# Vector Field Contours

T. Annen[*]
MPI Informatik
Germany

H. Theisel[†]
University of Magdeburg
Germany

C. Rössl[‡]
University of Magdeburg
Germany

G. Ziegler[§]
MPI Informatik
Germany

H.-P. Seidel[¶]
MPI Informatik
Germany

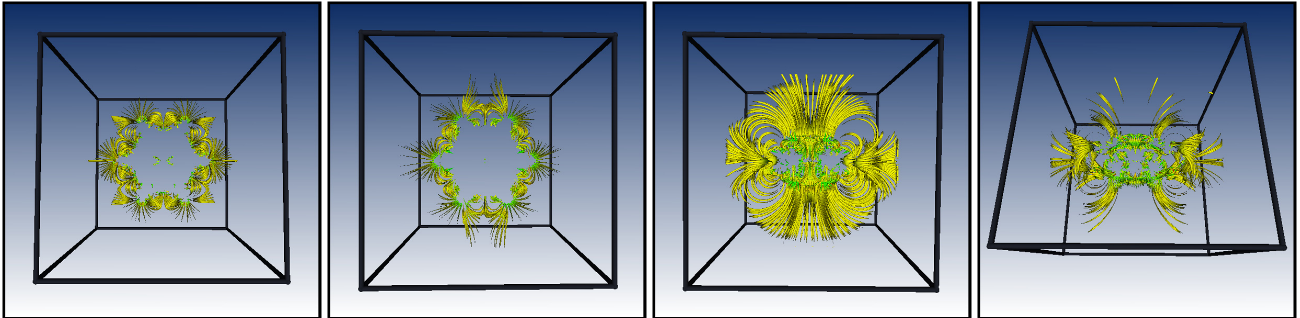


Figure 1: Vector field contours of the electrostatic field around a benzene molecule from different view directions. Using our new technique allows to select those stream lines which have tangent direction and osculating plane perpendicular to the current view direction in the seeding point.

**ABSTRACT**

We describe an approach to define contours of 3D vector fields and employ them as an interactive flow visualization tool. Although contours are well-defined and commonly used for surfaces and 3D scalar fields, they have no straightforward extension in vector fields. Our approach is to extract and visualize specific stream lines which show the most similar behavior to contours on surfaces. This way, the vector field contours are a particular set of isolated stream line segments that depend on the view direction and few additional parameters. We present an analysis of the usefulness of vector field contours by demonstrating their application to linear vector fields. In order to achieve interactive visualization, we develop an efficient GPU-based implementation for real-time extraction and rendering of vector field contours. We show the potential of our approach by applying it to a number of example data sets.

**Index Terms:** I.3.3 [Computer Graphics]: ,—Contours, Flow Visualization

## 1 INTRODUCTION

Contours, also called silhouettes, are a well-established and popular tool for the visualization of 3D shapes. Their expressiveness is the main benefit: the most significant surface features and hence the essential information is conveyed easily with only few line primitives.

Such information is easily comprehended, because human visual perception is highly trained for interpreting such contour based input. Contours are integral to technical illustration, and for more than a decade, non-photorealistic rendering (NPR) had adopted line drawing styles such artistic styles like pen-and-ink illustration [31]. In these, contours are of central importance as they efficiently convey the most relevant information.

Various kinds of contours of *surfaces* have been studied extensively, and they have emerged as standard features (see [6] and the references therein). For volume rendering, contours have been successfully applied to the visualization of 3D *scalar fields*: several approaches emphasize linear features by modulating opacity [5,8,18], or using curvature directions [14,21] or curvatures [16]. Typically, contours for scalar fields are defined as contours of implicit surfaces. There, the algorithmic focus is on efficient isosurface extraction or optimized extraction of the linear features [2, 4], possibly using high quality parametric curves [22] or graphics hardware to determine discrete silhouette pixels [20]. We remark that there is also considerable work on frame-coherent interactive methods and visualization of time-varying fields, however, this is not within the realm of this paper.

*Vector fields* constitute an important class of data in scientific visualization. They frequently result from numerical simulations such as computational fluid dynamics. Due to the size and complexity of the data, direct visualization of such 3D flow data is extremely challenging. The success of contours for representing surfaces and scalar fields and the fact that the human visual system is adapted at recognizing and interpreting contours motivates the central goal of this paper: determining vector field contours for flow illustration. We are only aware of one previous approach to using contours in vector field visualization: Svakhine et al. [25] consider those surfaces where the vector field is perpendicular to a certain view direction and use it to enhance the illustration of the flow. Contrary to this work, our approach extracts and processes geometry, i.e., line structures, based on additional criteria.

When considering *vector field contours* for visualization, we observe that:

1. A straightforward extension of the concept of contours to vector fields is *not* possible, since the original definition of a contour depends on the existence of underlying (iso)surfaces and their normals and such surfaces are generally undefined for 3D vector fields.

2. Stream lines have proven to give expressive visual representations if they are combined with appropriate seeding strategies.

*e-mail: tannen@mpi-inf.mpg.de
†e-mail:theisel@isg.cs.uni-magdeburg.de
‡e-mail:roessl@isg.cs.uni-magdeburg.de
§e-mail:gziegler@mpi-sb.mpg.de
¶e-mail:hpseidel@mpi-sb.mpg.de

In fact, stream lines are perhaps the most common standard tool for visualizing vector fields.

Combining these observations, we propose to extract a number of isolated stream lines which display the most similar behavior to classical contours on surfaces. In this way, our approach can be considered a stream line seeding approach: depending on the view direction and additional parameters, the seeding points are defined by certain local conditions. Starting from these points, the stream lines are integrated both in forward and backward directions until their similarity to surface contours is beyond a certain threshold.

Contrary to pre-existing stream line seeding approaches for 2D [15,19,28,29] and 3D [32] vector fields, our approach is, to the best of our knowledge, the first that seeds in a view direction dependent manner. Therefore, our approach is only applicable as an interactive visualization tool if the view direction can be changed under quasi-interactive frame rates. In order to achieve this, an efficient interactive GPU implementation is necessary.

The rest of the paper is organized as follows: In Section 2 we describe our contour approach. In particular, we describe how to define the similarity of a stream line to a surface contour and how to obtain local conditions for the seeding structures. Then an analysis follows in section 3 which details on the properties and usefulness of vector field contours as a visualization tool. Section 4 describes a fast GPU-based implementation of our approach which enables interactive adjustment of contour parameters, such as the view direction, and therefore enables interactive explorations of the vector field. In Section 5 we apply our approach to a number of test data sets and show results. Conclusions are given in Section 7.

