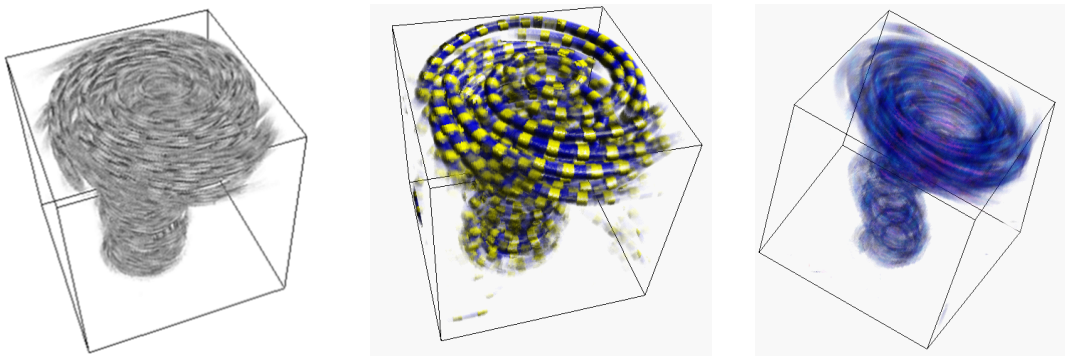
Chameleon: An Interactive Texture-based Rendering Framework for Visualizing Three-dimensional Vector Fields

Guo-Shi Li, Udeepta D. Bordoloi, Han-Wei Shen
Department of Computer and Information Science,
The Ohio State University
{lig,bordoloi,hwshen}@cis.ohio-state.edu



(a) (b) (c)

Figure 1: Tornado dataset rendered with different appearance textures. (a) with LIC texture pre-generated from straight flow. (b) with a color tube texture. Lighting is used to enhance the depth perception. (c) with a 2D paintbrush texture.

Abstract

In this paper we present an interactive texture-based technique for visualizing three-dimensional 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, streamlines are generated from the 3D vector data. Various geometric properties of the streamlines 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 underlying vector field. Users can change the input textures and instantaneously visualize the rendering results. 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.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture.

Keywords: 3D flow visualization, vector field visualization, volume rendering, texture mapping.

1 Introduction

Vector fields play an important role in many scientific, engineering and medical disciplines. Many visualization techniques have been proposed to assist observers 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[Hultquist 1992], flow volume[Max et al. 1993], to name a few) use shape, color, and motion of geometric primitives to convey the physical characteristics in the proximity of a certain point in the vector field. Texture-based methods, such as spot noise[van Wijk 1991], line integral convolution (LIC)[Cabral and Leedom 1993], and IBFV[van Wijk 2002], shade every pixel in the visualization using manipulated textures which express structural information of the vector field.

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 is 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 dataset. 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 severely impedes the interactive use of visual cues. In fact, 3D vector field visualizations by current visual cue-enhanced texture-based techniques are mostly generated in batch mode. 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. Keeping the above desirables in mind, we present a flexible and high-speed approach for three-dimensional vector field visualization.

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. For example, in LIC, streamline advection and output pixel value generation are done simultaneously. As a result, the look of the rendering result cannot be changed on the fly. We address 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 will refer to as the *trace volume*. In the second phase, the trace volume is combined with a desired *appearance texture* at run-time 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 re-process 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 run time. 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 enhanced structural perception using various visual cues can be rendered in real time.

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 has been directed towards integrating texture or icons into a volume rendering of the flow. [Crawfis and Max 1992] developed a 3D raster resampling technique where the volume rendering was built up in sheets oriented parallel to the image plane. These sheets were composited[Porter and Duff 1984] in a back-to-front order. The authors modified the volume integral 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[Crawfis and Max 1993] used for volume rendering. Here, small billboard images are overlapped and composited together to build up the final image. By placing a small icon within the billboard image, and orienting the image such that it lies both perpendicular to the viewing ray, and parallel to the projected vector direction at the splat's center point, a volume rendered image is produced.

Line Integral Convolution, or LIC[Cabral and Leedom 1993], 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 direction of the input vector field. While LIC is effective in visualizing 2D vector fields, it is quite computationally expensive. [Stalling and Hege 1995] proposed an extension to speed up the process. [Shen et al. 1996] proposed the advection of dyes in LIC computation. [Kiu and Banks 1996] used noises of different frequencies to distinguish between regions with different velocity magnitudes. [Shen and Kao 1998] proposed UFLIC for unsteady flow, and a level of detail approach was proposed by [Bordoloi and Shen 2002]. [Interrante and Grosch 1997] introduced the use of halos to improve the perceptual effectiveness when visualizing dense streamlines for 3D vector fields. [Rezk-Salama et al. 2000] 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.

