# Dynamic Line Integral Convolution for Visualizing Streamline Evolution

Andreas Sundquist

**Abstract**—The depiction of time-dependent vector fields is a central problem in scientific visualization. This article describes a technique for generating animations of such fields where the motion of the streamlines to be visualized is given by a second "motion" vector field. Each frame of our animation is a Line Integral Convolution of the original vector field with a time-varying input texture. The texture is evolved according to the associated motion vector field via an automatically adjusted set of random particles. We demonstrate this technique with examples from electromagnetism.

**Index Terms**—Line integral convolution, time-dependent, time-varying, vector fields, field lines, streamlines, electromagnetism.

## 1 Introduction

Since the introduction of the Line Integral Convolution (LIC) technique [3], it has been possible to depict static vector fields at an extraordinarily high spatial resolution. Traditionally, such fields had been depicted at relatively low spatial resolution using icons on a coarse grid or by using a few streamlines or field lines selected to illustrate important properties of the field. However, the LIC technique produces a representation of the field that shows its structural details at a resolution close to the resolution of the display. The images generated by LIC have outstanding resolution, but the random nature of the underlying texture used in the algorithm make it difficult to extend the technique to time-varying fields. Intuitively, we would like to generate animation sequences of time-varying fields in which every frame resembles the output of LIC for a static vector field, while the time dependence of the vector field is evident from the interframe coherence.

In this article, we describe a new technique, called Dynamic Line Integral Convolution (DLIC), that extends LIC to time-dependent fields in just this way, making it possible to visualize the evolution of its streamlines. The vector field is allowed to vary arbitrarily over time, with the motion of its streamlines described by a second "motion" vector field. Each rendered frame depicting the field is obtained using the LIC technique. To produce an animation of the field, we evolve the texture input to LIC in time in the manner prescribed by the associated "motion" vector field. The input texture is generated by advecting a dense collection of particles over time and adjusting them to maintain the appropriate level of detail. This produces an animated sequence of images of the original field with the desired properties—the time dependence of the field is evident from frame to frame by the interframe coherence and the spatial resolution is as detailed as in images created by LIC.

• *The author can be reached at 542 Addison Ave., Palo Alto, CA 94301. E-mail: asundqui@mit.edu.*

*Manuscript received 20 July 2001; revised 28 Feb. 2002; accepted 29 Apr. 2002.*
*For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 114575.*

Though this new technique is applicable to any time-dependent vector field for which we can also compute the motion of the streamlines, we have developed the DLIC method specifically for visualizing time-dependent electromagnetic fields. After describing the operation of DLIC, we demonstrate its use with time-varying electromagnetic fields in several different situations.

## 2 Previous Work

Vector fields will be denoted functionally as a mapping

$$f : \boldsymbol{D} \times \boldsymbol{T} \rightarrow \boldsymbol{R}^n, \tag{1}$$

where $\boldsymbol{D} \subset \boldsymbol{R}^n$ is the domain of interest and $\boldsymbol{T}$ is the time interval. Although the techniques discussed here would generalize to higher dimensions, from here on we will assume that $n = 2$. There are some cases where LIC is appropriate for visualizing three-dimensional fields [3], but often it is difficult to map all of the information to a two-dimensional display and detail is obscured. The algorithms discussed here frequently make use of numerical integrators to solve first-order differential equations:

$$\frac{d\boldsymbol{\sigma}}{du} = \boldsymbol{f}(\boldsymbol{\sigma}, t), \boldsymbol{\sigma}(0) = \boldsymbol{\sigma}_0. \tag{2}$$

The solution is called a *streamline*, which is an integral curve in the vector field $\boldsymbol{f}$ at a constant time $t$. To describe a particular integral curve, we will use the symbol $\boldsymbol{\sigma}$ parameterized in the following manner:

$$\frac{d}{du}\boldsymbol{\sigma}_t^u(\boldsymbol{x}_0) = \boldsymbol{f}(\boldsymbol{\sigma}, t), \boldsymbol{\sigma}_t^0(\boldsymbol{x}_0) = \boldsymbol{x}_0. \tag{3}$$

This use of $\boldsymbol{\sigma}$ denotes an integral curve at a constant time $t$ seeded at $\boldsymbol{x}_0$ with a parameter $u$. Additionally, we will use the symbol $\hat{\boldsymbol{\sigma}}$ to denote the same curve reparameterized by its arc length $s$. That is,

$$\frac{d}{ds}\hat{\boldsymbol{\sigma}}_t^s(\boldsymbol{x}_0) = \frac{\boldsymbol{f}(\boldsymbol{\sigma}, t)}{\|\boldsymbol{f}(\boldsymbol{\sigma}, t)\|}, \hat{\boldsymbol{\sigma}}_t^0(\boldsymbol{x}_0) = \boldsymbol{x}_0. \tag{4}$$

## 2.1 Line Integral Convolution

The LIC technique is a very general, high-quality method for visualizing static vector fields [3]. Although not originally described in this way, the essence of LIC is the integral operation introduced in [12]:

$$I(\boldsymbol{x}) = \int T(\hat{\boldsymbol{\sigma}}^s(\boldsymbol{x}))\kappa(s)ds. \tag{5}$$

Here, $T$ is a scalar field representing some input texture, and it is convolved with the kernel $\kappa$ along the streamline curve seeded at $\boldsymbol{x}$ in a vector field $\boldsymbol{f}$, yielding the output intensity $I$. Usually, the kernel $\kappa$ is chosen to be symmetric and have a limited extent, for example, a box filter shape. This operation can be approximated via a discrete convolution,

$$I(\boldsymbol{x}) \cong \sum_k T\big(\hat{\boldsymbol{\sigma}}^{k\Delta s}(\boldsymbol{x})\big)\tilde{\kappa}(k\Delta s), \tag{6}$$

taken with a step size $\Delta s$ and a discrete convolution kernel $\tilde{\kappa}$. The output *pixel* values are usually calculated by averaging one or more output intensity *samples*. In LIC, the scalar field $T$ is initialized to random white noise, producing an output that resembles particle streak images. As a result of the convolution operation, the texture values average together along the direction of the field to give a highly correlated, slowly varying intensity. In contrast, the texture values in the direction perpendicular to the field are completely uncorrelated. Thus, the field streamlines can be easily distinguished in the output pixel values.

