3D IBFV

Telea, Alexandru; Wijk, Jarke J. van

Published in:
EPRINTS-BOOK-TITLE

Publication date:
2003

3D IBFV: Hardware-Accelerated 3D Flow Visualization

Alexandru Telea Jarke J. van Wijk

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, the Netherlands

Abstract

We present a hardware-accelerated method for visualizing 3D flow fields. The method is based on insertion, advection, and decay of dye. To this aim, we extend the texture-based IBFV technique presented in [van Wijk 2001] for 2D flow visualization in two main directions. First, we decompose the 3D flow visualization problem in a series of 2D instances of the mentioned IBFV technique. This makes our method benefit from the hardware acceleration the original IBFV technique introduced. Secondly, we extend the concept of advected gray value (or color) noise by introducing opacity (or matter) noise. This allows us to produce sparse 3D noise pattern advections, thus address the occlusion problem inherent to 3D flow visualization. Overall, the presented method delivers interactively animated 3D flow, uses only standard OpenGL 1.1 calls and 2D textures, and is simple to understand and implement.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms

Keywords: Flow Visualization, Hardware Acceleration, Texture Advection, OpenGL

1 Introduction

Visualization of two and three dimensional vector data produced from application areas such as computational fluid dynamics (CFD) simulations, environmental sciences, and material engineering is a challenging task. Although not a closed subject, 2D vector field visualization can now be addressed by a comprehensive array of methods, such as hedgehog and glyph plots, stream lines and surfaces, topological decomposition, and texture-based methods. The last class of methods produces a "dense", texture-like image representing a flow field, such as spot noise and line integral convolution (LIC). For a comprehensive survey of these methods, see [Hauser et al. 2002].

Texture-based techniques have a number of attractive features. First, they produce a continuous representation of a flow quantity which covers every dataset point. In comparison, discrete methods, such as streamlines and glyph plots, sample the datset, leaving to the user the interpolation of the visualization from the drawn samples. Secondly, most texture methods relieve the user from the tedious task of deciding where to place the data to be advected, e.g. the streamline seed points. Finally, recent texture-based methods employ graphics hardware, to render *animated* visualizations of the flow field at interactive rates, thereby using motion to effectively convey the impression of flow.

IEEE Visualization 2003,
October 19-24, 2003, Seattle, Washington, USA
0-7803-8120-3/03/$17.00 ©2003 IEEE

However effective and efficient in 2D, texture-based techniques are not still extensively used for visualizing 3D flow. Indeed, 3D flow visualization exhibits a number of problems, some of them related to the use of texture-based techniques, others inherent to the extra spatial dimension. The main problem inherent to 3D flow visualization is the occlusion by the depth dimension. This problem is even more obvious in case of dense visualizations, such as produced by texture-based techniques, as compared to discrete methods, such as streamlines. Another important problem of texture-based 3D flow visualizations is the difficulty of quickly producing and rendering changing volumetric images that would convey the motion impression.

Recently, Image-Based Flow Visualization (IBFV) has been proposed for 2D flow fields [van Wijk 2001]. Based on a combination of noise injection, advection, and decay, IBFV is able to produce insightful and accurate animated flow textures at a very high frame rate, is simple to implement, and runs on consumer-grade graphics hardware. In this paper, we extend the IBFV technique and make it suitable to render 3D vector fields. In this extension, we keep IBFV's main features: a high frame rate, implementation simplicity, and independence on specialized hardware. Moreover, we address the issue of depth occlusion in 3D by extending the texture noise model the original IBFV introduces. Specifically, we add an opacity (or matter) noise to the gray value (or dye) noise already present in the IBVF. This gives a simple but powerful framework for tuning the visualization density without sacrificing the overall contrast.

Section 2 overviews the existing texture-based visualization methods for 3D flow, with a focus on hardware-accelerated methods. Section 2.1 presents the 2D IBFV method we dwell upon. In Section 3, we introduce the main concepts and the implementation of our method. Section 4 presents several results obtained with our method and discusses parameter settings. Finally, Section 5 draws the conclusions.

2 Related Work

As mentioned in Sec. 1, a number of texture-based methods have been developed for visualizing 3D flow. In this section, we give a brief overview of these methods. We focus on the hardware-accelerated ones, as our aim is to produce animated 3D flow imagery at an interactive rate.

One of the first papers to address the dense visualization of 3D vector data was published in 1993 by Crawfis et al. [1993]. Here, splats are used, following the line integral convolution (LIC) technique, to show the direction of the field.

More recently, in 1999, Clyne and Dennis [1999] and Glau [1999] presented volume rendering techniques for time-dependent vector fields which make use of parallel computing, respectively 3D texture on SGI Onyx machines. In the same year, Rezk-Salama et al. [1999] present a method based on 3D LIC textures. In this work, the impression of flow is given by animating a precomputed LIC texture via cycling colors in the hardware color tables. Additionally, clipping surfaces can be interactively adjusted to specify the user's volume of interest. The method achieves, on an SGI machine, 20 frames

Proceedings of the 14th IEEE Visualization Conference (VIS'03)
0-7695-2030-8/03 $ 17.00 © 2003 IEEE

per second (fps) without clipping and 3-4 fps when complex clipping surfaces are used. However effective, the method can not interactively address time-dependent fields, as this would involve recomputing the LIC texture. Moreover, the method uses OpenGL 3D textures, which are not yet hardware accelerated by consumer-grade graphics cards.

In contrast to the above, Weiskopf et al. propose a method for rendering time-varying vector fields as animated flow textures [Weiskopf et al. 2001]. The principle is the same as for IBFV, namely the method injects and advects a texture. However, less attention is dedicated to the noise generation as in the IBFV method [van Wijk 2001]. Special programmable per-pixel operations of the nVidia GeForce card family are used to offset, or advect, a texture image T_I, as function of another texture T_V that encodes the vector field. Specifically, the authors use the offset and dot product texture shaders of the GeForce cards, as well as multitexturing capabilities to combine several textures in a single pass. However, as the authors mention, the method would be applicable to 3D fields only when the needed per-pixel operations are supported for 3D textures by the GeForce cards. For 3D fields, a similar method is proposed that uses 3D textures, per-pixel texture extensions of SGI's VPro graphics cards, and the SGI-specific post-filtering bias and scale image operations. Given this specialized hardware, the method achieves 4 fps for an image of 320^2 pixels of a 3D flow dataset of 128^3 cells.

The previous method has been extended one year later by Reck et al. to handle 3D curvilinear grids [Reck et al. 2002]. However, the dye injection and advection process that drives the visualization remains the same, which means that the method handles 3D fields only using specialized SGI hardware. Given the extra overhead of handling curvilinear grids, this method achieves only 5 fps for a field of 16^3 cells, when an accelerating cell clustering technique that trades accuracy for speed is used. Without clustering, the method needs 7 seconds per frame.

