Lagrangian-Eulerian 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

In this paper, we propose a new technique to visualize dense representations of time-dependent vector fields based on a Lagrangian-Eulerian Advection (LEA) scheme. The algorithm produces animations with high spatio-temporal correlation at interactive rates. With this technique, every still frame depicts the instantaneous structure of the flow, whereas an animated sequence of frames reveals the motion a dense collection of particles would take when released into the flow. The simplicity of both the resulting data structures and the implementation suggest that LEA could become a useful component of any scientific visualization toolkit concerned with the display of unsteady flows.

1. Introduction

Traditionally, unsteady flow fields are visualized as a collection of pathlines or streaklines that originate from user-defined seed points [7,8]. More recently, several authors have developed techniques based on dense representations of the flow to maximize information content [3,5,6,10-12]. The fundamental challenge faced by this class of algorithms is to produce smooth animations with good spatial and temporal correlation.

In this paper, we propose a new visualization algorithm based on dense representations of time-dependent vector fields. The method combines the advantages of the Lagrangian and Eulerian formalisms. A dense collection of particles is integrated backward in time (Lagrangian step), while the color distribution of the image pixels are updated in place (Eulerian step). The dynamic data structures normally required to track individual particles, pathlines, or streaklines are no longer necessary since all information is now stored in a few two-dimensional arrays. The combination of Lagrangian and Eulerian updates is repeated at every iteration. A single time step is executed as a sequence of identical operations over all array elements. By its very nature, the algorithm takes advantage of spatial locality and instruction pipelining and can generate animations at interactive frame rates.

The rest of the paper is organized as follows. Section 2 gives an overview of related work. The general approach is described in Section 3 while the algorithm is examined in Section 4. Post-processing options are proposed in Section 5. Timing results are presented in Section 6. Conclusions are drawn in Section 7.

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 [12], and based on the Line Integral Convolution (LIC) technique [2]. The algorithm achieves good spatial and temporal correlation. However, the images are difficult to interpret: the paths are blurred in regions of rapid change of direction, and are thickest where the flow is almost uniform. The low performance of the algorithm is explained by the large number of particles (three to five times the number of pixels in the image) to process for each animation frame.

The spot noise technique, initially developed for the visualization of steady vector fields, has a natural extension to unsteady flows [3]. A sufficiently large collection of elliptic spots is chosen to entirely cover an image of the physical domain. The position of these spots is integrated along the flow, bent along the local pathline or streamline, and finally blended into the animation frame. The rendering speed of the algorithm can be increased by decreasing the number of spots in the image. The control of pixel coverage is done by assigning a fixed lifespan to each spot.

Max and Becker [10] propose a 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.

A technique to display streaklines was developed by Rumpf and Becker [11]. They precompute a two-dimensional noise texture whose coordinates represent time and a boundary Lagrangian coordinate. Particles at any point in space and time that originate from an inflow boundary are mapped back to a point in this texture.

More recently, Jobard *et al.* [5,6] extend the work of Heidrich *et al.* [4] to animate unsteady two-dimensional vector fields. The algorithm relies heavily on extensions to OpenGL proposed by SGI, in particular, pixel textures, additive and subtractive blending, and color transformation matrices. They pay particular attention to the flow entering and leaving the physical domain, leading to smooth animations of arbitrary duration. Excessive discretization errors associated with 12 bit textures are addressed by a tiling mechanism [5]. Unfortunately, the graphics hardware extension this algorithm relies on most, the pixel texture extension, was not adopted by other graphics card manufacturers. As a result, the algorithm only runs on the SGI Maximum Impact and the SGI Octane with the MXE graphics card.

3. Lagrangian- Eulerian Approach

We wish to track a collection of particles p_i, along a prescribed time-dependent velocity field, that densely covers a rectangular region. If we assign a property $P(p_i)$ to the i^{th} particle p_i, the property remains constant as the particle follows its pathline. At any given instant t, each spatial location $\mathbf{x}$ has an associated particle, labeled $p^t(\mathbf{x})$. One expresses that the particle property is invariant along a pathline by

$$\frac{\partial P\left(p^t(\mathbf{x})\right)}{\partial t} + \mathbf{v}^t(\mathbf{x}) \cdot \nabla P\left(p^t(\mathbf{x})\right) = 0 \tag{1}$$

The property attached to each particle takes on the role of a passive scalar. Its value is therefore not affected by diffusion or source terms (associated with chemical or other processes). This equation has two interpretations. In the first, the trajectory of a single particle, denoted by $\mathbf{x}^t(p)$ where p tags the particle, satisfies

$$\frac{d\mathbf{x}^t(p)}{dt} = \mathbf{v}^t\left(\mathbf{x}^t, p\right) \tag{2}$$

In this *Lagrangian* approach, the trajectory of each particle is computed separately. The time evolution of a collection of particles is displayed by rendering each particle by a glyph (point, texture spot [3], arrows). Except for recent work of Jobard *et al.* [5,6], current time-dependent algorithms are all based on particle tracking, e.g. [1,3,8,9,12]. While Lagrangian tracking is well suited to the task of understanding how dense groups of particles evolve in time, it suffers from several shortcomings. In regions of flow convergence, particles may accumulate into small clusters that follow almost identical trajectories, leaving regions of flow divergence with a low density of particles. To maintain a dense coverage of the domain, the data structures must support dynamic insertion and deletion of particles [12], or track more particles than needed [3], which decreases the efficiency of any implementation.