An important improvement over the basic LIC algorithm is an optimization called Fast Line Integral Convolution (FLIC) [12], [13], which gives an order-of-magnitude rendering speedup. The key to FLIC is the observation that the output intensities of two points along a given streamline are highly correlated. For example, with a discrete convolution kernel,

$$\tilde{\kappa}(k\Delta s) = \begin{cases} \frac{1}{2N+1}, & \text{for } -N \le k \le N \\ 0, & \text{for } |k| > N, \end{cases} \tag{7}$$

the difference in intensity between two points separated by a distance $\Delta s$ is:

$$\begin{aligned} I\big(\hat{\boldsymbol{\sigma}}^{\Delta s}(\boldsymbol{x})\big) - I(\boldsymbol{x}) \cong \frac{1}{2N+1}\Big[&T\Big(\hat{\boldsymbol{\sigma}}^{(N+1)\Delta s}(\boldsymbol{x})\Big) \\ &-T\big(\hat{\boldsymbol{\sigma}}^{-N\Delta s}(\boldsymbol{x})\big)\Big]. \end{aligned} \tag{8}$$

Thus, once we have the output intensity for a given point, we can find the output intensity of the next point on the streamline by a relatively inexpensive calculation. Applying this result optimally, the sample points are determined by the streamlines we choose to follow. As samples are computed, we accumulate the results in the output pixels lying closest to the sample points. A good scheme is to randomly pick the starting points for these incremental streamline computations with a bias against pixels with many samples. Although the cost of rendering an image with the original LIC algorithm is not prohibitive with today's computers, rendering an entire animation sequence with LIC is extremely time consuming. Making use of the FLIC optimization is therefore crucial for dynamic visualizations.

## 2.2 Dynamic Visualizations

There have been a number of articles describing methods for animating vector field images in the style of LIC. Several papers describe methods to depict cyclic motion along streamlines for static (or steady) vector flow fields [3], [4], [5], [8], [15]. Since the vector field itself is not allowed to vary in time, these methods do not allow us to visualize the motion of streamlines themselves.

Extending the algorithms to time-dependent vector fields is still an area of active inquiry. Most research in this area has involved the visualization of unsteady fluid-flow applications. Generally, methods for visualizing such fields do so by simulating the fluid motion. For example, Forssell and Cohen [4] describe a modification to LIC whereby *streaklines* instead of streamlines are followed, where streaklines are the set of points that pass through a seed point when advected through time. Unfortunately, the algorithm suffers from a number of coherence and accuracy problems, which Shen and Kao [10], [11] point out. They describe another algorithm, Unsteady Flow LIC, that advects the input texture forward in time instead of convolving it and feeds the output image back in as the input for the next frame. This improves the coherence of the animation both temporally and spatially.

Several other articles on visualizing time-dependent vector flows focus on the texture advection process. Max and Becker describe an algorithm for animating flow fields by using texture-mapped triangles whose coordinates or vertices are advected according to the flow. When the texture has become warped beyond usefulness, the coordinates and vertices are reset to their original values and the effect is faded in gradually to eliminate sudden popping [9]. Two papers by Jobard et al. describe methods for directly advecting a texture, one using a *Lagrangian-Eulerian* approach [7] and the other by making use of hardware acceleration [6]. Both techniques blend several frames of the advected texture together in order to introduce spatial correlation in the images. In order to avoid loss of detail in divergent regions, a small amount of random noise is continuously added to the texture. Unfortunately, injecting noise in a uniform fashion does not guarantee that the texture will have an even degree of randomness over the long run. Finally, Becker and Rumpf designed an algorithm for visualizing flow fields via the texture transport of value pairs. One coordinate varies perpendicular to the flow and assigns a random intensity to a particular flow line, while the other coordinate indicates the phase at which the flow entered the texture domain by color-coding. Unfortunately, this method also suffers from an inconsistent level of detail across the texture [1].

In any case, these methods are not directly applicable to electromagnetic visualizations because the assumption in flow problems is that the only motion of the field is along the direction of the field itself. For time-varying electromagnetic fields, this is no longer the case—the "motion" of the electric or magnetic field is generally *not* along the field lines. Thus, visualizing time-varying electromagnetic fields presents a somewhat different problem than that of visualizing fluid flow since the vector field and the direction of motion of the field lines are, in general, independent. In addition, unlike in fluid flow models where we are interested in path lines or

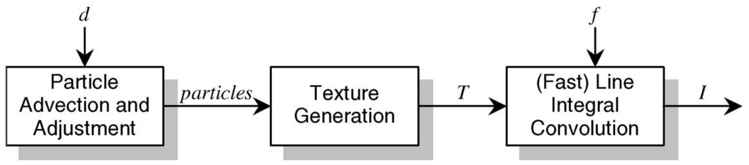
d
Particle Advection and Adjustment
particles
Texture Generation
T
f
(Fast) Line Integral Convolution
I

Fig. 1. Organization of DLIC.

streaklines, in electromagnetism, we are interested in visualizing the instantaneous field streamlines and how they evolve over time.

## 3 Dynamic Line Integral Convolution

### 3.1 Problem Formulation

We would like to visualize a dynamic vector field $\boldsymbol{f}$ in full generality, with $\boldsymbol{f}$ being an arbitrary mapping from some domain $D$ and a time interval $T$ to $R^2$. In addition, we describe the evolution of the field $\boldsymbol{f}$ via a motion vector field $\boldsymbol{d}$, where the value of $\boldsymbol{d}$ at a particular point in space and time is the instantaneous velocity of the streamline at the corresponding point of the vector field $\boldsymbol{f}$. Note that we are not attempting to visualize motion in a velocity field $\boldsymbol{f}$, as is typically the case in computational fluid dynamics (CFD), but rather the evolution of the streamlines in $\boldsymbol{f}$ as prescribed by $\boldsymbol{d}$. The field $\boldsymbol{f}$ alone is not sufficient to describe the situations we are interested in. For example, consider the vector field $\boldsymbol{f} = \hat{\boldsymbol{x}}$, where the horizontal streamlines are to translate vertically over time. We denote this movement via a motion field such as $\boldsymbol{d} = v\hat{\boldsymbol{y}}$. As a consequence, this algorithm may not be immediately applicable to CFD, where we typically do not even have a motion field $\boldsymbol{d}$.

Each image of the animation sequence must be an accurate and detailed depiction of the vector field $\boldsymbol{f}$ at a particular time. We wanted the images to resemble those produced by the LIC method, which achieves a very high level of detail. When the animation is viewed as a sequence, it should be clear how the streamlines evolve over time from the interframe coherence. Though some regions of the vector field may expand or contract over the course of the animation, we would like the level of detail to remain high throughout the image at all times. Finally, the rendering algorithm must be fairly time-efficient since hundreds of frames will need to be generated for a single sequence.

### 3.2 Algorithm Overview

In the following sections, we describe a new algorithm, Dynamic Line Integral Convolution (DLIC), that animates a time-varying vector field represented by the $\boldsymbol{f}$ and $\boldsymbol{d}$ fields described above. The method consists of three major components, as illustrated in Fig. 1. The sequence of computations begins with the *Particle Advection and Adjustment* stage, continues through the *Texture Generation* stage, and finishes with the *Fast Line Integral Convolution* stage. In the first stage, the algorithm tracks a large number of particles as they evolve over time according to $\boldsymbol{d}$, automatically consolidating excess particles and adding new particle detail where needed. In the second stage, these particles are used to generate an input texture, which in the final stage is passed to a modified FLIC algorithm to produce the output.

