Interactive Visualization of Three-Dimensional Vector Fields with Flexible Appearance Control

Han-Wei Shen, Guo-Shi Li, *Student Member*, *IEEE Computer Society*, and Udeepta D. Bordoloi

Abstract—In this paper, we present an interactive texture-based algorithm for visualizing three-dimensional steady and unsteady vector fields. The goal of the algorithm is to provide a general volume rendering framework allowing the user to compute three-dimensional flow textures interactively and to modify the appearance of the visualization on the fly. To achieve our goal, we decouple the visualization pipeline into two disjoint stages. First, flow lines are generated from the 3D vector data. Various geometric properties of the flow paths are extracted and converted into a volumetric form using a hardware-assisted slice sweeping algorithm. In the second phase of the algorithm, the attributes stored in the volume are used as texture coordinates to look up an appearance texture to generate both informative and aesthetic representations of the vector field. Our algorithm allows the user to interactively navigate through different regions of interest in the underlying field and experiment with various appearance textures. With our algorithm, visualizations with enhanced structural perception using various visual cues can be rendered in real time. A myriad of existing geometry-based and texture-based visualization techniques can also be emulated.

Index Terms—Flow visualization, vector field visualization, texture synthesis, appearance control, line integral convolution, volume rendering, graphics hardware.

1 Introduction

Effective analysis of vector fields plays an important role in many scientific, engineering, and medical disciplines. Various visualization techniques have been proposed in the past to assist the scientist in comprehending the behavior of the vector field. They can be loosely classified into two categories: geometry-based and texture-based methods. Geometry-based methods (such as glyph, hedgehog, streamline, stream surface [1], flow volume [2], to name a few) use shape, color, and motion of geometric primitives to convey the directional information in the proximity of user-supplied regions of interest in the vector field. Texture-based methods, such as spot noise [3], line integral convolution (LIC) [4], and IBFV [5], on the other hand, attempt to create a continuous visual representation for the vector field using synthetic textures to reveal the global characteristics of the underlying physical phenomena.

In two-dimensional vector fields or flows across a surface in three dimensions, the texture-based methods are capable of offering a clear perception of the vector field since the directions of the vector field can be seen globally in the visualization. For three-dimensional vector fields, however, the effectiveness is significantly diminished due to the loss of information when the three-dimensional data are projected onto a two-dimensional image plane. This drawback can be mitigated to some extent by providing additional visual cues. For example, lighting, animation, silhouettes, etc. can all provide valuable information about the three-dimensional structure of the data set. Comparing visualizations with different appearances also helps in understanding the anatomy of the vector field. Unfortunately, the high computational cost of 3D texture-based algorithms impedes the interactive use of visual cues. Another issue for 3D vector field renderings is occlusion, which significantly hinders visualization of internal structures of the volume. Interactivity becomes very important as a result: The user needs to be able to experiment freely with textures of different patterns, shapes, colors, and opacities and view the results at interactive speeds.

Recently, we proposed an interactive volume rendering framework, called *Chameleon*, to facilitate flexible appearance control when visualizing three-dimensional vector fields [6]. The relative inflexibility of existing texture-based methods is a result of the tight coupling between the vector field processing step and output texture generation step. We addressed this issue by decoupling the visualization pipeline into two disjoint stages. First, streamlines are generated from the 3D vector data. Various geometric properties of the streamlines are then extracted and converted into a volumetric form which we refer to as the *trace volume*. In the second phase, the trace volume is combined with a desired *appearance texture* at runtime to generate both informative and aesthetic representations of the underlying vector field. The two-phase method provides a general framework to modify the appearance of the visualization intuitively and interactively without having to reprocess the vector field every time the rendering parameters are modified. Just by varying the input appearance texture, we are able to create a wide range of effects at runtime. A myriad of existing visualization techniques, including geometry-based and texture-based, can also be emulated. Using consumer-level PC platform graphics hardware with dependent textures and per-fragment shading functionality, visualizations with

- *H.-W. Shen and U.D. Bordoloi are with the Department of Computer and Information Science, The Ohio State University, 395 Dreese Lab, 2015 Neil Ave., Columbus, OH 43210. E-mail: {hwshen, bordoloi}@cis.ohio-state.edu.*
- *G.-S. Li is with the Scientific Computing and Imaging Institute, University of Utah, 50 South Central Campus Dr., Room 3490, Salt Lake, City, UT 84112. E-mail: lig@sci.utah.edu.*

Manuscript received 30 Sept. 2003; revised 10 Nov. 2003; accepted 18 Nov. 2003.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-0094-0903.

enhanced structural perception using various visual cues can be rendered in real time.

In this paper, we extend the Chameleon algorithm with the following features: First, the user can now incorporate various procedural solid textures, such as cloud, fog, gas, etc. [7], to create visualizations. Using solid textures to visualize vector fields allows the user to focus on the global structure of the underlying field. It can also emulate certain experimental visualization techniques such as injecting dye or smoke into a flow field. The second enhancement to the Chameleon algorithm is to allow the user to update the trace volume interactively. Although the trace volume only needs to be created once and can be used for rendering different appearance textures, sometimes the user may wish to update the trace volume by inserting new flow lines or deleting unwanted ones. This is particularly useful when the user is navigating through a large data set and wants to explore different regions of interest in the data set. The third enhancement to the algorithm is the feature that allows the Chameleon algorithm to handle time-varying vector fields. Following the principal philosophy of decoupling the advection and rendering stages, the time-varying Chameleon algorithm creates the trace volume from a dense set of input pathlines. The main difference between the processing of steady state and unsteady vector fields comes from the fact that pathlines can intersect with themselves or each other, while streamlines do not. We address this issue by creating an initial trace volume and multiple update volumes as a result of the pathline voxelization. The time-varying data is visualized by rendering the trace volume and dynamically updating the trace volume. In addition to the above new features, we have adopted the latest graphics hardware such as nVIDIA GeForce FX and the high level shading language Cg to implement the Chameleon algorithm.

2 Related Work

Researchers have proposed various vector field visualization techniques in the past. In addition to the more traditional techniques such as particle tracing or arrow plots, there are algorithms that can provide a volumetric representation of the underlying three-dimensional fields. Some research had been directed toward integrating texture or icons into volume rendering of flow data. Crawfis and Max [8] developed a technique where the volume rendering was built up in sheets oriented parallel to the image plane. These sheets were composited [9] in a back-to-front order. The volume integral was modified to include the rendering of a tiny cylinder within a small neighborhood. A further refinement of this concept was to embed the vector icons directly into the splat footprint [10] used for volume rendering.

Line Integral Convolution, or LIC [4], developed by Cabral and Leedom, has been perhaps the most visible of the recent flow visualization algorithms. The algorithm takes a scalar field and a vector field as input and outputs another scalar field. By providing a white noise image as the scalar input, an output image is generated that correlates this noise function along the directions of the input vector field. While LIC is effective in visualizing 2D vector fields, it is computationally quite expensive. Stalling and Hege [11] proposed an extension to speed up the process. Shen at al. [12] proposed the advection of dyes in LIC computation. Kiu and Banks [13] used noises of different frequencies to distinguish between regions with different velocity magnitudes. Shen and Kao [14] proposed UFLIC for unsteady flow and a level of detail approach was proposed by Bordoloi and Shen [15]. Interrante and Grosch [16] introduced the use of halos to improve the perceptual effectiveness when visualizing dense streamlines for 3D vector fields. Rezk-Salama et al. [17] proposed a volume rendering algorithm to make LIC more effective in three dimensions. A volume slicing algorithm that utilizes 3D texture mapping hardware is explored to quickly adjust slice planes and opacity settings. More recently, Jobard et al. [18] proposed a Lagrangian-Eulerian Advection technique to visualize unsteady flows using hardware-assisted noise blending. Weiskopf et al. [19] used programmable graphics hardware to advect solid textures and animate moving particles. Van Wijk proposed a highly interactive Image Based Flow Visualization (IBFV) algorithm [5] for visualizing two-dimensional fluid flow using standard features of graphics hardware.

3 The Chameleon Rendering Framework

The primary goal of our research is to develop an algorithm that has a high degree of interactivity and flexibility. The traditional texture-based algorithm, such as LIC, is known for its high computation cost when applied to three-dimensional data. This high computational complexity makes it difficult for the user to change the output's visual appearance, such as texture patterns and frequencies, at an interactive speed. Although, in the past, researchers have proposed various texture-based rendering techniques for visualizing three-dimensional vector fields, there is no common rendering framework that allows an interactive mix-and-match of different visual appearances when exploring three-dimensional vector data. In this paper, we present our extended rendering framework to address this issue. In the following, we first give an overview of our approach and then present the details for the various stages of the algorithm.

