Image Based Flow Visualization

Jarke J. van Wijk*
Technische Universiteit Eindhoven
Dept. of Mathematics and Computer Science

Abstract

A new method for the visualization of two-dimensional fluid flow is presented. The method is based on the advection and decay of dye. These processes are simulated by defining each frame of a flow animation as a blend between a warped version of the previous image and a number of background images. For the latter a sequence of filtered white noise images is used: filtered in time and space to remove high frequency components. Because all steps are done using images, the method is named Image Based Flow Visualization (IBFV). With IBFV a wide variety of visualization techniques can be emulated. Flow can be visualized as moving textures with line integral convolution and spot noise. Arrow plots, streamlines, particles, and topological images can be generated by adding extra dye to the image. Unsteady flows, defined on arbitrary meshes, can be handled. IBFV achieves a high performance by using standard features of graphics hardware. Typically fifty frames per second are generated using standard graphics cards on PCs. Finally, IBFV is easy to understand, analyse, and implement.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques; I.6.6 [Simulation and Modeling]: Simulation Output Analysis

Keywords: Flow visualization, texture mapping, line integral convolution

1 Introduction

Fluid flow plays a dominant role in many processes that are important to mankind, such as weather, climate, industrial processes, cooling, heating, etc. Computational Fluid Dynamics (CFD) simulations are carried out to achieve a better understanding and to improve the efficiency and effectivity of manmade artifacts. Visualization is indispensable to achieve insight in the large datasets produced by these simulations. Many methods for flow visualization have been developed, ranging from arrow plots and streamlines to dense texture methods, such as Line Integral Convolution [Cabral and Leedom 1993]. The latter class of methods produces very clear visualizations of two-dimensional flow, but is computationally expensive.

We present a new method for the visualization of two-dimensional vector fields in general and fluid flow fields in particular. The method provides a single framework to generate a *wide variety* of visualizations of flow, varying from moving particles, streamlines, moving textures, to topological images. Three other features of the method are: handling of *unsteady flow*, *efficiency* and *ease of implementation*. More specific, all 512×512 images presented in this paper are snapshots from animations of unsteady flow fields. The animations were generated at up to 50 frames per second (fps) on a notebook computer, and the accompanying DVD contains a simple but complete implementation in about a hundred lines of source code.

The method is based on a simple concept: Each image is the result of warping the previous image, followed by blending with some background image. This process is accelerated by taking advantage of graphics hardware. The construction of the background images is crucial to obtain a smooth result. All operations are on images, hence we coined the term Image Based Flow Visualization (IBFV) for our method.

In the next section related work is discussed, and in section 3 the method is described and analysed extensively. The implementation and application are presented in section 4, in section 5 the results are discussed. Finally, conclusions are drawn.

2 Related work

Many methods have been developed to visualize flow. Arrow plots are a standard method, but it is hard to reconstruct the flow from discrete samples. Streamlines and advected particles provide more insight. A disadvantage of these techniques is that the user has to decide where to position the startpoints of streamlines and particles, hence important features of the flow can be overlooked.

The visualization community has spent much effort in the development of more effective techniques. Van Wijk [1991] introduced the use of texture to visualize flow. A *spot noise* texture is generated by inserting distorted spots with a random intensity at random locations in the field, resulting in a dense texture that visualizes data. Cabral and Leedom [1993] introduced *Line Integral Convolution* (LIC), which gives a much better image quality. Per pixel a streamline is traced, both upstream and downstream, along this streamline a random noise texture field is sampled and convolved with a filter.

Many extensions to and variations on these original methods, and especially LIC, have been published. The main issue is improvement of the efficiency. Both the standard spot noise and the LIC algorithm use a more or less brute force approach. In spot noise, many spots are required to achieve a dense coverage. In pure LIC, for each pixel a number of points on a streamline (typically 20-50) have to be computed. As a result, the computing time per frame is in the order of tens of seconds. One approach is to develop more efficient algorithms. Stalling and Hege [1995] achieved a higher performance by using a faster numerical method and a more efficient, streamline oriented scheme for the integration.

Another approach to achieve a higher efficiency is to exploit hardware. Firstly, parallel processing can be used [de Leeuw and van Liere 1997; Zöckler et al. 1997; Shen and Kao 1998]. Secondly, graphics hardware can be used. De Leeuw *et al*. [1995] use texture

*e-mail: vanwijk@win.tue.nl

Publication	Equipment	Procs.	fps
Cabral *et al.*[1993]	SGI Indigo	1	0.02-0.05
Stalling and Hege [1995]	SGI Indigo	1	0.23
Max and Becker [1995]	SGI Onyx	1	4.0
De Leeuw *et al.*[1997]	SGI Onyx	8	5.6
Zöckler *et al.*[1997]	Cray T3D	64	20.0
Shen and Kao [1998]	SGI Onyx2	7	0.09
Heidrich *et al.*[1999]	SGI Octane	1	3.3
Jobard *et al.*[2000]	SGI Octane	1	2.5
Jobard *et al.*[2001]	SGI Onyx2	4	4.5
Weiskopf *et al.*[2001]	PC, GeForce3	1	37.0
IBFV, 2002	PC, GeForce2	1	49.3

Table 1: Performance results for texture synthesis in frames per second.

mapped polygons to render the spots. Max and Becker [1995] visualized flow by distorting images. An image is mapped on a rectangular mesh, next either the texture coordinates or the vertex coordinates are changed. The mesh is rendered, and the mesh is distorted further. After a number of frames the distortion of the image accumulates, hence a new image is smoothly blended in, using alpha compositing [Porter and Duff 1984].

Heidrich *et al.* [1999] have presented the first version of LIC accelerated by graphics hardware. The integration of texture coordinates is delegated to the graphics hardware, and indirect addressing on a per-pixel basis via pixel textures [Hansen 1997] is used. They are used to store the velocity field and the texture coordinates. Pixels are updated via backward texture advection: The color of a pixel is found by tracing a virtual particle backwards.

Jobard *et al.* [2000] have further elaborated on the use of pixel textures. They presented a variety of extensions, including the handling of unsteady flow, dye advection and feature extraction. Several problems obtained special attention. Edge effects emerge near inflow boundaries. This is solved by adding new noise via image compositing. White noise is used as background, where per frame 3 % of the pixels are inverted to maintain high spatial frequencies at divergent areas. A sequence of frames (typically 10) is stored. At each time step the oldest frame is subtracted and the newest is added. All in all 18 texture applications per image are used. Recently Jobard *et al.* [2001] presented an alternative approach, which is a combination of a Lagrangian and an Eulerian approach: i.e. a mix of particles and noise. Again, backward texture advection is used. The method is implemented in a parallel fashion on a general purpose computer. Successive frames are blended.

Finally, Weiskopf *et al.* [2001] have used the programmable per-pixel operations of a nVIDIA GeForce 3 card. Backward texture advection is used to show moving particles. Short pathlines can be shown by combining the four last frames.

A summary of the performance results for the synthesis of dense textures for flow visualization reported is given in Table 1. *Procs.* refers to the number of processors used. This table should not be taken too seriously, because the results are not completely comparable. The resolution of the images varies, some consider only steady flow, and the image quality varies, for instance Weiskopf *et al.* produce only animations of moving particles.

Overall, we see that significant progress has been achieved in the synthesis of texture for flow visualization. Unsteady flows can be handled, framerates in the order of 3 to 40 LIC images per second can be achieved. However, to achieve real-time frame rates a supercomputer has to be used, or image quality has to be sacrificed. Framerates in the order of 1 to 6 fps require special features of graphics hardware and/or parallel machines such as an SGI Onyx. For comparison purposes, IBFV refers to the method presented in this article.

