Hardware-Accelerated Texture Advection For Unsteady Flow Visualization

Bruno Jobard*, Gordon Erlebacher* and M. Yousuff Hussaini*

School of Computational Science & Information Technology

Abstract

We present a novel hardware-accelerated texture advection algorithm to visualize the motion of two-dimensional unsteady flows. Making use of several proposed extensions to the OpenGL-1.2 specification, we demonstrate animations of over 65,000 particles at 2 frames/sec on an SGI Octane with EMXI graphics. High image quality is achieved by careful attention to edge effects, noise frequency, and image enhancement. We provide a detailed description of the hardware implementation, including temporal and spatial coherence techniques, dye advection techniques, and feature extraction.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Bitmap and framebuffer operations – Display algorithms; I.3.6 [Computer Graphics]: Graphics data structure and data types.

Additional Keywords: unsteady, vector field, pathlines, streak-line, advection, texture, hardware, OpenGL.

1. INTRODUCTION

Traditionally, unsteady flow fields are visualized by time integration of a collection of pathlines that originate from user-defined seed points [12]. Many experimental techniques are based on a combination of pathlines, streaklines, and timelines [2]. It is well known however, that the resulting structures are strongly dependent on the initial seed points. Following this realization, an increasing body of work aims to remove the influence of initial conditions on the final display. This led to a several approaches such as spot noise [6,7,18], LIC [4,14,15], texture advection [13], and most recently, equally-spaced streamlines [10,16], all developed for steady flows. Some of these approaches have recently been extended to the exploration of unsteady vector fields [7,8,14]. Most of these techniques are based on a dense representation of the flow. While the resulting displayed flow fields are more realistic and are to a large degree void of computational artifacts, they are very expensive to compute. The fastest algorithm developed to date for steady flows is fastLIC [15]. Straightforward generalizations of steady algorithms are even more expensive to compute since a large collection of particles is advected at each time step.

As the performance of commodity 3D graphics cards continues to increase at a rate substantially faster than that dictated by Moore's law (factor 2 every 18 months), an increasing fraction of the visual computations will be subsumed by the graphics processor, freeing the central processing unit for other tasks. Many algorithms currently written in software will be re-examined to take advantage of the latest hardware features. These algorithms will, within a couple of years, perform far faster than their older software-based siblings.

The trend towards increased reliance on hardware is clearly demonstrated in the evolution of OpenGL, a graphic standard introduced in 1992. Since then, a large number of extensions have been proposed, and a subset of them adopted. Among the more interesting proposed extensions advanced in 1997 is the notion of a pixel texture [1], a form of indirect addressing that allows many old algorithms to be recast on per pixel basis that were not possible before [9], an enhancement not previously possible.

In this paper, we propose a hardware-accelerated algorithm, based on texture advection, to animate a dense set of particles in two-dimensional unsteady flows. Based on a texture advection scheme, the algorithm is highly accelerated by available hardware features on advanced graphic workstations. Use is made of texture maps, hardware frame buffers, pixel textures, and blending. Several issues that plague texture advection methods are addressed: the treatment of domain boundaries, temporal and spatial correlation, and the loss of high frequency information. The animations simultaneously display velocity direction, velocity magnitude, particle path segments, and the trajectories that result from the continuous injection of colored dye into the flow. A dense collection of over 256×256 particles can be advected at a rate of two frames per second on an SGI Octane with EMXI graphics.

The rest of the paper is organized as follows. Section 2 gives an overview of related work. Texture advection is described in Section 3. The hardware implementation is detailed in Section 4. Several extensions of the algorithms are suggested in Section 5, and finally, we draw some conclusions in Section 6.

2. RELATED WORK

Several techniques have been proposed to produce dense representations of unsteady vector fields. Best known is perhaps UFLIC (Unsteady Flow LIC) developed by Shen [14], based on the Line Integral Convolution (LIC) technique [4]. The algorithm achieves good spatial and temporal correlation although the images are difficult to interpret. The paths are blurred in regions

* 411 Dirac Science Library, Tallahassee, FL, 32306-4120. jobard@csit.fsu.edu, erlebach@csit.fsu.edu, myh@csit.fsu.edu .

of rapid change of direction and are thick where the flow is almost uniform.

The spot noise technique, initially developed for the visualization of steady flowfields, has been extended to unsteady flows [7] by advecting the spots along pathlines. Control of the frame rate is possible by varying the number and size of the spots.

Max and Becker [13] proposed an alternative texture-based algorithm to represent steady and unsteady flow fields. The basic idea is to advect a texture along the flow either by advecting the vertices of a triangular mesh or by integrating the texture coordinates associated with each triangle backward in time. In both cases, if the flow has rotational shear, the advected texture eventually becomes excessively distorted. To counter this distortion, Max periodically reinitializes the texture coordinates to their initial values and blends the texture with a second advecting texture offset by half a period. When texture coordinates leave (or particles enter) the physical domain, an external velocity field is linearly extrapolated from the boundary. Rather than encode the direction of the velocity and its magnitude in a single frame, they are visualized through time animation. This technique attains interactive frame rates by controlling the underlying mesh resolution.

Heidrich *et al.* [9] described the first hardware-accelerated implementation of LIC to depict the directional information of 2D *steady* flow fields. A white noise texture is successively advected along streamlines, forward and backward, to generate N advected textures. When blended together, these textures produce the desired LIC image. Two major contributions of this algorithm are the delegation of the numerical integration of texture coordinates to the graphics hardware and the use of pixel textures to handle indirect addressing on a per-pixel basis. The exclusive use of graphics hardware results in a LIC algorithm that is several times the speed of fastLIC [15].