To conceptually understand what the algorithm is doing, first we notice that the LIC technique can be naturally extended to depend on a time-varying input texture:

$$I(\boldsymbol{x}, t) = \int T\big(\hat{\boldsymbol{\sigma}}_t^s(\boldsymbol{x}), t\big)\kappa(s)ds. \tag{9}$$

Ideally, at each time step, we would like this texture to resemble the white noise used in LIC vector field visualizations. This way, the resulting output will have as high contrast as possible perpendicular to the streamlines. At the same time, we need to make sure that the correspondence between streamlines evolves from frame to frame as prescribed by $\boldsymbol{d}$. We do this by warping the input texture over successive time steps via the mapping defined by $\boldsymbol{d}$. That is, as a first approximation for small $\Delta t$, we could use

$$T(\boldsymbol{x}, t + \Delta t) = T(\boldsymbol{x} - \boldsymbol{d}(\boldsymbol{x}, t + \Delta t)\Delta t, t). \tag{10}$$

Unfortunately, since the texture is typically stored as a discrete set of samples on an ordered grid, repeatedly warping and filtering it over time results in a loss of detail. In addition, the motion field $\boldsymbol{d}$ may have divergent or convergent regions, spreading out or compressing the detail. Finally, the texture mapping falls apart at the edge of the domain $D$ where the direction of motion given by $\boldsymbol{d}$ is pointing in.

To avoid these problems, instead of evolving the input texture according to equation (10), the DLIC algorithm tracks a large number of particles of random intensity, roughly on the same order as the number of pixels in the original input texture. As the particles move over time according to $\boldsymbol{d}$, the algorithm continuously monitors and adjusts their distribution to keep the level of detail roughly the same. This is the *Particle Advection and Adjustment* stage, which we describe in more detail below. At any instant of time for which we want to generate a frame of the animation, the texture at that time is generated by simply drawing all of the particles onto it. This is the *Texture Generation* stage. Once we have the texture for a given frame, the FLIC method is applied to this texture to render the streamlines at that time, in the *Fast Line Integral Convolution* stage. Although it may seem that tracking a large number of particles is expensive, its cost is substantially less than the time required for FLIC to run.

Intuitively, since the particles that produce the input texture advect according the motion field $\boldsymbol{d}$, the LIC convolutions along the time-evolving streamlines of $\boldsymbol{f}$ from one frame to the next sample the same particles as they move. Therefore, the streamlines of $\boldsymbol{f}$ appear to move from one frame to the next as we expect them to, that is, according to the motion field $\boldsymbol{d}$. Assuming we can maintain a roughly uniform distribution of particles over the texture and that their intensities are random and uncorrelated, the texture input to FLIC at each point in time will look like white noise. Therefore, each output image in the sequence will individually have the same properties as a static LIC rendering, but successive frames will have an interframe coherence of streamlines that depicts their motion.

## 3.3 Particle Advection and Adjustment

We now give more details about each of these stages, beginning with the particle stage. For each particle, we need to maintain its two-dimensional position $\boldsymbol{p}_i$ and its intensity $a_i$. The particles' positions need to have subpixel accuracy in order to track them accurately, so we use floating-point numbers to store them. Before the very first frame is rendered, we need to generate a complete new set of particles. To ensure a good distribution, we create one particle at the center of output image pixel, jittered slightly. The particle intensities are chosen randomly, ensuring that the first texture is essentially white noise. Ideally, there would be one particle used to generate each texture pixel, but, in reality, the density varies over time depending how the particle distribution evolves.

For each successive frame separated by time $\Delta t$, the particles are advected according to the motion field $\boldsymbol{d}$. Though more sophisticated methods can certainly be employed, for simplicity, we use a simple Euler integrator to move the particles, that is:

$$\boldsymbol{p}_i(t+\Delta t) = \boldsymbol{p}_i(t) + \boldsymbol{d}(\boldsymbol{p}_i(t), t)\Delta t. \tag{11}$$

In some applications, it may help to use a higher-order integrator, but, in our examples, the particles do not move very far over the time interval $\Delta t$. Hence, any errors between successive frames would be too small to notice, especially after the LIC stage filters the results. Errors accumulated over long periods of time are not perceived since the streamlines themselves evolve quite a bit over that time, creating and destroying detail along the way.

After the particles have been moved, we analyze the distribution of the particles and make adjustments as necessary. As a first step, we delete any particles that leave the bounds of the texture in order to save computational time, even though it is possible that the particles will return at a later time. Next, we compute a texture coverage map, which, in our implementation, is actually done in parallel with the texture generation stage since they share some of the same calculations. This "texture coverage" $T_C$ has the same resolution as the input texture and is used to determine how many particles cover each texture pixel. The texture coverage is the result of a sum of convolutions of the particles over the pixels, given by

$$T_C(\boldsymbol{x}, t) = \sum_i \iint A(\boldsymbol{x} - \boldsymbol{p}_i(t) + \boldsymbol{y})\chi(\boldsymbol{y})d\boldsymbol{y}. \tag{12}$$

Here, $A(\boldsymbol{y})$ describes the "shape" of the particle and $\chi(\boldsymbol{y})$ is the response function of a texture pixel. This convolution represents a fully general resampling of continuous particles onto the discrete texture grid. The result of this process is illustrated in Fig. 2.

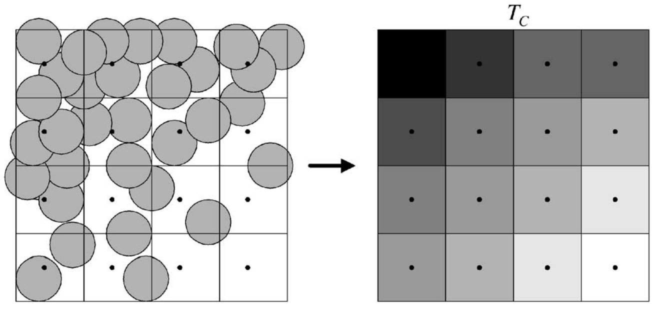

Fig. 2. Texture coverage.

To simplify the computation in our implementation, the particles are given a square shape and the pixels have a square, uniform response. In other words, $A(\boldsymbol{y})$ and $\chi(\boldsymbol{y})$ are both 1 inside the unit square and 0 otherwise. This simplification allows us to compute the contributions of the particles very quickly via bilinear interpolation coefficients.