IBFV uses advection of images via forward texture mapping on distorted polygonal meshes [Max and Becker 1995], in combination with blending of successive frames [Jobard et al. 2001]. The combination of these two approaches has not been presented before, and we think it is close to optimal. In contrast, the use of graphics hardware (f.i. via pixel textures) to calculate new texture coordinates requires special features of the hardware, introduces quantization problems, and requires interpolation of the velocity field on a rectangular grid. Advection of a flow simulation mesh matches more naturally with higher level graphics primitives, i.e. polygons instead of pixels, in combination with the use of a general purpose processor to perform the necessary calculations.

A major contribution of IBFV is the definition of the background images that are blended in. In other approaches [Jobard et al. 2000; Jobard et al. 2001] much work is spent afterwards to eliminate high frequency components. The approach used here is simpler: If the original images are low pass filtered, then there is no need for such a step afterwards.

3 Method

In this section we start with an informal introduction of IBFV, followed by a more precise description. Various aspects are analysed. In section 4 the model presented here is translated into an algorithm and mapped on graphics primitives. A discussion of the results can be found in section 5.

3.1 Concept

How can we generate a frame sequence that shows an animation of flow? Suppose we already have an image available (fig. 1). Typically, the next frame will look almost the same. If we project the image on, say, a rectangular, mesh, move each mesh point over a short distance according to the local flow and render this distorted image, we get an even better approximation [Max and Becker 1995].

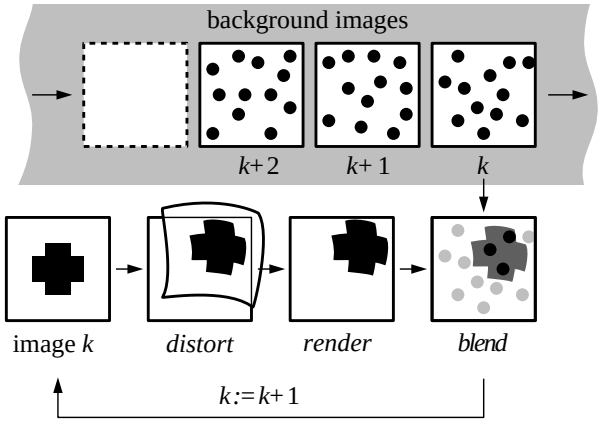

Figure 1: Pipeline image based flow visualization

We could continue this process, but then several problems show up. If we distort the mesh further for the next frames, the accumulating deformation will lead to cells with shapes that are far from rectangular. This problem can easily be solved: We just repeat exactly the same step as before, each time starting with the original mesh.

Nevertheless, the original image will be distorted more and more, and it will disappear from the viewport, advected by the flow. This

problem can be solved too: For each frame we take some new image and blend it with the distorted image [Jobard et al. 2001].

The next question is what new image to take. One could use a sequence of random white noise images, just like in standard LIC algorithms However, this leads to noisy animations. A better solution is to use pink noise, i.e. to remove high frequency components from the background images, both in space and time.

The preceding elements are the basic ingredients of Image Based Flow Visualization. Flow visualization methods like particle and streamline tracking operate in world space, and have geometric objects like points and lines as primitives. The Line Integral Convolution method starts from screen space: For each pixel a streamline is traced. We take complete images as our basic primitive. This leads to straightforward definitions and algorithms, which can be mapped effectively on graphics hardware.

3.2 Image generation

Consider an unsteady two-dimensional vector field $\mathbf{v}(\mathbf{x}; t) \in \mathbb{R}^2$

$$\mathbf{v}(\mathbf{x}; t) = [v_x(x, y; t), v_y(x, y; t)]. \tag{1}$$

We assume it has been defined for $t \geq 0$, and for $\mathbf{x} \in S$, where $S \subset \mathbb{R}^2$. The region S is typically rectangular, but this is not essential. We focus on flow fields, where $\mathbf{v}(\mathbf{x}; t)$ represents a velocity, but other applications, such as potential fields where $\mathbf{v}(\mathbf{x}; t)$ represents a direction and a strength, fall within the scope as well.

A pathline is obtained when we track the position of a particle in a dynamic flow field. A pathline is the solution of the differential equation

$$d\mathbf{p}(t)/dt = \mathbf{v}(\mathbf{p}(t); t), \tag{2}$$

for a given start position $\mathbf{p}(0)$. A streamline has an almost identical definition, except that here the velocity at a fixed time T is considered. For a steady flow field, pathlines and streamlines are the same. A first order approximation of equation (2) gives the well-known Euler integration method:

$$\mathbf{p}_{k+1} = \mathbf{p}_k + \mathbf{v}(\mathbf{p}_k; t)\Delta t, \tag{3}$$

with $k \in \mathbb{N}$ and $t = k\Delta t$. In the remainder we use the frame number k as unit of time.

Suppose we have a field $F(\mathbf{x}; k)$ that represents some property advected by the flow. Here F represents an image, hence $F(\mathbf{x}; k)$ is typically an RGB-triplet. The property is advected just like a particle, so $F(\mathbf{p}(t); t)$ is constant along a pathline $\mathbf{p}(t)$. A first order approximation of the transport of F can hence be given by:

$$F(\mathbf{p}_{k+1}; k+1) = \begin{cases} F(\mathbf{p}_k; k) & \text{if } \mathbf{p}_k \in S \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Here we set $F(\mathbf{p}_{k+1}; k+1)$ to 0 (black) if the velocity at a previous point $\mathbf{p}_k$ is undefined. Sooner or later most of $F(\mathbf{x}; k)$ for $\mathbf{x} \in S$ will be black; for many points $\mathbf{p}_k \in S$ a corresponding start point $\mathbf{p}_0$ will be located outside S. To remedy this, at each time step we take a convex combination of F and another image G. Specifically,

$$F(\mathbf{p}_k; k) = (1-\alpha)F(\mathbf{p}_{k-1}; k-1) + \alpha G(\mathbf{p}_k; k), \tag{5}$$

where the points $\mathbf{p}_k$ are defined by equation (3), and where $\alpha(\mathbf{x}; k) \in [0, 1]$ defines a blending mask. For many applications α will be constant in time and space. Equation (5) is central. It defines the image generation process and forms a startpoint for analysis.

The recurrency in (5) can be eliminated to give

$$F(\mathbf{p}_k; k) = (1-\alpha)^k F(\mathbf{p}_0; 0) + \alpha \sum_{i=0}^{k-1} (1-\alpha)^i G(\mathbf{p}_{k-i}; k-i). \tag{6}$$

The first term represents the influence of the initial picture. This term can be ignored if we either start with a black picture $F(\mathbf{x}; 0)$ or consider a large value for k. Hence we get

$$F(\mathbf{p}_k; k) = \alpha \sum_{i=0}^{k-1} (1-\alpha)^i G(\mathbf{p}_{k-i}; k-i). \tag{7}$$

In other words, the color of a point $\mathbf{p}_k$ of the image is the result of a line integral convolution of a sequence of images $G(\mathbf{x}; i)$ along a pathline through $\mathbf{p}_k$, with an exponential decay convolution filter $\alpha(1-\alpha)^i$. A low α gives a high correlation along a pathline.

This is a Eulerian point of view; We observe what happens at a point. We can also adopt a Lagrangian point of view; Consider what happens when we move along with a particle. Particles here are advected by the flow and fade away exponentially. Jobard *et al.* [2001] made the observation that this can be interpreted physically, and hence gives a natural effect. The particle moves two-dimensionally, and simultaneously sinks away with constant speed in a semi-transparent fluid.