3 The Chameleon Rendering Framework

The primary goal of our research is to develop an algorithm with greater 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 a mix-and-match of different visual appearances on the fly when exploring three-dimensional vector data. In this paper, a novel rendering framework is presented to address this issue. In the following, we first give an overview of our algorithm, and then provide the details of various stages in our algorithm.

3.1 Algorithm Overview

Figure 2 depicts the fundamental difference between our algorithm and the more traditional texture-based algorithm such as LIC. In LIC or similar texture-based algorithms, visual information is conveyed to the user through the correlation between the final voxel values. Texture synthesis is performed in a manner that the luminance of each pixel or voxel is computed and used as the rendering attribute. Once the process is completed, information about the vector field cannot be recovered from the resulting texture. If the user decides to alter the visual appearance, such as changing the frequency or the distribution of the noise, the entire texture synthesis process needs to be performed again.

To allow flexible run-time visual mapping, we devise an algorithm that decouples the processing of the vector field and the mapping of visual attributes. To establish visual coherence for the voxel along the flow direction, we store, in each voxel, a few attributes which are highly correlated along the flow direction. The attributes associated with each voxel will be referred to as the *trace tuple*. Trace tuples from the voxels collectively constitute a volume called the *trace volume*. At run time, the correlation between neighboring trace tuples will be translated to coherent visual properties along the flow direction. 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 pre-computed 2D/3D visual patterns, which will be warped and animated along the streamline directions to create the visualization. The appearance texture can be freely specified by the user at run time. For instance, it can be a pre-computed LIC image, or can be textures with different characteristics such as line

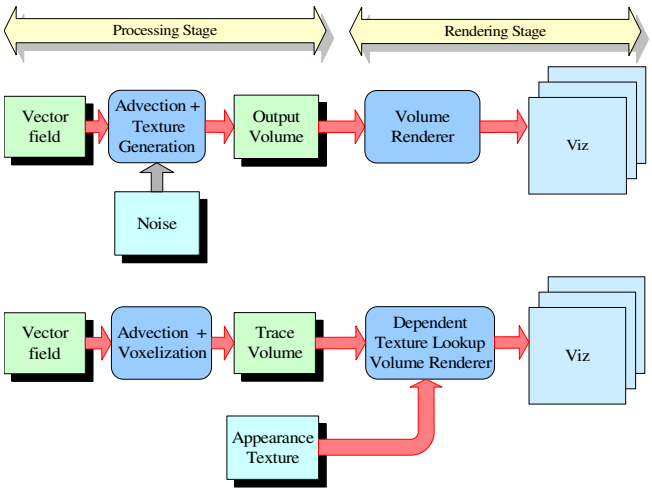


Figure 2: Visualization pipelines for LIC (above), and Chameleon (below). The Chameleon decouples the advection and texture generation stages. Once the trace volume is constructed, any suitable appearance texture can be used to generate varied visualizations of the same vector dataset.

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*.

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 an interpolated trace tuple. This sampled vector is 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 per-fragment shaders and dependent textures to achieve interactivity.

In the following sections, we elaborate each step of our algorithm in detail. We will focus on the topics of trace volume construction, including voxelization (sec.3.2), trace tuple assignment (sec.3.3), anti-aliasing (sec.3.4), and interactive rendering (sec.3.5).

3.2 Trace Volume Creation

The trace volume is created by voxelizing the input streamlines. Since the trace volume will be a texture input to the 3D texture mapping hardware (described later), it is defined on a 3D Cartesian grid. For the underlying vector fields, there is no preferred grid type because the trace volume is created from a dense set of streamlines but not the vector field. We use the method proposed by [Jobard and Lefer 1997] to control the density and the length of streamlines. The seeds are randomly selected, and the streamlines are generated by the fourth-order Runge-Kutta method. An adaptive step size based on curvature [Darmofal and Haimes 1992] is used. The advection process is stopped whenever the advected streamline gets too close to each other. This is to ensure that the thick lines discussed in sec.3.4 do not intersect with each other. Otherwise, the trace tuples will be overwritten during voxelization, which would result in undesirable dependent texturing artifacts in the rendering stage.