Using the texture coverage, we can easily find regions where the particle density deviates significantly from the norm. In our algorithm, we limit the per-pixel coverage to a maximum of 2.0 and a minimum of 0.5 particles. When the particle density in a pixel is too high by this definition, we are tracking more particles than necessary, so our algorithm consolidates several particles into a single particle. Since we have computed both the texture coverage and the texture itself at the same time, we know the intensity of the texture pixel whose coverage exceeds the maximum. First, we simply delete all the particles close to the pixel center that would have contributed significantly to the pixel. Next, we create a new particle located near the pixel center whose intensity $a_i$ is equal to that of the texture, thus combining the intensities of the old particles that produced that pixel. This process is illustrated in Fig. 3. Although one would expect to do some sort of center-of-mass averaging, this simple method is sufficient for DLIC.

Where the particle density is too low by the above definition, either the field motion is diverging, or the texture has no mapping in that region (for example, at an edge of the domain $D$ where the particles are moving in). In both cases, we add new detail by creating completely new particles with random intensities. In unmapped regions, the texture values are not constrained in any way, so, without any additional knowledge, we are clearly justified in assuming that white noise fills those areas. When the field is diverging, however, adding random detail may not seem reasonable. It turns out that adding new, random particles is a very simple solution that yields satisfactory results without exhibiting any surprising appearances of new features in those regions. This is because the number of particles created is generally small compared to the number of particles averaged together when the texture is fed through LIC. The particle creation process is depicted in Fig. 4.

The two simple operations of consolidation and creation described above succeed in maintaining a roughly uniform distribution of particles across the texture. The randomness in their intensities results in a texture with uncorrelated white noise, yielding good results when LIC is applied. There do exist scenarios where these particle adjustment techniques do

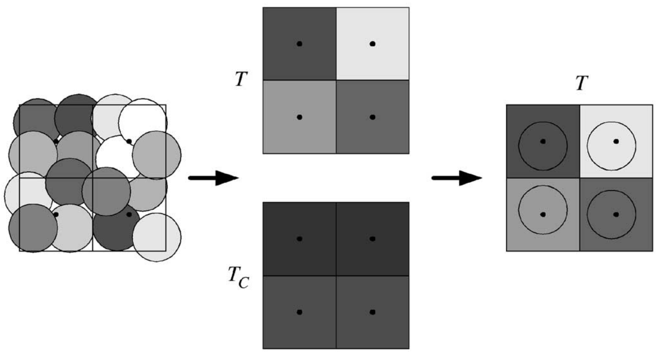

Fig. 3. Particle merging.

not work well, for example, with a motion field $\boldsymbol{d}$ that is uniformly contracting. In this case, particles average together excessively, lowering the contrast of the texture. Although new, high-contrast detail will be constantly created near the edges, toward the center of contraction the texture (and resulting output image) will appear grayed-out. Nonetheless, in our experience, when we have applied these particle adjustment methods to examples in electromagnetism, the results were generally good.

### 3.4 Texture Generation

Similar to the texture coverage computation, the texture image itself at any point in time is calculated from the particles via a sum of convolutions of the particles scaled by their intensities over the pixels, given by

$$T(\boldsymbol{x}, t) = \sum_i \iint a_i A(\boldsymbol{x} - \boldsymbol{p}_i(t) + \boldsymbol{y}) \chi(\boldsymbol{y}) d\boldsymbol{y}. \tag{13}$$

In our implementation, we use square profiles to simplify the computation. As a result, the algorithm to compute the texture simply iterates through the particles, summing the bilinear contributions scaled by the particle intensity to the four surrounding pixels. The result of this computation is illustrated in Fig. 5.

If the particles are distributed evenly and their intensities are random, the texture generated will resemble white noise. However, since we allow the texture coverage to range between 0.5 and 2.0 particles per pixel, the density of particles will most likely not remain completely uniform. Because the texture is computed as a sum over the particles, the distribution of intensities in the texture will vary with the density of particles. As it turns out, this is not a serious problem because we constrain the coarse distribution of particles, as described in the previous section. However, we have tested various schemes to better adjust for the varying particle density. One method is to divide the value of the texture at each pixel by the texture coverage. This confines the possible pixel intensities to the fixed range of $a_i$:

$$T'(\boldsymbol{x}, t) = \frac{T(\boldsymbol{x}, t)}{T_C(\boldsymbol{x}, t)}. \tag{14}$$

Another method is to use the "square coverage" of each pixel:

$$\begin{aligned} T_S(\boldsymbol{x}, t) &= \sum_i \left[ \iint A(\boldsymbol{x} - \boldsymbol{p}_i(t) + \boldsymbol{y}) \chi(\boldsymbol{y}) d\boldsymbol{y} \right]^2 \\ T'(\boldsymbol{x}, t) &= \frac{T(\boldsymbol{x}, t)}{\sqrt{T_S(\boldsymbol{x}, t)}}. \end{aligned} \tag{15}$$

This results in a texture with uniform variance. As it turns out, none of these methods produce results that are significantly better or worse than the others. In our implementation, since we already compute the texture coverage $T_C$ for the purpose of adjusting the particles, we use this value to normalize the texture pixels.

### 3.5 Fast Line Integral Convolution

Once we have generated the texture for the current time step from the particles, we use the LIC technique on the texture, yielding an image of the streamlines. Care must be taken to sample the image uniformly or the animation sequence will exhibit temporal aliasing. The particle profiles are first sampled to create the texture and then we resample them a second time to produce the output image with LIC. Ideally, we would sample the particles only once while integrating along the streamlines, but this is difficult to do efficiently. By creating the texture as an intermediary step, we can speed up the computation tremendously, especially when we make use of the FLIC optimization.

When implementing LIC, we have to choose which points of the output intensity $I(\boldsymbol{x}, t)$ to sample for each output pixel. For good but extremely slow results, we could take a large number of samples with a Gaussian distribution around the pixel center. Sampling only once uniformly at the pixel centers often produces identical results when the input texture does not have too high a frequency extent. In fact, for vector fields that vary slowly, we could prefilter the

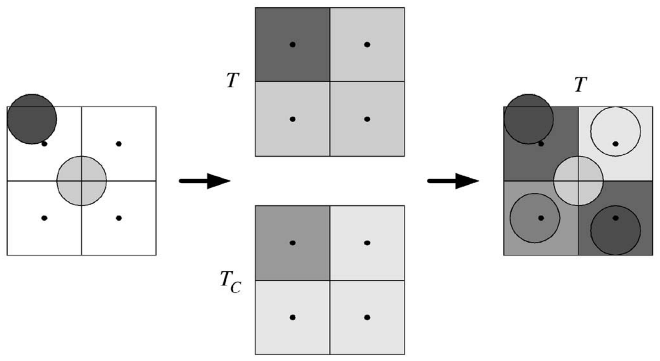

Fig. 4. Particle creation.

