Tiled Hardware-Accelerated Texture Advection for Unsteady Flow Visualization

Bruno Jobard, Gordon Erlebacher, and M. Yousuff Hussaini

School of Computational Science and Information Technology
Florida State University, USA

Abstract

Particle tracking in liquid and gaseous fluids is a very useful technique to better understand flow dynamics. In this paper, we develop a novel algorithm to track a dense collection of particles in unsteady two-dimensional flows. To capitalize on the rapid improvements in graphic hardware technology, the algorithm is exclusively based on subsets of the OpenGL implementation of SGI. Making use of several proposed extensions to the OpenGL-1.2 specification, animations of over 3×10^5 are produced at one frame/sec. on an SGI Octane with EMXI graphics. These animations are based on a multidomain implementation that overcomes hardware limitations such as insufficient memory in the hardware frame buffers to store temporary arrays and the low number of bits per color channel in buffers and textures. 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, multidomain tiling, and the effect of hardware constraints on the precision of the algorithm.

Keywords: unsteady, vector fields, multidomain, hardware, advection.

1. Introduction

Traditionally, unsteady flow fields are visualized by time integration of a collection of pathlines that originate from user-defined seed points. However, dense representations of the flow provide the highest information content [3,5,8,9]. While the resulting flow fields are more realistic, they are very expensive to compute.

As the performance of commodity 3D graphics cards continues to increase at a rate substantially faster than that dictated by Moore's law (doubling of CPU performance every 18 months), an increasing fraction of the visual computations will be subsumed by the graphics processor. New hardware capabilities of the next generation of graphics cards are leading to a new class of visualization algorithms [6,7]. They will within a few years outperform the current algorithms. 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 algorithms to operate on a per pixel basis that were not previously possible [6].

In this paper, we propose a tiled hardware-accelerated algorithm, based on texture advection, to animate a dense set of particles in two-dimensional unsteady flows. The algorithm is equivalent to a texture advection scheme applied to every pixel and highly accelerated by available hardware features on advanced graphic workstations. The algorithm makes use of texture maps, hardware framebuffers, pixel textures, color matrices, and blending. Seamless animations are achieved through a careful treatment of inflow and outflow boundaries, temporal and spatial correlation, and loss of high frequency information. To handle increased domain sizes, the size of several intermediate temporary buffers is decreased and the physical domain is tiled with a collection of overlapping textures. The basic algorithm described in [7] is applied in each domain with special treatment at the domain edges. Particles are represented by a two-color noise distribution, stored in a texture.

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 algorithm is examined in Section 4, which addresses notation, an overview of the algorithm, data structures, and the algorithm itself. Constraints imposed by the hardware are addressed in Section 5, followed by concluding remarks in Section 6.

2. Related Work

Several techniques have been advanced to produce dense representations of unsteady vector fields. Best known is perhaps UFLIC (Unsteady Flow LIC) developed by Shen [9], and based on the Line Integral Convolution (LIC) technique [2]. The spot noise technique, initially developed for the visualization of steady flowfields, has been extended to unsteady flows [3]. Advection of the spots along pathlines produces animations of unsteady flows. Max and Becker [8] propose 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. When texture coordinates or particles leave the physical domain, an external velocity field is linearly extrapolated from the boundary. This technique attains interactive frame rates by controlling the resolution of the underlying mesh. Heidrich *et al.* [6] describe the first hardware-accelerated implementation of LIC to depict the directional information of 2D steady flow fields. A white noise texture is advected along streamlines, forward and backward, to generate several 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.

3. Problem Formulation

We are interested in computing the temporal evolution of particles in an Eulerian framework. 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:

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

Equation (3) is the departure point for the algorithms of Max [8] and the one presented in this paper, both based on texture advection. Max computes the particle property at $\mathbf{x}$ by backward integration to $t=0$. As stated, the method does not work for unsteady flows and he has proposed alternative formulations [8]. Instead, we compute the particle property given the flow state at any previous time. Another distinction between the two algorithms is that Max advects the texture coordinates on a coarse triangular mesh while we advect every texel independently.

In the sections that follow, 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.

The basic advection algorithm follows the scheme proposed by Heidrich *et al.* [6]. Particle coordinates, stored in the red and green color components of a framebuffer, are blended with velocity textures to implement a first order discretization of (2). A pixel texture is then applied to these new coordinates to advect the property texture N computed at the previous time step.

We extend this implementation through several innovative steps to treat edge effects, to compensate for the nonzero divergence of the flow, and to address the loss of accuracy that results from the discrete nature of the algorithm. Moreover, we extend the maximum size of each animation frame from 256×256 [7] to an arbitrary size using texture tiling.