Alternatively, an *Eulerian* approach solves (1) directly. Particles lose their identity. However, the particle property, viewed as a field, is known for all time at any spatial coordinate. Unfortunately, any explicit discretization of (1) is subject to a *Courant condition*[1], so that in practice, the numerical integration step is limited to at most 1-2 cell widths. In turn, this imposes a maximum rate at which flow structures can evolve.

In our approach, we choose a hybrid solution. Between two successive time steps, coordinates of a dense collection of particles are updated with a Lagrangian scheme whereas the advection of the particle property is achieved with an Eulerian method. At the beginning of each iteration, a new dense collection of particles is chosen and assigned the property computed at the end of the previous iteration. We refer to the hybrid nature of this approach as a Lagrangian-Eulerian Advection (LEA) method.

To illustrate the idea, consider the advection of the bitmap image shown in Figure 1a by a circular vector field centered at the lower left corner of the image. With a pure Lagrangian scheme, a dense collection of particles (one per pixel) is first assigned the color of the corresponding underlying pixel. Each particle advects along the vector field and deposits its color property in the corresponding pixel in a new bitmap image. This technique does not ensure that every pixel of the new image is updated. Indeed, holes usually appear in the resulting image (Figure 1b).

A better scheme considers each pixel of the new image as a particle whose position is integrated backward in time. The particle position in the initial bitmap determines its color. There are no longer any holes in the new image (Figure 1c). Repeating the process at each iteration, any property can be advected while maintaining a dense coverage of the domain.

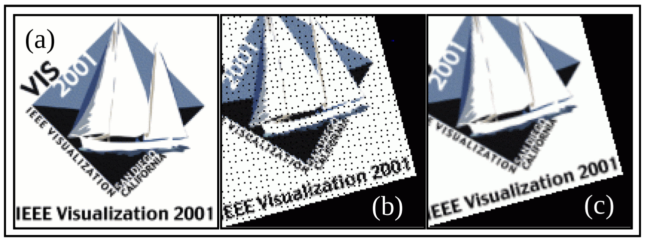

Figure 1. Rotation of bitmap image about the lower left corner. (a) Original image, (b) Image rotated with Lagrangian scheme, (c) Image rotated with Eulerian scheme.

The core of the advection process is thus the composition of two basic operations: *coordinate integration* and *property advection*.

Given the position $\mathbf{x}^0(i,j) = (i,j)$ of each particle in the new image, backward integration of Equation (2) over a time interval h determines its position

$$\mathbf{x}^{-h}(i,j) = \mathbf{x}^0(i,j) + \int_0^{-h} \mathbf{v}^{t-\tau}\left(\mathbf{x}^\tau(i,j)\right) d\tau \tag{3}$$

at a previous time step. h is the integration step, $\mathbf{x}^\tau(i,j)$ represents intermediary positions along the pathline passing through $\mathbf{x}^t(i,j)$, and $\mathbf{v}^\tau$ is the vector field at time τ.

An image of resolution $W \times H$, defined at a previous time $t-h$, is advected to time t through the indirection operation

$$\mathbf{I}^t(i,j) = \begin{cases} \mathbf{I}^{t-h}\left(\mathbf{x}^{-h}(i,j)\right) & \forall \mathbf{x}^{-h} \in [0,W) \times [0,H) \\ \text{user-specified value} & \text{otherwise} \end{cases} \tag{4}$$

which allows the image at time t to be computed from the image at any prior time $t-h$. This technique was used by Max[10]. However, instead of integrating back to the initial time to advect the same initial texture[10], we choose h to be the interval between two successive displayed images and always advect the last computed image. This minimizes the need to access coordinate values outside the physical domain. Notice that at least a linear interpolation of $\mathbf{I}^{t-h}$ pixels at the positions $\mathbf{x}^{-h}$ is necessary to obtain an image of acceptable quality.

[1] If the discrete time step exceeds some maximum value, severe numerical instabilities result.

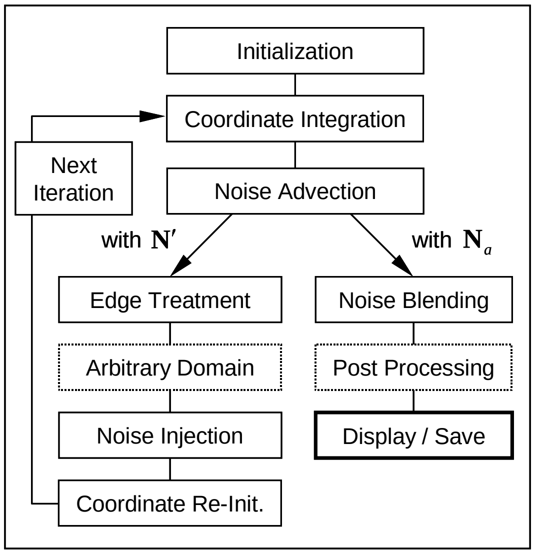

Figure 2. Flowchart of LEA algorithm.

4. Algorithm

With our Lagrangian-Eulerian approach, a full per-pixel advection requires manipulating exactly $W \times H$ particles. All information concerning any particle is stored in two-dimensional arrays with resolution $W \times H$ at the corresponding location (i,j). Thus, we store the initial coordinates (x,y) of those particles in two arrays $\mathbf{C}_x(i,j)$ and $\mathbf{C}_y(i,j)$. Two arrays $\mathbf{C}'_x$ and $\mathbf{C}'_y$ contain their x and y coordinates after integration. A first order integration method requires two arrays $\mathbf{V}_x$ and $\mathbf{V}_y$ that store the velocity field at the current time. Similarly to LIC, we choose to advect noise images. Four noise arrays $\mathbf{N}$, $\mathbf{N}'$, $\mathbf{N}_a$ and $\mathbf{N}_b$ contain respectively the noise to advect, two advected noise images, and the final blended image.