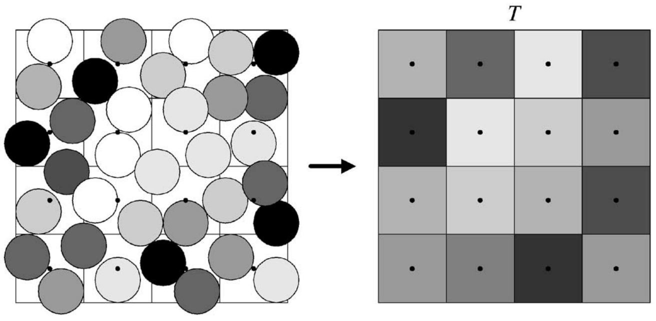

Fig. 5. Texture generation from the particles.

input texture with a Gaussian kernel to produce the same effect as using such a kernel to sample the output intensity. To avoid repeating artifacts, we can simply jitter the sample points. Whatever sampling pattern is chosen, we need to be consistent over the animation sequence, for any variation is very disruptive to the motion. For example, jittering the sample points should be done with a pseudorandom number generator so that we can reproduce the same jitter offsets for every frame. Prefiltering the input texture is a good way to control the amount of frame-to-frame sampling discrepancy.

Instead of sampling relative to the pixel centers, one alternative is to sample according to the positions of the particles we are tracking. This turns out to be especially important when we use the FLIC method. By the nature of the algorithm, we cannot choose all the output image sample points. Instead, we pick seed points that are used as starting points when following the streamlines incrementally. If the same set of starting points is used for every frame, then, as the particles and streamlines move, the animation will exhibit temporal sampling artifacts. To avoid this, we pick a random ordering of the particles and use those as starting points. As the particles move over time with the streamlines, so do the starting points, greatly reducing the amount of temporal sampling artifacts. When new particles are created, they are simply added randomly to the end of the ordering to minimize the effect of their sudden appearance. Particles that are deleted are simply removed from the list. This method allows us to use FLIC to render the animation sequence, which yields an order-of-magnitude increase in computational speed.

## 4 Application to Electromagnetism

### 4.1 Computing the Field Line Motion

In order to apply DLIC to time-varying electromagnetic fields, we need to compute the $f$ and $d$ vector fields. Note that, though the fields are three-dimensional in nature, we typically pick a planar "slice" to visualize using DLIC. The $f$ fields are usually straightforward to compute from the time-varying sources for the electric or magnetic fields by applying the methods of classical electromagnetism. However, it is not as clear how to define the motion of the field lines in electromagnetism and therefore how to compute the motion field $d$. Belcher and Olbert [2] give a (nonunique) definition for the motion of the electromagnetic field lines as the drift velocity of hypothetical low-energy test monopoles placed in that field. For example, in quasi-static magnetic fields, when we place low-energy test electric charges in the magnetic field, it can be shown that they will drift with a local velocity given by

$$v_{drift} = \frac{E \times B}{B^2}. \tag{16}$$

In this equation, $E$ is the electric field that arises due to the time changing magnetic field $B$. Note this motion is perpendicular to the streamlines in $B$. Similarly, in quasi-static electric fields, when we place hypothetical low-energy test magnetic monopoles in the electric field, it can be shown that they will drift with a local velocity given by

$$v_{drift} = c^2 \frac{E \times B}{E^2}, \tag{17}$$

where $c$ is the speed of light. In this equation, $B$ is the magnetic field that arises due to the time changing electric field $E$ and electric charge motions.

In quasi-static situations, these definitions yield a natural and physical interpretation of the motion of the electric or magnetic field and of the field lines. In situations that are not quasi-static (e.g., dipole radiation in the induction or radiation zones), the drift velocities given above may approach or even exceed the speed of light, so the simple physical interpretation of field motion as the drift of test monopoles breaks down. However, even in these cases, we can still choose the velocity field defined by (16) or (17) as the $d$ field. In fact, this choice for $d$ is still useful in that the motion so defined is in the direction of electromagnetic energy flow, parallel to the local Poynting vector. These issues are discussed at length in Belcher and Olbert [2], and we will not elaborate on them further here.

### 4.2 Examples

In addition to the image sequences presented in this article, complete animations for the examples discussed below are available on the web at: http://web.mit.edu/jbelcher/www/DLIC.html.

Our first example of DLIC applied to electromagnetism is for an electric charge moving against a uniform field. The charge is initially moving against the field and, as time progresses, the charge slows down, comes to rest, and then reverses direction. Fig. 6 shows an LIC representation of the $f$ field at one instant of time. We see that the electric field lines of the point charge, although radially close to the charge, are swept upward by the uniform electric field as we move away from the charge. Fig. 7 shows an LIC representation of the associated $d$ field at the same instant of time. We see that, near the charge, the field lines move vertically along with the movement of the charge. Far away from the charge, in the region dominated by the uniform field, the $d$ field causes the electric field lines to be "pushed" apart horizontally as the charge moves downward. Both these effects can be seen in the sequence of frames given in Fig. 8, as well as in the animation. The evolution of the uniform field lines is easy to interpret as they slide apart and come together again. Notice that, even though the field lines are compressed as the charge moves down, when they expand again, detail is created to fill in the diverging regions. The field lines near the charge, however, seem to "wrap out" from the charge as it moves down, rather than slide down with it as we would expect. It should be understood that our $d$ field is a description of how test charges would move in the field, which does not necessarily correspond with how we would intuitively expect points along a streamline to move.

Next, we turn to a quasi-static example in magnetostatics. The field sources consist of a conducting, nonmagnetic ring and a permanent magnetic dipole constrained to move along the ring's axis, with its dipole moment vector pointing along the same axis. We drop the dipole from above the center of the ring and, as it falls, the resultant change in magnetic flux through the ring gives rise to eddy

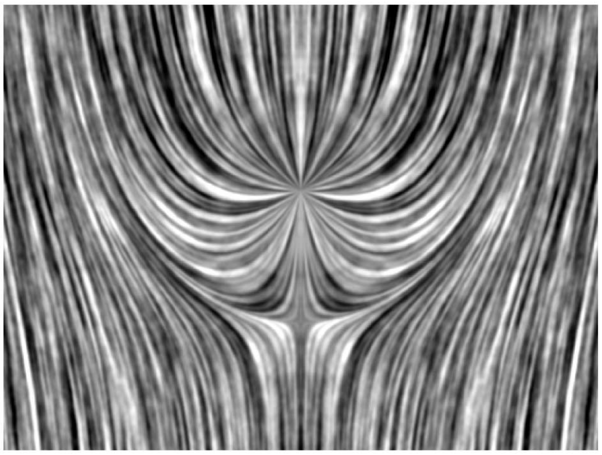
Fig. 6. The electric field configuration for an electric charge moving in a uniform electric field which points upward.