3.1 Algorithm Overview

In LIC or similar texture-based algorithms, texture synthesis is performed to establish pixel or voxel value coherence along the flow paths for depicting the vector directions. In addition to the algorithms' high computational complexity, one challenge for employing such texture synthesis methods is that the information about the vector field is difficult to recover from the resulting textures once the computation is complete. Consequently, if the user wants to alter the visual appearance, such as the texture shape or distribution pattern, the synthesis process needs to be performed all over again.

To allow flexible runtime mapping of textures with user desired visual characteristics, it will be beneficial if the processing of the vector field and the synthesis of textures can be decoupled. Specifically, if the processing of the vector field can output an intermediate renderable form which allows for a flexible mapping of different textures, better appearance control in the visualization can be achieved. In this paper, we present a novel visualization algorithm based on this idea. The intermediate renderable form produced by our algorithm is a volumetric object, which will be referred to as the *trace volume*. The main

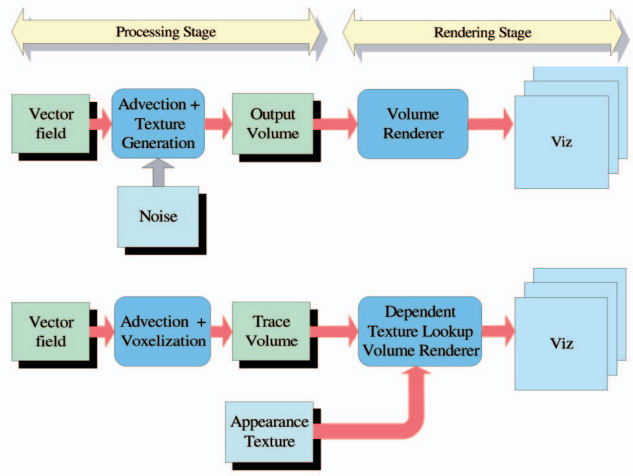


Fig. 1. Visualization pipelines for LIC (above) and Chameleon (below). The Chameleon decouples the advection and texture generation stages. Once the trace volume is constructed, any suitable appearance texture can be used to generate varied visualization of the same vector data set.

reason for choosing the volumetric form over other geometric representations, such as surfaces, lines, or points, is to display solid textures of various characteristics to reveal the global structure of the vector field. Although volume rendering used to be a computationally expensive process, the recent advancement of graphics hardware has made it possible to render volumes of moderate sizes at interactive speeds.

To depict the flow directions in the field, we store, at each voxel in the trace volume, a few attributes, called the *trace tuple*, which are used to establish visual coherence along the flow paths. Specifically, the attributes stored in the trace tuple are used as the texture coordinates to look up an input texture, which we will refer to as the *appearance texture*. The appearance texture contains precomputed 2D/3D visual patterns, which will be warped and animated along the flow directions to create the visualization. The appearance texture can be freely specified by the user at runtime. For instance, it can be a precomputed LIC image from a straight flow or can be a texture with different characteristics such as line bundles, particles, paint-brush strokes, etc. Each of these can generate a unique visual appearance. Our algorithm can alter the visual appearance of the data interactively when the user explores the underlying vector field and, hence, is given the name *Chameleon*. Fig. 1 depicts the fundamental difference between our algorithm and the more traditional texture-based algorithm such as LIC.

Rendering of the trace volume requires a two-stage texture lookup. Here, we give a conceptual view of how the rendering is performed. Given the trace volume, we can cast a ray from each pixel from the image plane into the trace volume to sample the voxels. At each step of the ray, we sample the volume attribute, which is a trace tuple. The components of this sampled vector are used as the texture coordinates to fetch the appearance texture. Visual attributes such as colors and opacities are sampled from the appearance texture and blended into the final image. Although here we use the ray casting algorithm to illustrate the idea, in our implementation, we use graphics hardware with fragment shaders and dependent textures to achieve interactivity.

In the following sections, we elaborate on each step of our algorithm in detail. We focus on the topics of trace volume construction and rendering, including voxelization, trace tuple and appearance texture configurations, anti-aliasing, incremental update of trace volumes, and interactive rendering. We then present the time-varying Chameleon algorithm.

3.2 Hardware-Assisted Trace Volume Creation

In this section, we describe the process for constructing the trace volume. We first assume that the underlying data set is a steady state vector field. Later, in Section 4, we extend our algorithm for time-varying fields.

In essence, the trace volume is created by voxelizing a dense set of input streamlines. Since the trace volume will be used as a texture input to 3D texture mapping hardware for rendering, as will be described later, it is defined on a 3D regular Cartesian grid. We note that there is no preferred grid type for our algorithm because the trace volume is constructed from the input streamlines instead of from the vector field itself. We use the method proposed by Jobard and Lefer [20] to control the density and the length of flow lines during the advection. The seeds are randomly selected and the flow lines are generated by the fourth-order Runge-Kutta method. An adaptive step size based on curvature [21] is used in the advection.

To convert the input flow lines into the trace volume, a hardware-assisted slice sweeping algorithm, inspired by the CSG voxelization algorithm proposed by Fang and Liao [22] is designed to achieve faster voxelization speed. The input to our voxelization process is a set of streamlines $\mathcal{S} = \{s_i\}$. Each streamline s_i is represented as a line strip with a sequence of vertices $\mathcal{P} = \{p_j\}$. Each vertex p_j will be given a three-dimensional vector, called the trace tuple, derived from the streamline geometry as well as the type of flow appearance to establish the visual coherence in the rendering. More specifically, the components of the trace tuple will be used as the texture coordinates to look up the appearance texture. In this section, we focus on the process of voxelization and defer the discussion of the trace tuple assignment to the next section.

We encode the trace tuples into the trace volume during the voxelization process using graphics hardware. Given an input streamline, we assign the trace tuple (u, v, w), according to the appearance schemes described in the next section as colors (red, green, blue) to the vertices of streamline segments. Using graphics hardware, our algorithm creates the trace volume by scan-converting the input streamlines onto a sequence of slices with a pair of moving clipping planes. For each of the X, Y, and Z dimensions, we first scale the streamline vertices by V/L, where V is the resolution of the trace volume in the dimension in question and L is the length of the corresponding dimension in the underlying vector field or a user-specified region of interest. Then, we render the streamlines orthographically using a sequence of clipping planes. The viewing direction is set to be parallel to the z axis, and the distance between the *near* and *far* planes of the view frustum is always one. Initially, the near and far clipping planes are set at $z = 0$ and $z = 1$, respectively. When each frame is rendered, the frame buffer content is read back and copied to one slice of the trace

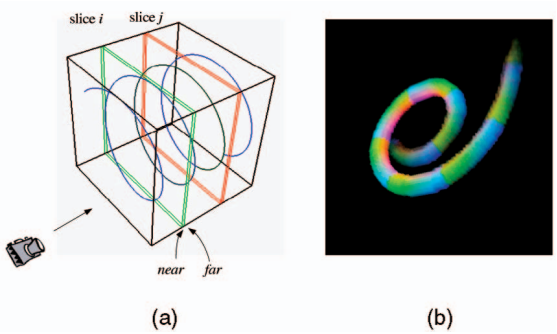


Fig. 2. (a) The slice sweeping voxelization algorithm. The near and far clipping planes are translated along the Z axis. At each position of the clipping planes, the streamlines are rendered to generate one slice of the trace volume. (b) A trace volume containing a collection of streamlines. The streamline parameterization is stored in the blue channel, while the streamline identifiers are stored in the red and green channels.

volume. As the algorithm progresses, the locations of the clipping planes are shifted by 1 along the Z axis incrementally until the entire vector field is swept. Fig. 2a illustrates our algorithm. Positions for the near and far clipping planes for two different slices are shown. Fig. 2b shows the voxelization result generated from a bundle of input streamlines, where different colors are used to encode the different values of trace tuples.