3. TEXTURE ADVECTION

We are interested in computing the temporal evolution of particles in an Eulerian frame of reference. A fluid particle at position $\mathbf{x}$ and time t is tagged by the value of a function $N(\mathbf{x},t)$, encoded as a two-color noise texture. This particle describes a trajectory $\xi(\mathbf{x},t,\tau)$ as a function of τ, called a pathline. At each point along the pathline, the velocity of the particle is $\mathbf{v}(\xi(\mathbf{x},t,\tau),\tau)$; the trajectory satisfies the evolution equation

$$\frac{d\xi(\mathbf{x},t,\tau)}{d\tau} = \mathbf{v}(\xi(\mathbf{x},t,\tau),\tau) \qquad (1)$$

From (1), if a particle passes through the point $\mathbf{x}$ at time t and the point $\mathbf{x}'$ at time t', the coordinates of these points are related by

$$\mathbf{x} = \mathbf{x}' + \int_{t'}^{t} \mathbf{v}(\xi(\mathbf{x}',t',\tau),\tau)d\tau \qquad (2)$$

Since $N(\mathbf{x},t)$ describes an invariant particle property, it is constant along a pathline:

$$N(\mathbf{x},t) = N(\mathbf{x}',t') \qquad (3)$$

A Taylor expansion of $N(\mathbf{x},t')$ about time t shows that $N(\mathbf{x},t)$ satisfies the advection equation

$$\frac{\partial N}{\partial t} + \mathbf{v}\cdot\nabla N = 0 \qquad (4)$$

valid for both steady and unsteady flows if $N(\mathbf{x},t)$ is a continuous function of $\mathbf{x}$. In this work, we consider discontinuous functions of the spatial coordinates so that (4) cannot be used.

To obtain an advected texture at any time t, equation (3) is solved at the center $\mathbf{x}$ of each texel of the noise texture by determining the property value at a previous location $\mathbf{x}'$ at time t'. This approach is a common starting point for the algorithms of Max [13], Heidrich [9] and the one presented in this paper, all based on texture advection.

The two former methods compute an advected texture at time t from the initial property texture at time $t' = 0$. Consequently, they require either an extrapolation of the vector field outside the physical domain [13] or the limitation of the advection to a few time steps to minimize artifacts in regions of incoming flow [9]. Our algorithm, dependent on a random property function, computes successive textures incrementally and suppresses artifacts by generating random property values for particles that enter the physical domain.

As in [9], our method advects textures on a per-pixel basis rather than on a coarse triangular mesh [13]. It also extends the implementation of Heidrich to track particles in *unsteady* flows over indeterminate time periods. This is made possible by an innovative treatment of incoming particles, compensating for the nonzero divergence of the flow, and a corrective procedure to address the loss of accuracy that results from the discrete nature of the algorithm.

In the following sections, we describe a new algorithm that computes $N(\mathbf{x},t)$ based only on OpenGL routines that directly access the hardware available on an Indigo 2 SGI with a Maximum Impact graphics board or on an SGI Octane with EMXI graphics.

4. HARDWARE IMPLEMENTATION

Our implementation largely capitalizes on new per-pixel operations and other recent OpenGL extensions provided by some SGI graphics boards. The core of the texture advection process relies mainly on two hardware features: 1) additive and subtractive blending between framebuffer content and incoming fragments from textured polygons or pixel arrays, and 2) an indirection operation, called pixel texture, that uses a buffer as a lookup table into a texture.

These hardware operations are further detailed in Section 4.1. Section 4.2 summarizes the different steps of the algorithm before Sections 4.3 through 4.8 describe them in detail.

4.1 Notation

This section introduces a simplified notation that maps to the hardware operations used in this paper. In our algorithm, data is drawn from, read to, and copied between a combination of buffers and textures. During these operations, incoming data can be blended into the destination buffer, colored using per-pixel color tables, and color transformed using color matrices.

Buffers and Textures. The physical variables used are coordinates, velocity, and particle property. They are stored either in a 2D/3D texture or in a 2D hardware RGB framebuffer. However, rather than using an entire visual allocated by the X window system for this purpose, the hardware back and front buffers are divided into several sub-buffers from which data can be read and to which data can be written. In the remainder of the paper, a buffer refers to any subset of a hardware framebuffer used as a storage area. Buffers and textures are denoted by $\mathbf{B}$ and $\mathbf{T}$ respectively with subscripts that characterize their function or content. All buffers and textures have the resolution of the discretized physical domain.

Blending operations. Blending is a per-pixel operation executed when an incoming fragment merges with the corresponding pixel in the destination buffer. Additive (B^+) and subtractive (B^-) blending of a texture $\mathbf{T}$ into a buffer $\mathbf{B}$ are denoted by

$$B^{\pm}(\alpha\,\mathbf{B},\beta\,\mathbf{T}) \quad \text{equivalent to} \quad \mathbf{B} \leftarrow \alpha\,\mathbf{B} \pm \beta\,\mathbf{T} \qquad (5)$$

The first argument of $B^{\pm}$ is always a buffer; the second argument can be a texture, a pixel texture, or another buffer.

Pixel texture. Proposed by SGI in 1997 as an extension to OpenGL [3], pixel textures have been used to advantage in a variety of algorithms ranging from steady-state LIC to a wide range of sophisticated lighting models [9]. Pixel textures allow the projection of a texture onto the framebuffer through the intermediary of a texture coordinate map [3]. Rather than directly affecting the color in the framebuffer (see Figure 1, left), the color components of the incoming fragment are interpreted as texture coordinates. The texel color at these coordinates is then sent to the framebuffer (Figure 1, right). Let $\mathbf{A}$ be an array of pixels and $\mathbf{T}$ be a texture. The action of a pixel texture operation, denoted by $P(\mathbf{A},\mathbf{T})$, can be viewed as the construction of an intermediate array of pixels $\mathbf{T}(\mathbf{A})$, where the RGB components of the pixels in $\mathbf{A}$, acting like texture coordinates (r,s,t), are replaced by the corresponding texel values of $\mathbf{T}$. The resulting pixel array can be stored or blended with the contents of a buffer $\mathbf{B}$. If a pixel array $\mathbf{A}$ is contained in a buffer $\mathbf{B}'$, the composite blending operation is expressed as

