EUROGRAPHICS 2002 / G. Drettakis and H.-P. Seidel
(Guest Editors)

Volume 21 (*2002*), *Number 3*

Hardware Accelerated Interactive Vector Field Visualization: A level of detail approach

Udeepta Bordoloi and Han-Wei Shen

E-mail: {bordoloi,hwshen}@cis.ohio-state.edu
Department of Computer and Information Sciences, The Ohio State University, Columbus, Ohio, USA

Abstract

This paper presents an interactive global visualization technique for dense vector fields using levels of detail. We introduce a novel scheme which combines an error-controlled hierarchical approach and hardware acceleration to produce high resolution visualizations at interactive rates. Users can control the trade-off between computation time and image quality, producing visualizations amenable for situations ranging from high frame-rate previewing to accurate analysis. Use of hardware texture mapping allows the user to interactively zoom in and explore the data, and also to configure various texture parameters to change the look and feel of the visualization. We are able to achieve sub-second rates for dense LIC-like visualizations with resolutions in the order of a million pixels for data of similar dimensions.

Categories and Subject Descriptors (according to ACM CCS): I.3 [Computer Graphics]: Applications
Keywords: Vector Field Visualization, Flow Visualization, Level of Detail, Line Integral Convolution, LIC

1. Introduction

Study, and hence, visualization of vector fields is an important aspect in many diverse disciplines. Visualization techniques for vector fields can be classified into local techniques and global ones. Examples of local techniques include particle traces, streamlines, pathlines, and streaklines, which are primarily used for interactive data exploration. Global techniques such as line integral convolution(LIC)[1] and spot noise[2], on the other hand, are effective in providing global views of very dense vector fields. These techniques are classified as global techniques because directional information at every point of the field are displayed, and the only limitation is the pixel resolution. The price for the rich information content of the global methods, however, is their high computational cost, which makes interactive exploration difficult.

We are concerned with the problem of interactive visualization of very dense two-dimensional vector fields with flexible level of detail controls. Even though the processor speeds have increased significantly in recent years, a global visualization technique for dense vector fields which realizes interactive rates still eludes us. Likewise, although large format graphics displays that consist of tens of millions of pixels have become increasingly available, our ability to generate very high resolution visualization images at an interactive rate is clearly lacking. Furthermore, currently very few global vector field visualization techniques allow the user to freely zoom into the field at various levels of detail. This capability is often needed when the size of the original vector field exceeds the graphics display resolution. Finally, most of the existing global techniques do not allow a flexible control of the output quality to facilitate either a fast preview or a detailed analysis of the underlying vector field.

In this paper we introduce an interactive global vector field visualization technique aiming to tackle the above problems. The performance goal of our method is set to produce dense LIC-like visualizations with resolutions in the order of a million pixels within a fraction of second. This is accomplished in part by utilizing graphics hardware acceleration, which allows us to increase the output resolution without linearly increasing the computation time. Additionally, our algorithm takes into account both a user-specified error tolerance and image resolution dependent criteria to adaptively select different levels of detail for different regions of the vector field. Moreover, the texture-based nature of our algo-

rithm allows the user to configure various texture properties to control the final appearance of the visualization. This feature provides extra flexibility to represent the directional information, as well as other quantities in various visual forms.

This paper is organized as follows. We first discuss previous relevant work (section 2) and present the level-of-detail algorithm for steady state flow (section 3). We discuss the level of detail selection process (section 4). The hardware acceleration issues are are presented next (section 5), followed by the results and discussion(section 6).

2. Related Work

Texture based methods are the most popular techniques for visualization of dense vector fields. Spot noise, proposed by Van Wijk[2], convolved a random texture along straight lines parallel to the vector field. Another popular technique is the Line Integral Convolution (LIC). Originally developed by Cabral and Leedom[1], it uses a white noise texture and a vector field as its input, and results in an output image which is of the same dimensions as the vector field. Stalling and Hege[3] introduced an optimized version by exploiting coherency along streamlines. Their method, called 'Fast LIC', uses cubic Hermite-interpolation of the advected streamlines, and optionally uses a directional gradient filter to increase image contrast. Forssell[4] applied the LIC algorithm to curvilinear grids. Okada and Kao[5] used post-filtering to increase image contrast and highlight flow features. Forssell[4] and Shen *et al.*[6] extended the technique to unsteady flow fields. Verma *et al.*[7] developed an algorithm called 'Pseudo LIC' (PLIC) which uses texture mapping of streamlines to produce LIC-like images of a vector field. They start with a regular grid overlaying the vector field grid, but they compute streamlines only over grid points uniformly sub-sampled from the original grid. Jobard and Lefer[8] applied texture mapping techniques to create animations of arbitrary density for unsteady flow.