Sometimes it is possible that some of the streamline segments are perpendicular to the $Z = 0$ plane. For orthographic projection, these segments will degenerate into a point. In certain graphics APIs, such as OpenGL, the degenerate points are not drawn, which will create unfilled voxels in the trace volume. To avoid this problem, such segments are collected and processed separately in another pass, where the viewing direction and the sweeping of the clipping volume is set to be along the X-axis. The voxelization results of the new segments are added (logically OR'ed) into the trace volume using a method similar to the updating technique mentioned in Section 3.5.

The performance of the voxelization depends on the rendering speed of the graphics hardware for the input streamline geometry. To reduce the amount of geometry to render for each slice, streamline segments are placed into bins according to their spans along the Z direction. During the voxelization, only the segments that intersect with the current clipping volume are sent to the graphics hardware. The performance for constructing the trace volume can be further increased by reading the slicing result directly from the frame buffer to the 3D texture memory. This can be done using OpenGL's *glCopyTexSubImage3D* command.

3.3 Trace Tuple and Appearance Texture

As mentioned earlier, the set of attributes stored at each voxel in the trace volume is referred to as the trace tuple. A trace tuple is a three-dimensional vector which can be divided into two main components: the streamline identifier (u, v), which is used to differentiate individual flow paths, and the streamline parameterization (w), which is to parameterize the voxels along the flow line. Trace tuples are used as the texture coordinates to look up a 3D appearance texture. The values assigned to the trace tuple at each voxel are determined based on the type of the underlying appearance texture in use. In the following, we explain

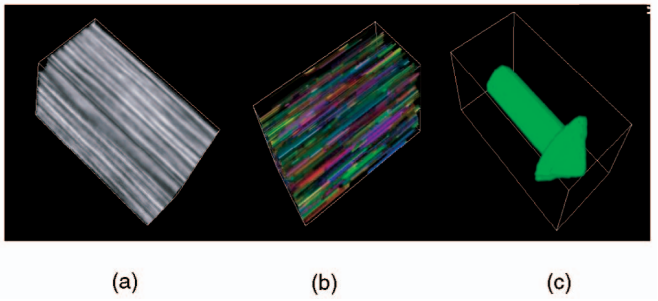


Fig. 3. Different appearance textures. (a) LIC, (b) line bundle, (c) arrow.

the configurations of the trace volume and the appearance textures in detail. Three types of appearances are currently supported by our algorithm: 1) stochastic lines, 2) local glyphs, and 3) global solid textures.

3.3.1 Stochastic Line Textures

LIC [4] or line bundles [23] are examples of stochastic line textures. In essence, these textures consist of a collection of 1D lines. Correlated values of luminance or color are assigned to each line for depicting the flow path, while different lines receive different colors stochastically to maintain the image contrast. To use this type of appearance in the visualization, we can precompute a 3D solid appearance texture from a straight flow using LIC or the line bundle algorithm. Fig. 3a and Fig. 3b show two examples of such 3D solid appearance textures. The 1D texture used to depict the flow direction is extended along the w direction in the texture space, while different (u, v) are used to represent different stochastic lines. To construct the trace volume, a collection of dense flow lines computed from the vector field are taken as the input. Each flow line will be assigned a randomly selected (u, v) tuple. Vertices that are on the same flow line will share the same (u, v), while the w values of the vertices along the flow path will be parameterized from 0 to 1 according to the arc length. When using the trace tuple as the texture coordinates to look up the appearance texture, the 1D straight line textures will be warped along the actual flow directions to create the visualization. It is worth mentioning that the trace volume only needs to be created once and different appearance textures of this type with various visual characteristics can be used at runtime without the need to recreate the trace volume. Fig. 4a and Fig. 4b show examples of using LIC and line bundle textures to create the visualization.

3.3.2 Local Glyph Textures

Graphical glyphs such as arrows, tubes, or spheres are commonly used in vector field visualization. These glyphs intuitively represent the vector directions and can also be rendered with enhanced shading effects to provide better depth cues. To display local glyphs in the vector field, we can voxelize glyphs of various shapes into 3D appearance textures. Fig. 3c shows an example of a voxelized arrow. For this type of appearance textures, special care is needed to compute the trace tuples. If we use a similar method as the one used for the stochastic line textures described above, i.e., randomly assigning the streamline identifier (u, v) to the input streamlines, the shape of the glyphs will not be

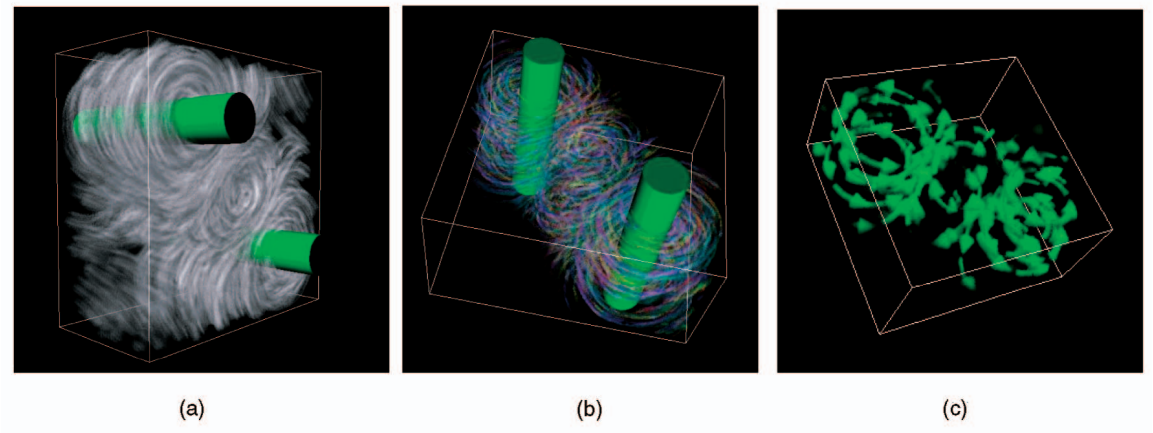


Fig. 4. Visualization of a vortices data set using (a) LIC, (b) line bundle, (c) arrow glyph appearance textures.

maintained in the final rendering since adjacent voxels may not be mapped to the adjacent texels in the texture space. To correctly map the appearance texture, we model each flow line as a bundle of thin lines surrounding a central line. This is done as follows: During advection, the streamlines are generated as a set of line segments. After the advection stage, each line segment is surrounded by a bundle of satellite lines, denoted as $\mathcal{B} = \{b_k\}$, where b_k is the kth satellite line in the bundle. The line bundle is created by extruding a mask $\mathcal{M} = \{m_k\}$ along the streamline during the advection process. Each point m_k on the mask corresponds to a vertex of the satellite strip. The distance between two adjacent strips should be small enough to avoid any vacant voxels within the thick line in the trace volume. Initially, the center of the mask is placed at the first vertex of the streamline. Then, the mask is swept along the streamline as the advection proceeds. During the sweep, the mask is always positioned perpendicular to the tangential direction of the streamline and the orientation of the mask is calculated based on the local curl of the flow, similar to the streamribbon construction algorithm used in [21] and [24]. When the advection of the central streamline completes, we construct the line strip b_k by connecting the vertices from the corresponding points in the mask along the sweep trace.

To determine the trace tuples for the trace volume voxels, all satellite lines in the bundle use the same w value as the corresponding points in the central streamline. The u and v coordinates of the satellite lines range from 0 to 1 according to their relative positions to the central line. The points at the central line always have the (u, v) identifier as (0.5, 0.5), which points to the central axis in the appearance texture along the w direction. When the trace tuples are organized this way, adjacent lines in the bundle are mapped to the adjacent texels of the appearance texture. Hence, any solid structure present in the appearance texture will be preserved after the trace volume is texture mapped. Fig. 2b shows the voxelization results for such a collection of lines where (u, v) values are encoded in the red and green channels and w is stored in the blue channel. Fig. 4c shows an example of using arrows as the appearance.

3.3.3 Global Solid Texture

Previously, researchers have created realistic rendering of nature phenomena such as cloud, fog, gas, etc. using procedural solid textures [7], where the intensity value of each texel is evaluated using stochastic noise or turbulence functions. To animate these fuzzy gaseous objects, a precomputed vector field is used to advect points in the solid texture space. The new positions of the points are input to the procedural texture module to evaluate the noise function at every frame to create the animation. Researchers have also proposed animations of global textures by advecting the texture coordinates defined at each grid node [25]. Using solid textures to visualize vector fields allows the user to focus on the global structure of the underlying field. It can also emulate certain experimental visualization techniques such as injecting dye or smoke into a flow field.