Figure 2 shows a flowchart of the algorithm. After the initialization of the coordinate and noise arrays (Section 4.2), the coordinates are integrated (Section 4.3) and the initial noise array $\mathbf{N}$ is advected (Section 4.4). The first advected noise array, $\mathbf{N}'$ is then prepared for the next iteration by subjecting it to a series of treatments (left column in Figure 2). Care is first taken to ensure that no spurious artifacts appear at boundaries where flow is entering the domain (Section 4.5). This is followed by an optional masking process to allow for non-rectangular domains (Section 4.6). A low percentage of random noise is then injected into the flow to compensate for the effects of pixel duplication and flow divergence (Section 4.7). Finally, the coordinate arrays are reinitialized to ready them for the next iteration (Section 4.8). The right column in the flowchart describes the sequence of steps that transform the second advected noise array $\mathbf{N}_a$ into the final image. $\mathbf{N}_a$ is first accumulated into $\mathbf{N}_b$ via a blending operation to create the necessary spatio-temporal correlation (Section 4.9). Two optional post-processing phases are then applied to $\mathbf{N}_b$ before its final display: a line integral convolution filter removes aliasing effects (Section 5.1) and features of interest are emphasized via an opacity mask (Section 5.2).

4.1 Notation

Array cell values are referenced by the notation $\mathbf{A}(i,j)$ with i and j integers in $\{0,..,W-1\}\times\{0,..,H-1\}$. We adopt the convention that an array $\mathbf{A}(x,y)$ with real arguments is evaluated from information in the four neighboring cells using bilinear interpolation. A constant interpolation is explicitly noted $\mathbf{A}(\lfloor x\rfloor,\lfloor y\rfloor)$, where $\lfloor x\rfloor$ is the largest integer smaller than or equal to x. To simplify the notation, array operations such as $\mathbf{A}=\mathbf{B}$ apply to the entire domain of (i,j).

The indirection operation $\mathbf{A}(i,j)=\mathbf{B}(r\mathbf{C}(i,j),s\mathbf{D}(i,j))$, where $\mathbf{C}(i,j)$ and $\mathbf{D}(i,j)$ lie in the range $[0,W-1]$ and $[0,H-1]$ respectively and r and s are scalars, is denoted by $\mathbf{A}=\mathbf{B}(r\mathbf{C},s\mathbf{D})$.

4.2 Coordinate and Noise Initialization

We first initialize the coordinate arrays $\mathbf{C}_x$, $\mathbf{C}_y$ and the noise arrays $\mathbf{N}$ and $\mathbf{N}_b$. Coordinates are defined as

$$\begin{cases}\mathbf{C}_x(i,j)=i+\operatorname{rand}(1)\\ \mathbf{C}_y(i,j)=j+\operatorname{rand}(1)\end{cases} \tag{5}$$

where $\operatorname{rand}(1)$ is a real number in $[0,1)$. The random offset distributes coordinates on a jitter grid to avoid regular patterns that might otherwise appear during the first several steps of the advection. Note that the integer part of the coordinates identifies the cell.

$\mathbf{N}$ is initialized with a two-valued noise function (0 or 1) to ensure maximum contrast and its values are copied into $\mathbf{N}_b$. Coordinates and noise values are stored in floating point format to ensure sufficient accuracy in the calculations.

4.3 Coordinate Integration

A first order discretization of Equation (3) is used to integrate the particle coordinates. After discretization with a constant time step h over the entire domain, (3) becomes

$$\begin{cases}\mathbf{C}'_x=\mathbf{C}_x-(h/V_{\max})\mathbf{V}_x\left(r_{W_V}\mathbf{C}_x,r_{H_V}\mathbf{C}_y\right)\\ \mathbf{C}'_y=\mathbf{C}_y-(h/V_{\max})\mathbf{V}_y\left(r_{W_V}\mathbf{C}_x,r_{H_V}\mathbf{C}_y\right)\end{cases} \tag{6}$$

where $V_{\max}$ is the maximum velocity magnitude in the whole dataset, $r_{W_V}=(W_V-1)/W$ and $r_{H_V}=(H_V-1)/H$ for a vector field resolution of W_V by W_H. The two scaling factors r_{W_V} and r_{H_V} ensure that the coordinates of the velocity arrays stay within proper bounds.

The velocity arrays at the current time are linearly interpolated between the two closest available vector fields. Therefore, h represents the maximal possible displacement of any particle over all iterations. The actual displacement of a particle is proportional to the local velocity and is measured in units of cell widths.

A useful property of a first order formulation is that the velocity is never required outside the physical domain. We have implemented a second order discretization, but found no noticeable effect due to the small extent of the spatio-temporal correlations in the final display.

4.4 Noise Advection

The advection of noise described by Equation (4) is applied twice to $\mathbf{N}$ to produce two noise arrays $\mathbf{N}'$ and $\mathbf{N}_a$, one for advection, one for display. $\mathbf{N}'$ is an internal noise array whose purpose is to carry on the advection process and to re-initialize $\mathbf{N}$ for the next iteration. To maintain a sufficiently high contrast in the advected noise, $\mathbf{N}'$ is computed with a constant interpolation. Before $\mathbf{N}'$ can be used in the next iteration, it must undergo a series of corrections to account for edge effects, the presence of arbitrary domains, and the deleterious consequences of flow divergence.

$\mathbf{N}_a$ serves to create the current animation frame and no longer participates in the noise advection. It is computed using linear interpolation of $\mathbf{N}$ to reduce spatial aliasing effects. $\mathbf{N}_a$ is then blended into $\mathbf{N}_b$ (Section 4.9).