## 2 Definition of Vector Field Contours

In order to define contours of vector fields, we start with a short description of contours of surfaces and 3D scalar fields. Given a regularly parameterized surface $\mathbf{x}(u,v)$, a contour (we consider the original notion of contours as silhouette lines not including suggestive contours [6]) is defined by a (unit-length) view direction $\mathbf{r}$. Here, we consider only the case of constant view direction, i.e., parallel projection. Then contours are surface curves consisting of points with $\mathbf{r}\cdot\mathbf{n} = \mathbf{r}\cdot(\mathbf{x}_u \times \mathbf{x}_v) = 0$ where $\cdot$ denotes the dot product of vectors.

The main idea for the definition of contours of a 3D scalar field $s(x,y,z)$ for a given view direction $\mathbf{r}$, is to pick an isovalue $s_0$ and to define the contour of the isosurface $s = s_0$ as described above. This means that given $\mathbf{r}$ and $s_0$, the contour consists of all points of the domain of $s$ with $[\,\mathbf{r}\cdot\nabla s = 0 \,\wedge\, s = s_0\,]$.

In order to extend the concept of contours from scalar fields to vector fields, we begin with the observation that stream lines and stream surfaces are probably the most recognized and most important line and surface features in vector fields. In 2D, smart placement of an appropriate number of stream lines has been proven to deliver expressive visualizations [15,19,28,29]. The generalization to 3D vector fields is not straightforward, and sophisticated filtering of stream lines is required to process moderately complex data [32]. Instead of trying to capture all topological features simultaneously, we perform a careful real-time selection of an efficient set of representative stream lines which are closest to contours for a given viewing direction.

As input, we are given a 3D steady vector field $\mathbf{v}(x,y,z)$ in a volume domain $D$ and a view direction $\mathbf{r}$. We search for parts of stream lines that locally act as contours. Two steps are required to execute this search: first, we set the seeding structures, second, we integrate stream lines from the seeds in both forward and backward directions until a certain threshold is exceeded.

Consider the isosurface $\mathbf{a} \subset D$ which is defined by

$$\mathbf{r}\cdot\mathbf{v} = 0 \tag{1}$$

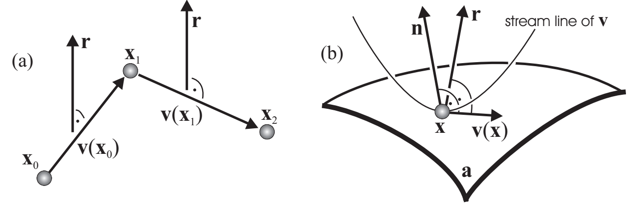



Figure 2: (a) A stream line $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, ...)$ locally acting as contour when: $\mathbf{r}\cdot\mathbf{v}(\mathbf{x}_0) = 0$ and $\mathbf{r}\cdot\mathbf{v}(\mathbf{x}_1) = 0$; (b) a point $\mathbf{x}$ is on the seeding structure if $\mathbf{r}\cdot\mathbf{v}(\mathbf{x}) = \mathbf{n}\cdot\mathbf{v}(\mathbf{x}) = 0$.

in $D$. Note that in general, $\mathbf{a}$ is neither a stream surface, nor a stream line is a surface curve on $\mathbf{a}$. Equation (1) corresponds to the condition $\mathbf{r}\cdot\nabla s = 0$ for contours of scalar fields. We provide two characterizations of the seeding structures of the vector field which stem from different interpretations and turn out to be equivalent:

1. We search for points $\mathbf{x} \in \mathbf{a}$ where one infinitesimal integration step preserves (1), i.e., the stream line starting from $\mathbf{x}$ acts locally as a contour. Consider an Euler integration (the Euler step serves as theoretical tool here. Our implementation uses forth order Runge-Kutta integration) with the step size $\varepsilon$, the condition can be formulated as $\mathbf{r}\cdot\mathbf{v}(\mathbf{x}_1) = 0$ with $\mathbf{x}_1 = \mathbf{x} + \varepsilon\,\mathbf{v}(\mathbf{x})$. Furthermore, since

   $$\lim_{\varepsilon\to 0} \frac{\mathbf{v}(\mathbf{x}_1) - \mathbf{v}(\mathbf{x})}{\varepsilon} = (\nabla\mathbf{v})\,\mathbf{v},$$

   this condition can be written as

   $$\mathbf{r}\cdot\mathbf{w} = 0 \tag{2}$$

   with $\mathbf{w} = (\nabla\mathbf{v})\,\mathbf{v}$. Figure 2a gives an illustration.

2. We search for all points $\mathbf{x} \in \mathbf{a}$ where the stream line starting from $\mathbf{x}$ is locally a surface curve of $\mathbf{a}$, i.e., $\mathbf{v}(\mathbf{x})$ is locally perpendicular to the normal of $\mathbf{a}$ in $\mathbf{x}$. Since the normal of $\mathbf{a}$ is parallel to $\nabla(\mathbf{r}\cdot\mathbf{v})$, we get the condition $\mathbf{v}\cdot\nabla(\mathbf{r}\cdot\mathbf{v}) = 0$. It is a straightforward exercise in vector algebra to show that this is equivalent to (2). Figure 2b gives an illustration.

The first formulation focuses directly on the local contour character of stream lines originating from $\mathbf{x}$. The second formulation is a little more abstract and based on the observation of tangent spaces of $\mathbf{a}$. Figure 2 illustrates both interpretations. The essential characterization of seeding structures in $\mathbf{a}$ is given by equation (2). It means that a contour is a part of a stream line with its osculating plane perpendicular to the view direction.

Enforcing (1) and (2) together yields a line structure $\mathbf{l}$ in $D$. The stream surface starting from $\mathbf{l}$ in both forward and backward direction corresponds to the surface $\mathbf{r}\cdot\nabla s = 0$ for a scalar field. There, the surface is further reduced by additionally considering the condition $s = s_0$, i.e., picking a particular isovalue. Since there is no equivalent to such isovalues for vector fields, we use the curvature of the stream lines

$$\kappa(x,y,z) = \frac{\|\mathbf{v}\times\mathbf{w}\|}{\|\mathbf{v}\|^3} \tag{3}$$

as a thinning criterion for $\mathbf{l}$: given two curvature values $\kappa_1 \le \kappa_2$, the thinned seeding structure $\mathbf{l}_c$ consists of all points $(x,y,z) \in D$ with

$$[\,(1)\,\wedge\,(2)\,\wedge\,\kappa_1 \le \kappa \le \kappa_2\,]. \tag{4}$$

If $\kappa_1 < \kappa_2$, $\mathbf{l}_c$ consists of (open or closed) lines, while for $\kappa_1 = \kappa_2$, $\mathbf{l}_c$ consists of isolated points.