To voxelize the streamlines, a hardware-assisted slice sweeping

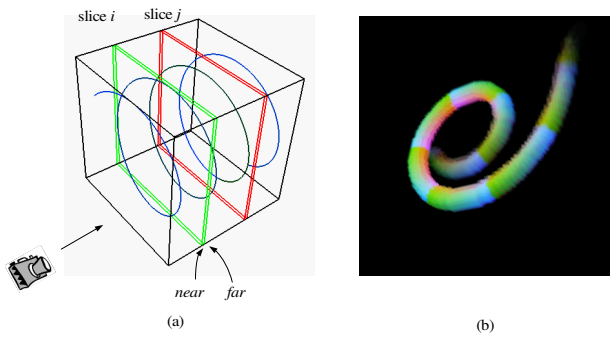


Figure 3: (a) The slice sweeping voxelization algorithm. The near and far clipping planes are translated along the Z axis. At each position of the clipping planes, the streamlines are rendered to generate one slice of the trace volume. (b) A trace volume containing a thick anti-aliased streamline. The streamline parametrization is stored in the blue channel, while the streamline identifiers are stored in the red and green channels (sec.3.3, sec.3.4).

algorithm, inspired by the CSG voxelization algorithm proposed by [Fang and Liao 2000], is designed to achieve faster voxelization speed. The input to our voxelization process is a set of streamlines $\mathscr{S} = \{s_i\}$. Each streamline s_i is represented as a line strip with a sequence of vertices $\mathscr{P} = \{p_j\}$. Each vertex p_j in the streamline s_i is assigned a trace tuple for the identification and parametrization of the streamline. The trace tuple for each streamline vertex is specified as a color for the vertex during our voxelization process. In this section, we focus on the trace volume scan conversion. More details about the trace tuple are provided in the next section.

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 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. Figure 3(a) illustrates our algorithm. Positions for the *near* and *far* clipping planes for two different slices are shown.

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, streamline segments are placed into bins according to their spans along the Z direction. During the voxelization, only the segments which intersect with the current clipping volume are sent to the graphics pipeline. 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.

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,

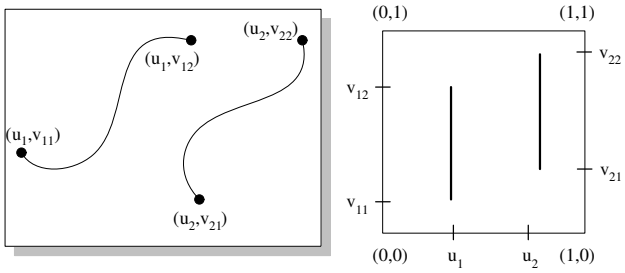


Figure 4: The use of trace tuples as texture coordinates. Left: Trace tuples are assigned to streamlines and stored in the color channels. Right : Trace tuples in the texture space.

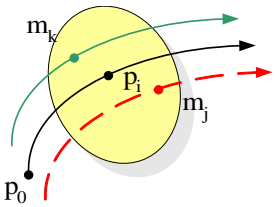


Figure 5: Construction of the thick line. A mask is swept along the central streamline. Points on the mask are used to generate vertices for the satellite lines. The parametrization for the lines in the bundle is the same as the central streamline. The satellite lines are assigned identifier values which map to adjacent texels in the appearance texture.

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.

3.3 Trace Volume Attributes Generation

As mentioned earlier, the set of attributes assigned to the voxels in a trace volume is referred to as the trace tuple. It stores two components: the streamline identifier, which differentiates individual streamlines, and the streamline parametrization, which parameterizes the voxels along the streamline. The dimensions of the trace tuple depend on the dimensions of the appearance texture. When a two dimensional appearance texture is used, the trace tuple is a two-dimensional vector, denoted as (u,v). The first component (u) is used to distinguish between different streamlines, and the second component (v) stores the parametrization of the voxels along the streamline. For example, in figure 4, the two streamlines have distinct u coordinates, which will be mapped to different vertical strips in the appearance texture. Along each streamline, the voxels are parameterized by v, which corresponds to a change in the texture coordinates along the vertical direction. When a three-dimensional appearance texture is used, the trace tuple is a three dimensional vector (u,v,w), where w is used to parameterize the streamline and a two-dimensional vector (u,v) is used to differentiate the streamlines.

We encode the trace tuples into the trace volume during the voxelization process using graphics hardware. Without loss of generality, here we assume that a three-dimensional appearance texture is used. Given an input streamline, we assign the trace tuple (u,v,w) as colors (red, green, blue) to the vertices of streamline segments. 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 identifiers, the interpolation will assign the same value for all intermediate voxels. The graphics hardware will interpolate the streamline parametrization linearly, which allows the appearance texture to map evenly across the streamline.

The precision limitation in the graphics hardware, however, poses a problem when using a color channel to parameterize the streamline, i.e., representing the w coordinate. In the current graphics hardware, colors and alpha values are represented by fix point numbers (8 bits per channel on most architectures). When we use an 8-bit number to represent the texture coordinate, the quality of the texture lookup result can suffer from quantization artifacts.