The Chameleon rendering framework supports the use of 3D solid textures as the appearance input to visualize the vector field. The main difference between this appearance and the previous two is that, when using stochastic lines or local glyphs, the same texture pattern is repeated everywhere to depict the flow paths. There is no attempt to create a global solid appearance covering the entire trace volume in the final visualization. To render and animate global solid texture appearances, the process for creating the appearance texture in the Chameleon framework needs to be modified. This is done as follows: Initially, a solid texture with the desired visual appearance that covers the entire trace volume domain is created. While it is straightforward to volume render this solid texture, the crux of the problem is how to advect the solid texture along the flow direction in the Chameleon framework. As mentioned previously, the input to the voxelization algorithm for creating the trace volume is a collection of dense flow lines. We assign each flow line a randomly selected (u, v), with its w ranging from 0 to 1 for the vertices along the line. With this arrangement, when the flow lines are voxelized, the voxels in the trace volume along each flow path correspond to a 1D array of texels in the 3D appearance texture space, which have a unique pair of (u, v) with w parameterized from 0 to 1. To correctly advect the input solid texture, we need to convert the solid texture to the appearance texture used by the Chameleon framework according to this trace volume configuration so that the global texture can advect along the flow paths when we shift the texture coordinate w using the animation technique described later.

To achieve this goal, we now describe how to configure and animate the appearance texture. First, we use the

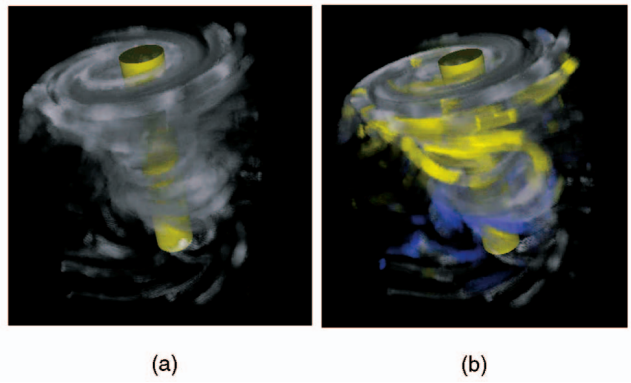


Fig. 5. Visualization of a tornado data set using: (a) a 3D turbulence function as the global texture, (b) dye advection.

positions of the input flow line vertices as the texture coordinates to sample the input 3D solid texture. This will result in one-dimensional line samples of the solid texture for each streamline. Then, we need to write these 1D samples to the corresponding 1D texels in the 3D appearance texture according to the flow line's (u, v) coordinates. This process can be implemented efficiently using graphics hardware as follows: For each input streamline, we treat the v component of the trace tuple assigned to each vertex as the y coordinate and the w component of the trace tuple as the x coordinate and then render the texture mapped line onto an orthographic 2D window. Since all the vertices on an input streamline have the same v component, the streamline will be drawn as a horizontal scanline onto the frame buffer. We can render all the input streamlines with the same u component onto the 2D window and then read back the frame buffer to the uth slice in the appearance texture using the OpenGL function *glCopyTexSubImage3D*. We repeat this process for all the u values until all the input streamlines are processed. Once this is done, the appearance texture has been successfully configured. We can then animate the procedural solid texture along the flow line directions by simply shifting the appearance texture coordinate w at each time step, as described in Section 3.6. Fig. 5 shows examples of (a) using a 3D turbulence function texture to render the trace volume and (b) advecting dye in the global texture.

3.4 Trace Tuple Precision and Anti-Aliasing

When we slice the streamlines during voxelization, the graphics hardware will interpolate the colors and, thus, the trace tuples for the intermediate voxels between the streamline vertices. Since all vertices along the same streamline share the same streamline identifier (u, v), the interpolation will result in the same values for all intermediate voxels. The graphics hardware will interpolate the streamline parameterization (w) linearly, which allows the appearance texture to map evenly across the streamline. It is worth mentioning that the precision limitation in the graphics hardware can pose a problem when using a color channel to parameterize the streamline, i.e., representing the w coordinate. Until recently, colors and alpha values were represented by fixed-point numbers in most graphics hardware. This can be problematic when representing the w coordinate of the trace tuple using a color channel since the quality of the texture lookup result can suffer from quantization artifacts. Although one can utilize the floating-point texture support by modern graphics hardware to alleviate this problem, the inflated texture memory requirement and the performance penalty make this option undesirable. To handle the limited precision problem when using a color channel to represent w, we can divide the streamline into multiple shorter segments and then map the full range of the texture coordinate, i.e., $[0, 1]$ onto each segment. In addition, we can have the appearance texture wrap around in the dimension that corresponds to the flow direction. We have found that this solution produces satisfactory rendering results.

When the resolution of the trace volume is limited, the above voxelization algorithm may produce jaggy results. In 2D, antialiasing lines can be achieved by drawing thick lines [26]. The opacities of the pixels occupied by the thick lines correspond to the coverage of their pixel squares. Since line antialiasing is widely supported by graphics hardware, one might attempt to use it when slicing through the streamlines during our hardware-accelerated voxelization process. However, we have found that this doesn't generate the desired effect since no antialiasing is performed across the slices of the trace volume. Hence, to achieve streamline antialiasing in the voxelization process, we need to model the thick lines and properly assign the opacities.

We model the 3D thick line using the method described previously for rendering local glyphs, i.e., by extruding a mask along the flow line to create a bundle of satellite lines. The mask is always perpendicular to the central flow line and the orientation of the mask is calculated based on the local curl of the flow, similar to the streamribbon construction algorithm used in [21] and [24]. All the lines in the bundle receive the same streamline parameterization as the central streamline and the streamline identifiers of the lines are assigned in a way that maps them to texels of the appearance texture in a close vicinity. We assign an opacity value to each vertex on the line bundle so that antialiasing can be performed in the rendering stage (Section 3.6). It is stored in the alpha channel of the vertex attribute. The opacity value is assigned in a way that the vertices near the surface and the endpoints of the thick line receive lower values to simulate the weighted area sampling algorithm [27].

3.5 Incremental Trace Volume Updates

Although the trace volume only needs to be created once and can be used for rendering different appearance textures, sometimes the user may wish to update the trace volume by inserting new flow lines or deleting unwanted ones. This is particularly useful when the user is navigating through a large data set and wants to incrementally explore different regions of interest. Although trace volume updates can be done in a straightforward manner, such as creating a different set of input flow lines and performing the voxelization algorithm again to generate a new trace volume, an incremental algorithm that allows the user to dynamically modify the trace volume and receive immediate visual feedback is more desirable. This way, the users can either start with an empty trace volume and incrementally "populate" it till the visualization is satisfactory or have the Chameleon algorithm generate an initial trace volume and then refine it.

To achieve the above goal, we devise an incremental trace volume update algorithm which works as follows: To

augment the trace volume, the user specifies new seeds, which will be used to advect additional flow lines. To render these flowlines into the trace volume, since there is already a trace volume residing in the texture memory, we need to make sure that the trace volume is only updated but not overwritten. To do so, when processing each slice of the trace volume during voxelization, a quadrilateral is first rendered which covers the entire viewport and texture-mapped using the corresponding slice from the existing trace volume. Then, we render the new flow lines using the texture mapped polygon as the background. After the rendering is completed, OpenGL function *glCopyTexSubImage3D* is called to copy the content of the frame buffer back to the trace volume slice.

Removing flow lines from the trace volume is very similar to augmenting the trace volume. First, the user picks the flow lines that are to be removed. Then, we perform the slice sweeping algorithm to process the relevant slices intersected by the selected flow lines. For each slice, we first render a polygon using the existing trace volume slice as the texture to fill the viewport. We then render the user selected flow lines. Since now those flow lines need to be removed from the trace volume, we turn on OpenGL blending function and utilize the *GL_FUNC_REVERSE_SUBTRACT_EXT* blending mode (defined in OpenGL extension *EXT_blend_subtract*) to cancel the trace tuples previously written by the flow lines. Fig. 6 shows an example of updating the trace volume.

3.6 Real-Time Rendering and Animation

Today, volumetric data sets can be rendered at interactive speeds using texture mapping hardware. When using hardware-based volume rendering methods, the volume data is stored as a solid texture in the graphics hardware. A stack of polygons, serving as proxy geometries, are used to sample the volume and blended together in a back-to-front order to create the final image. If the graphics hardware only supports 2D textures, the volume data set can be represented as three stacks of 2D textures and the slice polygons are axis-aligned. If 3D texture-mapping is supported, the data set is represented as a single 3D texture and view-aligned slicing polygons can be rendered.