The next question is what images G and masks α to use. We first consider the synthesis of dense textures. A standard way to obtain these is to use random noise. In the next section we discuss spatial constraints on such noise, in section 3.4 we consider temporal characteristics. Two aspects of IBFV that require further attention are edge and contrast problems, which are discussed in section 3.5 and 3.6. In section 3.7 the injection of dye is discussed.

3.3 Noise: spatial

To simplify the following discussion, without loss of generality, we consider a simple steady flow field $\mathbf{v}(\mathbf{x}; t) = [v, 0]$. As a result, pathlines are horizontal lines, and successive points $\mathbf{p}_k$ are spaced $d = v\Delta t$ apart. Furthermore, we consider a pathline through the origin, and assume that F and G are scalar functions (i.e. grey scale images). If we define f and g as

$$f(x; k) = F([x, 0]; k) \tag{8}$$

$$g(x; k) = G([x, 0]; k) \tag{9}$$

then equation (7) reduces to

$$f(x; k) = \alpha \sum_{i=0}^{k-1} (1-\alpha)^i g(x - id; k-i) \tag{10}$$

We consider some special cases for g. First, let's suppose that g is a steady, rectangular pulse with width δ and unit height, i.e.

$$g(x; k) = \begin{cases} 1 & \text{if } |x| \leq \delta/2 \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

The corresponding $f(x; k)$ for large k is a sequence of thin pulses, with exponentially decreasing height, spaced d apart (fig. 2). Obviously, this is not what we want. Discretization of the time dimension gives a sequence of ghost images of the pulse. The desired smooth result is obtained if we set $\delta = v\Delta t$, and let $\Delta t \to 0$

$$f(x) = \alpha(1-\alpha)^{x/d}. \tag{12}$$

What are the requirements on $g(x)$ to obtain such a smooth result? Obviously, this is a sampling problem, which can be studied in the frequency domain. A narrow pulse has a flat, white noise spectrum in its limit. Sampling such a signal gives aliasing artifacts. As an example, if we set $g(x; k) = \cos(2\pi nx/d)$, we get for the steady state $f(x) = \cos(2\pi nx/d)$. Hence, all components with a spatial frequency that is an integral multiple of the sampling frequency are not damped at all. Therefore, to prevent artifacts, g should not have high frequency components.

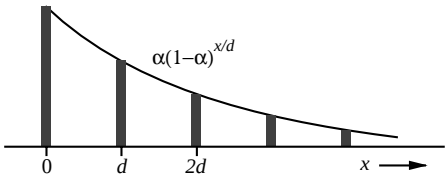

Figure 2: Stepwise advected pulse

We next consider this in the spatial domain. Let the function $h_w(x)$ represent a triangular pulse

$$h_w(x) = \begin{cases} 1 - |x|/w & \text{if } |x| \leq w \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

and let $g(x) = h_w(x)$. For small w, the corresponding $f(x)$ is a sequence of triangles with constant width and exponentially diminishing height. If the triangles overlap, they are added up. How can we get a result that approximates the continuous result of equation (12) satisfactorily? If we require that $f(x)$ should be at least monotonously decreasing, the requirement we get is simply $w \geq d$, or, in general

$$w \geq |\mathbf{v}|\Delta t. \tag{14}$$

This leads to a convenient way to construct low-pass filtered white noise. We use a linearly interpolated sequence of N random values G_i, with $i = 0, 1, \ldots, N-1$, i.e.

$$g(x) = \sum h_s(x - is)G_{i \bmod N}, \tag{15}$$

where the spacing s satisfies $s \geq |\mathbf{v}|\Delta t$. There are two ways to satisfy this constraint. First, a large value for s can be used. Figure 3(b) shows an example. The velocity field is modeled by a source in a linear flow, moving from left to right. Left of the source a saddle-point is located, where the magnitude of the velocity is 0. A stationary black-and-white noise was used, with s set to three pixels (fig. 3(a)). The texture shows both the direction and the magnitude of the local velocity. Concerning the latter: in regions with low velocity the texture is isotropic, whereas it is stretched in high velocity areas. A second way to satisfy the constraint is to use a modified velocity field $\mathbf{v}'(\mathbf{x}; k)$, with

$$\mathbf{v}'(\mathbf{x}; k) = \begin{cases} \mathbf{v}(\mathbf{x}; k)v_{\max}/|\mathbf{v}(\mathbf{x}; k)| & \text{if } |\mathbf{v}(\mathbf{x}; k)| > v_{\max} \\ \mathbf{v}(\mathbf{x}; k) & \text{otherwise} \end{cases} \tag{16}$$

in other words, we clamp the magnitude of the velocitity to an upper limit $v_{\max}$, with $v_{\max}\Delta t \leq s$. If we use a large value of Δt and a low value for $v_{\max}$, most of the velocity field will be clamped, i.e. magnitude information is ignored and only directional information is shown (fig. 3(c)). Or, in LIC terms, the field is sampled with points equidistant in geometric space, whereas in the previous way the points are equidistant in traveling time. In practice a combination of both approaches is most convenient. Fields defined with sinks and sources can locally lead to high velocities, and the use of $\mathbf{v}'$ is a useful safeguard here. Another approach would be to normalize the velocity by dividing through $\max(|\mathbf{v}|)$, but for fields defined with sinks and sources this gives poor results.

If we set $v_{\max}$ to high values, artifacts appear. For fig. 3(d) we set $v_{\max}\Delta t$ to nine pixels, and used a high value for Δt. As a result, components in the background noise with a wavelength of nine, eighteen, etc. pixels show up prominently, leading to intricate knitwear like patterns.

The scale parameter s can be used to influence the style of the visualization. In figure 4 three visualizations of the same field with

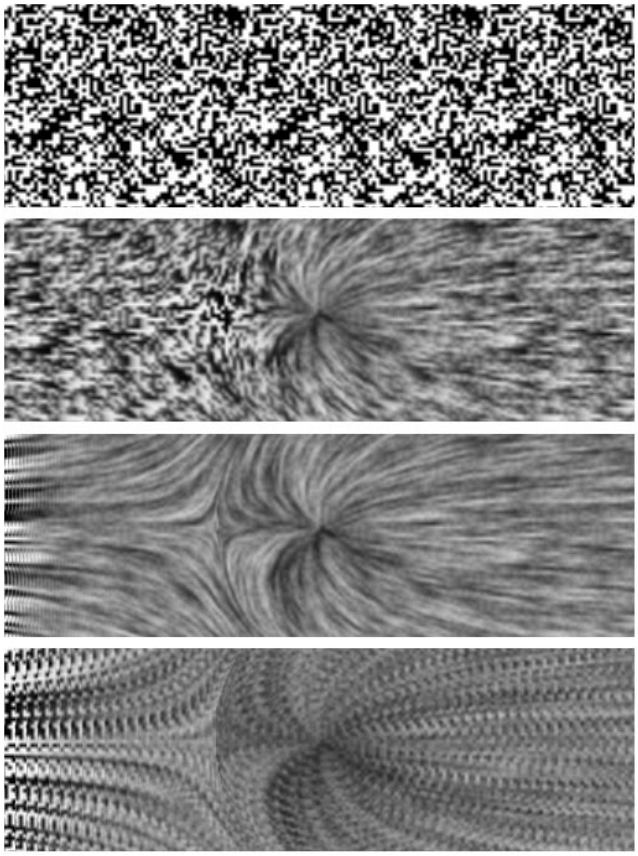

Figure 3: Source in linear flow. From top to bottom: (a) background noise, (b) direction and magnitude, (c) direction only, (d) artifacts

different settings for s are shown. Dependent on the scale, fine LIC-style images, coarser spot noise images, to fuzzy, smoke-like patterns are produced. The latter may seem unattractive, but such patterns give a lively result when used for animations of unsteady flow fields.