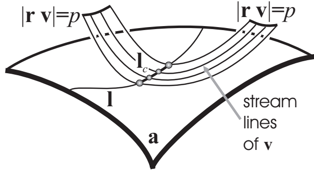



Figure 3: Concept of vector field contours: curvatures $\kappa_1 \leq \kappa \leq \kappa_2$ of the stream lines define a narrow band of seeding points on $\mathbf{l}$. Integration stops once stream lines get too far from contours, i.e., $|\mathbf{r} \cdot \mathbf{v}| \geq p$.

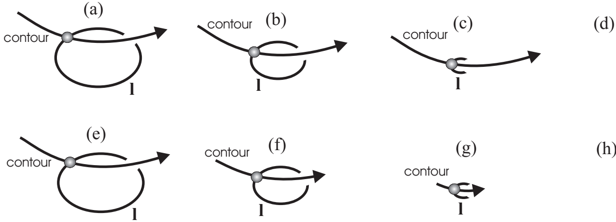



Figure 4: (a)-(d): seeding structure without relevance criterion: collapsing and disappearance of $\mathbf{l}$ leads to an abrupt disappearance of the contour; (e)-(h): seeding structure with relevance criterion gives a smooth disappearance of the contour.

The stream lines starting from $\mathbf{l}_c$ locally behave like contours. However, after leaving the seeding structure the contour-like behavior may be lost. We address this fact by monitoring $\mathbf{r} \cdot \mathbf{v}$ along the stream lines of $\mathbf{v}$: if $\|\mathbf{r} \cdot \mathbf{v}\|$ exceeds a certain threshold $p$, the integration stops. Figure 3 illustrates our concept of vector field contours.

The concept of vector field contours introduced up to now may still introduce temporal incoherencies while smoothly changing $\mathbf{r}$. The reason for this lies in the fact that the seeding structure defined by (4) may collapse and disappear while smoothly changing $\mathbf{r}$. Figures 4(a)-(d) illustrates an example for four directions of $\mathbf{r}$ adjacent to each other. The closed seeding structure $\mathbf{l}$ collapses to a point and disappears while smoothly changing $\mathbf{r}$. As a result, the contours starting from $\mathbf{l}$ disappear abruptly. As a solution for this, we introduce a *relevance criterion* $\rho$ for each point of $\mathbf{l}$ and incorporate this into the control of the length of the contours. Figures 4(e)-(h) illustrate this concept: a collapsing and disappearance of $\mathbf{l}$ leads to a smooth shortening and disappearance of the contour.

As the measure of relevance, we use

$$\rho = 1 - e^{-r \cdot \|\nabla(\mathbf{r}\cdot\mathbf{v}) \times \nabla(\mathbf{r}\cdot\mathbf{w})\|} \tag{5}$$

which can be computed locally for every point of $\mathbf{l}$. Note that $\rho$ is a number between 0 and 1: it becomes small if $\mathbf{l}$ is about to disappear. The number $r$ is a user-adjustable value steering the influence of the relevance criterion. The smaller $r$, the more impact does the relevance criterion have.

Once the relevance criterion is introduced, it steers the length of the contour in the following way: instead of stopping the integration of the contour if $\|\mathbf{r} \cdot \mathbf{v}\|$ exceeds $p$ (as introduced above as a first attempt), we stop the integration if $\|\mathbf{r} \cdot \mathbf{v}\|$ exceeds $\rho \cdot p$. Figure 11 illustrates the impact of the relevance criterion.

To summarize, our definition of vector field contours depends on the following parameters:

- $\mathbf{r}$ – view direction
- $\kappa_1, \kappa_2$ – curvature thresholds for seeding structure selection
- $p$ – measure of "contourness" to control the length of contours.
- $r$ – measure to steer the impact of the relevance criterion to the length of the contour.

In the subsequent sections we will present an analysis of vector field contours and show how this concept can be implemented efficiently for interactive flow visualization.

## 3 Analysis of Vector Field Contours

The human visual system is well-trained to recognize 3D shapes from their contours. This fact has been used is a variety of approaches to represent shapes by their contours. This is also the motivation of expanding the concept of contours to vector fields. However, contrary to 3D shapes the human visual system is not adapted to recognizing vector fields from their contours. This is because contours have not been applied to vector fields before.

3D shapes are recognized by the human visual system on an everyday basis. This is not the case for vector fields, describing flows and other phenomena: flows (e.g. of air) is rather invisible. Only its impact to certain objects may let the human infer its behavior. Therefore, vector field contours share a problem with all flow visualization techniques: they cannot rely on a human visual system which is well-trained for this.

In order to make vector field contours applicable, we have to analyze which properties of a vector field can be recognized by contours. We do so by analyzing linear vector fields of the kind

$$\mathbf{v}(\mathbf{x}) = \mathbf{J}\,\mathbf{x} \tag{6}$$

where $\mathbf{J}$ is the constant $3 \times 3$ Jacobian matrix. This vector field has a critical point at $(0,0,0)$. It is a well-known fact that its classification is obtained by an eigen-analysis of $\mathbf{J}$. Moreover, it is well-known that the critical points and the topological skeleton starting from them gives a rather complete description of a vector field [12], and that structurally stable critical points can be represented by a first-order approximation similar to (6) [1]. Therefore, we have to show that vector field contours can reveal the eigenvectors of $\mathbf{v}$ defined by (6).

Given $\mathbf{v}$ by (6), it is a straightforward exercise in algebra to show that $\mathbf{w}$ is a linear vector field as well which can be written as

$$\mathbf{w}(\mathbf{x}) = \mathbf{J}\,\mathbf{J}\,\mathbf{x} \tag{7}$$

This gives that for a given view direction $\mathbf{r}$ the seeding structure $\mathbf{l}$ is a straight line through the origin which can be generally written as

$$\mathbf{l}(t) = t \cdot (\nabla(\mathbf{r}\cdot\mathbf{v}) \times \nabla(\mathbf{r}\cdot\mathbf{w})). \tag{8}$$

It turns out that the eigenplanes of $\mathbf{J}$ (i.e., the eigenvectors of $\mathbf{J}^T$) can be observed in an intuitive way using vector field contours: if $\mathbf{r}$ moves over the direction of an eigenplane, (8) does not hold as solution for $\mathbf{l}$ any more. Instead, the whole eigenplane becomes the seeding structure, and all stream lines in the eigenplane become contours. Moreover, they do not leave this plane and therefore have maximal "contourness" and length. This means that by interactively moving $\mathbf{r}$ to explore $\mathbf{v}$, the eigenplanes can be found by searching for directions where the contours have a maximal number and length. Figures 9, 10 and the accompanying movie show an illustration.

## 4 Extraction and Visualization

In order to make vector field contours applicable as an interactive analysis tool, interactive frame rates must be achieved for extraction and visualization of contours while changing the parameters $\mathbf{r}, \kappa_1, \kappa_2, p$. We achieve this by an efficient GPU implementation of the algorithm and moderate preprocessing of the input data.