In the next section, we describe the notation used in the remainder of the paper. Section 4.2 summarizes the algorithm, Section 4.3 enumerates the required data structures, and Sections 4.4 and 4.5 describe the inner and outer algorithms.

4.1 Notation

This section describes 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 and velocity. They are stored either in a 2D/3D texture or in a 2D hardware RGB framebuffer. However, rather than using an entire buffer allocated by the X window visual 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. Finally, the depth of a buffer or texture refers to the number of bits in a single color component, unless otherwise noted.

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

$$B^{\pm}(\alpha\,\mathbf{B},\beta\,\mathbf{T}) \quad \text{equivalent to} \quad \mathbf{B} \leftarrow \alpha\,\mathbf{B} \pm \beta\,\mathbf{T} \qquad (4)$$

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 [1], pixel textures have been used to advantage in a variety of algorithms ranging from steady-state LIC to a wide

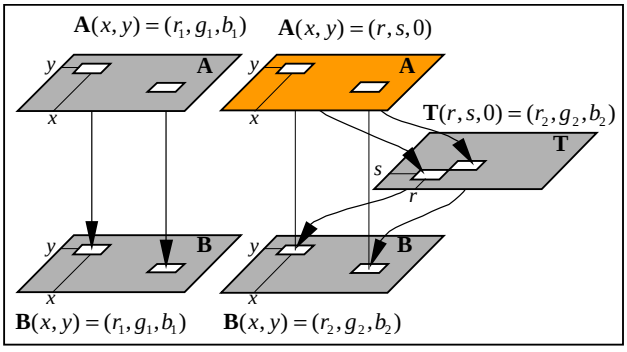

Figure 1. Pixel transfer: (left) A color triplet is transferred directory to the framebuffer. (right) Under the action of a pixel texture, a color triplet is used to address a texel, whose value is then sent to the frame buffer.

range of sophisticated lighting models [6]. Pixel textures allow the projection of a texture onto the framebuffer through the intermediary of a texture coordinate map [1]. Rather than directly affecting the color in the framebuffer (see Figure 1, right), 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, left). 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

$$B^{\pm}(\mathbf{B},P(\mathbf{B}',\mathbf{T})) \qquad (5)$$

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

Jobard *et al.* [7] described a hardware-based algorithm that computes the advection of a two-dimensional dense representation of particles in an Eulerian framework. Their algorithm consists of the following steps: noise advection, edge correction, fractional update, noise injection, and noise blending. All the calculations are performed in a physical domain of 256×256 pixels. As larger domains are considered, e.g. 512×512, the algorithm suffers from a lack of real estate in the available hardware-enabled buffers. It can no longer be applied. As an alternative, we break up the physical domain into a series of overlapping tiles (Figure 2). The main obstacle to overcome is the treatment of particles that advect across adjacent tiles.

The algorithm is composed of an inner and an outer part. The inner component operates successively on each subdomain. A noise is first advected on each tile. Simultaneously, the particles that originate from outside the tile are identified, and referenced through a mask. In the outer algorithm, these masks are used to reconstruct a global seamless texture. This global texture is then corrected to suppress artifacts caused by particles entering the physical domain, and for the effects of flow divergence. The rebuilt and corrected noise is then divided into tiles that serve as input textures for the next inner iteration. The global noise texture is finally blended with previous global textures to generate animation frames with appropriate degree of spatio-temporal correlation.

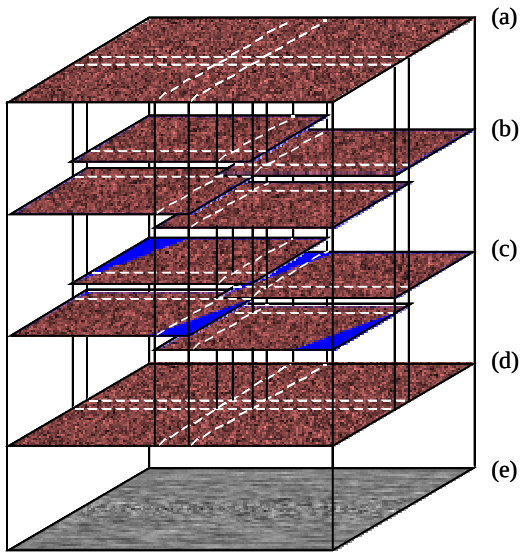

Figure 2. Tiling approach. (a) An initial noise texture $\mathbf{T}$ is (b) subdivided in smaller textures with an overlap region. (c) Individual sub-textures are advected and (d) are composited to form the advected version of $\mathbf{T}$. (e) Consecutive advected textures are blended together.

4.3 Data Structures

The algorithm makes use of textures and buffers of different sizes. The next two sections consider them in turn.