3.4 Noise: temporal

In the preceding section $G(\mathbf{x}; k)$ was assumed to be constant in time. This results in static images for steady flow. How can we produce a texture that moves along with the flow? The obvious answer is to vary $G(\mathbf{x}; k)$ as a function of time. We analyse this again using the one-dimensional model. We use a set of M images as background noise, linearly interpolated in space and discrete in time, i.e.

$$g(x; k) = \sum h_s(x - is)G_{i \bmod N; k \bmod M}, \tag{17}$$

with k again representing the frame number. One simple solution would be to use for each frame a different set of random values for G_{ik}. This would spoil the animation however, because the variation along a path line is too strong. In spectral terms, too much high frequency is introduced. Another solution is to produce two random images $G_{i;0}$ and $G_{i;N/2}$, and to derive the other images via linear interpolation, similar to [Max and Becker 1995] . However, this solution lso falls short. This texture is not stationary in time, in the sense that each frame has statistically the same properties. As an example, the variation in magnitude will be higher for $G_{i;0}$ than for $G_{i;N/4}$, because the latter is an average of two samples.

A convenient solution can be derived from the spot noise technique. The intensity of each spot varies while it moves, and to each

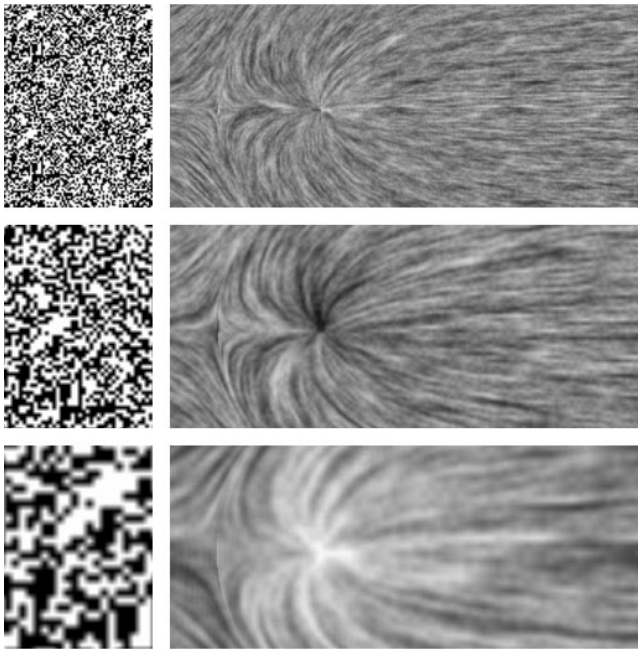

Figure 4: Scaling the texture. From top to bottom: (a) s = 1.5, (b) s = 3.0, (c) s = 6.0

spot a different phase is assigned. Here we do the same: We consider each point G_i of the background texture as a spot or particle that periodically glows up and decays according to some profile $w(t)$. In other words

$$G_{i;k} = w((k/M + \phi_i) \bmod 1) \tag{18}$$

where ϕ_i represents a random phase, drawn uniformly from the interval $[0, 1)$, and where $w(t)$ is defined for $t \in [0, 1]$.

We have experimented with various profiles $w(t)$. The use of $\cos 2\pi x$ (figure 5(a)) gives a result that is too soft and too diffuse, especially when seen in animations. A sharp edge in the profile is needed to generate contrast where the user can focus on. A simple choice is to use a square wave, i.e. $w(t) = 1$ if $t < 1/2$ and 0 elsewhere (figure 5(b)). Advantages are that a high contrast is achieved and crisp images are generated. Let's have a closer look which patterns are generated. Consider a single point, linearly interpolated, which is set to 1 and 0 alternatingly for a number of frames. This produces a dashed line, which fades away in the direction of the flow. And also, each dash itself fades away. This is somewhat unnatural: A standard convention in particle rendering is that the head should be the brightest and that the tail should faint. We can achieve this effect if we use $w(t) = \beta^t$. If we set $\beta < (1 - \alpha)^M$ the pulse falls off faster than the decay, and hence a texture with a decreasing intensity upstream is produced (figure 5(c)). However, the exponential decay has one disadvantage: the dynamic range is not used well during a cycle, leading to a poor contrast. A convenient alternative here is the sawtooth $w(t) = 1 - t$, which produces a similar effect and uses the dynamic range more effectively (figure 5(d)). The effect is strongest for animations, but also in still images the direction of the flow can be discerned.

Finally, we assumed so far that for each new frame a different background image is used, for frame k we used pattern $G(\mathbf{x}; k \bmod \mathrm{N})$. We can also use $G(\mathbf{x}; \lfloor (v_g k) \bmod \mathrm{N} \rfloor)$, where v_g denotes the rate of image change. If N is large enough, different frequencies for background image variation can be modeled without visible artifacts using the same set of background images.

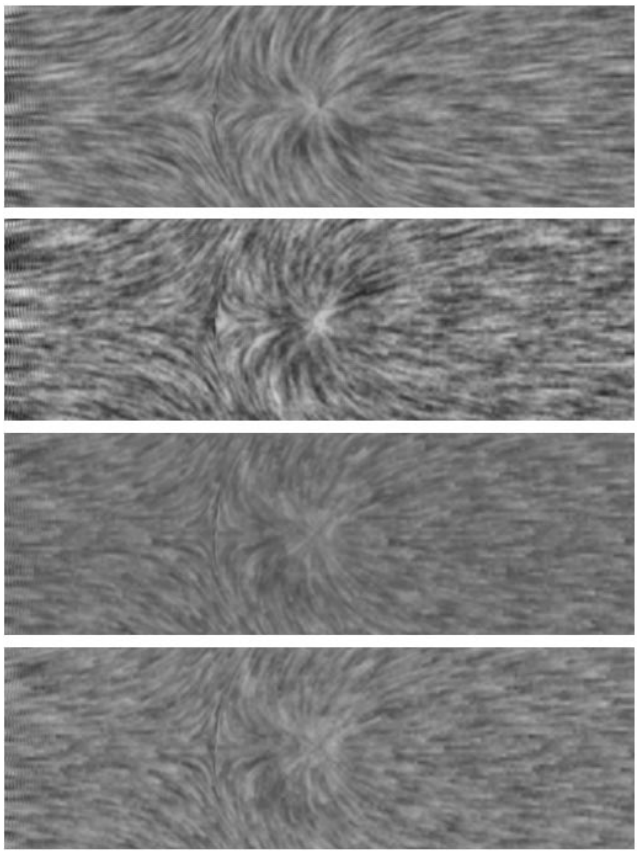

Figure 5: Different profiles $w(t)$. From top to bottom: (a) cosine, (b) square, (c) exponential decay, (d) saw tooth.

3.5 Contrast

One typical problem of flow visualization using texture mapping is the reduced contrast, and, as can be observed from figure 5, the method presented here is no exception. What happens with the contrast? Consider equation (10). The value of f is a sum of scaled samples of g. Let us assume that the values of the samples of g are independent (i.e. spaced s apart). We find then, using standard rules for summation and scaling of a number of independent variates, for the average μ_f and variance σ_f^2:

$$\mu_f = \mu_g \tag{19}$$

$$\sigma_f^2 = \frac{\alpha}{2 - \alpha}\sigma_g^2. \tag{20}$$

Hence, low values for α reduce the contrast significantly. For instance, $\alpha = 0.1$ reduces σ_f with about 77 percent, so a high value for σ_g^2 is welcome. Another result from statistics is that the histogram of the intensities approaches a normal distribution, according to the central limit theorem.