As usual, we assume that the vector field $\mathbf{v}$ is given as discrete samples on a regular grid defining a piecewise trilinear function.

We precompute an approximation of $\mathbf{w} = (\nabla\mathbf{v})\,\mathbf{v}$ as a second vector field over the same grid in the following way: $\mathbf{w}$ is estimated in the grid points by using central differences to estimate $\nabla\mathbf{v}$. Then the samples of $\mathbf{w}$ are trilinearly interpolated inside the grid cells.

For given $\mathbf{v}$ and $\mathbf{w}$, the extraction of $\mathbf{l}$ is equivalent to intersecting isosurfaces of $\mathbf{r}\cdot\mathbf{v}$ and $\mathbf{r}\cdot\mathbf{w}$. For this, efficient CPU realizations exist [4], which may trade speed for accuracy in the sense that there is no guarantee that all intersection curves are detected. Our technique differs from [4] in both, accuracy and efficiency. We guarantee the detection of all intersection points between the isosurfaces and the underlying grid faces while using a high-performance GPU-based approach. Efficient extraction is especially important because subsequent visualization steps frequently consume many resources depending on style choices and the size of data sets. Often this drastically influences the overall performance of the application.

Implementation details are presented in the subsequent sections which discuss the efficient generation of intermediate data structures. These seeding structures $\mathbf{l}_c$, represented as point sets, are finally represented as three packed textures. Such textures can be used as an input for many visualization techniques. We use them as starting point for rendering our vector field contours.

### 4.1 Seeding Structure Extraction

Extracting seeding structures associated with a certain view direction involves several intermediate rendering passes before the results can be used for visualization. An explanatory overview of our GPU-based rendering of vector field contours is given in Figure 5. The input data consists of the original vector field $\mathbf{v}$ and the finite difference approximation of $\mathbf{w}$ (see above) and curvature samples $\kappa$, which are obtained from (3). Note that $\mathbf{w}$ and the scalar field $\kappa$ are precomputed once for a given data set and are then used as input.

First we stream entries of both fields, $\mathbf{v}$ and $\mathbf{w}$, into a shader that computes the dot product between $\mathbf{r}$ and each vector in $\mathbf{v}$ and $\mathbf{w}$, respectively. Results are stored in a new 2D off-screen texture $STk$ (we arrange the 3D vector field slices into a larger 2D texture to ease intermediate computations) with three channels: the right-hand-sides of (1) and (2) are stored in red and green channels, respectively. We refer to these entries as $S$ and $T$. The blue channel, labeled $k$, contains the constant $\kappa$ which was loaded during initialization.

Each texel in $STk$ represents a reference vertex of a grid cell which is shared by three cell faces (see Figure 6). Points where both isosurfaces intersect one of these faces are computed by the intersection shader in three additional rendering passes. The output is a texture with only very few of the total number of texels encoding intersections. Using such texture as is would be inapplicable for visualization due to bandwidth limitations. Instead, we take advantage of the sparseness of the intersection data.

Therefore, we apply an efficient texture packing algorithm to efficiently convert sparse into the final compact seeding structure representation $\mathbf{l}$ (Figure 5). In the following we describe each part of the algorithm in more detail.

**Data representation:** Both vector fields $\mathbf{v}$ and $\mathbf{w}$ are stored as 3D textures using 16-bit floating point precision. Here, each texture tile of $2\times 2$ texel determines a grid cell front face (z face) as depicted in Figure 6. Half-float precision is favorable as state-of-the-art GPUs support trilinear filtering in hardware for this format. All other textures are IEEE 32-bit 2D float images. Prior to extraction we first fill the blue ($k$) bit plane of $STk$ texture with the precomputed curvature values. We mark it for read-only access as these values remain constant for a given data set.

The $STk$ texture size is determined by the number of slices of $\mathbf{v}$ combined with the individual slice resolution. Our current implementation trades texture memory for speed and uses the nearest power of two size as this alleviates the computational load of texture packing. In Figure 5 unused texture memory appears as an empty black column right at the end of the slices.

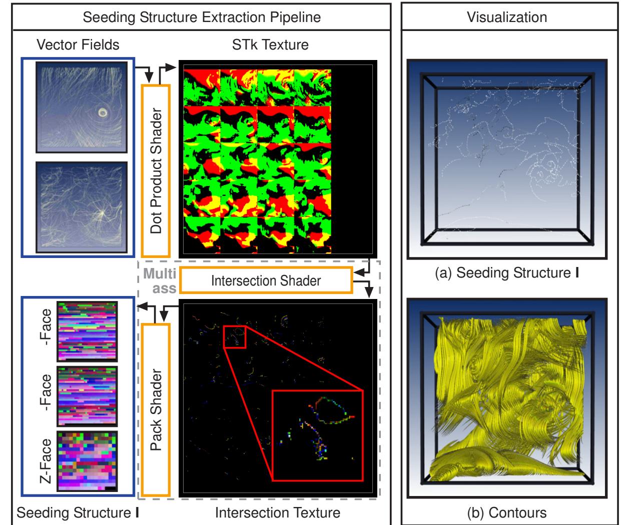



Figure 5: The pipeline for vector field contour computation (left) and visualization (right). A multi-pass process computes x, y, and z-face intersections and converts sparse intersection textures into a packed seeding structure **l**. Right (a) shows the resulting seeding structure found during the extraction process and (b) illustrates contours rendered as stream tubes.

**Intersection:** At the beginning of the extraction process a simple shader kernel fetches samples from both vector fields, computes (1) and (2) using the view direction and writes $S$ and $T$ into texture memory. View direction $\mathbf{r}$ is sent as shader parameter each time we execute the shader. For unfolding all vector field slices into a 2D texture we use a proxy geometry in the form of quads that have the same dimension as one slice. Each quad vertex is equipped with 3D texture coordinates where the $z$ coordinate indexes slices and $x, y$ perform inner slice addressing. Each time we unfold a slice we shift the viewport before each rendering to the new position in the large $STk$ texture. As soon as $STk$ texture computation has finished we bind it as an input for subsequent intersection tests. We search for those discrete intersections between two isosurfaces which are constrained to faces of grid cells. As intersections of isosurfaces with such faces are hyperbolas [27], the local search for each face consists only of finding roots of a quadratic equation. The coefficients of this equation are given as samples in our input texture $STk$.

We have to distinguish between $x$, $y$, and $z$ faces as texture addressing is different for each of them. Figure 6 illustrates a sample grid cell, its corner vertices and the faces shared by a reference vertex. The easiest case are intersections with $z$ faces: all vertices of the face are adjacent within the same slice as the base vertex (lower left here). However, for $x$ and $y$ faces we have to sample two texels from a neighboring slice (Figure 6). Using this scheme all data required to intersect both isosurfaces is available in the shader. Depending on the values $S$ and $T$ stored at the grid cell corners we can have either zero, one or two intersections (for instance $P_0$ and $P_1$ in Figure 7a).