$$B^{\pm}(\mathbf{B},P(\mathbf{B}',\mathbf{T})) \qquad (6)$$

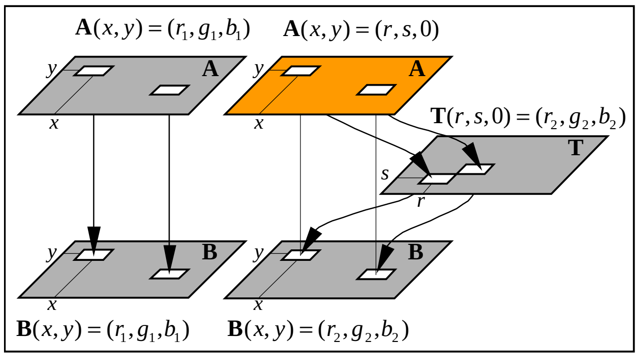

$\mathbf{A}(x,y)=(r_1,g_1,b_1)$
$\mathbf{A}(x,y)=(r,s,0)$
A
A
y
x
$\mathbf{T}(r,s,0)=(r_2,g_2,b_2)$
T
s
r
B
B
$\mathbf{B}(x,y)=(r_1,g_1,b_1)$
$\mathbf{B}(x,y)=(r_2,g_2,b_2)$

Figure 1. Pixel Texture. (left) Color triplets are transferred directly to the framebuffer. (right) When a pixel texture is applied, color triplets are used to address its texels, whose values are sent to the framebuffer.

Read, draw, and copy. A draw operation, denoted by $D(\mathbf{B},\mathbf{T})$, copies the contents of a texture $\mathbf{T}$ into a buffer $\mathbf{B}$. In practice, a polygon, texture-mapped with $\mathbf{T}$, is drawn into $\mathbf{B}$. A read operation, denoted by $R(\mathbf{T},\mathbf{B})$, takes the contents of a buffer, and transfers it to a subset of a texture, called a sub-texture, of equal size. In practice, we use the OpenGL extension glCopyTexSubImageEXT() to directly write to texture memory. Finally, a copy operation from a buffer $\mathbf{B}_1$ to a buffer $\mathbf{B}_2$ is denoted by $C(\mathbf{B}_2,\mathbf{B}_1)$. Although a part of the proposed SGI extensions to OpenGL, the copy operation does not work when the second argument is a pixel texture. In practice, the copy operator is replaced by the combination glReadPixels() and glDrawPixels() at the cost of accessing conventional memory. Both these routines work with pixel textures.

4.2 Algorithm Overview

The first phase of the algorithm implements a hardware version of the advection component described by equations (2) and (3). Hardware buffers and textures are used to encode the particle coordinates, velocity, and property. Components of each pixel in a coordinate buffer, $\mathbf{B}_{\mathbf{x}}$, encode texture coordinates in a noise texture $\mathbf{T}_N$. Initially each pixel in $\mathbf{B}_{\mathbf{x}}$ references its own location and no movement results when $\mathbf{T}_N$ is applied as a pixel texture $\mathbf{T}_N(\mathbf{B}_{\mathbf{x}})$. Adding a contribution of the velocity to every pixel of $\mathbf{B}_{\mathbf{x}}$ forces some of them to reference a neighboring texel in $\mathbf{T}_N$. Now, when $\mathbf{T}_N$ is applied as a pixel texture, property values are displaced, producing an advected version of $\mathbf{T}_N$ (Section 4.3).

Although most texels in the advected texture are assigned with a valid property value during the basic advection phase, supplementary treatments are necessary to correct and enhance the advected texture. Regions of incoming flow are first identified to simulate new particles entering the physical domain (Section 4.4), while the loss of spatial frequency due to the nonzero divergence of the flow is compensated for by a random injection of noise (Section 4.5). The corrected advected texture is then blended with the last blended frame to produce an animation frame with an acceptable level of spatio-temporal correlation (Section 4.6). Finally, the coordinate buffer $\mathbf{B}_{\mathbf{x}}$ is reinitialized in preparation for the next iteration. This initialization takes into account constraints imposed by the discrete nature of the algorithm (Section 4.7). Images are enhanced by additional post processing such as masking and dye advection (Section 4.8).

We will often refer to the different steps of the algorithm. They are numbered and summarized in Table 1 and Figure 2 (see color plate). Table 1 gathers the complete set of operations written using the notation described in Section 4.1, while Figure 2 represents all hardware resources as a list of buffers and textures along with the operations that link them together.

4.3 Basic Advection

In this section, we discuss the hardware implementation of the advection component of the algorithm described by equations (2) and (3). As proposed in [9], the red and blue components of buffers and textures encode both velocity and coordinate data. We store a time series of 2D vector fields, which cover the entire physical domain, in two 3D velocity textures whose third dimension represents time. The velocity components are normalized by the infinity norm over all the field slices. To accommodate the fact that texture values can only be positive, the velocity field is split into its negative and positive components $(\mathbf{v}=\mathbf{v}^+-\mathbf{v}^-)$ and stored in two separate 3D textures, $\mathbf{T}_{\mathbf{v}^-}$ and $\mathbf{T}_{\mathbf{v}^+}$. Furthermore, since the entire 3D vector field, (x,y,t), is often too large to completely reside in texture memory, only two time slices of the