4.3.1 Textures

The algorithm manipulates textures of three different sizes: global textures that cover the entire physical domain, tile textures that store information from each subdomain, and velocity textures that contain the velocity field. Their respective sizes are denoted by the subscripts l (large or global), s (small or tile), and v (velocity). The overall width, height, and depth of textures are powers of two and are identified with the uppercase letters W, H, and D respectively. We define the useful part of a texture as the subregion that contains the data. Its dimensions are denoted by the lowercase letters $w\,(\le W)$, $h\,(\le H)$, and $d\,(\le D)$. Uppercase dimensions are the smallest power of two greater than the associated lowercase dimensions. Uppercase (A) and lowercase (a) letters are related by

$$A = 2^{\lceil \log_2(a) \rceil}$$

The physical domain is w_l pixels wide and h_l pixels high and is covered by a global texture of dimension $W_l \times H_l$. Global textures include $\mathbf{T}_N$ (two-color noise texture), $\mathbf{T}_N^{\%}$ (a texture with 2-3 percent random noise), and $\mathbf{T}_N^R$ (a texture with random uniform noise). The physical domain is evenly divided into n_x tiles, along the x axis, and n_y tiles along the y axis. All the subdomains of dimension $w_s \times h_s$ are stored in a collection of textures of size $W_s \times H_s$. Tile textures are labeled by a superscript ij to indicate their position along the x and y axes. Adjacent subdomains overlap by c pixels along both axes. The width of the overlap region is at least the maximum distance traveled by a particle in a single iteration. The tile textures used

are $\mathbf{T}_{\mathbf{x}_0}$ (initial coordinates), $\mathbf{T}_{\mathbf{x}}^{ij}$ (coordinates in subdomain ij), and $\mathbf{T}_T^{ij}$ (noise in subdomain ij).

Finally, a 2D vector field defined on a $w_v \times h_v$ Cartesian grid at d_v time levels is stored in two 3D textures $(\mathbf{T}_{\mathbf{v}^-}, \mathbf{T}_{\mathbf{v}^+})$ of size $W_v \times H_v \times D_v$.

4.3.2 Buffers

Buffers come in two flavors: large and small. There are two large buffers. $\mathbf{B}_T$ (stored in the front buffer) contains the current advected noise that characterizes individual particles, while $\mathbf{B}_C$ (stored in the back buffer) contains the blended noise that encodes the spatio-temporal correlation of the particles. Four smaller buffers, defined by the size of a subdomain, are stored in the back buffer and share space with $\mathbf{B}_C$. These buffers are $\mathbf{B}_{\mathbf{x}}$, $\mathbf{B}_{\mathbf{x}'}$, $\mathbf{B}_{\Delta\mathbf{x}}$, and $\mathbf{B}_N$. They are used in the inner algorithm to update $\mathbf{T}_N^{ij}$, the noise texture inside a given subdomain. Their individual roles are described in Section 4.4.1.

Given a physical domain defined with $w_l \times h_l$ pixels, a tile size of at most $w \times h$ pixels, and a subdomain overlap of c pixels, the minimum number of buffers required in the x direction is

$$n_x = \left\lceil \frac{w_l - c}{w - c} \right\rceil \tag{6}$$

with a similar formula for n_y in the y direction. One easily shows that the minimum possible size for the tile is

$$w_s = \left\lfloor \frac{w_l + c(n_x - 1)}{n_x} \right\rfloor$$

4.4 Inner Algorithm

Each tile is processed sequentially to produce a collection of coordinate and noise textures ($\mathbf{T}_{\mathbf{x}}^{ij}$ and $\mathbf{T}_N^{ij}$). In succession, the particles are advected (Section 4.4.1), the tile edges are handled (Section 4.4.2), and corrections are introduced to offset the low number of available bits in the framebuffer (Section 4.4.3).

4.4.1 Advection

A series of 2D time-dependent vector fields that cover the entire physical domain are stored in 3D velocity textures whose third dimension represents time. The two velocity components $u(t)$ and $v(t)$ are stored in the red and green color components of the texture after normalization by

$$V_{\max} = \max_t(|u(t)|, |v(t)|)$$

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 $\mathbf{T}_{\mathbf{v}^+}$ [6]. Furthermore, since the entire 3D (x, y, t) vector field is often too large to completely reside in texture memory, only two time slices of the velocity texture are stored at any given time. They are updated any time the current time is outside the range encompassed by these slices.