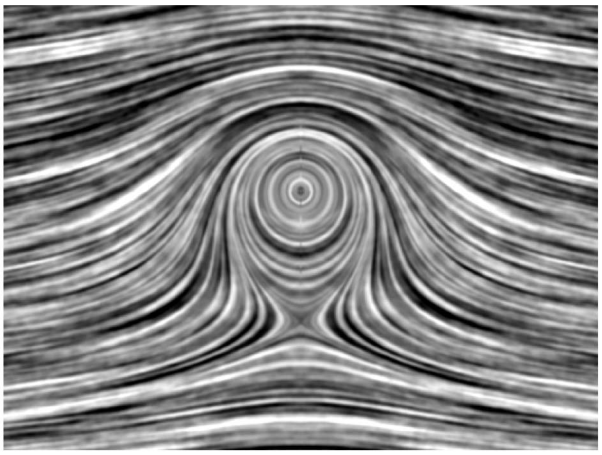
Fig. 7. The motion vector field $d$ corresponding to Fig. 6.

currents in the ring. These currents produce a magnetic field that tends to retard the motion of the dipole. Fig. 9 and Fig. 10 show two sequences of frames from an animation, first when the dipole is about to fall through the ring and second when the dipole has fallen through the ring. Though the frame-to-frame coherence is evident in the sequences from the evolution of its features, viewing the animation gives the best understanding of how the field lines move. Once again, even though the field lines around the magnet are "squeezed" into the ring, as they expand once again there is always enough detail created to ensure a high-contrast field line representation.

One area where DLIC needs improving is in dealing with singularities in the field. The animation of the falling magnet shows minor flaws near the center of the dipole, causing a slight flickering effect. Along the ring's axis, the motion field $d$ evaluates to zero, so the particles along the center streamline do not move at all. In some senses, this is the correct behavior since translating the streamline vertically along itself does not affect its identity. Unfortunately, as the center of the dipole falls, it overlaps a new set of stationary particles every frame. The lack of correlation between these particles is propagated visually to the radial streamlines emanating from the dipole. One way to reduce this flickering might be to diminish the amount that particles near the singularity contribute to the streamline convolution. Perhaps another way to solve this would be to use a slightly different motion field $d$ that distributes particles more naturally along the same streamline.

We now consider a non-quasi-static example. In this example, we animate the electric field associated with an electric dipole oscillating sinusoidally, including the induction and radiation zone fields. Fig. 11 shows a set of frames from an animation sequence of the time-dependent electric field of this oscillating dipole. The animation video is a wonderful demonstration of the radiating effect of an oscillating dipole and the interaction between the three zones of electromagnetic behavior. The streamline patterns retain their characteristics as they expand in a sphere from the dipole center, while, at the same time, new detail is being created continuously to fill the space. We can see a small amount of incoherent behavior near the center of the radiating wave. Once again, the $d$ field is partly at fault because it evaluates to zero at such radii and, consequently, the input texture does not move with the streamlines as it should. Near the singularity, only a very small subset of texture pixels are sampled, so any change in texture value is immediately apparent.

This animation has been made periodic as a postprocessing step so that it can be looped seamlessly. Normally, this is not possible with DLIC since entirely new, random particles are created as the radiation expands away from the dipole. Although both the $f$ and $d$ fields match at time

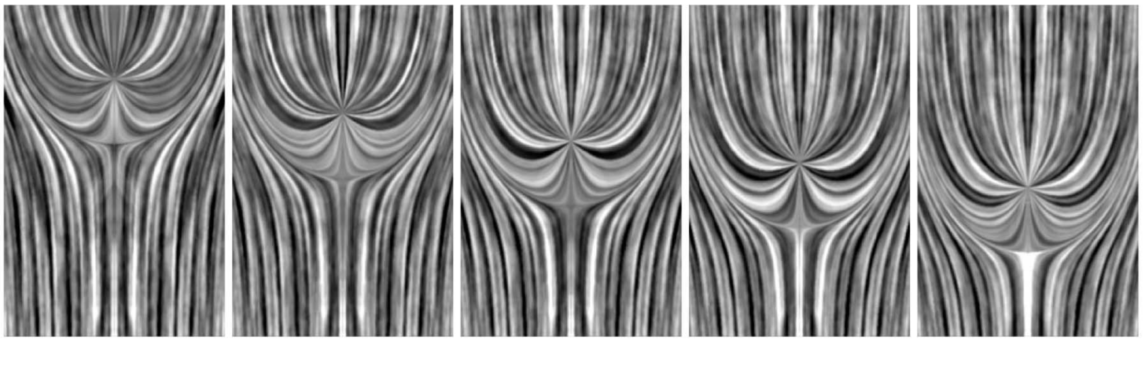
Fig. 8. Five frames of a DLIC animation of an electric charge moving downward in a uniform electric field that points upward.

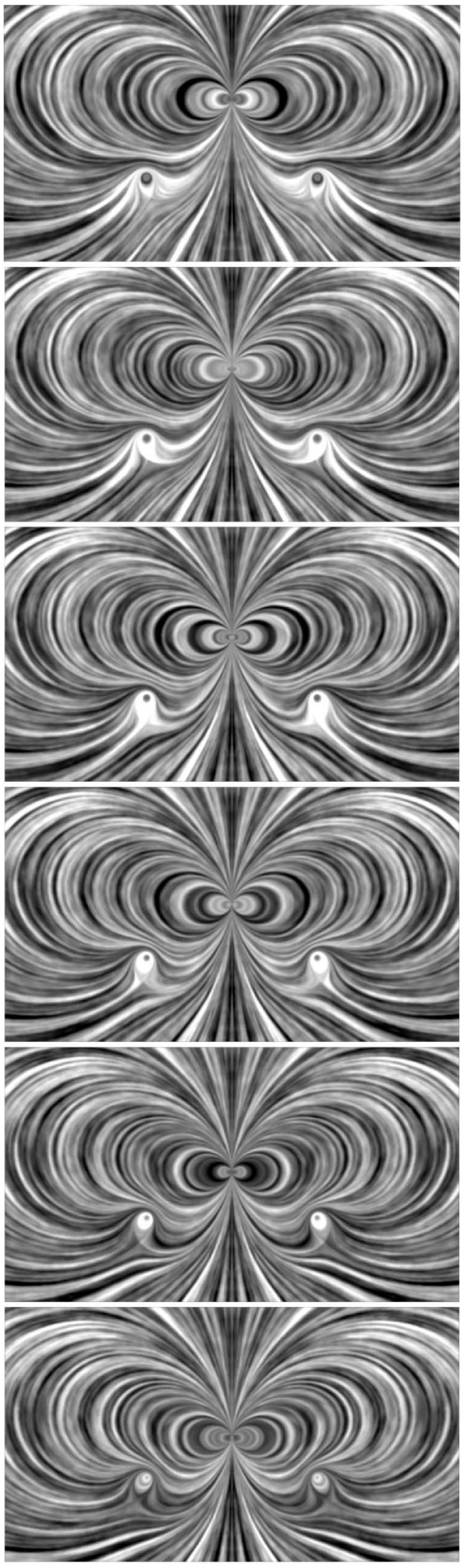
Fig. 9. Six frames of a DLIC animation of a magnet about to fall through a ring.