The goal of parameterizing the streamline and using the result as a texture coordinate to look up the appearance texture is to establish the visual correlation between the voxels along the streamline. However, we observe that it is sufficient to maintain only the local coherence within a nearby vicinity for the voxels along a streamline to depict the flow direction. It is similar to the fixed-length convolution kernel in the LIC. Therefore, to solve the limited precision problem when using a color channel to represent the last component of the trace tuple, we can divide the streamline into multiple 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 result.

The process of assigning streamline identifiers to different streamlines is dependent on the type of appearance texture being used. For LIC or line-bundle textures, for example, streamlines are randomly assigned identifier values in the range [0,1]. For textures containing a well defined solid structure, as in glyphs, it is important that adjacent voxels are assigned texture coordinates (u,v,w) which map to adjacent texels in the texture space. Otherwise the 3D structure present in the appearance texture would break down after texture mapping. As will be explained in the next section, we model streamlines as a set of lines surrounding a central line. This central line gets an identifier (u,v) which maps to the center of the 3D structure in the appearance texture. The outer lines are mapped to a close vicinity in the appearance texture. Figure 3(b) 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.

3.4 Anti-Aliasing

When the resolution of the trace volume is limited, the above voxelization algorithm will produce jaggy results. In 2D, anti-aliasing lines can be achieved by drawing thick lines[Segal and Akeley 2001]. The opacities of the pixels occupied by the thick lines correspond to the coverage of their pixel squares. Since line anti-aliasing 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 anti-aliasing is performed across the slices of the trace volume. Hence, to achieve streamline anti-aliasing in the voxelization process, one needs to model the thick lines and properly assign the opacities.

We model the 3D thick line as a bundle of thin lines surrounding a central line. 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 $\mathscr{B} = \{b_k\}$, where b_k is the kth satellite line in the bundle. The line bundle is created by extruding a mask $\mathscr{M} = \{m_k\}$ along the streamline during the advection process. Each point m_k on the mask corresponds

to a vertex of the satellite strip. Figure 5 shows two such points on the mask. 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. When the advection of the medial streamline completes, we construct the line strip b_k by connecting the vertices from the corresponding points in the mask along the sweep trace.

All the lines in the bundle are assigned the same streamline parametrization as the central streamline. As discussed in the previous section, the streamline identifiers of the lines are assigned in a way that maps them to adjacent texels of the appearance texture. Any solid structure present in the appearance texture is preserved after the trace volume is texture mapped. In addition, we assign an opacity value to each vertex on the line bundle so that anti-aliasing can be performed in the rendering stage(sec. 3.5). 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[Foley et al. 1990].

3.5 Real-Time Rendering Using Dependent Textures

Today volumetric datasets can be rendered at interactive speeds using texture mapping hardware. In the hardware based volume rendering methods, the volume data is stored as a texture in the graphics hardware. A stack of polygons are textured with the corresponding slices from the volume data and blended together in a back-to-front order to form the final image. If the graphics hardware only supports 2D textures, the volume dataset is represented as three stacks of 2D textures and the slice polygons are axis-aligned. If 3D texture-mapping is supported, the dataset can be represented as a single 3D texture and view-aligned slicing polygons can be rendered.

In our algorithm, rendering the trace volume requires a two-step texture lookup. The first texture lookup involves the usual slicing through the trace volume, where every fragment of the slicing polygon receives a color. This color represents the trace tuple, which is then used as the texture coordinates to look up the appearance texture to get the final color and opacity for the fragment. This two-step texture lookup can be performed in real time by employing the dependent texture capability provided by the `NV_TEXTURE_SHADER` extension on nVidia Geforce4 GPUs.

Figure 6 shows the texture shader setting for the fragment processing stage using the nVidia Geforce4 GPUs. The trace volume is represented by a RGBA 3D texture(*Tex0*) on the graphics hardware. With the texture coordinates (s,t,r) coming from the sliced polygon, an RGBA texel is fetched from the trace volume. It contains the trace tuple (u,v,w), as well as the opacity value α for the purpose of anti-aliasing described in section 3.4. The appearance texture is set to be the second texture, i.e., *Tex1*. The dependent texture shader is configured to use the trace tuple as the texture coordinates to sample *Tex1*. The anti-aliasing is done by using the register combiner (`NV_REGISTER_COMBINER`) to modulate α from *Tex0* with the opacity value from *Tex1* (figure 10). The normal volume, shown as *Tex2* in figure 6, is used for various depth cuing effects and will be discussed in the section 4.2. The last texture shader stage is assigned with a 2D texture (*Tex3*) which servers as the opacity modulation function and will be discussed later in section 4.3.