The buffers $\mathbf{B}_{\mathbf{x}}$, $\mathbf{B}_{\mathbf{x}'}$, $\mathbf{B}_{\Delta\mathbf{x}}$ and the textures $\mathbf{T}_{\mathbf{x}_0}$, $\mathbf{T}_{\mathbf{x}}^{ij}$ store x and y coordinate values in their red and green color components. Among them, only $\mathbf{T}_{\mathbf{x}_0}$ and $\mathbf{T}_{\mathbf{x}}^{ij}$ are explicitly initialized. The size of their useful region is $w_s \times h_s$. Anchored at the lower left corner of the texture, the initial values associated with each texel reflect their relative location within the physical domain:

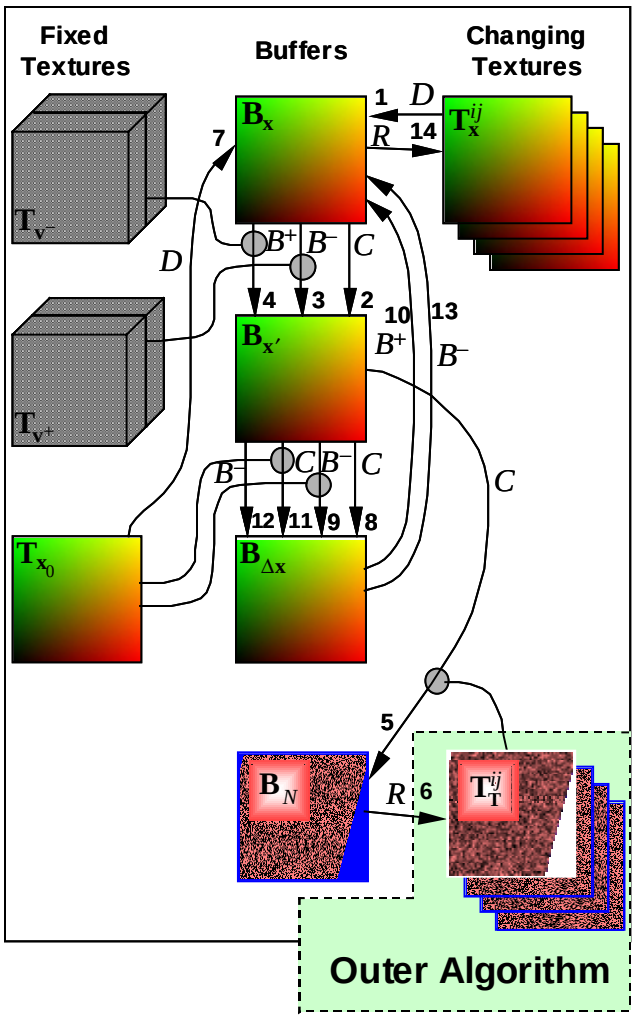

Figure 3. Flowchart of inner algorithm.

$$\mathbf{T}_{\mathbf{x}_0}(x, y) = \mathbf{T}_{\mathbf{x}}^{ij}(x, y) = \left(\frac{x + 0.5}{m_s}, \frac{y + 0.5}{m_s} \right)$$

where $m_s = \max(w_s, h_s)$, $x \in \{0,1,...,w_s - 1\}$, $y \in \{0,1,...,h_s - 1\}$, $i \in \{0,1,...,n_x - 1\}$, $j \in \{0,1,...,n_y - 1\}$.

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

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

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

In term of graphic hardware operators, $\mathbf{B}_{\mathbf{x}}$, which initially contains the initial particle locations stored in $\mathbf{T}_{\mathbf{x}}^{ij}$, is blended with texels from the velocity textures, and the result is stored in $\mathbf{B}_{\mathbf{x}'}$ (steps 2-4 in Table 1 and Figure 3). In the pixel textures of

```
    Initialize T_x0 and T_x^ij
    Generate global noise texture T_N
    Divide T_N into subdomain noise textures T_N^ij
    Insert blue frame one pixel wide around T_N^ij
    t = 0
    while (t < t_max)
      // Inner Algorithm
      for j = 0 to n_y
        for i = 0 to n_x
1         D(B_x, T_x^ij)
2         C(B_x', B_x)
3         B^-(B_x', hP(B_x, T_v+[t]))
4         B^+(B_x', hP(B_x, T_v-[t]))
5         C(B_N, P(B_x', T_N^ij))
6         R(T_N^ij, C^RGB1→RG0B B_N)
          // Fractional update
7         D(B_x, T_x0)
8         C(B_Δx, B_x')
9         B^-(B_Δx, P(B_x', T_x0))
10        B^+(B_x, B_Δx)
11        C(B_Δx, P(B_x', T_x0))
12        B^-(B_Δx, B_x')
13        B^-(B_x, B_Δx)
14        R(T_x^ij, B_x)
      // Outer Algorithm
15    D(B_T, T_N^R)
16    B_ij^α(B_T, T_N^ij)
17    B^XOR(B_T, T_N^%)
18    R_ij(T_N^ij, B_T)
      Insert blue frame one pixel wide around T_N^ij
19    R(T_N, B_T)
20    B^α(B_C, C^RGB1→RRR1 T_N)
      t = t + h
```