Visualizing 3D flow with dense imagery is difficult, as outlined in Sec. 1, due to the inherent occlusion problem. This problem is addressed by Interrante and Grosch [1989]. Essentially, a number of strategies for computing effective LIC textures is given, such as using region-of-interest (ROI) functions to limit the rendered volume, using sparse noise for the LIC to limit the volume fill-in, using 3D halos to give a shadow effect to the LIC splats, and using oriented fast LIC [Wegenkittl and Gröller 1997] to convey directional information by using assymetric filter kernels. However, these techniques trade speed for visual quality, and thus cannot generate interactive flow animations.

2.1 2D IBFV

As described above, an essential limitation of 3D texture-based flow visualizations is that they cannot be generated interactively, at least not on consumer-grade graphics cards. In the next section, we introduce our 3D image-based flow visualization method, or 3D IBFV for short, which addresses this issue. To better understand the method, we first overview 2D IBFV that serves as a basis for our method.

Consider an unsteady 2D vector field $v(x,t) \in \mathbb{R}^2$, defined for $t \geq 0$ and $x \in \Omega$, where $\Omega \subset \mathbb{R}^2$ is typically a rectangular region. v represents typically a flow field. However, other vector fields can be considered too.

The trajectory of a massless particle in the field, or a pathline, is the solution $p(t)$ of the ordinary differential equation:

$$\frac{dp}{dt} = v(p(t), t), \tag{1}$$

given a start position $p(0)$. Integrating the above equation by the Euler method gives the well known

$$p_{n+1} = p_n + v(p_n, t)\Delta t \tag{2}$$

Take now a 2D scalar property $A(x,t)$ advected by the flow, such as the color of advected bye. A will be, by definition, constant along pathlines defined by Eqn. 1. Thus, in a first order approximation, we have

$$A(p_{n+1}, n+1) = A(p_n, n), \quad \text{if } p_n \in S, \text{ else } 0 \tag{3}$$

However, one needs to initialize the dye advection by inserting dye into the flow. For this, we take a convex combination of the advection, as defined by Eqn. 3, and the dye injection, defined by a scalar image G, as follows

$$A(p_n, n) = (1-\alpha)A(p_{n-1}, n-1) + \alpha G(p_n, n) \tag{4}$$

Equation 4 is the essence of the IBVF, as it describes the advection, insertion, and decay of ink as function of time and space. The blending parameter α specifies the decay to injection ratio. In the original IBFV method, α was taken constant for all points in S.

```
initialize textures A and {Gn}
build warped polygon mesh P
while (true)
{
   draw mesh P textured with A
   select noise texture G from {Gn}
   draw polygon S textured with G and blend factor α
}
```

Figure 1: 2D IBFV algorithm

In terms of implementation, IBFV maps Eqn. 4 to OpenGL operations, as outlined by the pseudocode in Fig. 1. The domain S, representing the flow dataset, is modeled by a polygon mesh P, onto which the dye image A is textured. The advection of A is modeled by warping the textured polygon's vertices in the direction of the vector field with a small distance corresponding to the time step Δt. The ink image G is modeled by a set of noise textures G_n for the time instants $t = n\Delta t$. G_n's intensity is periodic in time, in order to achieve coherent pattern motion along pathlines. The ink injection, i.e. combination of A and G, is done by alpha blending the mesh P with a single polygon of the size of S textured with the noise G_n at the current time n. Finally, Equation 4 is made explicit in A by copying the warped and noise-injected image in the current frame buffer into the texture image A for the next frame. Overall, only OpenGL 1.1 operations are used, which makes the method simple and extremely fast.

3 3D IBFV

To extend 2D IBFV to 3D, three main problems are to be taken care of. First, a way must be found to perform ink advection in 3D. Secondly, 2D IBFV, as described in Sec. 2.1, produces an opaque image. If we are to extend the method to 3D, we must somehow be able to see the inside of the flow volume. We do this by varying both the noise sparsity and the opacity of the rendered volume. Finally, an efficient way must be found to render the 3D flow volume. These problems are discussed in the next sections.

3.1 Advection in 3D

The main problem in implementing advection in 3D along the lines of the original IBFV method is that there is no direct 3D analogue to the hardware-accelerated *textured mesh warping* in 2D. Although 3D meshes can be warped and 3D texture is supported by some graphics hardware, there is no *volumetric* graphics primitive equivalent to the 2D textured polygon. It is thus not possible to straightforwardly extend the 2D IBFV algorithm shown in Fig. 1 to 3D. We follow here another route, as described next.

We consider, for simplicity of the exposition, that the flow volume is aligned with the coordinate axes and that it is discretized as a regular grid $p_{ijk} = (i\Delta x, j\Delta y, k\Delta z)$, i.e. the grid cells have all sizes Δx, Δy, and Δz. We take now a 2D planar slice S parallel to the XY plane, at distance $k\Delta z$ measured along the Z axis (Fig. 2 a). Consider now a point p on slice S_k and three consecutive slices in the Z direction: S_- at $z = (k-1)\Delta z$; S; and S_+ at $z = (k+1)\Delta z$. If the advection time step Δt used for integrating Eqn. 1 is small enough compared to the Z sampling distance Δz and maximal Z velocity component max(v_Z), i.e. if $|v_Z|\Delta t < \Delta z$, then the advection that reaches S_k at p can come only from the volume between S_+ and S_-. We distinguish two situations: if $v_Z < 0$, the Z advection reaching S comes from S_+. If $v_Z > 0$, the Z advection reaching S comes from S_-. If $v_Z = 0$, there is no advection along the Z axis, so we subsume this case to any of the two former ones, e.g. consider the case $v_Z \geq 0$.