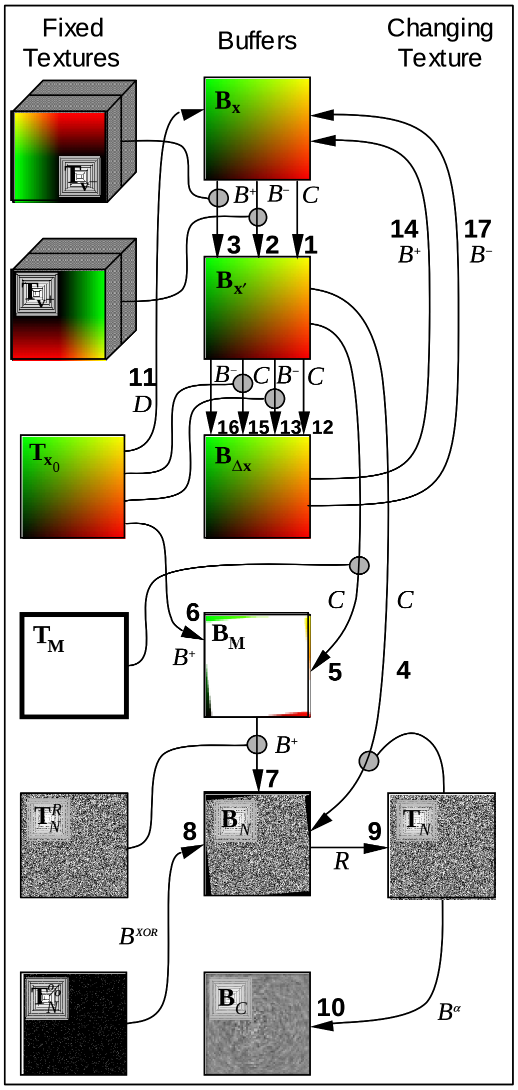


Figure 2. Algorithm for a single time step. The numbers indicate the operation number in the algorithm, which matches the line number in table 1. Pixel texture operations are shown as circles.

velocity texture are stored at any given time. They are updated whenever the current time is outside the range encompassed by these slices.

The texel coordinates of $\mathbf{T}_N$ are initially stored in the texture $\mathbf{T}_{\mathbf{x}_0}$ according to $\mathbf{T}_{\mathbf{x}_0}(i,j)=(i+0.5,j+0.5)/N$ where N is the texture size. Each texel of $\mathbf{T}_{\mathbf{x}_0}$ references the center of the corresponding texel in $\mathbf{T}_N$.

Coordinate update. Texture coordinates at time $t' = t - h$ are computed from a first order discretization of (2):

$$\mathbf{x}' = \mathbf{x} - h[\mathbf{v}^+(\mathbf{x},t) - \mathbf{v}^-(\mathbf{x},t)] \qquad (7)$$

Two buffers, $\mathbf{B}_\mathbf{x}$ and $\mathbf{B}_{\mathbf{x}'}$, are used for this operation. $\mathbf{B}_\mathbf{x}$, which initially contains the initial texture coordinates stored in $\mathbf{T}_{\mathbf{x}_0}$, is blended with texels from the velocity textures, and the result is stored in $\mathbf{B}_{\mathbf{x}'}$ (steps 1-3 in table 1 and Figure 2).

Since each velocity component is the range $[0.,1.]$, h is related to the maximum possible displacement p (in pixels) of a particle between two consecutive positions by $h = p/N$. To achieve a sufficient degree of spatio-temporal correlation during an animation sequence h must be sufficiently small. We find that $p \in [0.5,3]$ yields good results.

Noise update. The second part of the advection process (step 4) computes $N(\mathbf{x}',t)$ using the pixel texture $P(\mathbf{B}_{\mathbf{x}'},\mathbf{T}_N)$. In principle, any texture can be used for the advection. However, we use a noise texture for its lack of spatial correlation. This property is a necessary requirement for the treatment of particles entering the physical domain (Section 4.4), noise injection (Section 4.5), and noise blending (Section 4.6).

With the $\mathbf{x}$ buffer reinitialized between successive iterations, first four steps implement a basic texture advection. However, several issues must be addressed to correct and enhance the advected textures. They are explained in the following sections.

4.4 Edge Correction

A common problem with texture advection techniques is the inadequate treatment of particles that originate from outside the physical domain [13]. A proper treatment of edge effects requires that these particles be identified and new property values assigned to them without introducing extraneous visual artifacts. We capitalize on the OpenGL property that states that before storage into a buffer (or into a texture), floating point color values are clamped to the range $[0,1]$. Whatever the particle referenced outside the domain, its coordinates reference an edge texel. As an illustration, Figure 3 (a,b) shows the black and white striations, which result from the clamp operation, on a circular flow defined by $(u,v)=(-y,x)$. In this example, a particle at (x,y) originates from

$$\begin{cases} x' = x + hy \\ y' = y - hx \end{cases}$$

All particles with constant x' and $y' > 1$ acquire the value of the noise texture at $(x',1)$. These particles lie on a straight line with positive slope h, clearly seen in Figure 3 (a,b).

For clarity of exposition, let S_B be the set of pixels in $\mathbf{B}_{\mathbf{x}'}$ that reference a point on the boundary. We seek to replace pixels in S_B by a new random noise. This is achieved by the composition of two images I_1 and I_2 through an additive blend. The first