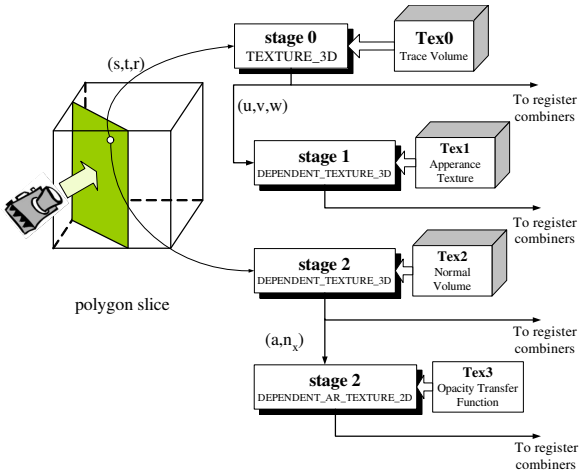


Figure 6: Texture shader configuration. The trace volume, the appearance texture, the and normal volume are represented as 3D color textures and assigned to the 1*st*, 2*nd* and 3*rd* texture units (*GL_TEXTURE0_ARB*, *GL_TEXTURE1_ARB*, and *GL_TEXTURE2_ARB*), respectively. The 1*st* and the 3*rd* texture units receive the texture coordinates interpolated from those specified by *glMultiCoord3f()*, then the 2*nd* and the 4*th* texture units take their results to perform dependent texturing.

4 Appearance Control

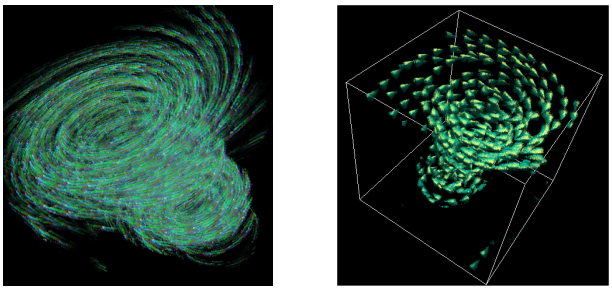
In this section, we show the use of different appearance textures and various visual cues in our algorithm. We also provide some additional implementation details that are not described in the previous sections.

4.1 Appearance Textures

Our chameleon rendering framework allows the user to experiment with different visual mappings at run time when exploring the underlying vector field. To demonstrate the utility of our algorithm, we have created several appearance textures. Each of them presents a different look and feel. Figure 1(a) shows a LIC-like visualization using a 96^3 tornado dataset. The appearance texture was generated using a 2D LIC texture precomputed from a straight flow, which can be computed very efficiently. We also generated a visualization using a texture that simulates streamtubes with illumination and saturated colors, as shown in figure 1(b). When using opaque surface-like textures, a better depth cue can be obtained. Figure 7(a) presents a visualization with an input appearance texture simulating the line bundle technique([Crawfis et al. 1994]). Similar to the LIC texture, the short strokes in the line bundle texture were generated using a straight flow. The tails of the strokes are made more transparent than the heads to emphasize the flow direction. When local glyphs are desired, the user can input a simple voxelized glyphs, such as the arrowhead-shaped solid shown in figure 7(b). All the visualizations were created from the same trace volume, which was created only once, in real time.

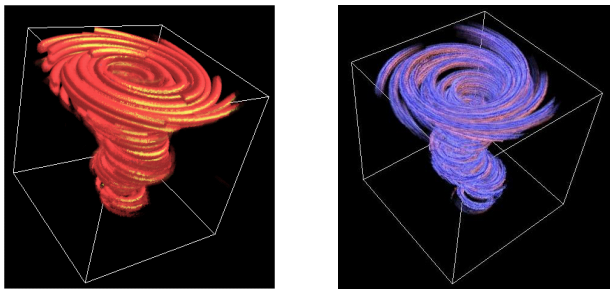
4.2 Depth Cues

Additional depth cues can be used to enhance the perception of the spatial relationship between flow traces. In our rendering framework, we can incorporate various depth cues such as lighting, silhouette, and tone shading. To achieve these effects, we need to



(a) (b)

Figure 7: Different appearance textures. (a) Line bundles. (b) Glyphs.



(a) (b)

Figure 8: Different depth cuing techniques. (a) Lighting. (b): Tone shading.

supplement the trace volume with a normal vector for each voxel. Although normal vectors are typically associated with surfaces and not uniquely defined for line primitives, when using 3D thick lines for anti-aliasing as described in section 3.4, the normal vector $\mathbf{n_i^j} = (n_x, n_y, n_z)$ for 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[Zöckler et al. 1996].