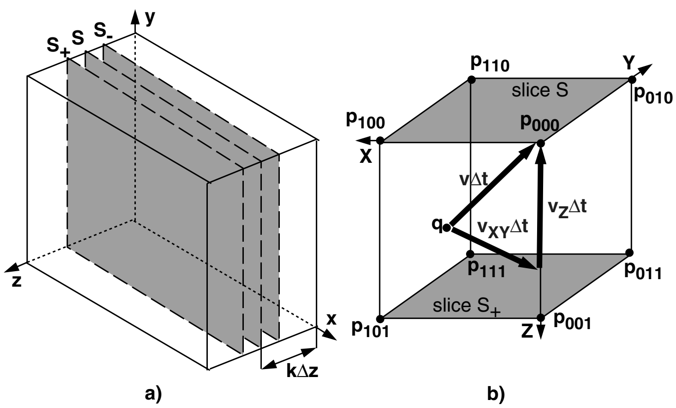

Figure 2: a) Slicing the volume. b) Decomposing 3D advection

Consider first the case $v_Z < 0$. With the above assumptions, the advection at the grid point p_{ijk} comes from the point $q = p_{ijk} - v(p_{ijk})\Delta t$ located between S and S_+. To simplify notation, without loss of generality, assume that $i = j = k = 0$ and that $v_x < 0$ and $v_y < 0$. The quantity A is advected by the flow, so $A(p_{000}, t + \Delta t) = A(q, t)$. We can evaluate $A(q)$ by trilinear interpolation of the eight vertices of the cell containing q (see Fig. 2 b)

$$\begin{aligned} A(q) = (1-z)(1-y)(1-x)A_{000} \quad & +z(1-y)(1-x)A_{001} \\ +\,(1-z)y(1-x)A_{010} \quad & +zy(1-x)A_{011} \\ +\,(1-z)(1-y)xA_{100} \quad & +z(1-y)xA_{101} \\ +\,(1-z)yxA_{110} \quad & +zyxA_{111} \end{aligned} \tag{5}$$

where x, y, and z are the local cell coordinates of point q, i.e. $x = -v_x\Delta t$, $y = -v_y\Delta t$, and $z = -v_z\Delta t$, and A_{ijk} are the values of A at the cell corner points p_{ijk}. Denote by A_k the sum of the terms in Eqn. 5 that have a factor $(1 - z)$, i.e.

$$\begin{aligned} A_k = (1-y)(1-x)A_{000} \quad & +y(1-x)A_{010} \\ +\,(1-y)xA_{100} \quad & +yxA_{110} \end{aligned} \tag{6}$$

and similarly by A_{k+1} the sum of the terms that have a factor z

$$\begin{aligned} A_{k+1} = (1-y)(1-x)A_{001} \quad & +y(1-x)A_{011} \\ +\,(1-y)xA_{101} \quad & +yxA_{111} \end{aligned} \tag{7}$$

We can thus rewrite Eqn. 5 as

$$A_-(p) = A(q) = (1-z)A_k + zA_{k+1} \tag{8}$$

where $A_-(p)$ is the advection at p if $v_Z(p) < 0$. The terms A_k and A_{k+1} allow a special interpretation. Indeed, A_k is the 2D advection caused by v_{XY}, the projection of v, onto the plane S, in the 2D rectangular cell $(p_{000}, p_{100}, p_{110}, p_{010})$.

Similarly, A_{k+1} is the planar 2D advection caused by v_{XY} in the 2D cell $(p_{001}, p_{101}, p_{111}, p_{011})$. Finally, Eqn. 8 denotes the advection of A from point p_{001} to point p_{000} along the Z axis, i.e. in the field v_Z which is the projection of v onto Z. Recall that the above held for $v_Z < 0$. If $v_Z \geq 0$, we deduce a similar relation to Eqn. 8, i.e.

$$A_+(p) = A(q) = (1-z)A_k + zA_{k-1} \tag{9}$$

where $A_+(p)$ is the advection at p if $v_Z(p) \geq 0$. The only difference here is that we consider the plane S_- instead of S_+, i.e. the Z advection brings information on the plane S from the opposite direction as in the former case $v_Z < 0$. Combining the two equations Eqn. 8 and Eqn. 9, we obtain the total advection $A(p)$

$$A(p) = A_-(p) + A_+(p) \tag{10}$$

Summarizing, the advection of the scalar quantity A from q to p in the field v can be decomposed in a series of 2D advection processes in the planes S, S_+, and S_-, followed by a 1D advection along the Z axis from S_- to S for those points p having $v_Z(p) \geq 0$ and a 1D advection along the Z axis from S_+ to S for the points having $v_Z(p) < 0$.

In the next section, the implementation of Eqn. 10 is presented.

3.2 Advection implementation

As explained in the previous section, we decompose the 3D advection of a scalar property A in a series of slice planar advections and a series of Z-axis aligned advections. Following this idea, the global 3D advection procedure for a single time step on a volume consisting of a set S_i of slices, $i = 0..N-1$, is given by the pseudocode in Fig. 3:

```
for i = 0 to N-1
{
  if (i>0)                                          (1)
     do 1D Z-axis advection from S_i-1 to S_i
  if (i<N-1)                                        (2)
     do 1D Z-axis advection from S_i+1 to S_i
  do 2D IBFV-based advection in the slice S_i       (3)
}
```

Figure 3: Advection procedure

The planar advection terms of the type A_{XY} in Eqns. 8 and 9 can be directly computed by applying the 2D IBFV method considering the projection v_{XY} of v to the plane S (step 3 of the algorithm). Denote now by A_k the value of A over all points of S_k, for a given slice k. Similarly, denote by v_{Zk} the velocity Z component over the points of S_k. From Eqn. 8, step 1 of the algorithm becomes (for Δt taken to be 1):

$$A_k := (1 - \max(v_{Zk-1}, 0))A_k + \max(v_{Zk-1}, 0)A_{k-1} \tag{11}$$

Similarly, step 2 becomes

$$A_k := (1 - \max(-v_{Zk+1}, 0))A_k + \max(-v_{Zk+1}, 0)A_{k+1} \tag{12}$$

In the above, the max function is used to consider the two cases $v_Z < 0$ and $v_Z \geq 0$ explained in Sec. 3.1. Note that we evaluate the component v_Z in the slices S_{k-1} and S_{k+1} to perform the advection. Alternative schemes can be used, e.g. evaluate v_Z as the average, or linear interpolation, of the velocity v_{Zk} in the current plane S_k and the velocities v_{Zk-1} and v_{Zk+1} in the planes S_{k-1} and S_{k+1}. Similarly, the planar and Z-advections (steps 1,2,3 in Fig. 3) can be done in different orders, leading to different integration schemes. Given

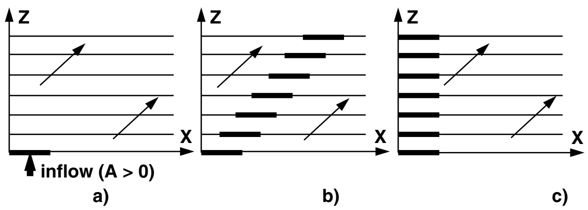

Figure 4: Effect of operation order: a) test configuration. b) correct advection. c) incorrect advection

that we require $|v_Z|\Delta t < \Delta z$, the chosen scheme in Fig. 3 performed well for all our datasets. Moreover, this scheme minimizes the number of drawing operations (see also Fig. 9 in Sec. 3.5).