A straightforward implementation of Equation (4) leads to conditional expressions to handle the cases when

$$\mathbf{x}' = \left(\mathbf{C}'_x(i,j), \mathbf{C}'_y(i,j)\right)$$

is exterior to the physical domain. A more efficient implementation eliminates the need to test for boundary conditions by surrounding $\mathbf{N}$ and $\mathbf{N}'$ with a buffer zone of constant width. From equation (6), $\mathbf{x}'$ refers to cells located at a maximum distance of $b = \lceil h \rceil$ cell width away from the array borders. An expanded noise array of size $(W+2b)\times(H+2b)$ is therefore sufficient to ensure that out of bound array accesses do not occur (see Figure 3). The advected arrays $\mathbf{N}'$ and $\mathbf{N}_a$ are computed according to

$$\begin{cases} \mathbf{N}'(i+b, j+b) = \mathbf{N}\left(\left\lfloor \mathbf{C}'_x(i,j)\right\rfloor + b, \left\lfloor \mathbf{C}'_y(i,j)\right\rfloor + b\right) \\ \mathbf{N}_a(i,j) = \mathbf{N}\left(r_W \mathbf{C}'_x(i,j) + b, r_H \mathbf{C}'_y(i,j) + b\right) \end{cases} \quad (7)$$

for all $(i,j) \in \{0,..,W-1\} \times \{0,..,H-1\}$, where $r_W = (W-1)/W$ and $r_H = (H-1)/H$. The two scaling factors r_W and r_H ensure a properly constructed linear interpolation.

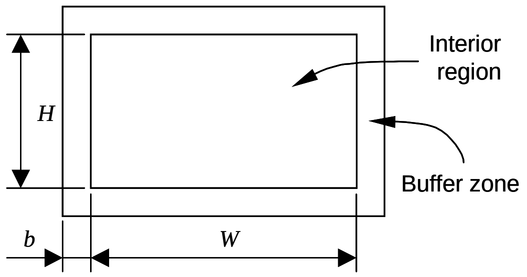

Figure 3. Noise arrays $\mathbf{N}'$ and $\mathbf{N}_a$ are expanded with a surrounding region $\mathbf{b} = \lceil \mathbf{h} \rceil$ cells wide.

4.5 Edge Treatment

A recurring issue with texture advection that must be addressed is the proper treatment of information flowing *into* the physical domain. Within the context of this paper, we must determine the user-specified value in Equation (4). To address this, we recall that the advected image contains a two-valued random noise with little or no spatial correlation. We take advantage of this property to replace the user-specified value by a random value (0 or 1). At each iteration, we simply store new random noise in the buffer zone, at negligible cost.

At the next iteration, $\mathbf{N}$ will contain these values and some of them will be advected to the interior of the physical domain by Equation (7). Since random noise has no spatial correlation, the advection of the surrounding buffer values into the interior region of $\mathbf{N}'$ produces no visible artifacts.

To treat periodic flows in one or more directions, the noise values are copied from an inner strip of width b along the interior edge of $\mathbf{N}'$ onto the buffer zone at the opposite boundary. As a result, particles leaving one side of the domain seamlessly reappear at its opposite side.

4.6 Incoming Flow in Arbitrary Shaped Domains

It often happens that the physical domain is non-rectangular or contains interior regions where the flow is not defined (e.g. shores and islands). Denote by B the boundaries interior to $\mathbf{N}$ that delineates these regions. LEA handles this case with no modification by simply setting the velocity to zero where it is not defined. The stationary noise in these regions is hidden from the animation frame by superimposing a semitransparent map that is opaque where the flow is undefined. For example, the opaque regions of this map might represent shorelines or islands (see right column on color plate).

When the flow velocity normal to B is nonzero and points into the physical domain, the advection of stationary noise values will create noticeable artifacts in the form of streaks. This might happen if an underground flow, not visible in the display, emerges into view at B. If necessary, we suppress these streaks with the help of a pre-computed boolean mask (or alternatively a boolean function) $\mathbf{M}(i,j)$ that determines whether or not the velocity field is defined at (i,j). $\mathbf{N}'(i,j)$ is then updated with random noise where $\mathbf{M}(i,j)$ is false.

4.7 Noise Injection

In this Section, we propose a simple procedure to counteract a *duplication effect* that occurs during the computation of $\mathbf{N}'$ in Equation (7). Effectively, if particles in neighboring cells of $\mathbf{N}'$ retrieve their property value from within the same cell of $\mathbf{N}$, this value will be duplicated in the corresponding cells of $\mathbf{N}$. Single property values in $\mathbf{N}$ may be duplicated onto neighboring cells in $\mathbf{N}'$ where coordinates $\left(\mathbf{C}'_x, \mathbf{C}'_y\right)$ have identical integer values.

To illustrate the source of noise duplication, we consider an example. Figure 4 shows the evolution of property values and particle positions for four neighboring pixels during one integration and one advection step. The vector field is uniform, is oriented at 45 degrees to the x axis, and points towards the upper right corner. At the start of the iteration, each particle has a random position within its pixel (Figure 4a). To determine the new property value of each pixel, the particle positions are integrated backwards in time (Figure 4b). The property value of the lower left corner pixel is duplicated onto the four pixels (worst-case sce-

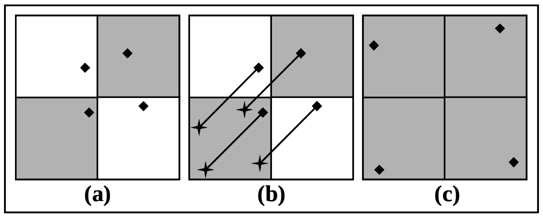

Figure 4. Noise duplication. A single noise value is duplicated into four cells in a uniform 45 deg flow.

nario) (Figure 4c). The fractional position of each particle is then re-initialized for the next iteration.