Like trace tuples, normal vectors can be 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 stored into the fixed-point color channels.

The normal volume is specified to the second texture unit (*Tex2*) in the texture shader program (Figure 6). The same trace tuple fetched from *Tex0* to look up *Tex1* is also used as the texture coordinates to sample the normal volume. The fetched normal vector is then fed to the register combiner stages on the nVidia GeForce4 GPU to perform various depth cue operations in a single rendering pass. In the following, we provide more details about creating the depth cues lighting, silhouette, and tone shading. Due to space constraints, we only provide the combiner settings for lighting in Figure 10.

Lighting The lighting equation for each voxel in the trace volume is defined as:

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

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. Figure 10 shows the configurations of the register combiner stages. Since the normal vector is scaled and shifted in the normal volume as discussed above, we use the `EXPAND_NORMAL_NV` input mapping functionality of the register combiner (shown as **E.N.** boxes in Figure 10) to map it back to the original [-1,1] range before the dot-product operation. The input mapping `UNSIGNED_IDENTITY_NV` (shown as **U.I.** boxes in Figure 10) clamps any negative dot product result to zero. Figure 8(a) shows the effect of using lighting.

Silhouette The spatial relationship between streamlines in the trace volume can be enhanced by using silhouettes to emphasize the depth discontinuity between distinct streamlines. We use the following formula to depict the silhouette of thick lines:

$$\mathscr{C} = C_{decal} \times (N \cdot E)^p + C_s \times (1 - (N \cdot E)^p)$$

where E is the eye vector and C_s is the silhouette color. Constant p is to control the thickness of the silhouette. The larger the p, the thicker the silhouette. An example of silhouette-enhanced rendering is given in Figure9(a).

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:

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

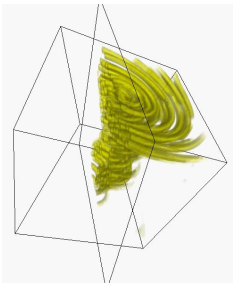
where C_w is the warmer color such as red or yellow and C_c is the cooler color such as blue or purple. Figure 8(b) shows the rendering supplemented by tone shading.

4.3 Interactive Volume Culling

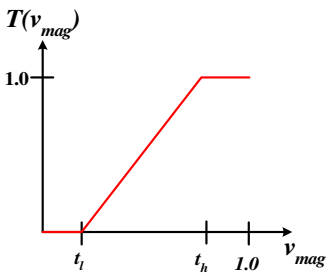
Clipping planes and opacity functions can be used to remove uninteresting regions from the trace volume. In our algorithm, since the trace volume is rendered using textured slicing polygons, we can easily utilize OpenGL's clipping planes to remove polygon slices outside the region of interest (Figure 9(a)).

We can also employ a transfer function T based on the velocity magnitude of the vector field to modulate the opacity of the trace volume. The final opacity value of the voxel becomes $\alpha \times T(v_{mag})$, where α is the opacity value of a voxel described in section 3.4, and v_{mag} is the velocity magnitude at that voxel normalized by the maximum velocity magnitude in the vector field. A simple transfer function, T, that we have used is shown in Figure 9(b).

We implement the transfer function lookup and opacity modulation using texture shader and register combiners. Recall that *Tex2* in Figure 6 is an RGB 3D texture which stores the normal vectors used in various depth cuing techniques. We store the normalized velocity magnitude v_{mag} in the alpha channel of *Tex2* and assign the transfer function T to the third texture unit *Tex3*. Although T is essentially a



(a)



(b)

Figure 9: Interactive Volume Culling. (a) culling with clipping plane and opacity modulation. Rendered with silhouette enhancement. (b) opacity transfer function $T(v_{mag})$

	resolution	
# of lines	128^3	256^3
7350	2.813	5.718
14700	3.891	7.671
22050	4.641	9.093

Table 1: Trace volume construction time (in seconds). The number of lines (first column) includes the satellite lines as well as the central streamlines used for constructing anti-aliased streamlines (sec. 3.4).

Image Resolution	600 x 600	800 x 800
128^3 volume	17.07	14.95
256^3 volume	14.31	12.13

Table 2: Trace volume rendering speed(frames/second).