One way to solve this problem is to postprocess the images. We have implemented this: Images can optionally be passed through a histogram equalization. For a sequence of textured images the histogram is fairly stable, so the calculation of the table has to be done only once. The result is however somewhat disappointing: Scaling up the narrow range of intensities gives rise to quantization artifacts. The use of more bits per color component per pixel (eight here) could remedy this. Also, other, more sophisticated techniques, like the removal of low pass components [Shen and Kao 1998] could be used. Most often however, we find that the textures look nice

enough on the screen, especially when they are used as a background for dye-based techniques, and that the reduction in frame rate is not worth the improvement in contrast. For none of the images presented in this paper extra contrast enhancement was used, so that the impact on the contrast can be compared for the various variations.

3.6 Boundary areas

Another issue that deserves extra attention is the treatment of boundary areas. Suppose that the flow domain S originally coincides with the image as viewed on the screen, and let S' be the distorted flow domain (fig. 6):

$$S' = \{\mathbf{x} + \mathbf{v}\Delta t | \mathbf{x} \in S\}. \tag{21}$$

We define the boundary area B as $S - S'$.

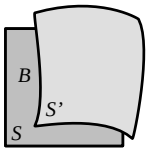

Figure 6: Boundary area $B = S - S'$

The use of eq. (4) and (5) comes down to the following procedure: 1) clear the screen; 2) render the previous image on a warped mesh; 3) blend in a background image. This procedure gives rise to artifacts (fig. 7 (a)). Area B is almost black for low α, and this leaks in from the boundary into the image. A better solution is to use a modified version of eq. (5)

$$F(\mathbf{p}_k; k) = (1-\alpha)F^*(\mathbf{p}_k; k-1) + \alpha G(\mathbf{p}_k) \tag{22}$$

$$\text{with } F^*(\mathbf{p}_k; k-1) = \begin{cases} F(\mathbf{p}_k; k-1) & \text{if } \mathbf{p}_k \in B \\ F(\mathbf{p}_{k-1}; k-1) & \text{otherwise} \end{cases}$$

Stated differently, we just do not clear the screen when we generate a new frame. We can show that now the average value is constant. Equation (22) can be modeled by setting $G(x; t)$ to 0 for $x < 0$. The boundary area B is here the interval $[0, d)$. The equivalent of eq. (7) is then for $k \to \infty$

$$f(x) = \alpha \sum_{i=m}^{\infty} (1-\alpha)^i g(x - md) + \alpha \sum_{i=0}^{m-1} (1-\alpha)^i g(x - id) \tag{23}$$

with $m = \lfloor x/d \rfloor$. The boundary area B is not advected, just constantly blended with new values of g. The average value of f is the same as the average value of g. For the variance σ_f^2 for a constant g in time, we find however

$$\sigma_f^2 = \frac{2(1-\alpha)^{2m+1} + \alpha}{2 - \alpha} \sigma_g^2. \tag{24}$$

In the boundary area the samples of g are all the same, hence for $m = 0$ $\sigma_f^2 = \sigma_g^2$, with increasing m the variance σ_f^2 approaches $\sigma_g^2 \alpha/(2-\alpha)$. In other words, the closer to the inflow boundary, the higher the contrast. Figure 7(b) shows this artifact. A linear flow field was used, $v\Delta t$ was 6 pixels, the texture scale s also to 6 pixels, $\alpha = 0.1$. This is a critical setting, near the boundary aliasing artifacts show up. In practice we use lower values $v_{\max}\Delta t$, typically 2-3 pixels, and a time varying G, and as a result the boundary artifacts are almost invisible. Also, these artifacts can be removed by using a slightly larger image than the image presented on the screen.

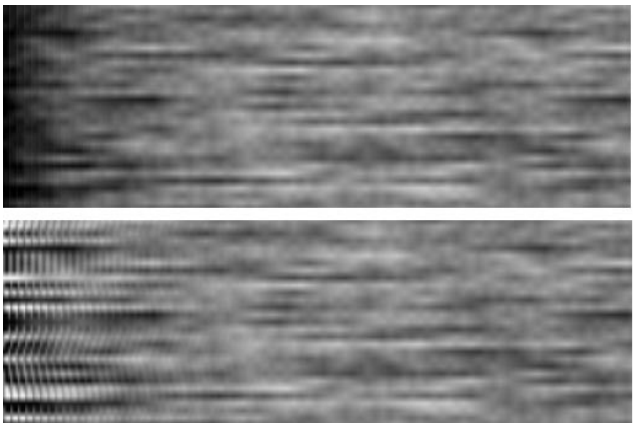

Figure 7: Boundary artifacts: black boundaries (top), increasing contrast towards edge (bottom).

3.7 Dye injection

The injection of dye is a convenient metaphor for many flow visualization methods. Shen *et al.* [1996] and Jobard *et al.* [2000] have already used dye injection in the context of LIC. Dye injection fits well in the model. Injection of dye with color G_D and blending mask α_D to an image F can be modeled as

$$F'(\mathbf{x}; k) = (1 - \alpha_D(\mathbf{x}; k))F(\mathbf{x}; k) + \alpha_D(\mathbf{x}; k)G_D(\mathbf{x}; k) \tag{25}$$

Multiple dye sources can be modeled by repetition of this step. The final modified image $F'(\mathbf{x}; t)$ is next used in the advection step, advecting the injected dye as well. Note that $\alpha_D(\mathbf{x}; k)$ varies with space. If we want to add a circular dye source to the image, we just draw a filled circle in the image, thereby implicitly using a mask that is 1 over the area of the circle and 0 elsewhere. Furthermore, both α and G_D can vary with time. If they are static, the result is a trace of dye, exponentially decaying. By varying them, we get objects that move and distort with the flow. The same profiles $w(t)$ as used for the background noise can be used, where the length of one cycle can be set independently from that of the background noise.

4 Implementation

So far we have presented a model, in this section we consider its implementation. In section 4.1 an algorithm is presented, in section 4.2 an application is described, and section 4.3 gives performance results.

4.1 Algorithm

All aspects of the model can easily be mapped on graphics primitives and graphics operations. The image F is represented as a rectangular array of pixels $F_{ij}, i = 0..N_X - 1, j = 0..N_Y - 1$. We use the framebuffer to store the image. The background images G are represented by a sequence of images G_{kij}, with $k = 0..M-1$ and $i, j = 0..N-1$, filled according to equation (18). Typically $N_X = N_Y = 512$, $N = 64$, and $M = 32$. The background patterns only have to be recalculated if the time profile $w(t)$ has changed, and can be stored in texture memory.

For the advection of the image we use a tesselation of S. For instance a rectangular mesh, represented by an array R_{ij}, $i, j = 0..N_m$, where for each array element the coordinates of the advected grid point are stored. We typically use $N_m = 100$. Other meshes could also be used, triangular meshes for instance lend

themselves also well to fast implementations. If the velocity can be calculated analytically for arbitrary points, the implementor is free to choose the most efficient mesh; If the velocity is given for a mesh already, it is obviously most convenient to use that mesh. Also, time varying meshes can be handled without extra complexity.

The algorithm to generate frame k of an animation now proceeds as follows.

1. If the flow field has changed, calculate a distorted mesh R;
2. Render R on the screen, texture mapped with the previous image;
3. Overlay a rectangle, texture mapped with noise pattern k mod M, blended with a factor α, whereas the image is weighted with $1-\alpha$;
4. Draw dye to be injected;
5. Save the image in texture memory;
6. Draw overlaid graphics.