Once we have identified those intersections which are in parametric range $[0,1]$ we can directly store those coordinates in the RGBA components of the fragment data. Later on during the texture packing we reconstruct the current position within the volume through a simple mapping using the texture coordinates. Results can therefore immediately be used for visualization.

Testing for intersections is done for each class of the faces $x$, $y$

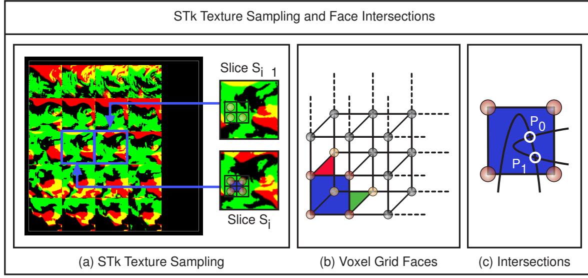



Figure 6: Grid cell faces and adjacency sampled in 2D. Four vertices from the z-face are four texel in the STk Texture slice $S_i$. The other three vertics required to compute all intersections are located in the next slice $S_{i+1}$.

and $z$ separately. Additionally, isosurface intersection generated incorrect results at slice boundaries as there are no associated faces. Checking for this case in the pixel shader is inefficient as it only applies for a small subset of fragments. Instead we simply overdraw a pattern to mask those region out. In total, this mask only consists of a few lines of one pixel width and an additional quad that masks the last slice. After the first intersection test is complete we have got a sparse intersection texture in which only a fraction of texels contribute to the seeding structure **I**. Using such kind of textures directly is infeasible for visualization. Thus we conduct an additional compaction pass to finally obtain packed textures illustrated in our pipeline (Figure 5).

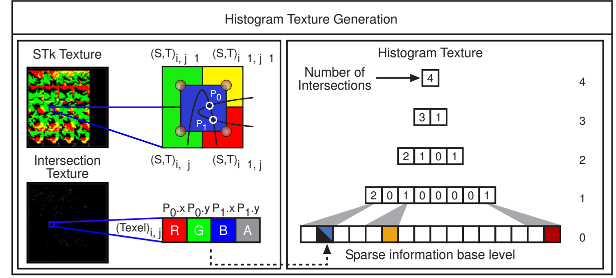
**Relevance criterion:** At the end of the intersection computation we check if both isosurfaces intersect in at least one point within the current voxel. If they do, we continue to compute the relevance value $\rho$ for this voxel according to Equation (5). The relevance value is then packed into the 32 bit floating point depth buffer attachment of the current frame buffer object rather than another color attachment. This is significantly faster, since the GL_EXT_framebuffer_object extension requires all color attachments to have the same format. So we would have to clear and write to 32 bit float RGBA channels otherwise. This would drastically reduce extraction performance.

The relevance texture has entries at exactly the same locations as the sparse intersection texture. One can imagine it as just an additional 5th component layer. In the packing process, described in the next section, this layer can be treated in almost the exact same way as the intersection texture. We will discuss the marginal shader code differences in the next section.

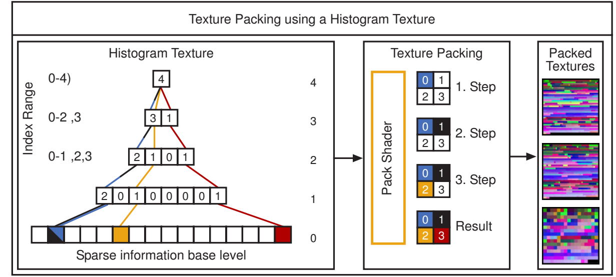
Later on, when we visualize contour stream lines we convolve the stop ("contourness") criterion $p$ with the relevance value $\rho$ to determine the final integration length for each stream line.

**Texture packing:** Sparse texture analysis is a common problem in graphics hardware programming. [13] introduces the concept of data compaction, i.e. filtering away unwanted data elements from a given data stream in $\log n$ iterations of successively producing a running sum. Alternatively, bitonic GPU-based merge sort [9] could be used for binary partitioning only, however, in $O(n(\log n))$.

The functionality of our texture packing algorithm is described in Figures 7 (a) and (b). The goal is to find and store all those texels in an image that contribute information and to ignore all other, irrelevant entries. Efficient queries are the key to high performance texture packing in stream processing architectures. Our solution introduces the *histopyramid texture*, an acceleration data structure that lets us retrieve each key index with $\log_2(\max\{xdim, ydim\})$ texture lookups. The building process for a histopyramid texture is adopting a well-known technique known as "reduction operation", a variant of custom mipmapping (see also [3]) which is able to sum $n^2$ elements in $\log_2 n$ passes building a Laplacian pyramid of par-




(a)




(b)

Figure 7: Building a histopyramid texture from the intersection textures (a). Each texel in the intersection texture can encode up to two face intersections in RGBA. (b) Histopyramid textures are used for the conversion from sparse into packed textures. The pack shader also converts from 2D texture space into 3D object space coordinates (using texture coordinates and intersection points) while writing the final result.

tial histograms. For fast rendering, we need to be able to output into mipmap levels of textures. This is provided by the OpenGL extension `GL_EXT_framebuffer_object`. In contrast to regular mipmapping, which averages the levels below, we sum up the valid intersections that were found in the $2 \times 2$ cells of the mipmap level underneath the current one. Figure 7a (right) illustrates such a mipmap pyramid. Level $L_0$ contains all possible combinations of intersections that we may encounter in this simple example. We can either have one (orange and red), two (split blue and black) or no intersection at all (white), and store this count in level $L_1$.

Now that level $L_1$ has counted the presence of data elements in level $L_0$, we repeatedly collapse four input cells with the reduction operator, starting at level $L_n$ and writing the resulting sum into the corresponding output cell at level $L_n + 1$. The iteration finishes when only one output cell remains. This single cell, situated at level $L_{\max}$, contains the total number of intersections and determines the packed texture size, with 2D dimensions based on $\lceil\sqrt{L_{\max}}\rceil^2$ of the number of entries.

The newly created histopyramid texture now enables efficient texture queries based on a given key index. This key index is a continuous integer, which we derive from the texel position in the packed output image. Now, we start at the top level of the histopyramid and descend every time the key index is within the range of indices covered by one cell's descendants (see Figure 7). The traversal stops at the base level, where it finds the intersection result with the corresponding key index. Finally, we convert the coordinates of the target cell and its content into 3D object coordinates, and output this result into the packed texture. (Note that since the packed texture is 2D, there can be more texels than key indices. Therefore, the pack shader discards all fragments whose index exceeds the number of valid entries.)