Over time, the average size of contiguous constant color regions in the noise increases. This effect is undesirable since lower noise frequency reduces the spatial resolution of the features that can be represented. This duplication effect is further reinforced in regions where the flow has a strong positive divergence.

To break the formation of uniform blocks and to maintain a high frequency random noise, we inject a user-specified percentage of noise into $\mathbf{N}'$. Random cells are chosen in $\mathbf{N}'$ and their value is inverted (a zero value becomes one and vice versa). The number of cells randomly inverted must be sufficiently high to eliminate the appearance of pixel duplication, but low enough to maintain the temporal correlation introduced by the advection step.

To quantify the effect of noise injection, we compute the energy content of the advected noise in $\mathbf{N}$ at different scales as a function of time. Although the Fourier transform would appear to be the natural tool for this analysis, the two-valued nature of the noise image suggests instead the use of the Haar wavelet (linear combination of Heaviside functions). We perform a two-dimensional Haar wavelet transform and compute the level of energy in different bands (the spatial scale of consecutive bands vary by a factor of two). The two-dimensional energy spectrum is reduced to a one-dimensional spectrum by assuming that the noise texture is isotropic at any point in time. (The smooth anisotropic flow result from *blending* multiple noise textures.) The energy in each band is scaled by its value after the initial noise injection. Ideally we would like to preserve the initial spectrum at all time.

Figure 5 illustrates the influence of the noise injection on the time evolution of the energy spectrum. Without injection, the energy in the larger scales (regions of pixel duplication) increases rapidly without stabilizing. This comes at the expense of some energy loss in the smaller scales (which decreases in the figure). As the percentage of noise injection increases, the spread of the scaled spectrum decreases continuously towards zero (the ideal state). However, excessive injection deteriorates the quality of the temporal correlation.

The necessary percentage of injected noise is clearly a function of the particular flow and depends on both space and time. It should be modeled as the contribution of two terms: a constant term that accounts for the duplication effects at zero divergence, and a term that is a function of the velocity divergence. In the interest of simplicity and efficiency, we use a fixed percentage of two to three percent, which provides adequate results over a wide range of flows.

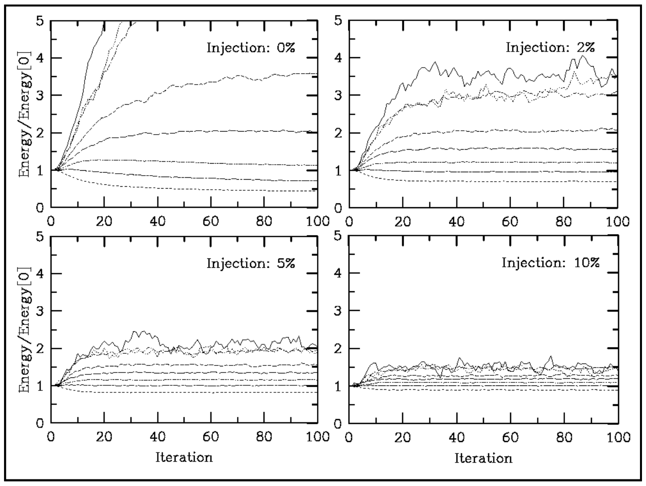

Figure 5. Energy content of the flow at different scales based on a 2D Haar wavelet decomposition of the two-valued noise function (assumed to be isotropic). The energy in each band is scaled with respect to its initial value. Results are shown for injection rates of 0%, 2%, 5% and 10%.

4.8 Coordinate Re-Initialization

The coordinate arrays are re-initialized to prepare a new collection of particles to be integrated backward in time for the next iteration. However, coordinates are not re-initialized to their initial values. The advection equations presented in Section 3 assume that the particle property is computed at the previous time step via a linear interpolation. Unfortunately, the lack of spatial correlation in the noise image would lead to a rapid loss of contrast, which justifies our use of a constant interpolation scheme. However, the choice of constant interpolation implies that a property value can only change if it originates from a different cell. If the coordinate arrays were re-initialized to their original values at each iteration, subcell displacements would be ignored and the flow would be frozen where the velocity magnitude is too low. This is illustrated in Figure 6, which shows the advection of a steady circular vector field. Constant interpolation without fractional coordinate tracking clearly shows that the flow is partitioned into distinct regions within which the integer displacement vector is constant (Figure 6a).

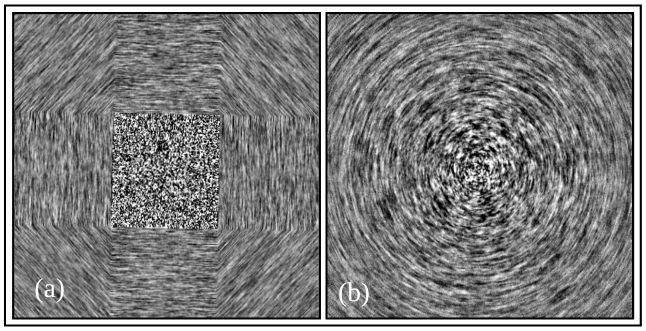

Figure 6. Circular flow without and with accumulation of fractional displacement $(h = 2)$.