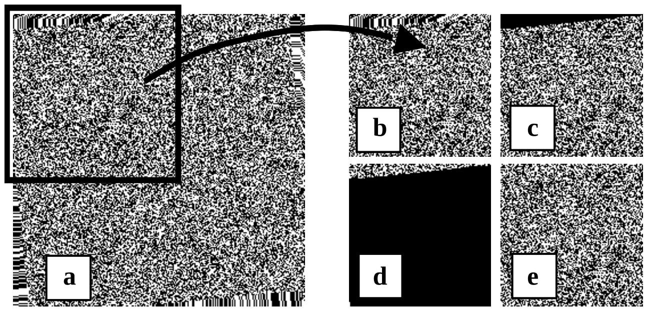


Figure 3. (a),(b) noise texture advected by a circular flow $(u,v)=(-y,x)$. (c) regions from particles exterior to the domain at the previous step are black (I_1). (d) new noise is injected into the edge region; complement region is black (I_2). (e) composite of I_1 and I_2.

```
    Initialize coordinates $\mathbf{T}_{\mathbf{x}_0}$ ; $D(\mathbf{B}_\mathbf{x},\mathbf{T}_{\mathbf{x}_0})$
    Initialize noise $\mathbf{T}_N$ with black border
    $t = 0$
    while $(t < t_{\max})$
      -------------------------- Basic Advection
1     $C(\mathbf{B}_{\mathbf{x}'},\mathbf{B}_\mathbf{x})$
2     $B^-(\mathbf{B}_{\mathbf{x}'},h.P(\mathbf{B}_\mathbf{x},\mathbf{T}_{\mathbf{v}^+}[t]))$
3     $B^+(\mathbf{B}_{\mathbf{x}'},h.P(\mathbf{B}_\mathbf{x},\mathbf{T}_{\mathbf{v}^-}[t]))$
4     $C(\mathbf{B}_N,P(\mathbf{B}_{\mathbf{x}'},\mathbf{T}_N))$
      -------------------------- Edge Correction
5     $C(\mathbf{B}_M,P(\mathbf{B}_{\mathbf{x}'},\mathbf{T}_M))$
6     $B^+(\mathbf{B}_M,\mathbf{T}_{\mathbf{x}_0})$
7     $B^+(\mathbf{B}_N,P(\mathbf{B}_M,\mathbf{T}_N^R))$
      -------------------------- Noise Injection
8     $B^{XOR}(\mathbf{B}_N,\mathbf{T}_N^{\%})$
      -------------------- Store Corrected Noise
      Draw black border around $\mathbf{B}_N$
9     $R(\mathbf{T}_N,\mathbf{B}_N)$
      --------------------------- Noise Blending
10    $B^\alpha(\mathbf{B}_C,\mathbf{T}_N)$
      ------------------------ Fractional update
11    $D(\mathbf{B}_\mathbf{x},\mathbf{T}_{\mathbf{x}_0})$
12    $C(\mathbf{B}_{\Delta\mathbf{x}},\mathbf{B}_{\mathbf{x}'})$
13    $B^-(\mathbf{B}_{\Delta\mathbf{x}},P(\mathbf{B}_{\mathbf{x}'},\mathbf{T}_{\mathbf{x}_0}))$
14    $B^+(\mathbf{B}_\mathbf{x},\mathbf{B}_{\Delta\mathbf{x}})$
15    $C(\mathbf{B}_{\Delta\mathbf{x}},P(\mathbf{B}_{\mathbf{x}'},\mathbf{T}_{\mathbf{x}_0}))$
16    $B^-(\mathbf{B}_{\Delta\mathbf{x}},\mathbf{B}_{\mathbf{x}'})$
17    $B^-(\mathbf{B}_\mathbf{x},\mathbf{B}_{\Delta\mathbf{x}})$
    $t = t + h$
```

Table 1. Hardware algorithm for one time step

image, I_1, is black if $\mathbf{x} \in S_B$ and is given by $\mathbf{T}_N$ elsewhere. The second image, I_2, has new noise in S_B, and is black elsewhere. Adding the two images produces a noise texture with new values only on S_B. There are no visible artifacts at the juncture between the images since we use a spatially uncorrelated noise. Next, we describe the construction of these two images.

As seen above, the spurious streaks at the edges of the domain take on the color of some boundary texel. We capitalize on this property by placing a one texel wide black border along the perimeter of $\mathbf{T}_N$ (initially and between steps 8 and 9). Consequently, the output of step 4 is a noise buffer that contains I_1. The second image is constructed with the help of a white *mask* texture $\mathbf{T}_M$ whose border texels are black. The pixel texture operation $P(\mathbf{B}_{\mathbf{x}'},\mathbf{T}_M)$ results in an intermediate black and white image whose black texels lie in S_B. This texture is then copied into a mask buffer $\mathbf{B}_M$ (step 5). The next two steps draw noise from $\mathbf{T}_N^R$ into S_B. Step 6 adds the initial coordinate texture $\mathbf{T}_{\mathbf{x}_0}$ to the mask buffer. Color clamping insures that the white pixels of $\mathbf{B}_M$ remain white. However, the black pixels in $\mathbf{B}_M$ acquire the color components associated with $\mathbf{T}_{\mathbf{x}_0}$. $\mathbf{T}_N^R$ is a 3D texture with two layers in the third dimension. The bottom layer has random noise; the top layer is black at coordinate (1,1,1). Thus, where the mask buffer is white, the color of $P(\mathbf{B}_M,\mathbf{T}_N^R)$ is black. At black texels, the mask buffer has the original coordinate values and the pixel texture returns a random noise. Finally, in step 7, the image I_2 output by the pixel texture is added to I_1, stored in the noise buffer generated in step 4. To insure that the new noise generated in step 7 has no temporal correlation $\mathbf{T}_{\mathbf{x}_0}$ is randomly translated at each iteration. This is accomplished using a texture transformation matrix.

4.5 Noise Injection

In regions of positive flow divergence, adjacent pixels in $\mathbf{B}_{\mathbf{x}'}$ that reference the same texel location in $\mathbf{T}_N$ after the backward integration step share the same color. Therefore, the overall frequency of the successive noise textures decreases. Figure 4 clearly demonstrates this decrease for a source flow after several time steps. To maintain a constant noise frequency, a small amount of new noise is injected into $\mathbf{B}_N$ at every iteration (step 8). Through experimentation, we found that randomly inverting the color of two to three percent of the noise texels at each time step is enough to maintain a high frequency noise that is approximately constant without a significant loss of temporal correlation.

In practice, an invariant black texture with a 2-3 percent random distribution of white texels, $\mathbf{T}_N^{\%}$, is XORed into $\mathbf{B}_N$ with an OpenGL blending mode. The injection process affects a different set of texels at each time step by applying a random texture translation matrix to $\mathbf{T}_N^{\%}$. The content of the noise buffer is read back into the noise texture $\mathbf{T}_N$ in step 9.

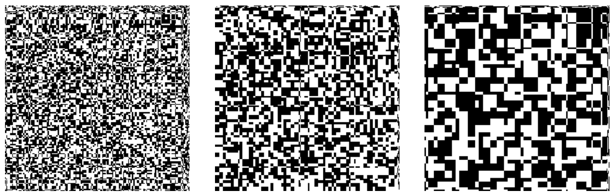


Figure 4. Noise texture advected by a source (the worst case scenario) with constant divergence: $(u,v) = (x,y)$. Notice the gaps of increasing size that result from the constant divergence of the particle paths.