In the above scheme, the order of the planar and Z-advections is, however, important. Consider, for example, a laminar diagonal flow $(v_x, v_y, v_z) = (1, 0, 1)$ and a property A (e.g. ink) nonzero in some area S_0 and zero elsewhere (Fig. 4 sketches this as seen along the Y axis). The advection should carry the 'inflow' value A from S_0 diagonally along the slices S_k, leading to the situation in Fig. 4 b. This is the result delivered by the algorithm in Fig. 3. If, however, we did, for each slice, first the 2D IBFV advection and then the Z-advection, to name one of the other possible orders, we would get the obviously wrong result in Fig. 3 c after one time step.

Let us look closer at Eqns. 11 and 12 which describe the algorithm steps 1 and 2. In essence, one performs a convex combination of the property A_k (planar advection) over consecutive slices, using the value z which represents the velocity Z component at grid points. As explained above, z is always greater or equal to zero. To generalize this for all points over a slice S_k, i.e. for other points than mesh points, we bilinearly interpolate z over the slice S_k from the z values at the mesh points.

If we were able to implement the above Z advection using hardware acceleration, then the complete 3D advection algorithm would be hardware accelerated, since the 2D IBFV method obviously is so. The next section describes how this can be done.

3.3 Hardware accelerated Z advection

We start by evaluating, for all mesh points i,j of all slices S_k, the quantities

$$\begin{aligned} v_{+Zijk} &= \max(\ v_{Zijk}, 0) \\ v_{-Zijk} &= \max(-v_{Zijk}, 0) \end{aligned} \tag{13}$$

where v_{+Zijk} and v_{-Zijk} are the absolute values of the Z velocity components in the direction, respectively in opposite direction of the Z axis, on slice S_k. For simplicity of notation, we drop the indices i and j, i.e. use v_{+Zk} instead of v_{+Zijk}, and similarly for v_{-Zk}. For every slice k, we encode the above as two OpenGL 2D luminance textures. All textures we use are named textures, as this ensures a fast access to texture data. The luminance textures store one component (luminance) per texel. The texture type (as passed to the OpenGL call `glTexImage2D`) is either `GL_INTENSITY8` or `GL_INTENSITY16`. This gives 8, respectively 16 bits resolution for the velocity Z component. Since two separate textures are used for v_{+Zk} and v_{-Zk}, an effective range of 9, respectively 17 bits is used for the velocity Z component. For all our applications, 16 bit textures have delivered good results, whereas the 8 bit resolution caused undersampling artifacts for vector fields with a high Z value range. Note also that the framebuffer resolution, used to accumulate the results, is also important for the overall computation accuracy. The spatial (X,Y) texture resolutions determine a trade-off between representation accuracy and memory use. As a simple rule, these textures shouldn't be larger than the vector dataset's XY resolution, since this is the complete Z velocity information to be encoded. Practically, resolutions of 64^2 and 128^2 have given very good results.

Now we can simply rewrite the Z advection equations (11) and (12) as

$$A_k := (1 - v_{+Zk-1})A_k + v_{+Zk-1}A_{k-1} \tag{14}$$
$$A_k := (1 - v_{-Zk+1})A_k + v_{-Zk+1}A_{k+1} \tag{15}$$

where :=, in the above, denotes assignment. Similarly, we implement the property A_k as a set of 2D RGBA textures, one for every slice plane S_k. The XY resolution of the property textures is exactly analogous to the resolution of the single RGB texture used 2D IBFV. Remark, however, that we need a texture alpha channel, whereas 2D IBFV did not. The use of this channel is explained in Sec. 3.5.

```
glDisable(GL_BLEND);
glBindTexture(GL_TEXTURE_2D,A[k-1]); drawQuad();
// A = A_k-1                          (draw previous slice)
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO,GL_SRC_COLOR);
glBindTexture(GL_TEXTURE_2D,v_+Z[k-1]); drawQuad();
// A = v_+Zk-1 A_k-1                  (modulate by v_+Zk-1)
glBindTexture(GL_TEXTURE_2D,temp);
glCopyTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,0,0,N,N,0);
// temp = v_+Zk-1 A_k-1               (save advection from previous slice)
glDisable(GL_BLEND);
glBindTexture(GL_TEXTURE_2D,A[k]); drawQuad();
// A = A_k                            (draw current slice)
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO,GL_ONE_MINUS_SRC_COLOR);
glBindTexture(GL_TEXTURE_2D,v_+Z[k-1]); drawQuad();
// A = (1-v_+Zk-1) A_k                (modulate by 1- v_+Zk-1)
glBlendFunc(GL_ONE,GL_ONE);
glBindTexture(GL_TEXTURE_2D,temp); drawQuad();
// A = v_+Zk-1 A_k-1 + (1-v_+Zk-1) A_k  (add of previous slice)
glBindTexture(GL_TEXTURE_2D,A[k]);
glCopyTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,0,0,N,N,0);
// A_k = A                            (save result in texture A_k)
```

Figure 5: OpenGL code implementing Z advection

The velocity-coding textures allow us to efficiently implement the above equations using graphics hardware. Equation 14 is implemented by the OpenGL code in Fig. 5. The OpenGL `GL_TEXTURE_ENV_MODE` value passed to the `glTexEnv` function is always `GL_REPLACE`, i.e. we do not use mesh vertex colors for the textures. All effects are obtained by varying the blending modes via the `glBlendFunc` function. A similar code is needed to implement Eqn. 15. Just as in the 2D IBFV case, the textures are created with `GL_LINEAR` values for the `GL_TEXTURE_MIN_FILTER` and the `GL_TEXTURE_MAX_FILTER` parameters of `glTexParameter`, to ensure bilinear interpolation. The function `drawQuad` draws a single textured quadrilateral that covers the whole image. The variable `temp` is one RGBA texture used as a temporary workspace. The viewing parameters are such that there is a one to one mapping of the slices S_k to the viewport.

Overall, our advection uses five textured quad drawing operations and two `glCopyTexImage2D` operations per slice and per advection direction, done into P-buffers, for speed reasons. This has the extra advantage of not needing an on-screen window showing the inner working of the advection process.