Table 1. Algorithm for unsteady flow advection on tiled domains.

steps 3 and 4, the (R,G,B) components of $\mathbf{B}_{\mathbf{x}}$ are interpreted as texture coordinates into $\mathbf{T}_{\mathbf{v}^-}$ and $\mathbf{T}_{\mathbf{v}^+}$. Red and green values range from 0.0 to w_s/m_s and h_s/m_s respectively; blue values are set to zero. The active region of the velocity textures have coordinates that range from 0.0 to w_v/W_v, h_v/H_v, and d_v/D_v respectively. Scale and bias operations are used to map the subregion of the velocity texture corresponding to the current tile into $\mathbf{B}_{\mathbf{x}}$. The use of 3D textures lets the graphics hardware interpolate the velocity field in space and time. The velocity is finally scaled by the integration step size h and added to $\mathbf{B}_{\mathbf{x}}$ using blending operations via a call to the OpenGL extension routine *glBlendColorEXT()*.

Noise update. The second part of the advection process (steps 5,6) computes $N(\mathbf{x}',t)$ in each subdomain using the pixel texture $P(\mathbf{B}_{\mathbf{x}'},\mathbf{T}_N^{ij})$ and stores the result in $\mathbf{T}_N^{ij}$. Scaling factors $s_{red}=m_s/W_s$ and $s_{green}=m_s/H_s$ applied to the colors stored in $\mathbf{B}_{\mathbf{x}'}$ serve to reference the useful part of $\mathbf{T}_N^{ij}$. Although any texture can be used for the advection, we choose a two-color noise texture for its lack of spatial correlation. This property is also required for the treatment of domain boundary effects (Section 4.4.2), noise injection (Section 4.5.3), and noise blending (Section 4.5.4).

The re-initialization of the coordinate buffer to $\mathbf{T}_{\mathbf{x}_0}$ in step 7 completes the translation of (3) into hardware-based operations and implements a basic texture advection. However, several issues must be addressed to correct and enhance the advected textures. They are addressed in the following sections.

4.4.2 Edge Correction

A common problem with texture advection techniques is the inadequate treatment of particles that originate from outside the physical domain [8]. 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 $[0,1]$. Consequently, particles that enter the current tile are pixels in $\mathbf{B}_{\mathbf{x}'}$ that have at least one of their red and green components set to zero or one; they reference tile boundaries. For clarity of exposition, let S_B be the set of pixels in $\mathbf{B}_{\mathbf{x}'}$ that reference a point on the boundary, and let S_I be the remaining set of pixels in $\mathbf{B}_{\mathbf{x}'}$. By construction, all the particles in S_I originate from within the interior of the physical domain. We seek to replace pixels in S_B by a new random two-color noise.

To achieve this, we store in a single RGBA texture, both the advected noise, and a mask that defines S_I. The noise is stored in the red color component of the RGBA texture and is mapped to a black and white noise during the noise-blending phase (steps 19,20). The mask is built during the noise advection phase (steps 1-6) by making the pixels in S_B transparent in the noise texture $\mathbf{T}_N^{ij}$. By placing a one pixel wide blue border around $\mathbf{T}_N^{ij}$ (as an initial condition and between steps 18 and 19), all pixels in S_B acquire a blue component during step 5. While reading the noise buffer into $\mathbf{T}_N^{ij}$, the blue channel is transferred to the alpha channel using a color matrix transformation $C^{RGB1\to RG0B}$ in step 6. Values of zero and one in the alpha channel are finally exchanged by applying a scale and bias of –1 and 1 respectively. At this point, $\mathbf{T}_N^{ij}$ is transparent in S_B (shown as a white region on $\mathbf{T}_N^{ij}$ in Figure 3 and 5) and opaque at all other locations. In the outer algorithm, the partially transparent noise textures $\mathbf{T}_N^{ij}$ are used to reconstruct a global noise texture where only texels that correspond to pixels in S_I are visible (Section 4.5.3).

4.4.3 Coordinate Re-initialization

During the texture advection phase, the coordinate buffer $\mathbf{B}_{\mathbf{x}'}$ was updated to reference the location of incoming particles and a new noise texture was computed from the advection of the

current noise texture. The subdomain 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 particles 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 7, any subpixel displacement (also called fractional displacement) is lost and cannot be recovered: the motion of the particle is suppressed (step 5). This is very well illustrated in the central region of the circular flow shown in Figure 4 (left).