4.6 Noise Blending

We introduce an acceptable level of spatial and temporal correlation into each frame by applying a one-sided exponential filter to the sequence of frames. This effect is implemented with standard alpha blending (step 10):

$$\mathbf{B}_C = (1-\alpha)\mathbf{B}_C + \alpha\mathbf{T}_N$$

The use of noise textures implies that the only spatial correlation after filtering is along a pathline segment. Besides smoothing the animation, the blending process adds directional information to static frames, a feature not present in [13] for example. A two-color black and white noise maximizes the contrast of the final blended image. Good visual results are obtained with $\alpha = 0.1$. The image in $\mathbf{B}_C$ can be saved as a final animation frame or be used for further image enhancements (Section 4.8).

4.7 Coordinate Reinitialization

During the texture advection phase, the coordinate buffer $\mathbf{B}_{\mathbf{x}'}$ was updated to reference the location of incoming particle properties and a new noise texture was computed from the advection of the current noise texture. The coordinate buffers must now be reinitialized in preparation for the next iteration, taking into account certain constraints imposed by the discrete nature of the algorithm.

The displacement of the particle property between successive frames must be small enough to maintain a good spatio-temporal correlation. However, if the displacement of a particle is such that both old and new positions lie within the same pixel, the updated noise texel remains unchanged. Even worse, once the coordinates are reinitialized to their initial values (stored in $\mathbf{T}_{\mathbf{x}_0}$) in step 11, any subpixel displacement (also called fractional displacement) is lost and cannot be recovered: the motion of the particle property is suppressed (step 5).

The above discussion suggests that the fractional displacements of particles be accumulated, and the noise texture be updated, once the accumulated displacement exceeds the width w_p of a pixel. The distance from $\mathbf{x}_0$ to $\mathbf{x}'$ is the sum of an integer displacement vector

$$\mathbf{n}(\mathbf{x}'-\mathbf{x}_0) = (\text{int})[(\mathbf{x}'-\mathbf{x}_0)w_p^{-1}]\,, \qquad (8)$$

whose components are each an integral number of pixel widths, and a fractional displacement vector

$$\boldsymbol{\beta}(\mathbf{x}'-\mathbf{x}_0) = (\mathbf{x}'-\mathbf{x}_0) - \mathbf{n}(\mathbf{x}'-\mathbf{x}_0)\,w_p\,,$$

whose components each have a magnitude less than w_p. If the fractional displacements were neglected (omit steps 12-17), $\mathbf{B}_{\mathbf{x}}$ would receive $\mathbf{x}_0$ in step 11. The goal of the additional steps 12 through 17 is to extract $\boldsymbol{\beta}(\mathbf{x}'-\mathbf{x}_0)$ from $\mathbf{B}_{\mathbf{x}'}$ and store $\mathbf{x}_0 + \boldsymbol{\beta}(\mathbf{x}'-\mathbf{x}_0)$ into $\mathbf{B}_{\mathbf{x}}$ in preparation for the next iteration.

Figure 5 shows the effect of the proposed correction on a circular flow. The basic advection algorithm (left) leaves points of low velocity fixed in space as evidenced by the lack of blending in the central regions. In addition, dye injected into the flow slowly drifts to the center as inaccuracies accumulate in time. The corrected version is seen on the right.

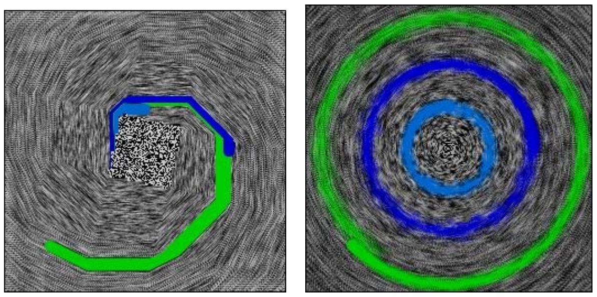


Figure 5. Dye advection in a circular flow defined by the circular flow $(u,v)=(-y,x)$. Left: fractional coordinate correction is disabled. Right: fractional correction is enabled.

4.8 Image Enhancement

We propose two techniques to augment the information content of the animations. First, the brightness of regions of low velocity is reduced to better identify strong currents in the flow. Second, colored dye is introduced into the flow to visualize streaklines.

4.8.1 Velocity Mask

Figure 6 shows a single frame of wind patterns over Europe [17]. High and low velocity regions are discerned thanks to the spatial correlation of the velocity field. However, the high frequency noise associated with regions of low velocity detracts the user from regions of interest related to higher-speed currents. To increase the contrast between these regions, we subtract the linear function $f_{\mathbf{v}} = 1 - \|\mathbf{v}\|$ from $\mathbf{B}_C$ and store the result in the final display buffer $\mathbf{B}_{C'}$. For this purpose, we store $f_{\mathbf{v}}$ in the blue component of the negative velocity texture during the initialization phase. At each iteration, the corresponding time slice of $\mathbf{T}_{\mathbf{v}^-}$ is mapped into a temporary buffer $\mathbf{B}_{\mathbf{v}}$. The blue component of $\mathbf{B}_{\mathbf{v}}$ is duplicated into the red and green components through the intermediary of a color matrix and subtracted from $\mathbf{B}_{C'}$. These operations are expressed as $C(\mathbf{B}_{C'},\mathbf{B}_C)$, $D(\mathbf{B}_{\mathbf{v}},\mathbf{T}_{\mathbf{v}^-})$, and $B^-(\mathbf{B}_{C'},\mathbf{B}_{\mathbf{v}})$. The velocity mask is applied between steps 10 and 11. A larger class of functions can be constructed with color maps to implement more general feature enhancements or feature extractions.

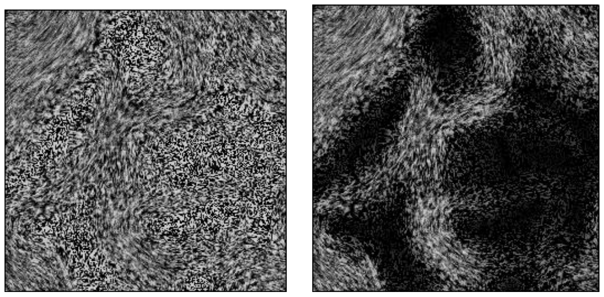


Figure 6. Winds over Europe. Regions of low velocity included (left) and excluded (right).