3.4 Noise injection: Preliminaries

The last section has shown how 3D advection can be implemented using hardware acceleration. The second important question is now which scalar property to advect. We follow 2D IBFV, i.e. inject a spatially random, temporally periodic noise signal (see [van Wijk 2001] for a detailed analysis). In the following, denote by A_n =

Proceedings of the 14th IEEE Visualization Conference (VIS'03)
0-7695-2030-8/03 $ 17.00 © 2003 IEEE

$(A_{In}, A_{\alpha n})$ a 2D image at moment $t = n\Delta t$, consisting of an intensity component A_{In} and an alpha (opacity) component $A_{\alpha n}$. The original 2D IBFV equation (4) becomes now, for every 3D slice:

$$A_n = (1-\alpha)A_{n-1} + \alpha G_n \tag{16}$$

(we drop the spatial p_n parameter for conciseness). Here, G_n is a 2D texture modelling the noise injected at time step n, consisting of an intensity G_{In} and an opacity $G_{\alpha n}$ component. Just as in the 2D IBFV case, the noise should be periodical in time, as this creates the impression of color continuity along streamlines. In our 3D case, we add an extra spatial dimension, i.e. the Z axis, to the noise. Just as for the temporal dimension, the noise must be a continuous, periodic signal along the Z axis, to minimize high frequency artifacts. For a full discussion hereof, see the original paper [van Wijk 2001].

In 2D IBFV, both $A_{\alpha n}$ and $G_{\alpha n}$ were identically one (i.e. opaque) everywhere. More precisely, the actual 2D IBFV implementation used RGB, and not RGBA, textures. Hence, if we directly implement Eqn. 16 in the 3D case, we obtain a fully opaque flow volume consisting of gray value ink. Although 'correct', this result is useless for visualizing the flow.

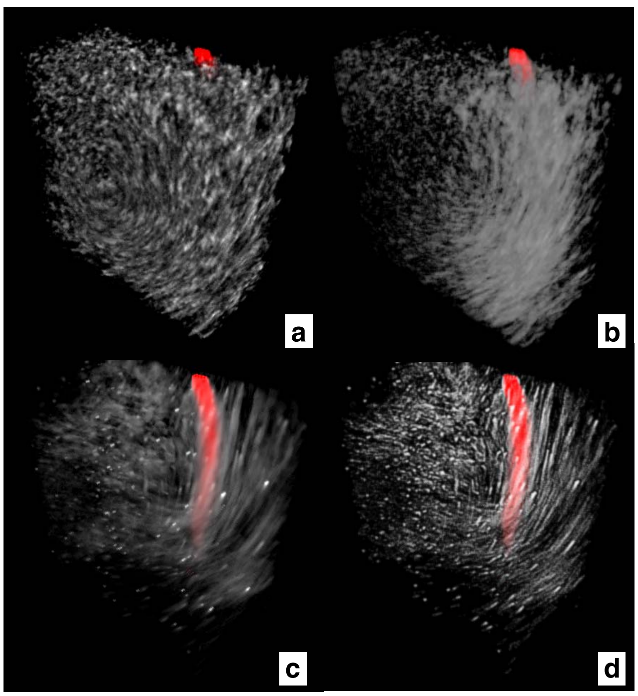

Figure 6: Noise injection: a) and b) 2D IBFV noise mode. c) and d) 3D IBFV improved noise

A first idea to tackle this problem is to use the texture alpha channel to model *transparency*, i.e. to use a $G_{\alpha n}$ which is not identically 1, but a noise signal similar to G_{In}. Texels with a low alpha would correspond to 'transparent' noise, i.e. holes in the texture slices, through which one could see deeper into the flow volume. Unfortunately, this method produces visualizations which have a poor contrast and which, after a while, tend to fill up the volume with texels having the same (average) transparency. When rendered by alpha compositing (see Sec. 3.6), little is seen in the depth of the flow. Figure 6 a (see also Color Plate) shows this for a vortex flow in which red ink has been injected to trace a streamline. Discarding lower alpha values by using OpenGL alpha testing improves the results somewhat. However, the red streamline is still not visible (Fig. 6 b).

Parameter tuning can produce only mild improvements, as the following analysis shows. If an α close to zero is used, in order to diminish the noise injection, then the 'holes' in the noise have no chance to show up, and the visualization becomes quickly blurred. If a high α is used, to make the noise more prominent, then this will erase the current information (second term). The 'ink decaying' effect diminishes, and one sees only the noise variation in time. If one uses a sparse noise pattern, i.e. a $G_{\alpha n}$ which has mostly (very) low values, in conjunction with an average (0.5) to high (0.9..1) α, then the occlusion effect diminishes indeed. However, the method may now easily fall into the other extreme, where too little ink is injected, and the injected 'holes' quickly erase it.

Overall, we have found that it is very hard to get a parameter setting which produces a high-contrast visualization, both in terms of injected noise color and transparency. The visualization quickly tends to get blurred and produce low-contrast slices in the alpha channel. Although such visualizations do convey some insight into the flow, due to their animated nature, we need a better solution for the contrast problem. This is described in the next section.

3.5 Noise injection in 3D

The key to producing a high-contrast 3D flow visualization is to refine Eqn 16. The main problem of this model is that it is not able to express *what* to inject (i.e. ink or 'holes') and *where* to inject it in the volume independently. Instead of the original formulation, we propose to use

$$A_n = (1-H_n)A_{n-1} + H_n G_n \tag{17}$$

Here, we replace the constant α parameter from Eqn. 16 with a noise function H_n. In other words, we use now two noise signals G_n and H_n instead of a single one. Both signals are periodic functions of Z position and time, as before, so they are stored as two sets of 2D textures, two textures per Z slice. The noise signal H_n is a single-channel alpha-texture. It describes *where* to inject the noise. If $H_n = 1$ at some point, it means we inject the noise signal in G_n at that point. If $H_n = 0$, we inject nothing at that point.

The signal G_n may be a luminance-alpha (LA) or RGBA signal, depending whether we wish to inject gray, respectively colored noise. It describes *what* to inject at a given point, both in terms of the color (L or RGB) and the 'hole' or 'matter' injection (the A channel). Although we have experimented with color noise of random hue, monochrome noise (whether gray or color) has given the best results.

Using two noise signals instead of one allows us to specify what and where we inject independently. To inject fully transparent holes at a few locations, we use a G_n having texels with a zero alpha and a H_n sparsely populated with high values. In comparison with the model given by Eqn. 16, we can now inject a fully transparent hole close to a fully opaque ink spot or a location where no injection at all takes place. Figure 6 c and d clearly show that this approach delivers a more transparent, but still highly contrasting visualization. Now the red streamline inside the flow is clearly visible.

We next describe the design of the G_n and H_n noise signals. Just as for the 2D IBFV, we compute the noise H_n at a moment t and spatial position x, y, and slice k, by using a periodic 'transfer function' h, phase-shifted with t, from a start moment given by a white noise signal N:

$$H(x,y,k,t) = h((N(x,y,k)+t) \bmod T) \tag{18}$$

where $T = t_N\Delta t$ is the time period of t_N different moments. The noise components G_{In} and $G_{\alpha n}$ are computed analogously, using two more functions g_I and g_α. Note also that the phase offset (i.e. N signal) used to compute H is different than the one used for G. If it were the same, the two noises G and H would be in phase, which would create visible artifacts.