Our current implementation enhances the basic histopyramid building and packed texture retrieval with vectorization. We store the partial sum of a $2 \times 2$ region in RGBA components, and first add up the cell's sum vector when the cell becomes a partial sum at the next higher level itself. This way, we avoid unnecessary texture lookups to compute index ranges, as we can predict the index range of each leaf cell from the parent's sum vector without actually *accessing* the leaf cell. Note that in this implementation, the top level still contains a sum vector. Its four entries have to be added up on the CPU to get the total amount of entries in $L_0$.

After packing an intersection texture we immediately pack its corresponding relevance value layer because we can reuse the histogram texture. Remember that both sparse textures, the intersection as well as the relevance texture, have their data entries at the exact same positions. Then packing the relevance texture layer only requires small changes in the packing shader. Firstly, we only work on a scalar value which doesn't need to be mapped into another domain. Hence, there is no coordinate transformation necessary. Secondly, indices are unambiguous, because there can only be one data entry in each texel.

It should be noted that only image dimensions which are identical powers of two can provide the algorithm with a constant number of input and output cells. The histogram computation algorithm itself could easily be adapted to rectangular and non-power-of-two textures. However, current GPU programmability restrictions (namely, the inability to provide explicit texture sizes for each pyramid level) would then severely limit the performance of the subsequent packing algorithm. In the meanwhile, we choose to pad the input dimensions of the image in order to be equal powers of two.

As soon as all three textures are packed we have everything necessary to start visualizing vector field contours. This is the topic of the next section.

### 4.2 Contour Visualization

There exist some techniques for interactive visualization of vector fields, for instance [17, 24]. Our aim is to visualize vector field contours, i.e., stream line segments emerging from the seeding structure. The seeds (represented as packed textures) are fed into a stream line integrator. A similar approach is used in [17] for particle tracing. There, textures are used to inject a large number of particles into a flow. Following [17], we employ a fourth order Runge-Kutta formula for accurate integration. After every single integration step, new positions are written into an off-screen texture which is then used as input for the next step. Integration continues within the domain as long as stream lines locally resemble contours (see Section 2). This criterion is expressed as $\|\mathbf{r} \cdot \mathbf{v}_{i+1}\| \leq (\rho \cdot p)$ and can be evaluated directly. In addition, we ultimately limit the maximum number of integration steps for practical reasons. In our examples up to 400 integration steps we used depending on the number of seeding point which determine the read back burden. The result of each stream line integration is a polyline. We finally render all polylines using the approach in [24]. This enables us to also encode the local "contourness" $\|\mathbf{r} \cdot \mathbf{v}\|$ of a stream line by setting color and thickness relative to the threshold $(\rho \cdot p)$.

We conclude this section with the following remarks. In fact, the setup of stream lines is currently the bottleneck of our implementation: even though state-of-the-art PCI Express graphics cards allow very fast data exchange with the host performance is hindered by the large number of read backs required depending on the number of seed points multiplied by the number of intergation depth. In practice, we often have to stream back textures more than 200 times (see also Table 1). We are aware of sophisticated workarounds with some potential to alleviating these limitations.

However, we see more elegant and promising solutions in reach with the announced Shader Model 4: with the introduction of the geometry shader, which can create new geometry, performance of stream line integration will be increased significantly. We have decided to leave these optimizations for future work. This decision is also justified by the fact that our extraction can be done in real-time still allowing interactive frame rates for all our examples. We finally emphasize that the focus of our GPU-based implementation is on the efficient extraction of seeding structures using texture packing.

| Data Set | Size | Ext. | Vis. | SLI | SP |
|---|---|---|---|---|---|
| Isabel | $100^2 \times 50$ | 30 | 2-8 | 200 | $< 5519$ |
| Benzene | $100^3$ | 21 | 5-8 | 250 | $< 9852$ |
| ABC | $64^3$ | 64 | 8-12 | 400 | $< 787$ |
| Saddle | $64^3$ | 63 | 8-16 | 400 | $< 186$ |
| Focus | $64^3$ | 63 | 11-13 | 400 | $< 180$ |

Table 1: Summary of our example data sets. The columns show the name, size, performance of extraction and visualization (both in frames per second), (maximum) number of stream line integration steps, and number of seed points, respectively.

## 5 Results

We apply our approach to a number of test data sets. The results are shown in this section.

We start with a synthetic benchmark: The so-called ABC (Arnold-Beltrami-Childress) flow field

$$\mathbf{v}(\mathbf{x},t) = \begin{pmatrix} (A + (1 - e^{-t/10})\sin(2\pi t))\sin z + C\cos y \\ B\sin x + (A + (1 - e^{-t/10})\sin(2\pi t))\cos z \\ C\sin y + B\cos x \end{pmatrix}$$

has recently attracted attention in the fluid dynamics community because it describes an unsteady solution of Euler's equation [11].

We visualize $\mathbf{v}$ for $A = \sqrt{3}, B = \sqrt{2}, C = 1, t = 0$ within the domain $[0, 2\pi]^3$. The data set is sampled on a regular $64^3$ grid. Figure 11 (i)–(l) shows visualizations for different view directions. Figure 11 (a)–(d) show the color-coded relevance criterion for a disappearing seeding structure. The impact of the relevance dependent scaling of stream lines on temporal coherence is emphasized in Figure 11 (e)–(h).

Figure 1 visualizes the electrostatic field around a benzene molecule from varying view directions. This data set was computed on a $100^3$ regular grid using the fractional charges method described in [23]. This data set is highly symmetric and rather complex. It has been used for demonstration of other visualization techniques recently [26, 30]

Figure 8 shows visualizations of the Hurricane Isabel data set, which recently has been considered in a number of papers [7, 10]. Using vector field contours the wind tunnels are well visible under the according view directions. We used the vector field at the starting time of the simulation on a regular a grid. In Figure 9 we visualize a saddle and Figure 10 shows vector field contours of a repelling focus. This examples refer to the analysis we presented in section 3. Our technique can reveal inherent structures of linear vector fields by identifying its eigenvectors.

The screen shots and the accompanying videos demonstrate the potential of vector field contours as an interactive flow visualization tool. Table 1 summarizes information on the data sets of our GPU-based visualization. Performance was measured on a computer equipped with a 2.6GHz AMD CPU and a GeForce FX7800 graphics card with 512 MByte memory. The timings confirm the efficiency of our approach.