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 8-13), $\mathbf{T}_{\mathbf{x}}^{ij}$ would receive $\mathbf{x}_0$ in step 14. The goal of the additional steps 8 through 13 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 $\mathbf{T}_{\mathbf{x}}^{ij}$ in preparation for the next iteration.

The result of tracking fractional displacements is shown in Figure 4 (right), and is contrasted with the advection of particles in a circular flow without this correction. (left)

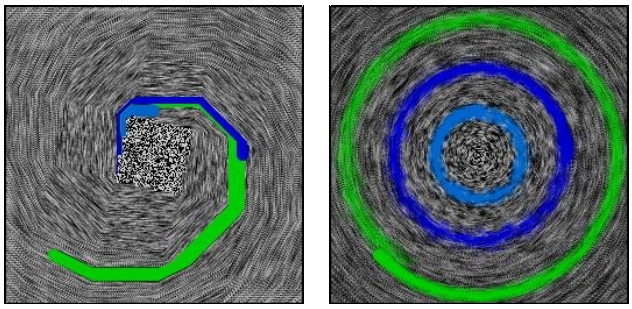

Figure 4. Texture advection along a circular flow defined by $(u,v) = (-y,x)$. Left: fractional coordinate correction is disabled. Right: fractional correction is enabled. A dye has been released to reveal the effects of long time integration. (See [7] for details on dye advection)

4.5 Outer Algorithm

Starting from the collection of tiles $\mathbf{T}_N^{ij}$ (generated in step 6 in the inner algorithm), the outer algorithm reconstructs a global noise texture $\mathbf{T}_N$ and blends it into the cumulative [GE1]buffer $\mathbf{B}_C$.

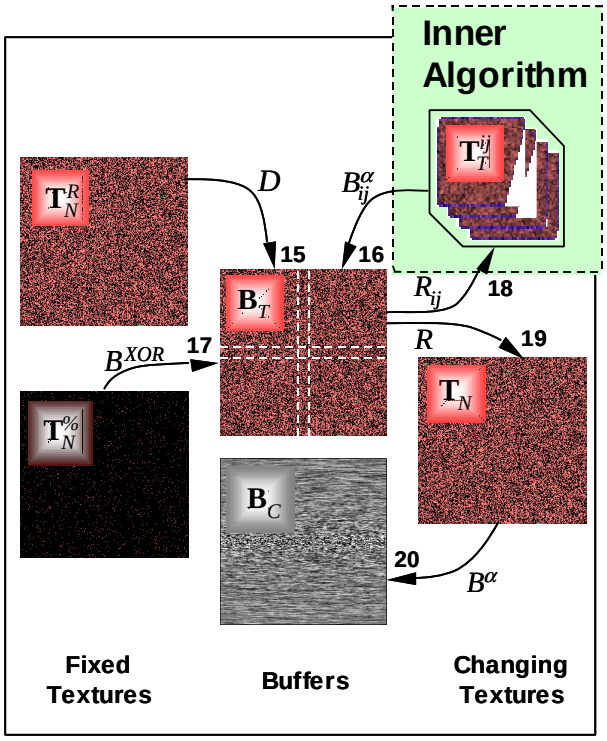

Figure 5. Flowchart for outer algorithm.

4.5.1 Extended Notation

A new notation is necessary to describe operations between textures and buffers of different sizes. For example, it may be necessary to read a subregion of a buffer into a tile texture (e.g., step 18), or to blend a tile into a subregion of a buffer (e.g., step 16). The size disparity between the two arguments of an operator is denoted by a subscript ij. In practice, the operator is applied successively to all tiles.

4.5.2 Noise Reconstruction

The initial phase of the outer algorithm constructs the global advected noise in a global buffer $\mathbf{B}_T$ from the textures $\mathbf{T}_N^{ij}$ (steps 15-17). Before compositing the individual tiles into this buffer, it is necessary to draw a background random texture $\mathbf{T}_N^R$ into $\mathbf{B}_T$. This serves to provide new noise in edge regions where particles originate from outside the physical domain. Recall that in these regions, the $\mathbf{T}_N^{ij}$ have a zero α component. Rather than construct a new random texture at every step, a random texture translation is applied to $\mathbf{T}_N^R$, which avoids any possible temporal correlation. This is accomplished with a texture coordinate transformation matrix.