Proceedings of the 14th IEEE Visualization Conference (VIS'03)

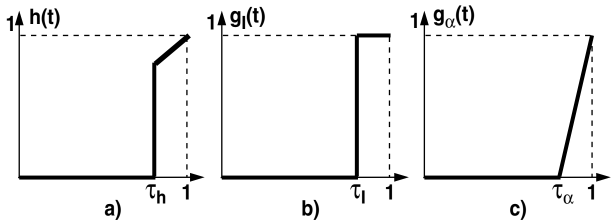

Figure 7: Transfer functions for a) H_n, b) G_{In}, and c) $G_{\alpha n}$

We choose the functions h,g_I, and g_α as follows (see also Fig. 7). For h we propose

$$h(t) = \begin{cases} 0, \; t < \tau_h \\ t, \; t > \tau_h \end{cases} \tag{19}$$

In other words, nothing is injected for $t < \tau_h$, whereas strong injection takes place for $t > \tau_h$.

For g_I, we can use a similar function, controlled by a parameter τ_I. However, we found the step function

$$g_I(t) = \begin{cases} 0, \; t < \tau_I \\ 1, \; t > \tau_I \end{cases} \tag{20}$$

better. This would inject black ink for $t < \tau_I$ and white ink otherwise. Since the variable opacity already modulates the noise blending, we found the above ink injection to be sufficient, i.e. we didn't make use of gray ink.

Finally, for g_α we propose

$$g_\alpha(t) = \begin{cases} 0, \; t < \tau_\alpha \\ \frac{t-\tau_\alpha}{1-\tau_\alpha}, \; t > \tau_\alpha \end{cases} \tag{21}$$

Hence, holes are injected for $t < \tau_\alpha$. For $t > \tau_\alpha$, ink is injected, its color being given by g_I. The above g_α produces asymmetric advection patterns, thinner in the sense of the flow and thicker in the opposite direction. This may serve as an indication of the flow sense. Choosing

$$g_\alpha(t) = \begin{cases} 0, \; t < \tau_\alpha \\ \frac{1-t}{1-\tau_\alpha}, \; t > \tau_\alpha \end{cases} \tag{22}$$

will reverse the orientation of the flow patterns. For concrete settings for the τ_h, τ_I, and τ_α parameters, and a summary of all parameters, see Sec. 4.

The noise injection modelled by Eqn 17 can be efficiently implemented using graphics hardware. For this, we actually precompute and store the signals H_n and $Q_n = G_n H_n$ as 2D textures. As in the original 2D IBFV, these textures may be smaller than the property textures A, to save memory and rendering time. Texture repeat and stretch operations are used to map the H and Q textures' size PQ_N to the size A_N of the textures A. Moreover, we store only textures for t_N time instants and HQ_{ZN} values of the Z coordinate (slice index k) and then repeat them periodically, both in Z direction and time (see Eqn. 18). Normally, $HQ_{ZN} < Z_N$, where Z_N is the number of Z slices. Overall, we store thus $HQ_{ZN} t_N$ pairs of H and Q textures.

The noise injection algorithm for a given Z slice k is shown in Fig. 8. Here, we denote the noises G and H at time step n in slice k by G_{kn} and H_{kn} respectively. As noise injection is done after the Z advection and 2D IBFV steps, the drawable contains the signal A_k when the injection starts. Recall also that, when alpha-textures are used in the `GL_REPLACE` mode of `glTexEnv`, they only affect the destination alpha channel (e.g. step 1 of the code in Fig. 8). Overall, the noise injection we propose uses just two textured quad draws, as compared to the original 2D IBFV which used one similar operation.

```
glEnable(GL_BLEND);
glBlendFunc(GL_ZERO,GL_ONE_MINUS_SRC_ALPHA);
glBindTexture(GL_TEXTURE_2D,P[k][t]); drawQuad();
// A = (1-Hkn) Ak                  (clear areas where we inject
                                    something)

glBlendFunc(GL_ONE,GL_ONE);
glBindTexture(GL_TEXTURE_2D,Q[k][t]); drawQuad();
// A = (1-Hkn) Ak + HknGkn         (add ink and/or holes to the
                                    injection areas)
```

Figure 8: OpenGL code implementing noise injection

```
for (k = 0; k<ZN; k++)          // initialization
{
  for (n=0; n<tN; n++)
      precompute noise textures Hkn and Qkn = Gkn Hkn
  build warped polygon mesh Pk
  create texture Ak
}
while(true)                      // execution
{
  for(k = 0; k<ZN; k++)
  {
     clear drawable
     perform the Z advection from slice k-1 to k
     perform the Z advection from slice k+1 to k
     draw mesh Pk textured with current drawable
     inject noise using textures Qkn and Hkn
     copy drawable to texture Ak
  }
  display results (draw textures Ak back to front)
}
```

Figure 9: Complete IBFV 3D method

Putting it all together, we obtain the complete 3D IBFV method, shown in pseudocode in Fig. 9. The method has two phases, just like in the 2D IBFV case (compare Fig. 9 with Fig. 1). In the initialization phase, the noise and property textures (H_{kn}, Q_{kn}, and A_k) as well as the warped polygon meshes P_k are built for all Z slices k (and all time instants n, for the noise textures). In the execution phase, the method is structurally similar to the 2D IBFV case, except for the new Z advection step.

3.6 Rendering

As the 3D IBFV method is running, we visualize its results by drawing, in back to front order, the RGBA texture slices A_k. Blending is enabled and the `glBlendFunc` function's source and destination factors are, as usually for this technique, set to `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA`. An important enhancement is obtained by enabling the OpenGL alpha test and cutting off all alpha values below a given α_{cut}. Setting α_{cut} between 0.01 and 0.1 allows one to quickly 'coarsen' the flow volume by discarding the almost transparent texels. Although not visible in separate slices, such texels can accumulate in the back to front blend and increase the overall opacity. Figure 11 shows a flow volume rendered for three different α_{cut} values, from three viewpoints. Clearly, the higher α_{cut} values allow more insight in the flow.

We should remark that the choice of the Z slicing direction is, so far, arbitrary. For a regular dataset, an efficient choice is to minimize the Z slices count, i.e. choose Z as the axis having the least cell count from the three axes. Visualizing the back-to-front rendered slices (Sec. 3.6) will definitely produce poor results if the slicing direction is orthogonal to the line of sight, as we then tend to look through the slices. Still, for the various flow volumes we visualized, this didn't seem to be a major hindrance for the users. If desired, as usually done in many volume rendering applications, the method can detect this situation and change the slicing direction interactively, as the viewpoint changes.