4.8.2 Dye Advection

Experimentalists have long tracked tracer particles, dye, and smoke to help understand the structure of unsteady flows [2]. In our implementation of dye advection, the advected texture acts like a physical surface upon which dye is released. Dye is introduced into the flow between steps 8 and 9 by drawing geometric primitives (dots, lines, etc.) into the noise buffer $\mathbf{B}_N$. The algorithm then automatically advects the dye at no additional cost. In the final image, the dye is automatically subject to a temporal convolution of successive frames for increased smoothness. The dye is stored in the green and blue texture components while the noise is stored in the red component. Thus, multiple colored streaks can be tracked. Figures 5, 7, and 8 were produced in this way.

It is well known that in unsteady flows, streaklines, streamlines, and particle paths are different from one another. We test the validity of the dye advection algorithm on the uniformly rotating uniform flow $(u,v)=(\cos(t),\sin(t))$. In this flow, streamlines are straight, while streaklines and particle paths have circular trajectories. Figure 7 shows two frames of an animation in which dye is released at three points in the flow. As expected, the streaklines are circular.

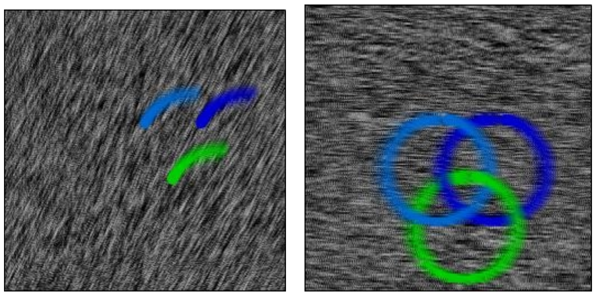


Figure 7. Three streaklines in the rotating uniform flow $(u,v) = (\cos(t), \sin(t))$ (see [5]).

5. DISCUSSION

The use of graphics hardware implies some constraints and restrictions due to the limited number of bits available to encode data.

Resolution limitation. The depth of the framebuffer has a direct impact on the spatial and temporal accuracy of the texture advection. In particular, it affects the number of bits that encode the fractional part of the coordinate displacement in the coordinate buffers. Figure 5 shows the extreme case when no bits are available for the fractional part. Through experimentation, we found that four bits are necessary for good visual results. Under this constraint, a visual of 12 bits per color component only provides 8 bits to encode texture coordinates, which then limits the advection to 256×256 textures. We have since addressed this limitation using a tiling algorithm [11].

Hardware resources. Hardware buffers and texture memory are limited resources. Our algorithm uses seven textures (see

Figure 2). Using internal texture formats available in OpenGL that require the minimum amount of memory, the advection of 256 by 256 textures consumes less than 1.2 Mbytes of texture memory. In practice, the incomplete implementation of the pixel texture extension on SGI graphic boards requires that the textures mapped by this operation be 3D and RGBA (see manual page for glPixelTexGenSGIX). This limitation increases the required texture memory to 2.3 Mbytes, which can still reside in the four Mbyte texture memory of the Octane.

Multiple buffers are stored in each hardware framebuffer. They are arranged in a single hardware buffer as a non-overlapping array of 3 by 2 buffers of size $N \times N$. We used the fact that $\mathbf{B}_{\Delta\mathbf{x}}$, $\mathbf{B}_M$, can share in turn the same space without conflict to reduce the required number of stored buffers from seven to six. 256 pixels wide buffers are easily accommodated.

6. CONCLUSION

This paper describes the first complete hardware-accelerated implementation of an algorithm to visualize unsteady flow based on a per-texel advection technique. It can simultaneously display velocity direction, velocity magnitude, and dye advection. A major advantage of this system is its ability to interactively compute long animation sequences.

We solved intrinsic problems that plague texture advection algorithms, particularly when they are applied to time-dependent data over extended periods:

- Incoming flow regions are handled with uncorrelated noise textures and image compositing.
- Long time advection is achieved through a restoration of the texture frequency at each time step without significant loss of temporal correlation.
- Spatio-temporal correlation is enhanced by applying a temporal filter on advected textures. As an additional bonus, flow direction is available in static frames.

We demonstrated how to use masks to control the regions of interest through the intermediary of a control function stored in the velocity texture. Finally, we capitalized on the possibility of long time integration to transport dye and visualize streaklines. Figure 8 displays three frames of a 1000 frame animation of unsteady wind patterns over Europe using all the above. Dye is released both from a point, and from a line segment. Regions of rotation are easily discerned, along with the regions of high velocity.

This paper further demonstrates the usefulness of new hardware capabilities and advanced graphic functionality, such as the SGI pixel texture extension. Interactive frame rates are achieved with buffer sizes of 256×256 and over 65,000 individual particles. The small texture size led to a novel algorithm for long time advection. At present, only the Maximum Impact and the Octane have the required hardware in their graphics engines to implement the algorithm. However, these features deserve to be incorporated into a wider class of machines. Each 256×256 frame takes 0.4 seconds to compute on an Octane with EMXI graphics. Although 18 texture applications per step are required, we expect the algorithm to be increasingly superior to the best software implementations. On the other hand, the precise control afforded by a software implementation will most probably lead to higher quality images.

7. ACKNOWLEDGMENTS

We would like to thank David Banks for lively discussions in all areas of visualization, including several valuable suggestions to improve the quality of this paper, and van Liere for the use of the wind data from CWI. We also appreciate the comments of the reviewers, which led to improved clarity of exposition. We acknowledge the support of NSF under grant NSF-9872140.

8. REFERENCES

[1] OpenGL ARB. OpenGL Specification, Version 1.2, 1998.

[2] *Visualized flow : Fluid Motion in Basic and Engineering Situations Revealed by Flow Visualization*, Pergamon Press, 1988.

[3] Advanced Graphics Programming Techniques Using OpenGL. SIGGRAPH '98 Course, http://www.sgi.com/-software/opengl/advanced98/notes/notes.html, 1998.