Next, each tile $\mathbf{T}_N^{ij}$ is composited into $\mathbf{B}_T$ with the simple alpha blend

$$\mathbf{B}_T \leftarrow [1-\alpha(\mathbf{T}_N^{ij})]\mathbf{B}_T + \alpha(\mathbf{T}_N^{ij})\mathbf{T}_N^{ij}$$

If a particle lies in an overlap region, there is always at least one texture whose corresponding texel is opaque (by construction). In the event there are two such textures, the noise value is the same. Consequently, the result of the blending is independent of the order in which the tiles are processed. This property may

prove useful in parallel extensions of the algorithm that harness the power of multiple graphics cards. If one refers back to the discussion of edge correction in Section 4.4.2, it is clear that the succession of alpha blends of $\mathbf{T}_N^{ij}$ into $\mathbf{B}_T$ results in a seamless noise pattern with new noise only appearing at the edges of the physical domain.

4.5.3 Noise Injection

In regions of positive flow divergence, adjacent pixels in $\mathbf{B}_{\mathbf{x}'}$ that reference the same texel location in $\mathbf{T}_N^{ij}$ after the backward integration step will share the same color. Therefore, the overall spatial frequency of the successive noise textures decreases. Figure 6 clearly demonstrates this decrease for a source after several time steps. To maintain a constant noise frequency, a small amount of new noise is injected into $\mathbf{B}_T$ at every iteration (step 17). Through experimentation, we find that randomly inverting the color of two to three percent of the noise texels at each time step is sufficient to maintain an approximately constant high frequency noise without significant loss of temporal correlation.

In practice, a black texture $\mathbf{T}_N^{\%}$ with a 2-3 percent random distribution of white texels is XORed into $\mathbf{B}_T$ 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 contents of $\mathbf{B}_T$ are then redistributed among the tiles $\mathbf{T}_N^{ij}$ (step 18) for use in the next inner iteration.

4.5.4 Noise Blending

We introduce an acceptable level of spatial and temporal correlation into each frame by emulation of long time exposure photography, which integrates an image over a finite time interval. This effect can be simulated with standard alpha blending formula

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

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 [8] for example. A two-color black and white noise maximizes the contrast of the final blended image. Good visual results are obtained with $\alpha = 0.1$.

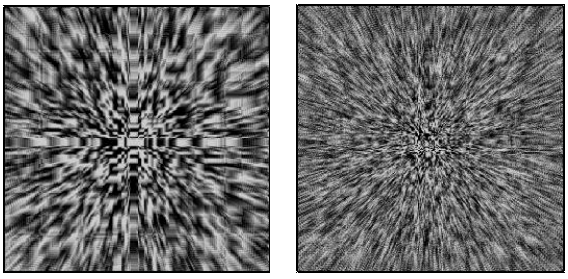

Figure 6. Result of blending 10 successive noise textures. Without noise injection (left), with continuous three percent noise injection (right). Source field $(u,v) = (x,y)$.

5. Hardware Constraints

When developing hardware-accelerated visualization algorithms, the finite depth and size of buffers and textures affect both the accuracy of the algorithm and its computational cost. These effects are most pronounced when correcting the texture advection for fractional displacements and when choosing an optimal depth for the velocity textures.

The notion of fractional displacement is intrinsically linked to the interplay between the depth of the framebuffer and the width of the tiles. To illustrate this notion, consider a framebuffer with 12 bits per color component and a square tile 256 pixels wide. In step 5 of the algorithm, the new noise texture $\mathbf{T}_N^{ij}(\mathbf{x}'(\mathbf{x}))$ is computed with a pixel texture; $\mathbf{x}'$ is stored in $\mathbf{B}_{\mathbf{x}'}$ with a precision of 12 bits. Two elements conspire to reduce the number of significant bits in $\mathbf{x}'$. First, $\mathbf{T}_N^{ij}$ is only 256 texels wide. Second, the texture interpolation uses GL_NEAREST to maximize noise contrast. The combined result is that only the first eight bits of $\mathbf{x}'$ affect the advected noise. We call the remaining four bits the fractional part of the particle displacement, used to increase the accuracy of the texture advection (Section 4.4.3).

To quantify the above statements, we denote by n_b the number of bits per color component in the framebuffer, and by 2^{n_s} the size of the smallest square texture that covers a tile, where

$$n_s = \lceil \log_2 m_s \rceil$$

Clearly, there are $n_b - n_s$ bits available to store the fractional part of the particle displacement. With a framebuffer depth of 12 bits and a range of tiles with sizes that range from 256 down to 32 pixels, the number of fractional bits ranges from 4 to 7, the latter providing for advection with the highest accuracy but at the highest cost per frame. Commodity graphics boards provide 8-bit framebuffers. They would require tiles that are 16 times smaller to achieve a comparable accuracy.