A few words per step. The calculation of the distorted mesh is done per grid point R_{ij} by calculating a displacement $\mathbf{d} = \mathbf{v}(R_{ij}, k)\Delta t$, which is clamped if $|\mathbf{d}|$ exceeds a threshold, and adding this displacement to R_{ij}. In step 2 the texture coordinates are set evenly distributed between 0 and 1. In step 3 the background noise is blended into the image. The texture coordinates are set such that the spacing of the points of the pattern equals s. Linear interpolation is done by the hardware. In step 4 dye is injected. This can be done in many ways, typically by drawing a shape or overlaying an extra image. In step 5 the result is copied to texture memory ready for the next frame. Finally, overlaid graphics that should not be advected, such as markers and time indicators, are drawn.

One little problem of step 4 and 6 is that when something is drawn in the boundary region B, this is not removed in step 1 for the next frame. As a result, such dye is persistent and leaks into the flow. A simple solution is just not to render additional imagery too close to the boundary of the image or to display the computed image minus its border.

Nearly all steps match very well with graphics hardware. The main action is the rendering of texture mapped quadrilaterals, possibly blended. Modern graphics hardware, such as which can be found in standard PCs nowadays, is made exactly for this purpose, and hence high speeds can be achieved. The algorithm can be implemented using only OpenGL1.1 calls, without extensions, hence the portability is high. As an illustration of the simplicity and portability of IBFV, the accompanying DVD contains a minimal but complete texture based flow visualization demo in the form of about one hundred lines of C-code. This enables the reader to make a quick start and to experiment by varying the parameters and extending visualization options.

The only step that does not take advantage of the graphics hardware is the calculation of the distorted mesh. However, this is a great task for the central processor. These calculations are most easily done in floating point arithmetic, and data must be calculated, retrieved from disk or from a network. The use of graphics hardware for this purpose does not pay off.

4.2 Results

We have implemented IBFV in an interactive flow simulation and visualization system. The application was implemented in Delphi 5, using ObjectPascal. It consists of about 3,500 lines of code, most of which concerned user interfacing. For the modeling of the flow a potential flow model, in line with [Wejchert and Haumann 1991], was used. A flow field is defined by superposition of a linear flow

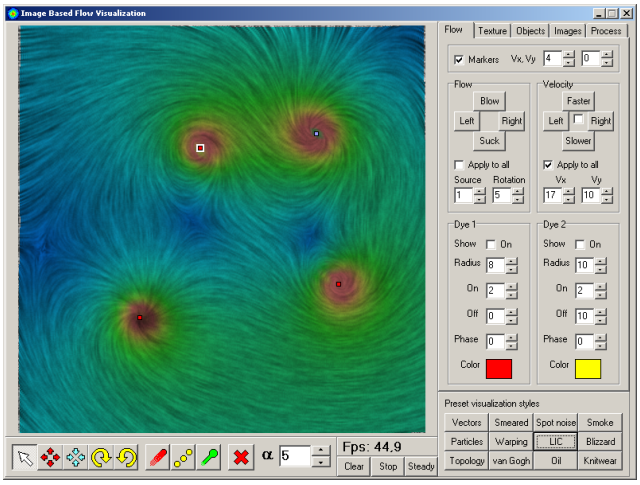

Figure 8: User interface flow modeling and visualization system

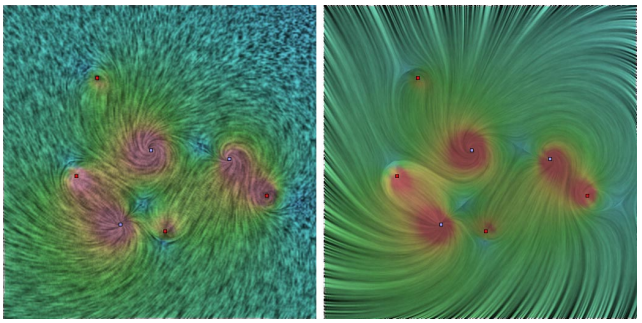

Figure 9: Texture: (a) coarse, (b) fine

field $\mathbf{v}_L$ and a number of fields defined by flow elements. Each flow element has a position $\mathbf{p}_i$, a source strength s_i and a rotation strength r_i as attributes, and contributes

$$\mathbf{v}_i(\mathbf{x};t) = \begin{bmatrix} s_i & -r_i \\ r_i & s_i \end{bmatrix} \frac{\mathbf{d}}{|\mathbf{d}|^2}, \tag{26}$$

with $\mathbf{d} = \mathbf{x} - \mathbf{p}_i$, to the flow field.

Figure 8 shows a screen-shot of the user interface. The user can interactively add and remove flow elements and change their properties. To generate a time dependent flow field, to each flow element a velocity is assigned, such that the flow element moves through the field. Each flow element can optionally produce dye, various attributes of the circles to be drawn can be set. The image is overlaid here with an extra transparent texture, colored according to the magnitude of the velocity, similar to [Jobard et al. 2000]. Many options for visualization were implemented. With different variations of dye injection a wide variety of effects can be achieved. We present these using the same flow field. All images have a 512×512 resolution, for the flow field a 100×100 mesh was used.

Figure 9 shows two visualizations with texture. The scale of the texture can be varied, left we used a coarse texture, on the right a fine texture; by changing Δt and v_{max} either both the magnitude and direction of the velocity (left) or only the direction can be shown. In the latter version saddle points can be located more easily.

A simple model for injecting dye is to use a rectangular grid, possibly jittered, and to draw a circle on the gridpoints, either continuously or intermittently. Figure 10(a) shows that continuous release gives arrow-like glyphs. In figure 10(b) particles are released, using an exponential decay profile. This gives lively animations. Near the

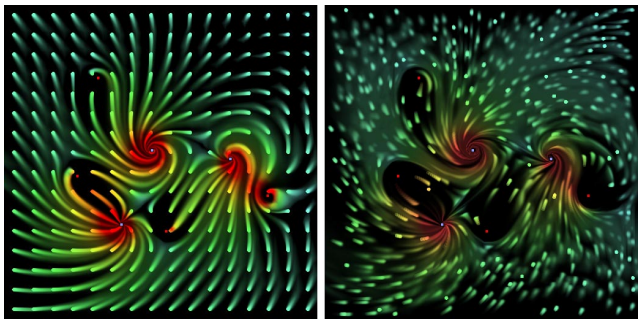

Figure 10: Classic: (a) arrow, (b) particles

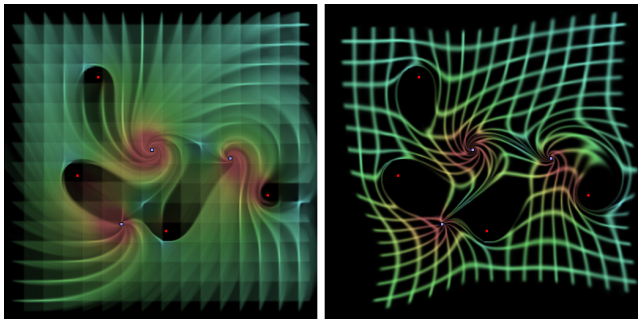

Figure 11: Image distortion: (a) smeared, (b) warped

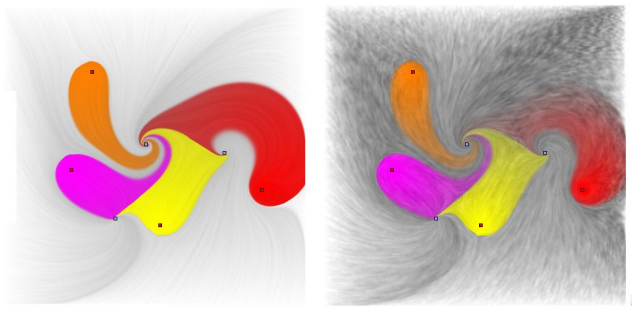

Figure 12: (a) Topology, (b) Topology with texture

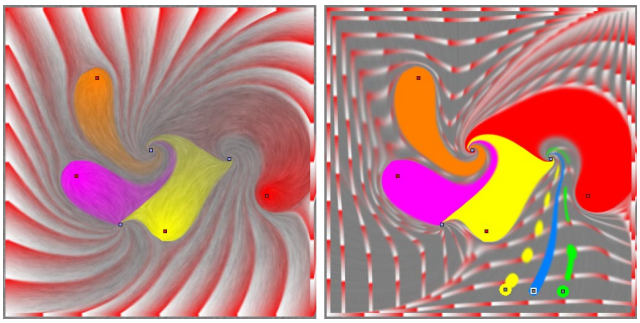

Figure 13: Boundary: (a) streamlines, (b) timelines and dye sources

sources (indicated with a red marker) particles are blown away, in other areas dye accumulates.