1D function hence can be realized by 1D dependent texture lookup, we construct *Tex3* as a 2D texture with identical rows because currently only 2D/3D dependent texture lookups are supported by the Geforce4 GPU. The shading operation in *stage3* is then configured as `DEPENDENT_AR_TEXTURE_2D_NV`, which uses the alpha and red components of the texel fetched from *Tex2* as the texture coordinates to lookup the transfer function bound as *Tex3*. In the register combiners, we modulate the opacity described in section 3.4 with the value fetched from *Tex3*(Figure 10). The user can interactively modify the opacity transfer function and render the trace volume in real time.

4.4 Animation

For non-directional textures (like a LIC texture), animation provides a way to visualize the flow direction. Using the chameleon algorithm, one can easily create animations by looping through a series of appearance textures, which can be generated easily by continuously shifting the appearance texture along the flow direction in the local texture space. Alternatively, an additional stage in the texture shader program can be introduced to translate the texture coordinates, represented by the trace tuples, along the streamline direction at run time when rendering the trace volume. The advantage of this approach is that multiple appearance textures need not be loaded when producing animations. When the 2D trace tuple (u,v) is used, this translation can be achieved by multiplying $(u,v,1)$ with the following 2×3 matrix M:

$$M = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & \delta \end{vmatrix}$$

where δ is the translation amount along the streamline direction and is incremented at each animation step. The translated trace tuple $(u,v+\delta)$ is then used as the texture coordinates to sample the appearance texture. We implement this by assigning the trace tuple (u,v) for each vertex on the streamline as color $(u,v,1)$, and perform the matrix multiplication by the `DOT_PRODUCT_NV` and `DOT_PRODUCT_TEXTURE_2D_NV` texture shader operation. To show the effectiveness of our algorithm, we have generated several animations showing the results of our work on the supplementary files accompanying this paper.

5 Performance

We implemented our chameleon algorithm on a standard PC using OpenGL (for rendering) and MFC (for creating user interface) libraries. The machine is equipped with a single Pentium4 2.0GHz PC with 768MB RAM and nVidia Geforce4 Ti4600 GPU (128MB video RAM). Table 1 shows the performance of constructing 128^3 and 256^3 trace volumes for the 96^3 tornado data set. The timings include the advection and rendering of the streamlines, as well as transferring the voxelization results from the frame buffer to the 3D texture memory for all the 128 or 256 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 are able to construct the trace volumes very efficiently. The number of lines in the first column includes the satellite lines as well as the central streamlines used for constructing anti-aliased streamlines (sec. 3.4).

Once the construction of the trace volume is completed, the rendering speed is independent from the streamline geometries. Since Chameleon performs hardware texture-based volume rendering, which is essentially fill-rate limited, the rendering speed is only dependent on the resolution of the trace volume as well as the size of the viewport. Table 2 shows the speeds for rendering 128^3 and 256^3 trace volumes. Using graphics hardware, we are able to perform interactive rendering of the trace volumes at a speed of more than ten frames per second. This allows the user to explore the vector field interactively.

6 Conclusion and Future Work

We have presented an interactive texture-based technique for visualizing three-dimensional vector fields. By decoupling the calculation of streamlines and the mapping of visual attributes into 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 multi-resolution trace volumes and next-generation graphics hardware which provides full programmability in the rasterization stage. With the support of the floating-point datatype on the new hardware, the image quality can be further improved. Many traditional volume rendering techniques can also be incorporated into the Chameleon framework.

7 Acknowledgements

This work is supported in part by NSF grant ACR 0222903, NASA grant NCC-1261, Ameritech Faculty Fellowship, and Ohio State