Recently, we have seen a drastic change in the design of PC graphics processing units (GPUs). They have evolved from being a fixed-function state-based pipeline to being highly programmable and are capable of producing sophisticated rendering effects at interactive speeds. In the latest GPUs such as nVidia GeforceFX or ATI Radeon 9800, the programmable vertex and fragment stages of the graphics pipeline, usually referred to as vertex shader and fragment shader, are exposed to the user as streaming processors with general purpose registers and SIMD instructions. Each of the programmable stages can execute user-defined programs on a per-vertex or per-fragment basis. The vertex or fragment program can be specified using either assembly-like opcodes (such as ARB_fragment_program, [28]) or via high-level shading languages (such as Cg [29] or DirectX HLSL [30]) which can be translated into assembly code.

Our algorithm utilizes programmable graphics hardware to facilitate runtime appearance control at interactive speeds. Specifically, the trace volume is rendered using a two-step texture lookup performed in real time by employing the dependent texture read instruction in the fragment shader. The first texture

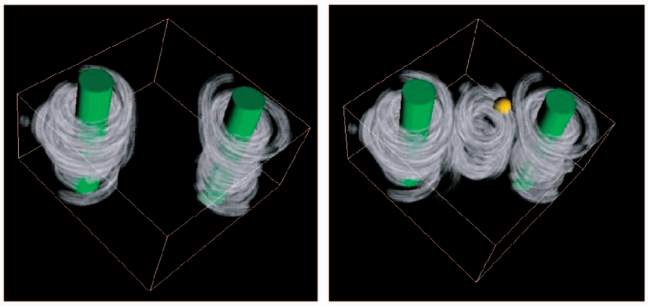


Fig. 6. The Chameleon algorithm allows the user to interactively update the trace volume. (a) Initial trace volume. (b) A 3D cursor (the sphere in the center) is used to augment the trace volume.

lookup involves the usual slicing through the trace volume. When a slice polygon is rendered, each of its vertices is accompanied by texture coordinates, which in turn are the locations of the intersection between the slice polygon and the bounding box of the trace volume. These texture coordinates are interpolated using the nearest neighbor scheme when the slice polygon is rasterized and then fed to the fragment shader. The inputs to the fragment shader also include the trace volume and the appearance texture, which are both represented as RGBA 3D texture objects *trace_volume_tex* and *appearance_tex*, respectively. Suppose the interpolated texture coordinates for a given fragment is $iTexCoord = (s, t, r)$, the dependent texture read can be performed by first fetching *trace_volume_tex* with $iTexCoord$, which returns the trace tuple $trace\_tuple = (u, v, w)$, then using $trace\_tuple$ as the texture coordinate to index *appearance_tex* to get the final color (RGB portion) of the fragment. The antialiasing is done by modulating the α value from *trace_volume_tex* with the opacity value from *appearance_tex*. These operations can be implemented in Cg as the following code fragments, where *tex3D* denotes the texture sampling instruction. For more information about Cg, please refer to [29].

```
fixed4 trace_tuple
    = tex3D(trace_volume_tex, iTexCoord);
fixed4 oColor
    = tex3D(appearance_tex, trace_tuple.rgb);
oColor.a = oColor.a*trace_tuple.a;
return oColor;
```

Using the dependent texture reads, the Chameleon algorithm can easily generate animations to assist the viewer to comprehend flow directions. This can be done by shifting the appearance texture along the straight flow direction in the texture space, which effectively equals translating the w component of the trace tuple before using it to index the appearance texture in the fragment program. When using Cg, this can be done as the following code segment, where *delta* is the translation amount along the streamline direction and is updated at each animation frame.

```
fixed4 trace_tuple
    = tex3D(trace_volume_tex, iTexCoord);
trace_tuple.b += delta;
fixed4 oColor
    = tex3D(appearance_tex, trace_tuple.rgb);
```

4 Time-Varying Chameleon Algorithm

Following the principal philosophy of decoupling the advection and rendering stages, we now present an extension of the Chameleon algorithm to unsteady flow. The algorithmic pipeline for time-varying fields remains similar to the one shown in Fig. 1 and, as such, Chameleon retains the feature of interactive appearance control as in the steady state case. In our time-varying algorithm, pathlines which describe the trajectories of massless particles moving in an unsteady flow [14] are voxelized to form a trace volume in the offline stage of the pipeline. Voxels in the trace volume store a three-component trace tuple: the pathline identifier (u, v) and the time-stamp (w). During the rendering phase, the trace volume is displayed using dependent texturing with a user-selected appearance texture. This texture is designed such that, when it is mapped to the trace volume, only those voxels in the trace volume whose timestamp equals the time of the current frame are visible. The texture in the volume rendered trace volume is animated along the pathlines by shifting the w component of the trace tuple, as described in Section 3.6.

For our algorithm, the main difference between the processing of steady state and unsteady flows comes from the fact that pathlines can intersect with themselves or each other, while streamlines do not. As a result, for time-varying data sets, there can be voxels in the trace volume that intersect pathlines more than once and thus need to store more than one trace tuple. We will explain this situation with the help of an example, shown in Fig. 7a. The pathlines starting from both A and B pass through the voxel shown. The pathline with identifier (u_1, v_1) passes through the voxel at time $t = 3$, while the pathline (u_2, v_2) intersects the voxel at time $t = 10$. Thus, the voxel needs to store both $(u_1, v_1, 3)$ and $(u_2, v_2, 10)$. For a correct rendering, the voxel should contain the first tuple at the time step $t = 3$ and then switch to the second one when the time step equals 10. To achieve this, our time-varying algorithm will perform interactive trace tuple updates during rendering. Initially, this voxel contains the trace tuple with the smallest time-stamp, i.e., $(u_1, v_1, 3)$. At $t = 3$, the voxel is rendered with these values. This trace tuple is not needed after $t = 3$ and the voxel should contain the tuple $(u_2, v_2, 10)$ while rendering the frame for $t = 10$. So, the voxel is updated with the second trace tuple information after rendering $t = 3$, but before $t = 10$.

The visualization pipeline of the Chameleon framework is modified for time-varying data to handle multiple trace tuples in the following manner: In a preprocessing stage, pathlines are advected and the voxel contention information (i.e., which voxel to replace at what time step) is collected. A bookkeeping operation is performed to organize the voxels that will intersect the pathlines multiple times. Since those voxels require runtime updates during rendering, if neighboring voxels need to be updated at the same time step, then it is more efficient to update all of them in one go instead of using multiple texture writes for each individual voxel. To do this, the bookkeeping operation stores each group of such neighboring voxels into a single *update volume*. Bookkeeping is followed by a voxelization stage, which scan-converts pathlines and creates the trace volume. The trace tuple values that will go into each of the update volumes will be stored separately

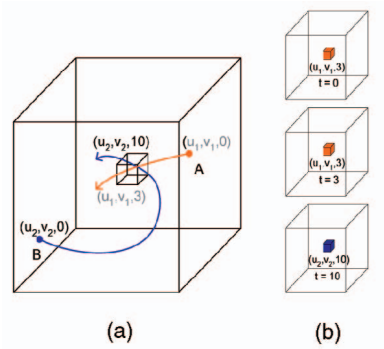


Fig. 7. Time-Varying Chameleon: example of voxel update. (a) Two pathlines pass through the voxel shown, at $t = 3$ and $t = 10$. (b) The trace-tuples corresponding to the pathlines should be written to the voxel *before* the voxel is rendered using those values. $(u_1, v_1, 3)$ is written when creating the initial trace volume and $(u_2, v_2, 10)$ is written after rendering the frame for $t = 3$, but before rendering $t = 10$.

and there can be multiple update volumes for each time step. After the preprocessing stages complete, the data is visualized by rendering the trace volume and dynamically updating the trace volume texture with update volumes as necessary. In the remainder of this section, we discuss in greater detail the stages of bookkeeping and voxelization.