To prevent this from happening, we track the fractional part of the displacement within each cell. Instead of re-initializing the coordinates to their initial values, the fractional part of the displacement is added to cell indices (i, j):

$$\begin{cases} \mathbf{C}_x(i,j) = i + \mathbf{C}'_x(i,j) - \lfloor \mathbf{C}'_x(i,j) \rfloor \\ \mathbf{C}_y(i,j) = j + \mathbf{C}'_y(i,j) - \lfloor \mathbf{C}'_y(i,j) \rfloor \end{cases} \tag{8}$$

The effect of this correction is shown in Figure 6b.

The coordinate arrays have now returned to the state they were in after their initialization phase (Equation (5)); they verify the relations $\lfloor \mathbf{C}_x(i,j) \rfloor = i$ and $\lfloor \mathbf{C}_y(i,j) \rfloor = j$.

4.9 Noise Blending

Although successive advected noise arrays are correlated in time, each individual frame remains devoid of spatial correlation. By applying a temporal filter to successive frames, spatial correlation is introduced. We store the result of the filtering process in an array $\mathbf{N}_b$. We have found the exponential filter to be convenient since its discrete version only requires the current advected noise and the previous filtered frame. It is implemented as an alpha blending operation

$$\mathbf{N}_b = (1-\alpha)\mathbf{N}_b + \alpha \mathbf{N}_a \tag{9}$$

where α represents the opacity of the current advected noise array. A typical range for α is $[0.05, 0.2]$. Figure 7 shows the effect of α on images based on the same set of noise arrays.

The blending stage is crucial because it introduces spatial correlation along *pathline* segments in every frame. To show clearly that the spatial correlation occurs along pathlines passing through each cell, we conceptualize the algorithm in 3D space; the x and y axes represent the spatial coordinates, whereas the third axis is time. To understand the effect of the blending operation, let's consider an array $\mathbf{N}$ with black cells and change a single cell to white. During advection, a sequence of noise arrays (stacked along the time axis) is generated in which the white cell is displaced along the flow. By construction, the curve followed by the white cell is a pathline. The temporal filter blends successive noise arrays $\mathbf{N}_a$ with the most recent data weighted more strongly. The temporal blend of these noise arrays produces the projection of the pathline onto the $x-y$ plane, with an exponentially decreasing intensity as one travels back in time along the pathline. When the noise array with a single white cell is replaced by a two-color noise distribution, the blending operation introduces spatial correlation along a dense set of short pathlines.

Streamlines and pathlines passing through the same cell at the same time are tangent to each other, so a streamline of short extent is well approximated by a short pathline. Therefore, the collection of short pathlines serves to approximate the instantaneous direction of the flow. With our LEA technique, a single frame represents the instantaneous structure of the flow (streamlines), whereas an animated sequence of frames reveals the motion of a dense collection of particles released into the flow.

The filtering phase completes one pass of the advection algorithm. The image $\mathbf{N}_b$ can be displayed to the screen or stored as an animation frame. $\mathbf{N}'$ is used as the initial noise texture $\mathbf{N}$ for the next iteration. It is worthwhile to mention that each iteration ends with data having the exact same property as when it started. In particular, the coordinate arrays satisfy

$$\lfloor \mathbf{C}_x(i,j) \rfloor = i$$
$$\lfloor \mathbf{C}_y(i,j) \rfloor = j$$

and $\mathbf{N}$ contains a two-color noise without degradation of contrast.

In the next section, we describe several optional post-processing steps to enhance the display of the animation frames, both in terms of quality and in terms of content.

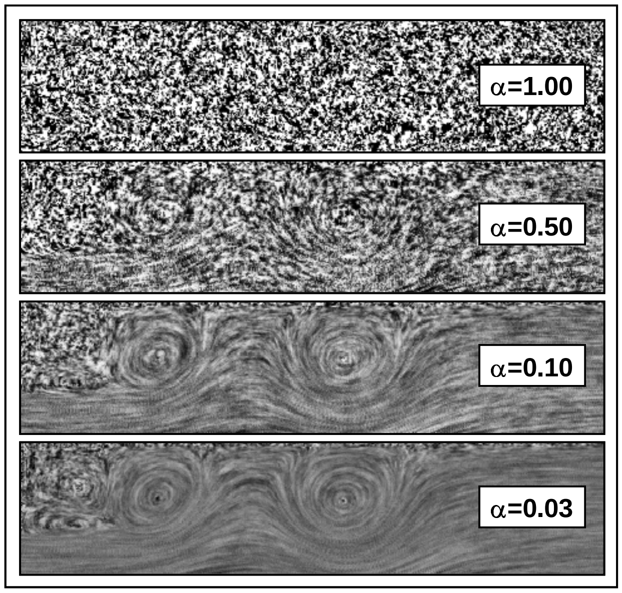

Figure 7. Frames obtained with different values of α.

5. Post-Processing

A series of optional postprocessing steps is applied to $\mathbf{N}_b$ to enhance the image quality and to remove features of the flow that are uninteresting to the user. We present two filters. A fast version of LIC can be applied to remove high frequency content in the image, while a *velocity mask* serves to draw attention to regions of the flow with strong currents.

5.1 Directional Low-Pass Filtering (LIC)

Although the temporal filter (noise blending phase) converts high frequency noise images into smooth spatially-correlated images, aliasing artifacts remain visible in regions where the noise is advected over several cells in a single iteration. As a rule, aliasing artifacts become noticeable where noise advect more than one or two cells in a single time step (see Figure8 bottom). Experimentation with different low-pass filters led us to conclude that a Line Integral Convolution filter applied to $\mathbf{N}_b$ is the best filter to remove the effect of artifacts while preserving and enhancing the directional correlation resulting from the blending phase. This follows from the fact that temporal blending and LIC bring out the structure of pathlines and streamlines respectively, and these curves are tangent to one another at each point. Although the image quality is often enhanced with longer kernel lengths, it is detrimental here since the resulting streamlines will have significant deviations from the actual pathlines. The partial destruction of the temporal correlation between frames would then lead to flashing effects in the animation. A secondary effect of longer kernels is decreased contrast.