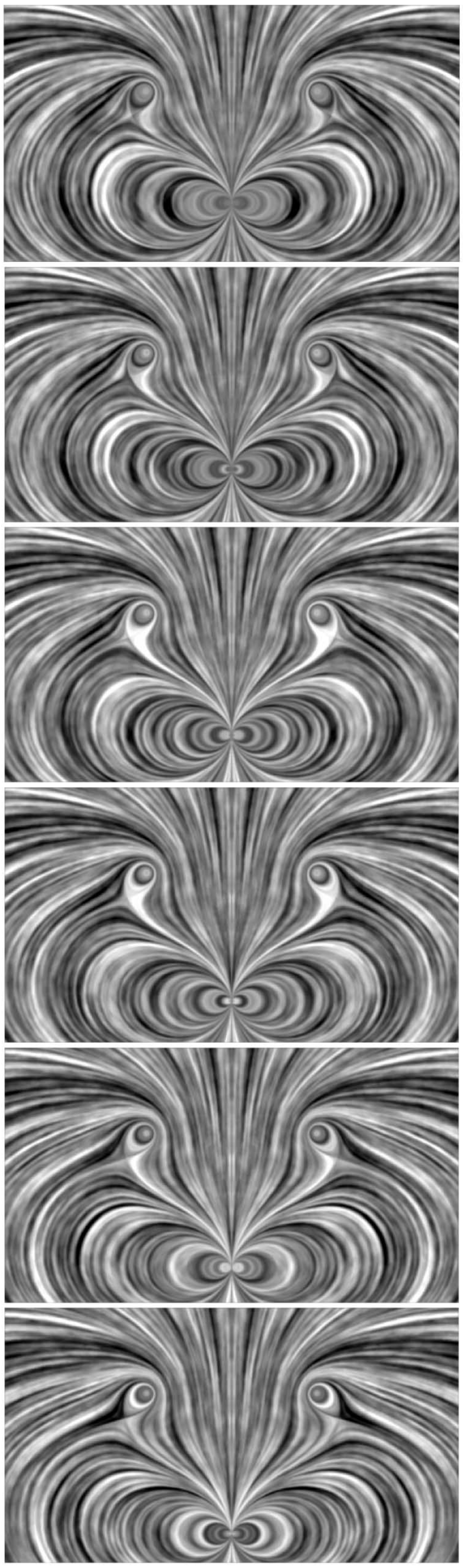
Fig. 10. Six frames of the magnet after falling through the conducting ring.

intervals of the period $T$, the particles will have completely different intensities $a_i$. Fortunately, we can rectify this by smoothly blending between pairs of images separated by time $T$. That is, we first render the animation sequence over a time $T + \tau$, where $\tau$ is some blending interval. We then create a new animation sequence so that:

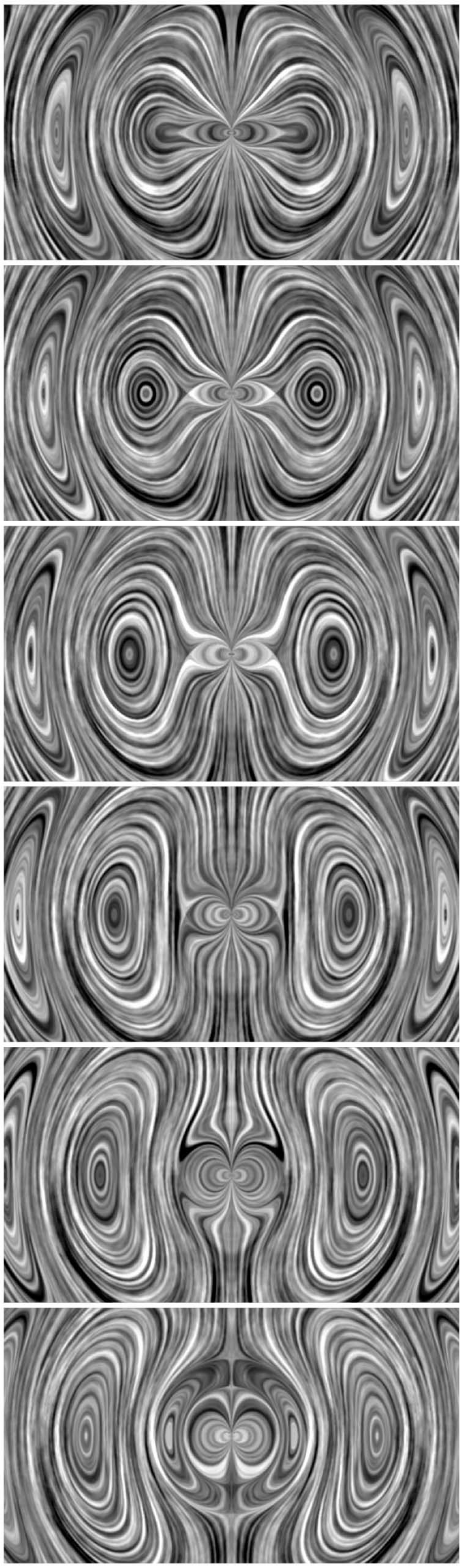
Fig. 11. Six frames of the DLIC animation of electric dipole radiation for a sinusoidally varying electric dipole moment.

$$I'(\boldsymbol{x},t) = \begin{cases} \gamma(\alpha)[\alpha I(\boldsymbol{x},t) + (1-\alpha) I(\boldsymbol{x}, T+t)], & 0 < \alpha = \frac{t}{\tau} < 1 \\ I(\boldsymbol{x},t), & \text{otherwise.} \end{cases} \quad (18)$$

The parameter $\alpha$ varies linearly from 0 to 1 over the time $\tau$, shifting from the image at time $T+t$ to the corresponding image at time $t$. An additional function $\gamma(\alpha)$ is used to renormalize the contrast of the blend. The overall result of this process is an animation that can be looped since the end frames now transition seamlessly into the beginning frames, with no discontinuity in pattern.