4.1 Pathline Advection and Bookkeeping

The algorithm starts with a set of seed points, which are placed randomly throughout the volume. Any other placement strategy can be used as it is independent of the proposed visualization technique. The advection of each seed point returns a set of line segments representing the pathline originating from the seed. The entire vector field volume is divided into bins using a low resolution grid and the pathline segments are placed in respective bins, sorted by time. If the two vertices of a segment lie across bin boundaries, the segment is placed in the bin corresponding to the vertex with the larger timestamp. The grid helps reduce the number of tests for intersection of pathline segments. Additionally, due to efficiency reasons, a bin will be used as the smallest unit for volume updates. Advection is followed by a bookkeeping stage, which has two main functions. First, it creates a history of updates required for each bin. Second, it merges neighboring (in both space and time) bins that need to be updated so as to minimize the number of texture updates required.

During bookkeeping, pathline segments are tested for intersections with segments from all previous timesteps and a history of intersections is recorded for each bin. During rendering, the trace volume voxels inside a bin will be updated once for every intersection within that bin. To reduce the number of OpenGL calls to update the trace volume, the bins which need to be updated are collected together—first in the space neighborhood and then in time. For the grouping in the spatial neighborhood, we superimpose an octree structure on the trace-volume. The root node of the octree represents the whole volume. A child node represents one of eight subvolumes obtained by dividing the parent node's volume. The leaf nodes correspond to the bins. For a given time step t, if enough (more than a threshold) of the bins within a nonleaf node (subvolume) need to be overwritten, then the bookkeeping stage decides that the whole subvolume should be updated

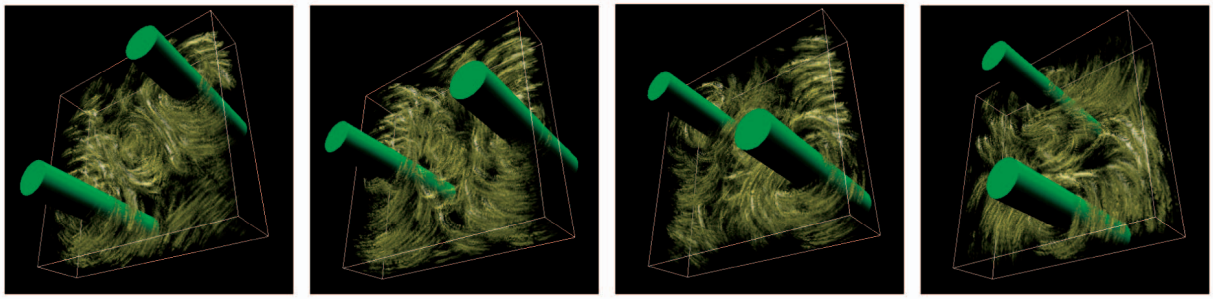


Fig. 8. Four snapshots from an animation of the vortices data set using the time-varying Chameleon algorithm. The images were generated using a line bundle texture with lighting.

using a single call. This test is done in a top-down manner, so the highest octree nodes (largest subvolumes) satisfying the condition are selected for updating. Within a subvolume that has been selected for update, there can be bins which do not need an update at time t *and* whose voxels contain timestamps (w) less than t. These bins can be written with the values of their next update at a future time step. At the end of the bookkeeping stage, the updating information for every time step is known (which subvolumes need to be updated with which pathline segments). If a vertex of a pathline segment lies outside of a subcube to be updated (as can happen when vertices lie across bin boundaries), the cube is expanded to enclose the vertex.

4.2 Voxelization and Rendering

The bookkeeping stage generates information about the subvolumes that need to be updated for each time step, which is fed to the voxelization stage. The voxelizer scan converts the pathlines associated with each of these subvolumes to create the update volumes, in order of time steps. Within the same time step, the order does not matter. An update volume is generated by a voxelization procedure similar to the steady flow voxelization. An important consideration for time-varying voxelization is that only those voxels which have new trace tuples should be updated. For example, consider a subvolume with multiple bins which is being voxelized to generate an update volume. The subvolume might contain a bin that does not need to be updated because the voxels in the trace volume corresponding to this bin are already up-to-date. These trace volume voxels should remain intact after the subvolume is used to update the trace volume; otherwise, incorrect results will be generated. To achieve this, during each scan conversion, the trace volume contents of all the previous voxelizations are used as a background, similar to our trace volume update algorithm described in Section 3.5. Thus, only voxels with new trace tuples are updated and voxels with no new information retain their previous values. After each subvolume is voxelized, it is read back from the graphics hardware and stored as an update volume to be used later during rendering.

During the rendering phase, the initial trace volume is used to visualize the first time step of the unsteady flow. For each of the subsequent time steps, the bookkeeping data structure is used to recall corresponding update volumes. The size and number of update volumes for each time step can vary and there can be time steps without any updates. Using *glTexSubImage3D*, we overwrite regions of the current trace volume with corresponding update volumes in the same order that was used for voxelization. After the updates for each time step, the trace volume is rendered to display the current time step. Fig. 8 shows four snapshots from a vortices animation using the time-varying Chameleon algorithm. A real-time animation of 50 time steps of the same data set accompanying this paper can be found in the IEEE digital library. Before rendering each time step, the trace volume texture is modified using update volumes, which are stored in main memory. The size of the texture updates depends on the number of pathlines used in the animation. For the accompanying animation, a total of 178,000 pathlines were used over time. On average, 70 percent of the trace volume texture was updated for each frame.

In order to shade only those voxels belonging to the current time step during the flow animation, the appearance texture must be configured with care. The user supplies a temporal window Δt which specifies that, at any time step t, the pathline segments corresponding to the time range $(t - \frac{\Delta t}{2}, t + \frac{\Delta t}{2})$ should be visible. This can be done by designating a fixed length (derived from Δt) of the appearance texture along the w direction to be visible and shifting the texture coordinates accordingly to make sure that the current time step falls into the nontransparent window.

5 Enhanced Depth Cues

Previously, researchers have proposed various techniques to enhance the perception of spatial features in volumetric data. Examples include the volume illustration techniques proposed by Ebert and Rheingans [31], a point-based volume stippling technique by Lu et al. [32], a 2D incompressible flow visualization technique using concepts from oil painting by Kirby et al. [33], and an enhanced LIC technique using halos by Interrante and Grosch [16]. In this section, we discuss the generation and use of various enhanced depth cues in our algorithm. Several related implementation details that are not described in the previous sections will also be described.

Additional depth cues can be used to enhance the perception of the spatial relationship between flow traces. Fig. 9 illustrates the effect of having enhanced depth cues, where the image on the right was produced with lighting on while the one on the left was not. In our rendering framework, we can incorporate various depth cues such as lighting, silhouette, and tone shading.

To achieve these effects, we need to supplement the trace volume with a normal vector for each voxel. Although normal vectors are typically associated with surfaces and not

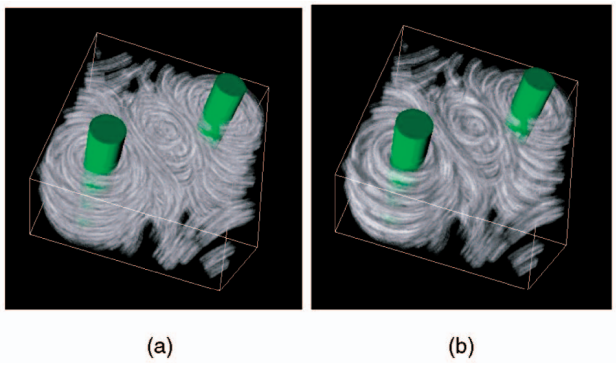


Fig. 9. Rendering of the vortices data set: (a) without lighting, (b) with lighting.

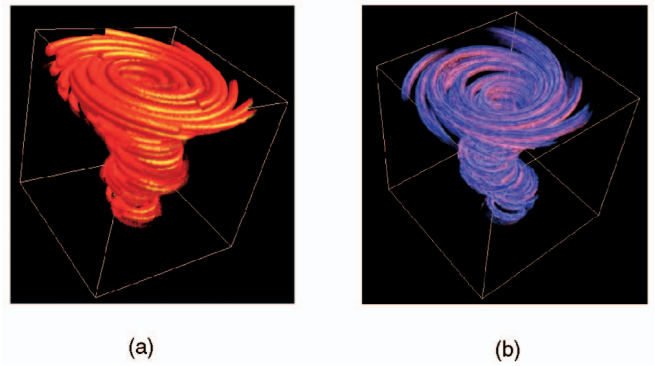


Fig. 10. Visualization of a tornado data set using different depth cues: (a) lighting, (b) tone shading.

uniquely defined for line primitives, when using 3D thick lines for antialiasing as described in Section 3.4, the normal vector $\mathbf{n_i^j} = (n_x, n_y, n_z)$ for the jth vertex m_i^j on strip i can be defined as $\mathbf{m_i^j} - \mathbf{v_j}$, where v_j is the center of the extruding mask. Alternatively, when the light vector $\mathbf{L}$ is fixed, the normal vector can be defined as the one lying on the $\mathbf{L} - \mathbf{T}$ plane, where $\mathbf{T}$ is the tangential vector. This is the technique used by the illuminated streamline algorithm [34].