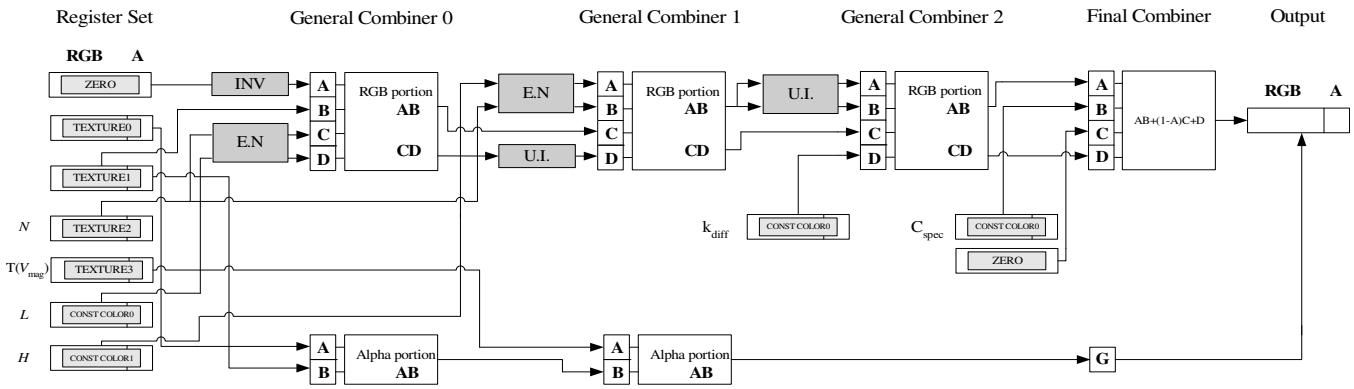


Figure 10: The register combiner configuration for lighting calculation. **U.I.**, **E.N.** stands for *UNSIGNED_IDENTITY_NV* and *EXPAND_NORMAL_NV*, respectively. **INV** represent *UNSIGNED_INVERT_NV*, which maps value x to *abs(1-x)*.

Seed Grant. We thank Dr. Roger Crawfis for his help. We also thank the anonymous reviewers for their insightful comments.

References

BORDOLOI, U. D., AND SHEN, H.-W. 2002. Hardware accelerated interactive vector field visualization: A level of detail approach. In *Proceedings of Eurographics '02*, Springer, 605–614.

CABRAL, B., AND LEEDOM, C. 1993. Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH 93*, ACM SIGGRAPH, 263–270.

CRAWFIS, R., AND MAX, N. 1992. Direct volume visualization of three-dimensional vector fields. In *Proceedings of the 1992 workshop on Volume visualization*, ACM Press, 55–60.

CRAWFIS, R., AND MAX, N. 1993. Texture splats for 3d vector and scalar field visualization. In *Proceedings Visualization '93*, IEEE CS Press, 261–266.

CRAWFIS, R., MAX, N., AND BECKER, B. 1994. Vector field visualization. In *IEEE Computer Graphics and Applications*, IEEE CS Press, 50–56.

DARMOFAL, D., AND HAIMES, R. 1992. Visualization of 3-d vector fields: Variations on a stream. In *AIAA 30th Aerospace Science Meeting and Exhibit*.

FANG, S., AND LIAO, D. 2000. Fast csg voxelization by frame buffer pixel mapping. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, ACM Press, 43–48.

FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. 1990. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc.

HULTQUIST, J. 1992. Constructing stream surfaces in steady 3d vector fields. In *Proceedings of Visualization '92*, IEEE Computer Society Press, 171–178.

INTERRANTE, V., AND GROSCH, C. 1997. Strategies for effectively visualizing 3d flow with volume lic. In *Proceedings of Visualization '97*, IEEE Computer Society Press, 421–424.

JOBARD, B., AND LEFER, W. 1997. Creating evenly-spaced streamlines of arbitrary density. In *Proceedings of the eight Eurographics Workshop on visualization in scientific computing*, 57–66.

KIU, M.-H., AND BANKS, D. C. 1996. Multi-frequency noise for lic. In *Proceedings of the conference on Visualization '96*, IEEE Computer Society Press, 121–126.

MAX, N., BECKER, B., AND CRAWFIS, R. 1993. Flow volumes for interactive vector field visualization. In *Proceedings Visualization '93*, IEEE CS Press, 19–24.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, 253–259.

REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER, G., AND ERTL, T. 2000. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings 2000 SIGGRAPH/EUROGRAPHICS workshop on on Graphics hardware*, ACM Press, 109–118.

SEGAL, M., AND AKELEY, K. 2001. *The OpenGL Graphics System: A Specification (Version 1.3)*. OpenGL Architecture Reference Board.

SHEN, H.-W., AND KAO, D. 1998. A new line integral convolution algorithm for visualizing time-varying flow fields. *IEEE Transactions on Visualization and Computer Graphics 4*, 2.

SHEN, H.-W., JOHNSON, C., AND MA, K.-L. 1996. Visualizing vector fields using line integral convolution and dye advection. In *Proceedings of 1996 Symposium on Volume Visualization*, IEEE Computer Society Press, 63–70.

STALLING, D., AND HEGE, H.-C. 1995. Fast and resolution independent line integral convolution. In *Proceedings of SIGGRAPH '95*, ACM SIGGRAPH, 249–256.

VAN WIJK, J. 1991. Spot noise: Texture synthesis for data visualization. *Computer Graphics 25*, 4, 309–318.

VAN WIJK, J. 2002. Image based flow visualization. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, 745–754.

ZÖCKLER, M., STALLING, D., AND HEGE, H.-C. 1996. Interactive visualization of 3d-vector fields using illuminated stream lines. In *Proceedings of the conference on Visualization '96*, IEEE Computer Society Press, 107–114.