Arbitrary images can be inserted in the field. Figure 11 shows what happens if a picture of a grid is smeared (continuous release with low α) or warped (release once, with $\alpha = 1$) by the flow. The method presented here can be used to apply artistic effects on arbitrary images by adding sources, sinks, and vortices, and by overlaying texture.

Topological analysis [Helman and Hesselink 1989] aims at segmenting the flow area into distinct areas, such that flow is constrained within the area. Here, streamlines can start only at the boundary or at sources. If we now paint the boundary and let each source produce a differently colored dye, automatically a topological decomposition is achieved when the dye is advected (fig. 12). For the flow model used here this process can be fully automated, because sources are modeled and known explicitly, for arbitrary flow fields this can be realized by searching for points where the velocity magnitude vanishes. Combination of this technique with a soft texture ($\alpha = 0.01 - 0.02$, fig. 12(b)) gives an especially attractive result. The texture indicates the flow, the color of the area faints with increasing distance from the source or boundary.

Inflow from a boundary can be studied by painting strips along the boundary (figure 13). If a pattern for the boundary is used, streamlines appear; intermittent painting gives so called time lines. Furthermore, on the right image three user positioned dye sources are shown.

We have applied the method to visualize the result of CFD simulations. Figure 14 shows a 2D slice from a 3D time dependent simulation of turbulent flow around a block [Verstappen and Veldman 1998]. A rectilinear 536 $\times$ 312 grid with a strongly varying density was used. This grid could be handled without special problems. We only added an option to zoom and pan on the data set, such that the turbulent flow can be inspected at various scales. In figure 14 we zoom in on a corner of the block, thereby reveiling intricate detail. When sufficiently close (i.e. about 50 $\times$ 50 gridlines in view), zooming and panning on the animated flow can be done smoothly at 30–40 fps.

All applications presented here were easy to implement. All that has to be done is injection of dye, and the remainder is done by the underlying process. Almost no data structures are needed, for instance, there is no need to keep track of the position of advected particles. Another strong effect of real-time animation is that it becomes interesting to make all parameters for dye injection time-dependent. Positions, colors, shapes, can all be defined as functions of time, leading to interesting special effects.

4.3 Performance

The images shown were made on a Dell Inspiron 8100 notebook PC, running Windows 2000, with a Pentium III 866 MHz processor, 256 MB memory, and a nVidia GeForce2Go graphics card with 32 MB memory. The following table shows framerates in fps for four different values of the mesh resolution N_m and for three different configurations. A distinction is made between steady flow (mesh requires no recalculation), one moving flow element (mesh requires recalculation), and seven moving flow elements (the configuration of figure 9 to 12). As test animation a moving black and white texture was used, but there is little difference for the various flow visualization styles. The only exceptions are the use of a transparent velocity overlay or the use of a constant extra image (see figure 11(a)), which

mesh	steady	one source	seven sources
50×50	49.3	49.3	49.3
100×100	49.3	37.0	27.0
200×200	49.3	14.6	9.1
512×512	18.5	2.5	1.4

Table 2: Frames per second, 512×512 images

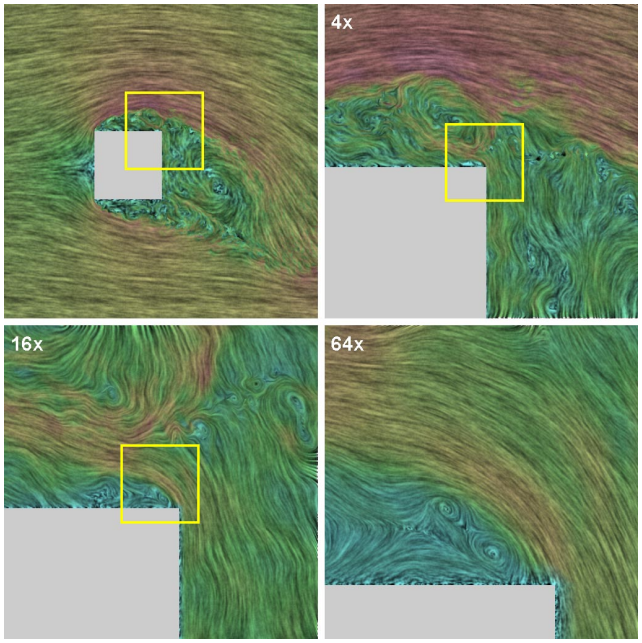

Figure 14: Visualization of turbulent flow simulation data [Verstappen and Veldman 1998] at different scales

cost some 10 to 50% extra, dependent on the resolution of the mesh or image.

What does this table show? Above all, IBFV is *fast*. Unsteady flow is visualized via moving textures in real-time. Parameters can be changed on the fly, the effect of changes is smooth and immediate. The maximum speed of the flow as shown on the screen is typically in the order of 150 pixels per second, hence lively animations result. We have tried our application on several other machines. On a PC running Windows'98 with a 350 MHz processor, 128 MB memory, and a graphics card with a nVidia TNT processor and 16 MB memory we achieved a frame rate of 22 fps for the standard case. In other words, also on somewhat older machines real-time frame rates can be achieved.

In the table from left to right the amount of work for the processor increases with more moving sources. For steady flow the same mesh can be reused, for dynamic flow it has to be recalculated. Dependent on the complexity of the field (one vs. seven sources) the process slows down. From top to bottom the amount of work for the graphics hardware increases with increasing mesh resolution: More coordinates have to be transformed. For steady flow the amount of work for the processor is independent of the mesh resolution, for dynamic flow it depends on the complexity of the field.

The impact of variation of the mesh resolution on the result is limited. Actually, for the fields modeled here no differences can be seen in animations. As an example, figure 15 shows a dye source at a distance of 100 pixels from a vortex. The left image was made using $N_m = 50$, for the right $N_m = 512$ was used. The dye source produces an almost perfect circle. We do see that some diffusion shows up, due to resampling of the distorted mesh on the regular pixel grid of the image. This smoothing has also a very positive aspect: Standard aliasing artifacts such as jagged edges never show up. Anyhow, these images are quite artificial, for texture visualization the visible trace of a particle is much shorter.

We did not perform yet a formal assessment of the accuracy of IBFV, but judging from test images such as fig. 15, it does not seem an urgent problem. The use of a more advanced ODE solver than the very simple Euler scheme used here could give further improvements if necessary. The topology method is interesting from the viewpoint of stability. Standard approaches require careful programming, to make sure that streamlines starting from critical points end up in other critical points. IBFV produces stable results, also for hard cases like a pure vortex. The result here is that the boundary of the vortex area and the remainder of the field is blurred and smooth, which seems to be a natural representation for this situation.