Like trace tuples, normal vectors are assigned to vertices along the thick lines as colors and scan-converted during the voxelization process. Since a normal vector is a 3-tuple and the number of color channels is not sufficient to represent both the trace tuple and the normal vector simultaneously, we employ a second voxelization pass to process the streamlines with normal vectors as the colors. Because each component of a normalized normal vector $\mathbf{n_i^j}$ is in the range of $[-1, 1]$, they are shifted and scaled into the $[0, 1]$ range in order to be represented as OpenGL vertex colors.

Similar to the trace volume, the normal volume is also represented as a 3D RGBA texture object in the graphics hardware and used as one of the inputs to the fragment shader. The same trace tuple used to look up the appearance texture is also used as the texture coordinates to sample the normal volume. After being remapped to the range of $[-1, 1]$, it can be used by subsequent fragment program statements to perform various depth cue operations. In the following, we provide some details about creating the depth cues, such as lighting, silhouette, and tone shading, as well interactive volume culling.

5.1 Lighting

We use the Phong illumination model [27] to calculate lighting on each voxel. The lighting equation for each voxel in the trace volume is defined as:

$$\mathcal{C} = C_{decal} \times k_{diff} \times (N \cdot L) + C_{spec} \times (N \cdot H)^{k_s}),$$

where N, L, H are the normal vector, light vector, and halfway vector, respectively. C_{decal} and C_{spec} are the colors fetched from the appearance texture and the color of the specular light. k_{diff} is a constant to control the intensity of the diffuse light. The intensity of the specular light is controlled by the magnitude of C_{spec} and k_s is the shininess of the specular reflection. For simplicity and performance reasons, we assume parallel lights and nonlocal viewer. Hence, all these parameters except N remain constants for all fragments and are placed in the registers of the fragment shader. Note that, since the per-voxel normal N is defined in the object space, L and H need to be transformed accordingly. The transformation of L and H can be done by either the application or by using the vertex shader. Fig. 10a shows a rendering of the tornado data set with lighting.

5.2 Tone Shading

Unlike lighting, which only modulates the pixel intensity, tone shading varies the colors of the pixels to depict the spatial structure of the scene. Objects facing toward the light source are colored with warmer tones, while the opposite are in cooler tones. We achieve the tone shading effect with the following formula:

$$\mathcal{C} = C_w \times C_{decal} \times (N \cdot L) + C_c \times (1 - (N \cdot L)),$$

where C_w is the warmer color, such as red or yellow, and C_c is the cooler color, such as blue or purple. Fig. 10b shows the rendering supplemented by tone shading.

5.3 Silhouette

The spatial relationship between flow lines in the trace volume can be enhanced by using silhouettes to emphasize the depth discontinuity between distinct streamlines. They are depicted by assigning the silhouette color to those voxels which satisfy $E \cdot N \leq \rho$, where E is the eye vector, N is the normal vector, and ρ is the parameter to control the thickness of the silhouette. An example of silhouette-enhanced rendering is shown in Fig. 11a.

5.4 Interactive Volume Culling

Clipping planes and opacity functions can be used to remove uninteresting regions from the trace volume. In our algorithm, since the trace volume is rendered using textured slicing polygons, we can easily utilize OpenGL's clipping planes to remove polygon slices outside the region of interest. We can also employ a transfer function based on some scalar quantities (such as pressure or velocity magnitude) associated with the vector field to modulate the opacity of the trace volume. An example of interactive clipping is shown in Fig. 11b.

6 Performance

We implemented our Chameleon algorithm on a standard PC using OpenGL (for rendering) and MFC (for creating

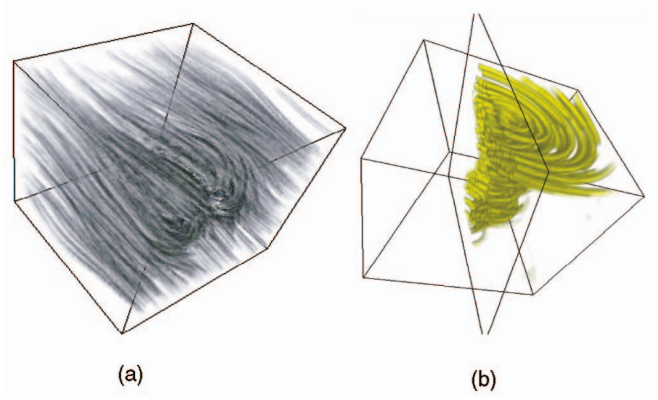


Fig. 11. (a) Silhouette rendering of the Argon Bubble data set, (b) interactive culling using transfer function and OpenGL clipping planes.

user interface) libraries. The fragment programs were written in the Cg shading language and compiled with the nVidia Cg runtime compiler (version 1.1). The machine is equipped with a single Pentium4 2.66GHz PC with 1024MB RAM and nVidia GeforceFX 5900Ultra GPU (256MB video RAM). The table in Fig. 13 shows the performance of constructing and rendering 256^3 static trace volumes and normal volumes for three data sets—tornado, moving vortices, and argon bubble data sets. The table in Fig. 12 provides the performance measurements for the time-varying Chameleon algorithm using the moving vortices data set for 50 time steps. For the time-varying data, the resolutions of the trace volume and the normal volume were both set to $128 \times 128 \times 32$. In both tables, we list the timings for advection and rendering of the flow lines, as well as transferring the voxelization results from the frame buffer to the 3D texture memory for all volume slices. The construction time increased as we increased the number of streamlines. However, rendering and frame buffer transfer are all done using graphics hardware. Therefore, we were able to construct the trace volumes very efficiently.

Once the construction of the trace volume is completed, the rendering speed is independent from the size of input streamline geometry. Since Chameleon performs hardware texture-based volume rendering, the rendering speed is only dependent on the resolution of the trace/normal volume, the number of fragments in the viewport, and the complexity of the fragment programs. Using the modern graphics hardware, we are able to perform interactive rendering of the trace volumes with various shading effects at a speed from eight to 18 frames per second. This allows the user to explore the vector field interactively.

Dataset	**Vortices** (50 steps)		
# of lines	89000	178000	267000
Advection (sec.)	4.42	9.27	11.29
Voxelization (sec.)	41.01	72.78	85.57
Plain (frame/sec.)	29.10	14.75	11.34
Lighting (frame/sec.)	18.33	10.32	9.75
Silhouette (frame/sec.)	15.76	10.01	7.82
Tone Shading (frame/sec.)	19.40	11.60	9.49

Fig. 12. Performance measurements for the time-varying Chameleon algorithm.

7 Conclusion and Future Work

We have presented an interactive texture-based technique for visualizing three-dimensional vector fields. By decoupling the computation of streamlines and the mapping of visual attributes to two disjoint stages in the visualization pipeline, we allow the user to use various appearance textures to visualize the vector field with enhanced visual cues. We plan to extend our work to achieve level of detail by using multiresolution trace volumes. Flow topology analysis can be incorporated to assist better seed placement strategy, as well as use of nonuniform resolution trace volumes. Various existing and upcoming volume rendering techniques, originally devised for visualizing scalar volumes, can also be incorporated into the Chameleon framework.

Acknowledgments

This work is supported in part by the US National Science Foundation grant ACR 0222903, NASA grant NCC-1261, US Department of Energy Early Career Principal Investigator Award, Ameritech Faculty Fellowship, and Ohio State Seed Grant. The authors thank Roger Crawfis, Milan Ikits, Miriah Meyer, and J. Dean Brederson for their generous help. The Argon Bubble data set is provided courtesy of John Bell and Vince Beckner, Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory. The authors also thank anonymous reviewers for their valuable comments.