A considerably better rendering can be obtained if 3D textures are

Proceedings of the 14th IEEE Visualization Conference (VIS'03)
0-7695-2030-8/03 $ 17.00 © 2003 IEEE

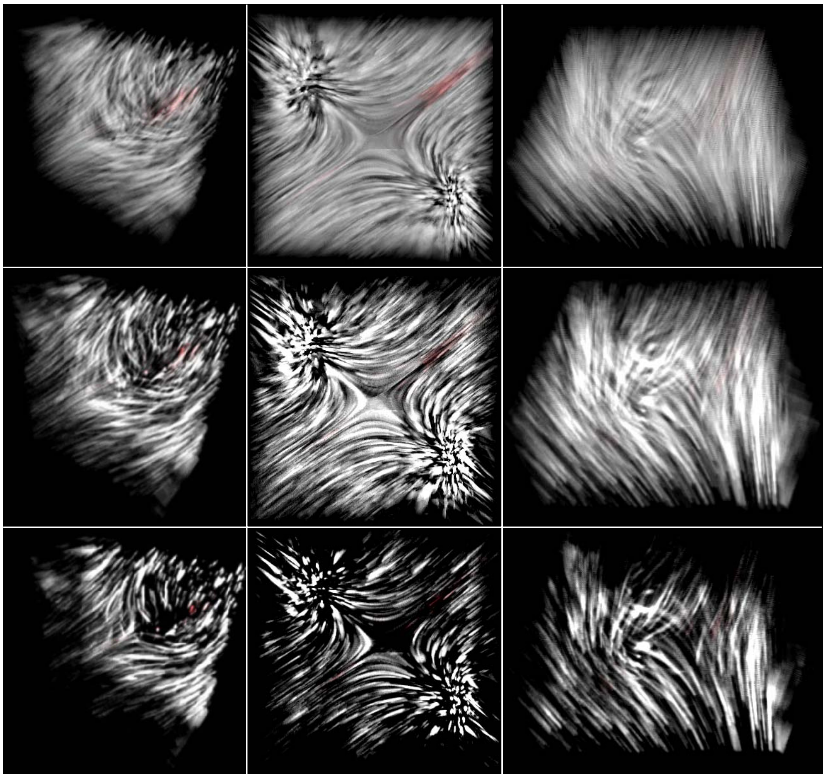

Figure 11: 3D flow rendered with $\alpha_{cut} = 0.01$ (top), $\alpha_{cut} = 0.02$ (middle), and $\alpha_{cut} = 0.05$ (bottom)

used. In this case, the 2D textures A_k can be slices in a single 3D texture volume. The whole 3D IBFV method stays the same. However, a visibly improved rendering can be done by drawing, back to front, a number of 2D polygons parallel to the viewport, that use the 3D texture. This is exactly what most volume rendering methods do. Remark that the use of 3D texture is, in this case, strictly needed for rendering the 3D IBFV results, not for producing them. Our main problem here was that (reasonably large) 3D textures are seldom present with hardware acceleration in the graphics cards we availed of (GeForce 2 and 3).

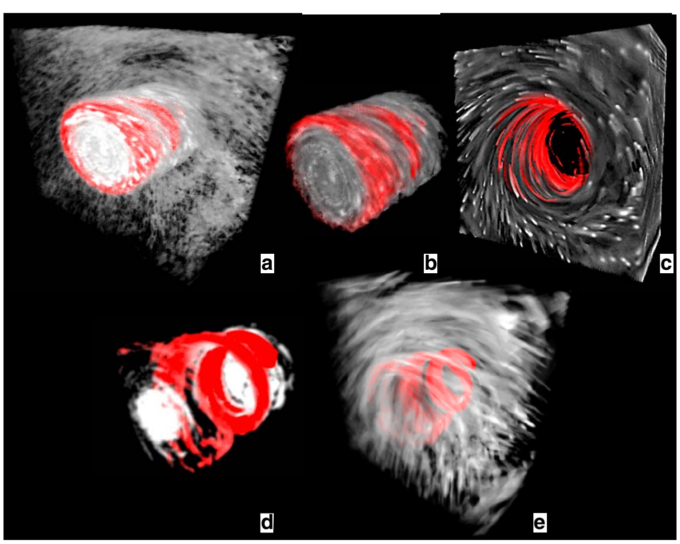

Figure 10: Parameter settings for 3D IBFV

4 Discussion

We shall discuss now the parameter settings and the method's performance and memory requirements.

Compared to 2D IBFV, the 3D method introduces several new parameters. Here follows the complete parameter list of 3D IBVF (see Sec. 3.5 for details):

- A_N: resolution of the (4 bytes, RGBA) property textures A_k
- Z_N: number of Z slices
- v_N: resolution of the velocity (2 byte, alpha) textures
- HQ_N: resolution of textures H (1 byte, luminance) and Q (2 bytes, luminance alpha)
- HQ_{ZN}: Z resolution of the noise textures H and Q
- t_N: time resolution of noise textures
- X_N,Y_N: resolution of meshes (12 bytes per vertex)
- τ_h: noise injection strength (see Sec. 3.5)
- τ_α: ink to hole injection ratio
- τ_I: black to white noise ink ratio
- α_{cut}: alpha test threshold

The visualization is strongly affected by the last four parameters (explained in Sec. 3.5). τ_h controls how much noise, consisting of holes and/or ink, is injected. Good values for τ_h range from 0.01 to 0.1. Higher values tend to produce too noisy images. τ_α is the ink to hole injection ratio. If close to 1, holes are injected, i.e. matter is 'carved out' of the flow volume. If close to 0, ink is injected. Good

Proceedings of the 14th IEEE Visualization Conference (VIS'03)
0-7695-2030-8/03 $ 17.00 © 2003 IEEE

values for τ_α are 0.9 or higher, in order to produce a sparse flow volume. τ_I controls the ink luminance. If close to 0, more white ink is used. If close to 1, more black ink is used. Good values range around 0.1, to favor white ink.

As for the 2D IBFV, we can inject ink at chosen locations to trace streamlines. Figure 10 shows a helix flow in which red ink was injected close to the back plane's center. Besides color, we set the ink's alpha to 1 (opaque). Using a high α_{cut} discards most of the noise but keeps the opaque ink (Fig. 10 d). Another idea is to set, for the noise injected close to the inflow's center, a high alpha. This alpha is advected into the helix core, which becomes opaque. Next, we set a high α_{cut} and discard all outside the core (Fig. 10 b). If low alpha noise is used instead, we 'carve out' the core (Fig. 10 c). Another effective option is to inject strong noise for a few steps (Fig. 10 a) and then turn it off by setting τ_P to zero. This creates a more pleasant, smoother visualization (Fig. 10 e). This technique was used also for the flow in Fig. 11.