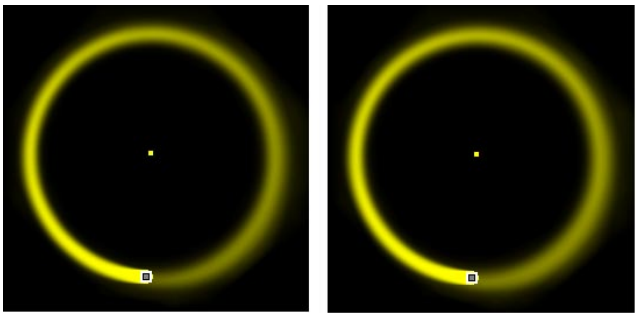

Figure 15: (a) 50×50 mesh, (b) 512×512 mesh

5 Discussion

5.1 Efficiency

IBFV is about one to three orders of magnitude faster than other methods for the synthesis of dense texture animations of flow (see table 1). This efficiency comes from four different effects. Firstly, graphics hardware is used. The warping, blending, and linear interpolation of images is taken care of by the graphics hardware. Communication between graphics hardware and processor is limited to the coordinates of warped meshes, frames are copied directly to texture memory. Secondly, the number of steps in the generation of a single image is limited. Only two full-screen operations are used per frame. Thirdly, frame to frame coherence is exploited. Fourthly, the velocity field can typically be sampled at a lower resolution than the image resolution, where the central processor takes care of velocity field calculations.

We see no simple options to increase the performance further. One limit is the refresh rate of the monitor used. Another limit is the simplicity of the algorithm: The inner loop is almost empty.

In the near future we will experiment with higher image resolutions to see if the same frame rates are achieved. Limitations in texture memory will not be a problem. The total amount of texture memory required is $3N_xN_y$ bytes for the image, and $4N^2M$ bytes for the background patterns. For the set-up used here, in total 1.25 MB is used, which fits well within graphics memory.

5.2 Versatility

IBFV can be used for arbitrary flow fields. So far, we have used IBFV for unsteady flow fields with high vorticity defined for rectangular meshes, but we expect that arbitrary triangular meshes, and even time varying meshes can be handled without special measures.

IBFV can be used to produce a variety of different types of visualizations. Various kinds of texture can be produced. These can be defined with a few parameters: the scale s of the texture, the type of time profile $w(t)$, the image variation speed v_g and the blending factor α. Dependent on these settings, LIC, spot noise, or smoke like textures are produced. Changes in parameter settings can be judged in real-time, hence tuning can be done efficiently. Furthermore, injection of dye and images can be used to generate effects like stream-

lines, vectors, particles, and to analyse the topological structure of the field.

5.3 Simplicity

Maybe the main virtue of IBFV is its simplicity (or elegance?), independent of the point of view. IBFV is based on injection, advection, and decay of dye. It can be described mathematically in a few simple equations. It is based on warping and blending images. It can be coded compactly using a few standard OpenGL calls. This is attractive for application purposes, but also for its analysis. Issues such as contrast and boundary artifacts can be analysed precisely.

5.4 Future work

We have applied IBFV to external two-dimensional CFD datasets. A next step is the use of the method for three-dimensional simulations. Slicing through 3D rectangular data sets does not seem to bring in special problems. Also, generation of texture on arbitrary surfaces in 3D should be feasible. A larger step is to transpose the 2D images to 3D volumes. In [Weiskopf et al. 2001] such an approach is presented for particles. The efficiency will depend critically on the capabilities of the graphics hardware.

Besides flow visualization for scientific purposes, other applications can be foreseen. A system such as presented here is attractive for educational purposes. Games and animations could use IBFV to generate smoke and other special effects. The flow modeling method used here is already effective to create and draw fairly arbitrary flow fields. In combination with a more sophisticated CFD method, such as presented in [Witting 1999], a higher realism can be achieved. The method produces cyclic imagery (after transient effects) for steady flow. These can be used as animated gifs to liven up web pages. The method can also be used for artistic effects on arbitrary images.

Finally, also the method itself can be extended. So far we have dealt with injection of dye, decay of dye and advection. A similar method could be derived to handle anisotropic diffusion.

6 Conclusions

We have presented a simple, efficient, effective, and versatile method for the visualization of two-dimensional flow. Unsteady flow on arbitrary meshes is visualized as animations of texture, vectors, particles, streamlines, and/or advected imagery at fifty frames per second on a PC, using standard features of consumer graphics hardware. As there remains little to be desired, we think we are close to having solved the problem of visualization of two-dimensional flow.

Acknowledgements

I thank Ion Barosan, Alex Telea, Huub van de Wetering, and Frank van Ham (TU/e), for their inspiring and constructive support during the preparation of this paper, and Roel Verstappen, Arthur Veldman (RuG), and Wim de Leeuw (CWI) for providing highly challenging turbulent flow simulation data.

References

CABRAL, B., AND LEEDOM, L. C. 1993. Imaging vector fields using line integral convolution. In *Proceedings of ACM SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, 263–272.

DE LEEUW, W., AND VAN LIERE, R. 1997. Divide and conquer spot noise. In *Proceedings SuperComputing'97*.

DE LEEUW, W., AND VAN WIJK, J. 1995. Enhanced spot noise for vector field visualization. In *Proceedings IEEE Visualization'95*.

HANSEN, P. 1997. Introducing pixel texture. *Developer News*, 23–26. Silicon Graphics Inc.

HEIDRICH, W., WESTERMANN, R., SEIDEL, H.-P., AND ERTL, T. 1999. Applications of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics*, 127–134.

HELMAN, J., AND HESSELINK, L. 1989. Representation and display of vector field topology in fluid flow data sets. *Computer 22*, 8 (August), 27–36.

JOBARD, B., ERLEBACHER, G., AND HUSSAINI, M. 2000. Hardware-accelerated texture advection for unsteady flow visualization. In *Proceedings IEEE Visualization 2000*, 155–162.

JOBARD, B., ERLEBACHER, G., AND HUSSAINI, M. 2001. Lagrangian-eulerian advection for unsteady flow visualization. In *Proceedings IEEE Visualization 2001*, 53–60.

MAX, N., AND BECKER, B. 1995. Flow visualization using moving textures. In *Proceedings of the ICASW/LaRC Symposium on Visualizing Time-Varying Data*, 77–87.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. *Computer Graphics 18*, 253–259. Proceedings SIGGRAPH'84.

SHEN, H.-W., AND KAO, D. L. 1998. A new line integral convolution algorithm for visualizing time-varying flow fields. *IEEE Transactions on Visualization and Computer Graphics 4*, 2, 98–108.

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

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

VAN WIJK, J. 1991. Spot noise: Texture synthesis for data visualization. *Computer Graphics 25*, 309–318. Proceedings ACM SIGGRAPH 91.

VERSTAPPEN, R., AND VELDMAN, A. 1998. Spectro-consistent discretization of Navier-Stokes: a challenge to RANS and LES. *Journal of Engineering Mathematics 34*, 1, 163–179.

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. In *Vision, Modeling, and Visualization VMV '01 Conference Proceedings*, 439–446.

WEJCHERT, J., AND HAUMANN, D. 1991. Animation aerodynamics. *Computer Graphics 25*, 19–22. Proceedings ACM SIGGRAPH 91.

WITTING, P. 1999. Computational fluid dynamics in a traditional animation environment. In *Proceedings of ACM SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 129–136.

ZÖCKLER, M., STALLING, D., AND HEGE, H.-C. 1997. Parallel line integral convolution. *Parallel Computing 23*, 7, 975–989.