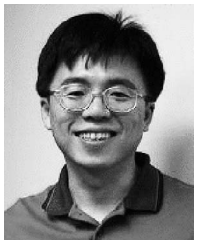
References

[1] J. Hultquist, "Constructing Stream Surfaces in Steady 3D Vector Fields," *Proc. Visualization '92*, pp. 171-178, 1992.
[2] N. Max, B. Becker, and R. Crawfis, "Flow Volumes for Interactive Vector Field Visualization," *Proc. Visualization '93*, pp. 19-24, 1993.
[3] J. van Wijk, "Spot Noise: Texture Synthesis for Data Visualization," *Computer Graphics*, vol. 25, no. 4, pp. 309-318, 1991.

Dataset	**Vortices** (100x100x50)			**Bubble** (256x256x256)			**Tornado** (96x96x96)		
# of lines	34250	68500	102750	34250	68500	102750	34250	68500	102750
Advection (sec.)	2.11	3.66	5.18	1.72	3.82	4.82	3.46	4.67	5.49
Voxelization (sec.)	4.71	6.10	7.11	7.40	16.34	21.67	5.31	6.72	7.71
Plain (frame/sec.)	18.23	17.40	16.11	10.40	9.11	8.90	11.30	10.70	10.37
Lighting (frame/sec.)	11.30	10.11	10.03	6.60	6.23	6.02	7.84	7.41	7.34
Silhouette (frame/sec.)	14.11	11.30	11.20	7.91	7.30	7.14	8.21	8.10	7.93
Tone Shading (frame/sec.)	11.70	12.76	12.30	8.10	7.89	7.40	9.11	8.91	8.22

Fig. 13. Performance measurements for the static Chameleon algorithm. The resolution for the trace volume and normal volume are both 256^3.

[4] B. Cabral and C. Leedom, "Imaging Vector Fields Using Line Integral Convolution," *Proc. SIGGRAPH '93,* pp. 263-270, 1993.
[5] J. van Wijk, "Image Based Flow Visualization," *ACM Trans. Graphics (Proc. ACM SIGGRAPH 2002),* vol. 21, no. 3, pp. 745-754, 2002.
[6] G.-S. Li, U. Bordoloi, and H.-W. Shen, "Chameleon: An Interactive Texture-Based Rendering Framework for Visualizing Three-Dimensional Vector Fields," *Proc. Visualization '03,* pp. 241-248, 2003.
[7] D. Ebert, F. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling, A Procedural Approach,* third ed. Morgan Kaufmann, 2002.
[8] R. Crawfis and N. Max, "Direct Volume Visualization of Three-Dimensional Vector Fields," *Proc. 1992 Workshop Volume Visualization,* pp. 55-60, 1992.
[9] T. Porter and T. Duff, "Compositing Digital Images," *Proc. ACM SIGGRAPH '84,* pp. 253-259, 1984.
[10] R. Crawfis and N. Max, "Texture Splats for 3D Vector and Scalar Field Visualization," *Proc. Visualization '93,* pp. 261-266, 1993.
[11] D. Stalling and H.-C. Hege, "Fast and Resolution Independent Line Integral Convolution," *Proc. SIGGRAPH '95,* pp. 249-256, 1995.
[12] H.-W. Shen, C. Johnson, and K.-L. Ma, "Visualizing Vector Fields Using Line Integral Convolution and Dye Advection," *Proc. 1996 Symp. Volume Visualization,* pp. 63-70, 1996.
[13] M.-H. Kiu and D.C. Banks, "Multi-Frequency Noise for LIC," *Proc. Conf. Visualization '96,* pp. 121-126, 1996.
[14] H.-W. Shen and D. Kao, "A New Line Integral Convolution Algorithm for Visualizing Time-Varying Flow Fields," *IEEE Trans. Visualization and Computer Graphics,* vol. 4, no. 2, Apr.-June 1998.
[15] U.D. Bordoloi and H.-W. Shen, "Hardware Accelerated Interactive Vector Field Visualization: A Level of Detail Approach," *Computer Graphics Forum,* vol. 21, no. 3, pp. 605-614, 2002.
[16] V. Interrante and C. Grosch, "Strategies for Effectively Visualizing 3D Flow with Volume LIC," *Proc. Visualization '97,* pp. 421-424, 1997.
[17] C. Rezk-Salama, P. Hastreiter, C. Teitzel, and T. Ertl, "Interactive Exploration of Volume Line Integral Convolution Based on 3D-Texture Mapping," *Proc. IEEE Visualization '99,* pp. 233-240, 1999.
[18] B. Jobard, G. Erlebacher, and Y. Hussaini, "Lagrangian-Eulerian Advection for Unsteady Flow Visualization," *Proc. Visualization '01,* pp. 53-60, 2001.
[19] D. Weiskopf, M. Hoph, and T. Ertl, "Hardware-Accelerated Visualization of Time-Varying 2D and 3D Vector Fields by Texture Advection via Programmable Per-Pixel Operations," *Proc. Vision, Modeling, and Visualization '01,* pp. 439-446, 2001.
[20] B. Jobard and W. Lefer, "Creating Evenly-Spaced Streamlines of Arbitrary Density," *Proc. Eighth Eurographics Workshop Visualization in Scientific Computing,* pp. 57-66, 1997.
[21] D. Darmofal and R. Haimes, "Visualization of 3-D Vector Fields: Variations on a Stream," *AIAA 30th Aerospace Science Meeting and Exhibit,* 1992.
[22] S. Fang and D. Liao, "Fast CSG Voxelization by Frame Buffer Pixel Mapping," *Proc. 2000 IEEE Symp. Volume Visualization,* pp. 43-48, 2000.
[23] R. Crawfis, N. Max, and B. Becker, "Vector Field Visualization," *IEEE Computer Graphics and Applications,* pp. 50-56, 1994.
[24] S. Ueng, K. Sikorski, and K. Ma, "Fast Algorithms for Visualize Fluid Motion in Steady Flow on Unstructured Grids," *Proc. Visualization '95,* pp. 313-320, 1995.
[25] N. Max and B. Becker, "Flow Visualization Using Moving Textures," *Proc. ICASE/LaRC Symp. Visualizing Time-Varying Data,* pp. 77-87, 1995.
[26] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification (Version 1.3).* Reference Board, 2001.
[27] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics: Principles and Practice,* second ed. Addison-Wesley Longman Publishing, 1990.
[28] "OpenGL Extension Registry," http://oss.sgi. com/projects/ogl-sample/registry/, 2003.
[29] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-Like Language," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 896-907, 2003.
[30] "Microsoft directx High-Level Shader Language," http://msdn. microsoft.com/library/, 2003.
[31] D. Ebert and P. Rheingans, "Volume Illustration: Non-Photorealistic Tendering of Volume Models," *Proc. Visualization '00,* pp. 195-202, 2000.
[32] A. Lu, C. Morris, D. Ebert, P. Rheingans, and C. Hansen, "Non-Photorealistic Volume Rendering Using Stippling Techniques," *Proc. Visualization '02,* pp. 211-218, 2002.
[33] R. Kirby, H. Marmanis, and D. Laidlaw, "Visualizing Multivalued Data from 2D Incompressible Flows Using Concepts from Painting," *Proc. Visualization '99,* pp. 333-340, 1999.
[34] M. Zöckler, D. Stalling, and H.-C. Hege, "Interactive Visualization of 3D-Vector Fields Using Illuminated Stream Lines," *Proc. Conf. Visualization '96,* pp. 107-114, 1996.



Han-Wei Shen received the BS degree from National Taiwan University in 1988, the MS degree in computer science from the State University of New York at Stony Brook in 1992, and the PhD degree in computer science from the University of Utah in 1998. From 1996 to 1999, he was a research scientist with MRJ Technology Solutions at NASA Ames Research Center. He is currently an assistant professor at The Ohio State University. His primary research interests are scientific visualization and computer graphics. In particular, his current research and publications are focused on topics in flow visualization, time-varying data visualization, isosurface extraction, volume rendering, and parallel rendering.

Guo-Shi Li received the BS degree from National Taiwan University in 1999 and the MS degree in computer science from The Ohio State University in 2003. He is currently a PhD student in computer science at the University of Utah and a member of the Scientific Computing and Imaging Institute. His research interests include computer graphics and interactive scientific visualization techniques. He is a student member of the IEEE Computer Society.

Udeepta D. Bordoloi received the BEng degree (1997) from Delhi University, Delhi, India, and the MS degree (1999) in electrical engineering from Washington University, St. Louis, Missouri. He is currently a PhD student in computer science at The Ohio State University. His research interests include computer graphics and scientific visualization.