As 2D IBFV, we are aware that our 3D method is limited in the range of velocities it can display. Specifically, both the maximum values for the XY and Z velocities must be smaller than the XY and Z cell sizes divided by the time step. In practice, we handle this by either using a small time step or by clamping the higher velocities.

We consider now the total memory (in bytes) our method needs to store the textures and polygon meshes:

$$M = 2Z_N(6X_NY_N + 2A_N^2 + 2v_N^2) + 3HQ_{ZN}HQ_Nt_N \quad (23)$$

Given the settings: $A_N = 512$, $X_N = Y_N = Z_N = 50$, $v_N = HQ_N = HQ_{ZN} = 64$, and $t_N = 32$, we obtain $M = 59$ MBytes. This just fits into our 64 MB GeForce cards. If M exceeds the graphics card memory, transfer to the normal memory takes place, which severely degrades the rendering performance.

The rendering time is, as expected, proportional with the texture sizes, the number of slices Z_N, and the mesh resolution X_NY_N. For the GeForce 2 and GeForce 3 Ti 400 cards, the mesh resolution X_NY_N was by far the dominant factor in the rendering time. This was much less severe for the standard GeForce 3 cards, where the dominant factor seems to be the texture resolution A_N. A few rendering timings are shown in Fig. 12. We preferred using smaller Z_N than X_N and Y_N values as this delivered higher performance, as expected. Besides X_N, Y_N, and A_N, all configurations use the settings described for the memory estimation above, and run on a Pentium III PC at 800 MHz with Windows 2000. Given that the 2D IBFV produced 60 frames per second on the same hardware and that we render 50 Z slices, 3D IBFV delivers, so to speak, more throughput per rendered slice.

X_{res}=Y_{res}	20	30	40	60	80	A_{res}
GeForce 3 Ti	9.0	4.5	3	1.6	0.9	512
	10.2	6.3	3.9	2.2	1.2	256
GeForce 3	3.0	2.9	2.9	2.4	2.2	512
	11.0	10.8	10.6	10.4	9.0	256

Figure 12: 3D IBFV timings, frames per second

Another discussion point is the usage of the velocity-encoding luminance textures (Sec. 3.3). An alternative approach would be to encode the Z velocities as 2D mesh vertex colors. In this case, the Z advection would be done by drawing the textured mesh with the `GL_MODULATE` mode of the `glTexEnv` OpenGL function, instead of blending two textures, as we do now. This approach (which we tried first) has several disadvantages. First, storing a N^2 quad mesh is much more expensive than storing a N^2 luminance texture. Even if mesh coordinates are somehow shared, we must still store a full RGBA value per vertex, as OpenGL 1.1 does not allow storing only vertex luminance values. Using velocity textures, we store just the luminance values. Secondly, we found out that, on several nVidia GeForce 2 and 3 cards, drawing a single quad with a N^2 texture is much faster than drawing an N^2 quad mesh with vertex colors. Note, however, that the two approaches are functionally identical.

5 Conclusions

3D IBFV is a method for visualizing 3D fluid flow as moving texture patterns using consumer graphics hardware. As its 2D counterpart, 3D IBFV offers a framework to create several flow visualizations (stream 'tubes', LIC-like patterns, etc), high frame rates, and a simple OpenGL 1.1 implementation, without 3D textures. However, while 2D IBFV easily handles instationary fields, 3D IBFV currently handles only the time independent case, as instationary fields require a continuous update of the velocity textures. 3D IBFV produces higher frame rates on less specialized hardware than other 3D flow visualization methods. We extend the 2D IBFV noise concept by adding 'opacity noise'. Combined with alpha testing, we get a simple and interactive way to examine flow volumes in the depth dimension. We present the implementation and parameter settings in detail, so that one can readily apply it. We see several extensions of 3D IBFV. New noise and/or ink injection designs can produce 3D flow domain decompositions, stream surfaces, curved arrow plots, and many other visualizations. Secondly, using 3D texture, as these become more widely available, may lead to a simpler, more accurate 3D IBFV. Finally, using DirectX floating-point textures may provide better accuracy. We plan to investigate these options in the near future.

References

CLYNE, J., AND DENNIS, J. 1999. Interactive direct volume rendering of time-varying data. *Proc. IEEE VisSym '99, eds. E. Gröller, H. Loffelmann, W. Ribarsky, Springer, pp. 109-120.*

CRAWFIS, R., MAX, N., BECKER, B., AND CABRAL, B. 1993. Volume rendering of scalar and vector fields at llnl. *Proc. IEEE Supercomputing '93, IEEE CS Press, pp. 570-576.*

GLAU, T. 1999. Exploring instationary fluid flows by interactive volume movies. *Proc. IEEE VisSym '99, eds. E. Gröller, H. Loffelmann, W. Ribarsky, Springer, pp. 277-284.*

HAUSER, H., LARAMEE, R. S., AND DOLEISCH, H. 2002. State-of-the-art report 2002 in flow visualization. *Tech. Report TR-VRVis-2002-003, VRVis Research Center, Vienna, Austria.*

INTERRANTE, V., AND GROSCH, C. 1989. Visualizing 3d flow. *IEEE Computer Graphics and Applications, 18(4), 1998, pp. 49-52.*

RECK, F., REZK-SALAMA, C., GROSSO, R., AND GREINER, G. 2002. Hardware accelerated visualization of curvilinear vector fields. *Proc. VMV '02, eds. T. Ertl, B. Girod, G. Greiner, H. Niemann, H.P. Seidel, Stuttgart, pp. 228-235.*

REZK-SALAMA, C., HASTREITER, P., CHRISTIAN, T., AND ERTL, T. 1999. Interactive exploration of volume line integral convolution based on 3d texture mapping. *Proc. IEEE Visualization '99, eds. D. Ebert, M. Gross, B. Hamann, IEEE CS Press, pp. 233-240.*

VAN WIJK, J. J. 2001. Image based flow visualization. *Computer Graphics (Proc. SIGGRAPH '01), ACM Press, pp. 263-279.*

WEGENKITTL, R., AND GRÖLLER, E. 1997. Fast oriented line integral convolution for vector field visualization via the internet. *Proc. IEEE Visualization '97, IEEE CS Press, pp. 119-126.*

WEISKOPF, D., HOPF, M., AND ERTL, T. 2001. Hardware-accelerated visualization of time-varying 2d and 3d vector fields by texture advection via programmable per-pixel operations. *Proc. VMV '01, eds. T. Ertl, B. Girod, G. Greiner, H. Niemann, H.P. Seidel, Stuttgart, pp. 439-446.*

Proceedings of the 14th IEEE Visualization Conference (VIS'03)
0-7695-2030-8/03 $ 17.00 © 2003 IEEE