[4] B. Cabral and L. C. Leedom. Imaging Vector Fields Using Line Integral Convolution. *Computer Graphics Proceedings*. In James T. Kajiya, editor, Annual Conference Series, 263-272, ACM, August 1993. ISBN 0-201-58889-7.

[5] W. C. de Leeuw and R. van Liere. Comparing LIC and Spot Noise. *IEEE Visualization '98.* In David Ebert, Hans Hagen, and Holly Rushmeier, editors, pages 359-366, ISBN 0-8186-9176-X.

[6] W. C. de Leeuw and R. van Liere. Enhanced Spot Noise for Vector Field Visualization. *Proceedings Visualization '95.* In D. Silver and G. M. Nielson, editors, 359-366, IEEE Computer Society Press, 1995.

[7] W. C. de Leeuw and R. van Liere. Spotting Structure in Complex Time Dependent Flow. Technical Report CWI - Centrum voor Wiskunde en Informatica, Technical Report SEN-R9823, September 1998.

[8] L. K. Forssell and S. D. Cohen. Using Line Integral Convolution for Flow Visualization: Curvilinear Grids, Variable-Speed Animation, and Unsteady Flows. *IEEE Transactions on Visualization and Computer Graphics*, 1(2) , 133-141, June 1995.

[9] W. Heidrich, R. Westermann, H.-P. Seidel, and T. Ertl. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. *ACM Symposium on Interactive 3D Graphics.* 127-134, ACM, April 1999.

[10] B. Jobard and W. Lefer. Creating Evenly-Spaced Streamlines of Arbitrary Density. *Visualization in Scientific Computing.* In W. Lefer and M. Grave, editors, pages 43-56, Springer Verlag, 1997.

[11] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Tiled Hardware-Accelerated Texture Advection for Unsteady Flow Visualization. *Graphicon 2000.*

[12] D. A. Lane. Visualizing Time-Varying Phenomena In Numerical Simulations Of Unsteady Flows NASA Ames Research Center, Visualizing Time-Varying Phenomena In Numerical Simulations Of Unsteady Flows NAS-96-001, February 1996.

[13] N. Max and B. Becker. Flow visualization using moving textures. *Proceedings of ICASE/LaRC Symposium on Visualizing Time Varying Data.* In David C. Banks, Tom W. Crockett, and Stacy Kathy, editors, NASA Conference Publication, 3321, 77-87, 1996.

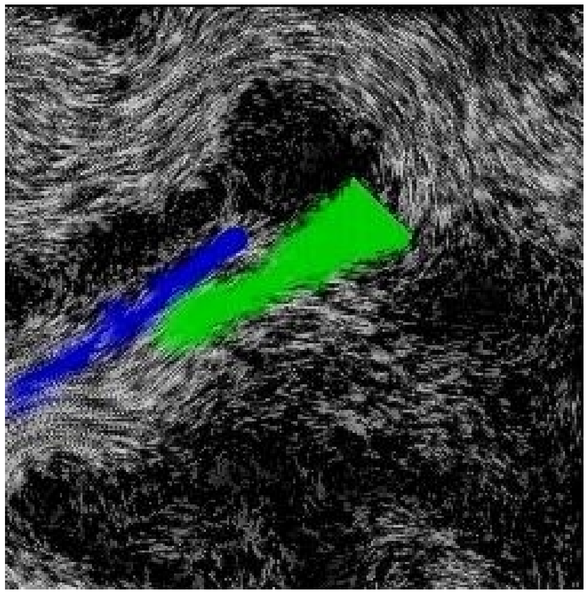
[14] H.-W. Shen and D. L. Kao. A New Line Integral Convolution Algorithm for Visualizing Time-Varying Flow Fields. *IEEE Transactions on Visualization and Computer Graphics*, 4(2), 98-108, 1998.

[15] D. Stalling and H.-C. Hege. Fast and Resolution Independent Line Integral Convolution. *ACM SIGGRAPH Computer Graphics Proceedings.* Annual Conference Series, 249-256, 1995.

[16] G. Turk and D. Banks. Image-Guided Streamline Placement. *Proceedings of SIGGRAPH 96.* In Holly Rushmeier, editor, Computer Graphics Proceedings, Annual Conference Series, 453-460, ACM SIGGRAPH, ISBN 0-201-94800-1.

[17] Wind field over Europe Compiled by: van Liere, R., CWI, Amsterdam, the Netherlands.

[18] J. J. van Wijk, Spot Noise-Texture Synthesis for Data Visualization. *Computer Graphics (Proceedings of* SIGGRAPH *91)*, vol. 25, 309-318, July, 1991.



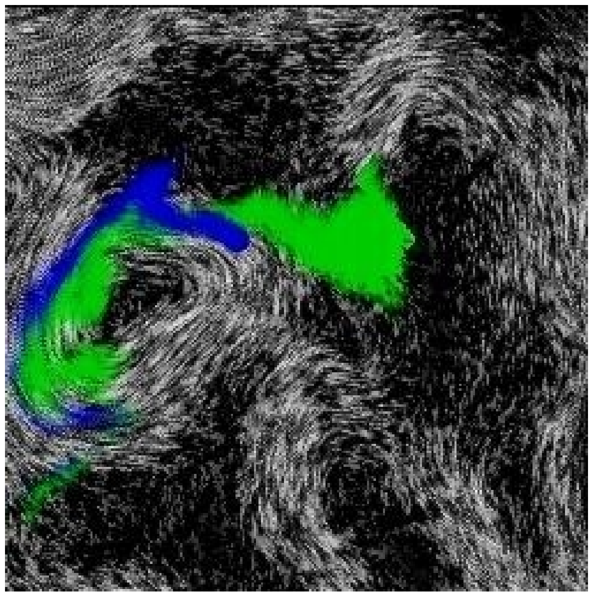


Figure 8. Two frames from a 1000 frame animation (computed in 11 minutes) sequence of wind currents over Europe [17]. Note the lack of artifacts at the domain boundaries. Dye is released from a point and from a line segment. Vortical patterns are evident in the lower frame.