## 6 Limitations

As every flow visualization technique, our technique is not perfectly suited for every data set. In fact, our technique of vector field con-

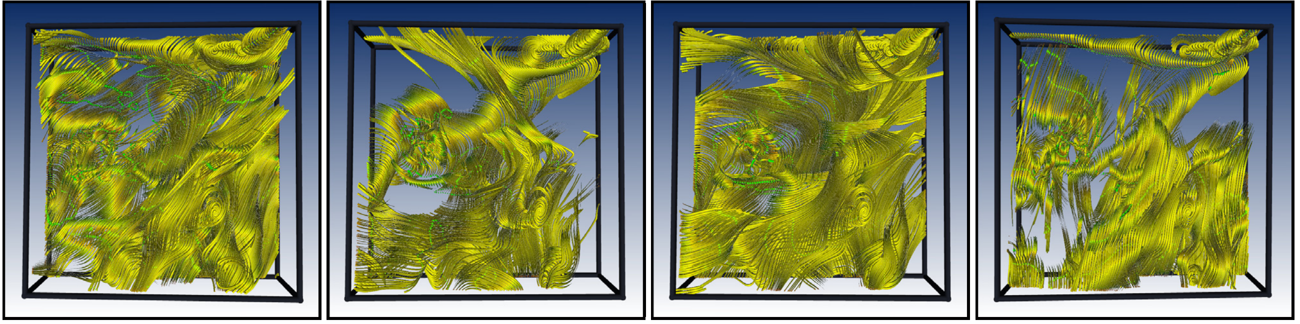


Figure 8: Hurricane Isabel data set: vector field contours rendered as stream tubes. Please note that the tunnels of the storm are well separated in certain views from other stream lines.

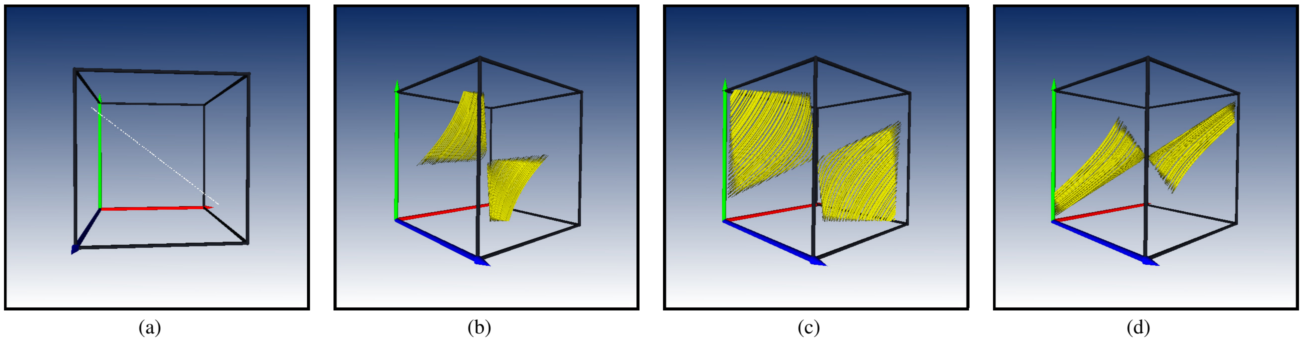



Figure 9: A saddle. (a) illustrates the seeding structure. (b)–(d) capture the sweep through the eigenplane. The view (c) corresponds to the eigenvector $(1,0,-1)^T$ of the linear vector field.

tours shares the property of scalar contours that there are configurations in which too many or not enough contours appear, making the visualization cluttered or sparse. For vector field contours, this happens if a 3D vector field is in fact pseudo-2D, i.e., on vector component is everywhere almost vanishing. In this case, all stream lines are contours for a particular view directions, where for any other direction no contours appear.

## 7 Conclusions and Future Research

The visualization of 3D vector fields constitutes a challenging problem and is an active area of research. Any approach to direct visualization by rendering certain stream lines must carefully select the most significant lines even for moderately complex data. This leads to stream line seeding methods: these are well-established for 2D vector fields. However, good direct visualization of 3D flow fields is much more involved, and so far we are only aware of one non-trivial seeding strategy [32]. At the same time we realize that NPR techniques and in particular contours have emerged as a powerful tool for the visualization of surfaces and 3D scalar fields. Here, significance of features is prioritized by human visual perception, and relatively few line primitives are required to convey the essential information. We are aware of only one approach that takes advantage of such NPR-like techniques to emphasize features in 3D flows [25].

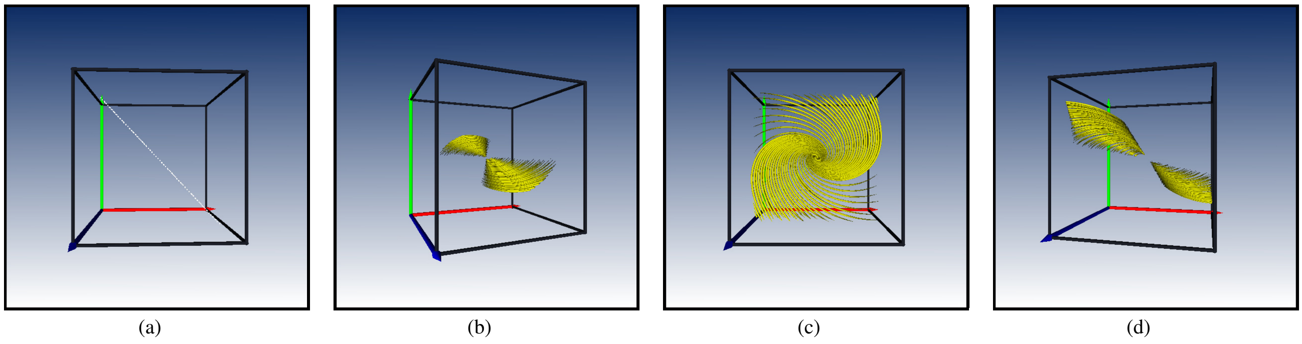
We presented a new approach to 3D flow visualization introducing *vector field contours*. Our method is related to the mentioned recent approaches in that it computes certain seeding structures based on ideas from NPR. In particular, a novel notion of contours for flow data is used to select contour-like stream lines for rendering. This combination makes our approach different to others. Most importantly and contrary to previous work, our visualization method is not static but is designed as a tool for dynamic, view-dependent exploration of flow data. Contours depend on the view direction and only a few other parameters which are subject to change. As vector field contours are not intuitive to our visual system we have conducted an analysis to show that our contour technique reveals interesting structures such as the eigenvectors of linear vector fields. We present a sophisticated implementation on the GPU that enables real-time visualization. In fact, seeding *and* rendering is achieved in real-time, which was not even achieved for 2D vector fields before. By utilizing latest graphics hardware feature sets we are able to map our entire extraction algorithm to the GPU. We see further possibilities for performance tuning in the announcement of shader model 4.0 and the accompanying geometry shader. Our results show the potential of vector field contours for flow visualization.