An internal texture format that is consistent with the depth of the buffers is necessary to prevent a loss of accuracy during pixel transfer operations. OpenGL accommodates several different internal texture formats with various depths. We use a GL_RGBA12_EXT internal format (12 bits per component) for coordinate textures. We minimize the use of texture memory without loss of accuracy by storing the velocity field and the noise textures in the GL_RGBA8_EXT and GL_RGBA2_EXT internal formats respectively.

6. Conclusion

This paper describes a hardware-accelerated implementation of an algorithm to visualize unsteady flow visualization based on a per-texel advection technique.

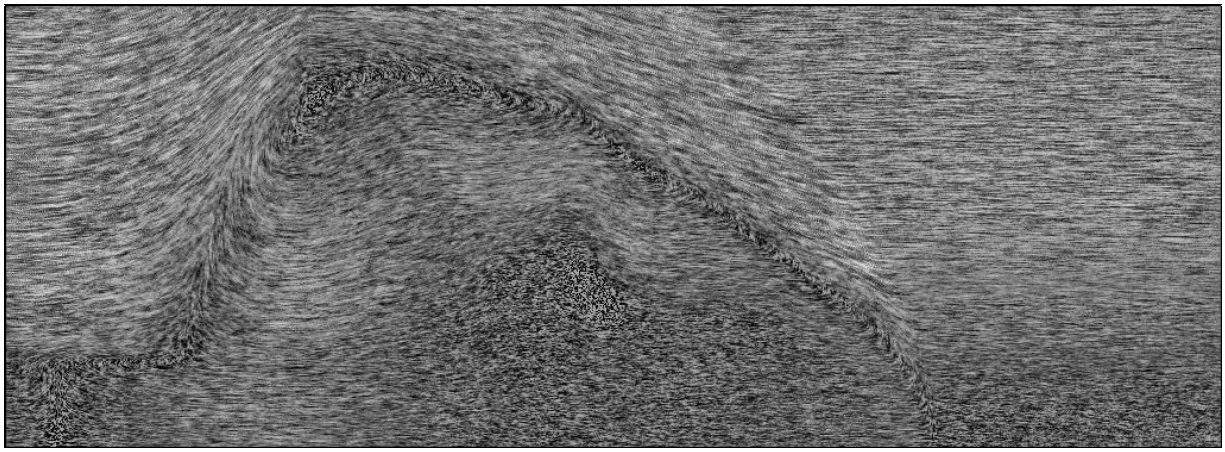

Figure 7. The figure was extracted from a 500-frame animation sequence of a longitudinal vortex interacting with an unsteady shock [4] on a domain of 1024×372 pixels. Note that both magnitude and direction of the velocity field are visible in this *static* picture. To enhance the contrast, we blended the final image with a texture of velocity magnitude. Lighter areas denote regions of higher velocity.

We solved the problems that arise in texture advection algorithms when they are applied to the long-time integration of time-dependent data. Incoming flow regions are handled by injecting uncorrelated noise where particles enter the physical domain. 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 the application of a temporal filter on advected textures. A tiling procedure is introduced to increase the useful size of the image, to increase the accuracy of the texture advection and to reduce the memory requirements associated with the temporary hardware buffers.

A 2D cut of an axisymmetric flow that represents the interaction of a Mach 7 shock with a longitudinal vortex [4] is shown in Figure 7. Each frame took one second to compute on an Octane. Had the complete pixel texture specification been implemented, the time would have been reduced to 0.55 seconds per frame.

At present, only the Maximum Impact and the Octane have the required hardware in their graphics engines. However, these features deserve to be incorporated into a wider class of machines. We expect the algorithm presented in this paper to become increasingly competitive with the best software implementations as the power and cost of graphics cards continues to outpace the development of new computational chips. On the other hand, the precise control afforded by a software implementation will often 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. We acknowledge the support of NSF under grant NSF-9872140.

8. References

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

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

[3] Wim C. de leeuw and Robert van Liere. Spotting Structure in Complex Time Dependent Flow. Technical Report CWI - Centrum voor Wiskunde en Informatica, Technical Report SEN-R9823, September 1998.

[4] Gordon Erlebacher, M. Yousuff. Hussaini, and Chi-Wang Shu. Interaction of a Shock With a Longitudinal Vortex. *Journal Fluid Mechanics*, 337 :129-153, April 1997.

[5] Lisa K. Forssell and Scott 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.

[6] 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*. pages 127-134, ACM, April 1999.

[7] Bruno Jobard, Gordon Erlebacher, and M. Y. Hussaini. Hardware-Accelerated Texture Advection for Unsteady Flow Visualization. *IEEE Visualization 2000. To appear*.

[8] Nelson Max and Roger Crawfis. 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, pages 77-87, 1996.

[9] Han-Wei Shen and David 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.