While any LIC implementation can be used, our algorithm can advect an entire texture at interactive rates. Therefore, we are interested in the fastest possible LIC implementation. To the best of our knowledge, FastLIC [13] and Hardware-Accelerated LIC [4] are the fastest algorithms to date, and both are well suited to the task. However, we propose a simple, but very efficient, software version of Heidrich's hardware implementation to postprocess the data when the highest quality is desired.

Besides the input noise array $\mathbf{N}_b$, the algorithm requires two additional coordinate arrays Cxx and Cyy, and an array $\mathbf{N}_{LIC}$ to store the result of the line integral convolution. The length of the convolution kernel is denoted by L. For reference, we include the pseudo code for *Array-LIC* in Figure 9.

In general, $L \approx h$ produces a smooth image with no aliasing. However, large values of h speed up the flow, with a resulting increase in aliasing effects. If the quality of the animation is important, L must be increased with a resulting slowdown in the frame rate. The execution time of the LIC filter is commensurate with the timings of FastLIC for $L < 10$. Beyond 10, a serial FastLIC [13] should be used instead. An OpenMP implementation of our ALIC algorithm on shared memory architectures is straightforward. Results are presented in Section 6. As shown in Table 1, smoothing the velocity field with LIC reduces the frame rate by a factor of three across architectures. We recommend exploring the data at higher resolution without the filter or at low resolution with the filter.

```
float* ALIC(const float* Vx, const float* Vy,
            int Wv, int Hv,
            const float* $N_b$, int W, int H
            int L, float* $N_{LIC}$)
rWv = Wv/W    ; rHv = Hv/H
rW  = W /(W-1); rH  = H /(H-1)
L2 = L div 2
r  = 1/(2*L2+1)
Loop over pixels i,j { $N_{LIC}$(i,j) = r*Nb(i,j) }
sgn = 1/Vmax
for n = 1 to 2  // Forward and backward advection
  Loop over pixels i,j
      Cxx(i,j)=i; Cyy(i,j)=j
  for k = 0 to L2
    Loop over pixels i,j //Coordinate integration
      Cxx(i,j)= ( Cxx(i,j)
         + sgn*Vx(rWv*Cxx(i,j),rHv*Cyy(i,j))+W)%W
      Cyy(i,j)= ( Cyy(i,j)
         + sgn*Vy(rWv*Cxx(i,j),rHv*Cyy(i,j))+H)%H

    Loop over pixels i,j // Noise advection
                         // and Accumulation
        $N_{LIC}$(i,j) += r*Nb(rW*Cxx(i,j),rH*Cyy(i,j))
  end for
  sgn = -1/Vmax
end for
return $N_{LIC}$
```

Figure 9. Pseudo-code for ALIC (Array LIC).

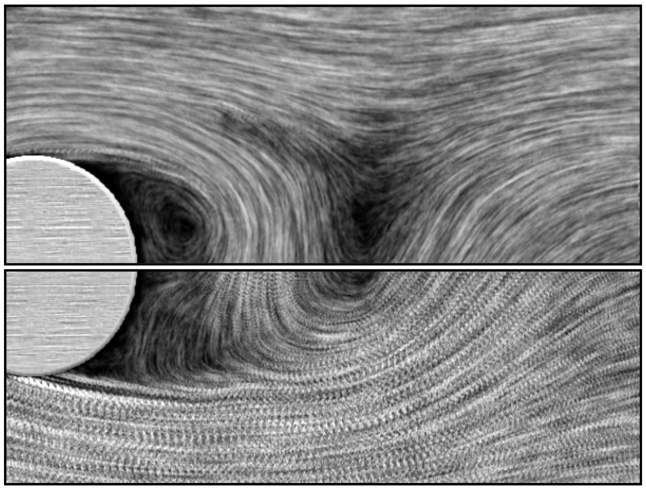

Figure 8. Frame without (bottom) and with (top) LIC filter. A velocity mask is applied to both images.

5.2 Velocity Mask

To fade out high frequency noise in $\mathbf{N}_b$ occurring in low velocity regions, we construct an opacity map, referred to as a velocity mask, that we store in an alpha layer. With a partially transparent noise layer, a background image such as a geographical map (see color plates) can greatly enhance the information content provided by the flow by providing additional context. For maximum control, $\mathbf{N}_b$ should become more transparent in regions of low intensity. Regions of the flow with strong currents can be emphasized further by maximizing the opacity where the velocity magnitude is high. Once computed, the velocity mask is combined with $\mathbf{N}_b$ into an intensity-alpha texture that is blended with the background image (see color plate). We compute the opacity map

$$\mathbf{A} = \left(1 - (1 - \mathbf{V})^m\right)\left(1 - (1 - \mathbf{N}_b)^n\right) \tag{10}$$

as a product of a function of local velocity magnitude and a function of the noise intensity. Higher values of the exponents m and n increase the contrast between regions of low and high velocity and low and high intensity respectively. When $m = n = 1$, equation (10) takes the linear form $\mathbf{A} = \mathbf{N}_b \mathbf{V}$.

6. Results

The next section presents timings of our algorithm. We conducted experiments to evaluate the efficiency of the algorithm at four resolutions (300^2 through 1000^2 pixels). We present in Table 1 timings in frames/second, using several of the available options. Three different computers were used.