Furthermore, we achieve inter frame coherence by scaling the length of stream lines using an additional parameter which determines the relevance of each seed point. Please see the supplementary videos for an illustration and the difference in coherence when such a mechanism is absend.

Other possible research directions are the extension of *suggestive* contours [6], extensions to time-varying data, and combinations with other GPU-based interactive techniques like for instance particle tracing.

## 8 Acknowledgments

We thank Tino Weinkauf (ZIB) for his constant support and fruitful discussions. We thank Hans-Christian Hege for providing the Benzene data set. Also thanks to Kaleigh Smith and Wolfram von Funck for their help. The Hurricane Isabel data produced by the Weather Research and Forecast (WRF) model, courtesy of NCAR and the U.S. National Science Foundation (NSF).



(a) (b) (c) (d)

Figure 10: A repelling focus. (a) illustrates the seeding structure. (b)–(d) capture the sweep through the eigenplane. The view (c) corresponds to the eigenvector $(0,0,-1)^T$ of the linear vector field.

(a) (b) (c) (d)

(e) (f) (g) (h)

(i) (j) (k) (l)

Figure 11: The influence of the relevance criterion vizualized on a vanishing line loop. (a)–(d) show the color-coded seeding structure for the ABC dataset. The relevance increases from dark to light color. The importance of the line structure in the red box decreases from left to right. (e)–(h) illustrate the corresponding illuminated stream lines without using the relevance criterion. Note how the length of the stream lines decreases rapitly in (g) and finally vanishes completely (h). (i)–(l) show stream lines which are scaled accodring to the relevance criterion and therefore increase the temporal coherence of renderings.

## REFERENCES


[1] D. Asimov. Notes on the topology of vector fields and flows. Technical report, NASA Ames Research Center, 1993. RNR-93-003.

[2] D. Bremer and J. Hughes. Rapid approximate silhouette rendering of implicit surfaces. In *Implicit Surfaces*, pages 155–164, 1998.

[3] I. Buck and T. Purcell. A toolkit for computation on gpus. In *GPU Gems*, pages 621–363. Addison-Wesley, 2004.

[4] M. Burns, J. Klawe, S. Rusinkiewicz, A. Finkelstein, and D. DeCarlo. Line drawings from volume data. *ACM Transactions on Graphics*, 24(3):512–518, 2005.

[5] B. Csébfalvi, L. Mroz, H. Hauser, A. König, and E. Gröller. Fast visualization of object contours by Non-Photorealistic volume rendering. *Computer Graphics Forum*, 20, 3:452–460, 2001.

[6] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella. Suggestive contours for conveying shape. *ACM Transactions on Graphics*, 22(3):848–855, 2003.

[7] H. Doleisch, P. Muigg, and H. Hauser. Interactive visual analysis of hurricane isabel. *VRVis Technical Report*, 2004-058, 2004.

[8] D. Ebert and P. Rheingans. Volume Illustration: Non-Photorealistic Rendering of Volume Models. In *IEEE Visualization*, pages 195–202, 2000.

[9] N. Govindaraju, S. Redon, M. Lin, and D. Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. In *Eurographics/SIGGRAPH Graphics Hardware Workshop*, 2003.

[10] K. Gruchalla and M. J. Immersive visualization of the hurricane isabel dataset. 2004.

[11] G. Haller. An objective definition of a vortex. *J. Fluid Mech.*, 525:1–26, 2005.

[12] J. Helman and L. Hesselink. Visualizing vector field topology in fluid flows. *IEEE Computer Graphics and Applications*, 11:36–46, May 1991.

[13] D. Horn. Stream reduction operations for gpu applications. In *GPU Gems 2*, pages 573–587. Addison-Wesley, 2005.

[14] V. L. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. *ACM SIGGRAPH*, 31:109–116, 1997.

[15] B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. In *Proceedings 8th Eurographics Workshop on Visualization in Scientific Computing*, pages 57–66, Boulogne, 1997.

[16] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *IEEE Visualization*, pages 513–520, 2003.

[17] 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), 11 2005.

[18] E. B. Lum and K.-L. Ma. Hardware-Accelerated Parallel Non-Photorealistic Volume Rendering. In *NPAR 2002*, pages 67–74, New York, 2002. ACM Press.

[19] A. Mebarki, P. Alliez, and O. Devillers. Farthest point seeding for efficient placement of streamlines. In *IEEE Visualization*, pages 163–170, 2005.

[20] Z. Nagy and R. Klein. High-Quality Silhouette Illustration for Texture-Based Volume Rendering. *Journal of WSCG*, 12:301–308, 2004.

[21] Z. Nagy, J. Schneider, and R. Westermann. Interactive Volume Illustration. In *Vision, Modeling and Visualization*, pages 497–504, 2002.

[22] S. Schein and G. Elber. Adaptive extraction and visualization of silhouette curves from volumetric datasets. *The Visual Computer*, 20(4):243–252, 2004.

[23] D. Stalling and T. Steinke. Visualization of vector fields in quantum chemistry. Technical report, ZIB Preprint SC-96-01, 1996. ftp://ftp.zib.de/pub/zib-publications/reports/SC-96-01.ps.

[24] C. Stoll, S. Gumhold, and H.-P. Seidel. Visualization with stylized line primitives. In *IEEE Visualization*, pages 695–702, 2005.

[25] N. Svakhine, Y. Jang, D. Ebert, and K. Gaither. Illustration and photography inspired visualization of flows and volumes. In *IEEE Visualization*, pages 687–694, 2005.

[26] H. Theisel, T. Weinkauf, H.-C. Hege, and H.-P. Seidel. Saddle connectors - an approach to visualizing the topological skeleton of complex 3D vector fields. In *Proc. IEEE Visualization 2003*, pages 225–232, 2003.

[27] J.-P. Thirion and A. Gourdon. The 3D marching lines algorithm. *Graph. Models Image Process.*, 58(6):503–509, 1996.

[28] G. Turk and D. Banks. Image-guided streamline placement. In *ACM SIGGRAPH*, pages 453–460, 1996.

[29] V. Verma, D. T. Kao, and A. Pang. A flow-guided streamline seeding strategy. In *IEEE Visualization*, pages 163–170, 2000.

[30] T. Weinkauf, H. Theisel, K. Shi, H.-C. Hege, and H.-P. Seidel. Topological simplification of 3d vector fields by extracting higher order critical points. In *Proc. IEEE Visualization 2005*, pages 559–566, 2005.

[31] G. A. Winkenbach and D. H. Salesin. Computer-Generated Pen-and-Ink Illustration. In *ACM SIGGRAPH*, pages 91–100, 1994.

[32] X. Ye, D. Kao, and A. Pang. Strategy for seeding 3D streamlines. In *IEEE Visualization*, pages 163–170, 2005.