## 5 Summary

Given two vector fields $\boldsymbol{f}$ and $\boldsymbol{d}$, where $\boldsymbol{f}$ is the time-varying vector field we would like to visualize and $\boldsymbol{d}$ describes the motion of the field lines in $\boldsymbol{f}$, the *Dynamic Line Integral Convolution* algorithm described here produces a spatially detailed animation depicting the time evolution of the field. We track a large number of random particles that move according to $\boldsymbol{d}$ and use them to create the input texture for FLIC. We are able to retain the high-contrast properties of the input texture by adjusting the particles over time to maintain a random distribution of intensities. Every frame of the resulting animation is similar to the results of a static LIC rendering and, together, the frames exhibit a temporal correlation that depicts the time-evolution of the field lines. In this paper, we have detailed the methods we use to advect the particles and adjust their distribution, generate the texture, and, finally, apply the FLIC algorithm to render the frame.

Unfortunately, the workings of DLIC require a motion field $\boldsymbol{d}$, which is not available in all dynamic vector field applications. In actuality, this is a property of the visualization problem itself. For example, in fluid dynamics, the $\boldsymbol{f}$ field characterizes both the direction and motion of the field, while, in electrodynamics, $\boldsymbol{f}$ only describes the direction of the field, not its motion. For our purposes, we chose to describe this motion via a secondary field $\boldsymbol{d}$. We welcome further research into methods for representing vector field dynamics in other applications.

The success of DLIC can be measured by how well the animation sequences reflect the evolution of the sample fields. For the examples in this article, all taken from electromagnetism, the animations give a good understanding of how the structure of the electromagnetic field evolves with time, with the high spatial resolution characteristic of LIC. Indeed, to our knowledge there is no other existing method for depicting the time evolution of electromagnetic fields at anything approaching this level of detail.

Although the animation sequences in this paper demonstrate the successful application of DLIC to time-varying vector fields, the algorithm could stand to be improved in some areas. As described in the previous section, when dealing with singularities, the frame-to-frame coherence is not as robust as we would like, sometimes producing a slight flickering near the singularity. Though part of the reason is that the $\boldsymbol{d}$ field does not advect particles near the singularity as we would intuitively expect, it might be possible for the algorithm to detect this problem and adapt to it. It is also possible that a rendering algorithm different from FLIC, for example, one that scatters the texture values to the output instead of collecting them via a convolution, may produce higher quality results. Another area that needs more exploration is the techniques used to advect the particles and adjust their distribution. Although the simple methods we have employed work well, there may be other ways that produce even better results. Finally, although the use of FLIC instead of LIC as the rendering core provides a significant speedup, it is still very time-consuming to render a lengthy sequence. Accelerating the computations is highly desirable and would make it an even more powerful tool for exploring the dynamics of time-varying vector fields.

## Acknowledgments

This paper is based on an MIT master's thesis in engineering in electrical engineering and computer ecience and eenior thesis in physics entitled *Dynamic Line Integral Convolution For Visualizing Electromagnetic Phenomena* [14]. The author would like to thank Professor John Belcher for suggesting that he investigate the extension of the Line Integral Convolution technique to time-varying electromagnetic fields and for discussions about how to define motion in such fields. He is also indebted to Andrew McKinney and Mark Bessette of the MIT Center for Educational Computing Initiatives for technical support. This work is supported by US National Science Foundation Grant #9950380, "Force Field: Using Animation In Teaching Electromagnetism." It is part of a larger effort at MIT funded by the d'Arbeloff Fund For Excellence In MIT Education, the MIT/Microsoft iCampus Alliance, and the MIT School of Science and Department of Physics to reinvigorate the teaching of freshman physics using advanced technology and innovative pedagogy.

## References

[1] J. Becker and M. Rumpf, "Visualization of Time-Dependent Velocity Fields by Texture Transport," *Proc. Visualization in Scientific Computing '98, Eurographics Workshop,* pp. 91-101, 1998.

[2] J. Belcher and S. Olbert, "Field Line Motion in Classical Electromagnetism," *Am. J. Physics,* submitted, 2001.

[3] B. Cabral and C. Leedom, "Imaging Vector Fields Using Line Integral Convolution," *Proc. SIGGRAPH '93,* pp. 263-270, 1993.

[4] L. Forssell and S.D. Cohen, "Using Line Integral Convolution for Flow Visualization: Curvilinear Grids, Variable-Speed Animation, and Unsteady Flows," *IEEE Trans. Visualization and Computer Graphics,* vol. 1, no. 2, pp. 133-141, June 1995.

[5] W. Freeman, E. Adelson, and D. Heeger, "Motion without Movement," *ACM Computer Graphics,* vol. 25, no. 4, pp. 27-30, 1991.

[6] B. Jobard, G. Erlbacher, and Y. Hussaini, "Hardware-Accelerated Texture Advection for Unsteady Flow Visualization," *IEEE Proc. Visualization 2000,* 2000.

[7] B. Jobard, G. Erlbacher, and Y. Hussaini, "Lagrangian-Eulerian Advection for Unsteady Flow Visualization," *IEEE Proc. Visualization 2001,* 2001.

[8] B. Jobard and W. Lefer, "The Motion Map: Efficient Computation of Steady Flow Animations," *IEEE Proc. Visualization '97,* pp. 323-328, 1997.

[9] N. Max and B. Becker, "Flow Visualization Using Moving Textures," *Proc. ICASE/LaRC Symp. Visualizing Time Varying Data,* Nov. 1995.

[10] H.-W. Shen and D. Kao, "UFLIC: A Line Integral Convolution for Visualizing Unsteady Flows," *IEEE Proc. Visualization '97,* pp. 317-322, 1997.

[11] H.-W. Shen and D. Kao, "A New Line Integral Convolution Algorithm for Visualizing Time-Varying Flow Fields," *IEEE Trans. Visualization and Computer Graphics,* vol. 4, no. 2, pp. 98-108, Apr.-June 1998.

[12] D. Stalling and H.-C. Hege, "Fast and Resolution Independent Line Integral Convolution," *Proc. SIGGRAPH '95,* pp. 249-256, 1995.

[13] D. Stalling, "Fast Texture-Based Algorithms for Vector Field Visualization," doctoral dissertation, Dept. of Scientific Visualization, Konrad-Zuse-Zentrum Berlin (ZIB), 1998.

[14] A. Sundquist, "Dynamic Line Integral Convolution for Visualizing Electromagnetic Phenomena," master's thesis, Dept. of Electrical Eng. & Computer Science, Massachusetts Ins. of Technology, May 2001.

[15] R. Wegenkittl, E. Gröller, and W. Purgathofer, "Animating Flowfields: Rendering of Oriented Line Integral Convolution," *Proc. Computer Animation '97,* pp. 15-21, June 1997.

**Andreas Sundquist** received the MEng degree in electrical engineering and computer science and BS degrees in EE&CS, physics, and mathematics from the Massachusetts Institute of Technology in 2001. His interests include computer architecture and visualization, particularly in application to physical phenomena.