The organization of the algorithm as a series of array operations makes it particularly straightforward to parallelize on shared memory architectures. Furthermore, operations on the array elements only make accesses within h rows or columns. For small h, the locality of these accesses is sufficient not to produce cache misses on a CPU with a cache of moderate size (e.g., 512 kbytes). Table 1 also includes timings from an OpenMP implementation running on four processors of an Onyx2.

Options / Resolutions	**Advection**		**Advection + Velocity Mask** $(m = n = 3)$		**Advection + Velocity Mask + ALIC filter** $(L = 6)$	
$\mathbf{300 \times 300}$	3.4	14.0	2.2	8.8	0.8	3.0
	16.3	39.0	10.4	27.0	3.6	11.6
$\mathbf{500 \times 500}$	1.2	4.7	0.8	3.1	0.3	1.0
	6.3	18.0	3.7	10.5	1.3	4.5
$\mathbf{1000 \times 1000}$	0.3	1.2	0.2	0.7	0.07	0.2
	1.4	4.1	0.9	2.7	0.3	1.1

Table 1: Timings in frames/second as a function of options and resolutions. Each configuration has been tested on four different configurations: O2[1] (upper left), Octane[2] (upper right), Onyx2[3] (lower left) and Onyx2 with four processors (lower right).

7. Conclusion

This paper describes an algorithm to visualize time-dependent flows based on an original per-pixel Lagrangian-Eulerian Advection approach. A noise image is advected from a time step to the next. The color of every pixel in the current image is determined in two steps. A dense collection of particles (one per pixel) is first integrated backward in time for a fixed time interval (Lagrangian phase) to determine their positions in the previous frame. The color at these positions determine the color of each pixel in the current frame (Eulerian phase). We described how to seamlessly handle regions where the flow enters the physical domain. A temporal filter is applied to successive images to introduce a good level of spatio-temporal correlation. Thus, every still frame represents the instantaneous structure of the flow, whereas an animated sequence of frames reveals the motion of a dense collection of particles released into the flow. When necessary, spatial correlation is enhanced through a fast LIC algorithm. A post-processing filter has been described to control the contrast between regions of high and low velocity magnitude. The advected noise is controlled by the percentage of noise injection, while the final image is influenced by the temporal blending coefficient and the LIC parameters. Although fixed default parameters gives good results for any vector field, these parameters can be chosen interactively, to generate images suitable to the user. Transparency makes it possible to view a background image through the flow; this leads to our current work on multiple layer texture advection. We demonstrated the efficiency of the algorithm on a variety of computers, including a multiprocessor workstation. The interactivity made possible by this work has made it possible to explore 2-D unsteady flows in real time, and suggests that in the near future interactive three-dimensional texture advections will become a reality.

[1] SGI O2, R5000, 200MHz, 64MB, 1MB L2 cache, 64KB L1 cache

[2] SGI Octane, R12000, 300MHz, 2GB, 2MB L2 cache, 64KB L1 cache

[3] SGI Onyx2, R12000, 300MHz, 2GB, 8MB L2 cache, 64KB L1 cache

8. 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. Some of the datasets used to illustrate the techniques presented in this paper were provided courtesy of Z. Ding (FSU), J. O'Brien (COAPS, FSU), J. Quiby (MeteoSwiss, Switzerland), and R. Arina (Ecole Polytechnique of Turin, Italy). We acknowledge the support of NSF under grant NSF-9872140.

9. References

[1] B.G. Becker, D.A. Lane, and N.L. Max. Unsteady Flow Volumes. *Proceedings IEEE Visualization '95.* In G.M. Nielson and D. Silver, editors, IEEE Computer Society Press, October 1995.

[2] B. Cabral and L.C. Leedom. Imaging Vector Fields Using Line Integral Convolution. *Computer Graphics Proceedings.* In J.T. Kajiya, editor, Annual Conference Series, ACM, pp. 263-269, August 1993.

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

[4] 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.* ACM, pp. 127-134, April 1999.

[5] B. Jobard, G. Erlebacher, and M.Y. Hussaini. Tiled Hardware-Accelerated Texture Advection for Unsteady Flow Visualization. *Graphicon 2000.* pp. 189-196, August 2000.

[6] B. Jobard, G. Erlebacher, and M.Y. Hussaini. Hardware-Accelerated Texture Advection for Unsteady Flow Visualization. *Proceedings Visualization 2000.* In T.E. Ertl, B. Hamann, and A. Varshney, editors, IEEE Computer Society Press, pp. 155-162, October 2000.

[7] D.A. Lane. UFAT - A Particle Tracer for Time-Dependent Flow Fields. *Proceedings IEEE Visualization '94.* In R.D. Bergeron and A.E. Kaufman, editors, IEEE Computer Society Press, pp. 257-264, 1994.

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

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

[10] N. Max and B. Becker. Flow visualization using moving textures. In: *Data Visualization Techniques,* Chandrajit Bajaj, editor, John Wiley and Sons, Ltd., pp. 99-105, 1999.

[11] M. Rumpf and J. Becker. Visualization of Time-Dependent Velocity Fields by Texture Transport. *Proceedings of the Eurographics Workshop on Scientific Visualization '98.* Springer-Verlag, pp. 91-101, 1998.

[12] 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), pp. 98-108, 1998.

[13] D. Stalling and H.-C. Hege. Fast and Resolution Independent Line Integral Convolution. *Proceedings of SIGGRAPH '95.* Computer Graphics Annual Conference Series, pp. 249-256, 1995.