Level-of-detail algorithms have been applied in various forms to almost all areas of visualization, including flow visualization[9],[10]. Cabral and Leedom[11] used an adaptive quad-subdivision meshing scheme in which the quads are recursively subdivided if the integral of the local vector field curvature is greater than a given threshold. We use a similar subdivision for our level-of-detail approach. Depending on the error-threshold, our algorithm can produce visualizations spanning the whole range from high-fidelity images to preview-quality (high frame-rate) images. Being resolution independent, it allows the user to freely zoom in and out of the vector field at interactive rates. Unlike many variations of LIC which require post-processing steps like equalization, a second pass of LIC, or high-pass filtering[5], our method does not need any extra steps. Moreover, changing textures and/or the texture-mapping parameters can allow us to produce a wide range of static representations and animations.

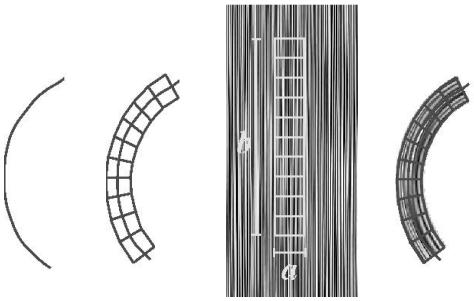
3. Algorithm Overview

In this section we present an interactive algorithm for global visualization of dense vector fields. The interactivity is achieved by level-of-detail computations and hardware acceleration. Level-of-detail approximations make it possible to save varying amounts of processing time in different regions based on the local complexity of the underlying vector field, thus providing a flexible run-time user-controlled trade-off between quality and execution time. Hardware acceleration allows us to compute dense LIC-like textures more efficiently than line integral convolution. Use of graphics hardware makes it possible to display the vector field at very high resolutions while maintaining the high texture frequency and low computation times.

To perform level-of-detail estimation, we define an error measure over the vector field domain (section 4.1). As a preprocessing step, we then construct a branch-on-need (BONO)[12] quadtree which serves as a hierarchical data structure for the error measure. The error associated with a node of the quadtree represents the error when only one representative streamline is computed for all the points within the entire region corresponding to the node. At run time, the quadtree is traversed and the error measure stored in each node is compared against a user-specified tolerance. Using the level-of-detail traversal we are able to selectively reduce the number of streamlines required to generate the flow textures. In section 4.2, we discuss how the quadtree traversal is controlled based on the resolution of the display.

Hardware accelerated texture mapping is used to generate a dense image from the scattered streamlines output by the traversal phase of the algorithm. During the above mentioned quadtree traversal, quad blocks of different levels corresponding to different spatial sizes are generated. For each region, a streamline is originated from its center. A quadrilateral strip, with a width equal to the diagonal of the region, is constructed following the streamline. Henceforth we will refer to this quadrilateral strip as a 'stream-patch' and to the streamline as 'medial streamline'. The stream-patch is then texture mapped with precomputed LIC images of a straight vector field. The texture coordinates for the quad-strip are derived by constructing a corresponding quad-strip at a random position in texture space. Figure 1 shows the construction and texture mapping of a stream-patch, and details are presented in section 5.1. The stream-patches for different regions are blended together (section 5.2). Each stream-patch extends beyond the originating region, covering many regions lying on its path. If a region has already been drawn over by one or more adjacent regions' stream-patches, it is no longer necessary to render the stream-patch for the region. In section 5.3, we discuss the use of the stencil buffer in graphics hardware to skip such regions.

In the following sections, we first explain the level-of-detail selection criteria, and then the hardware acceleration features of the algorithm.



(a) (b) (c) (d)

Figure 1: *Construction of the stream-patch: (a) the medial streamline, (b) the quadrilateral strip constructed on the streamline, (c) texture coordinates corresponding to the vertices of the quad-strip, and the texture parameters a, b, (d) the quad-strip after it has been texture mapped.*

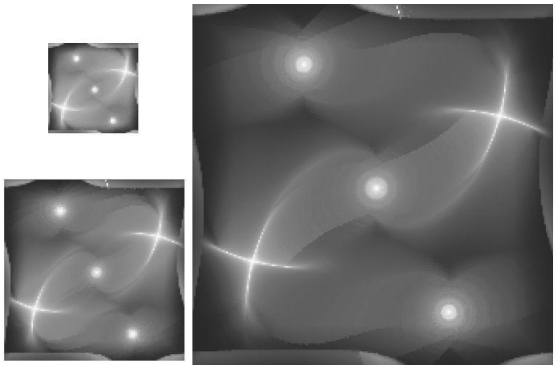


Figure 2: *Multi-level error for the vector field shown in figure 7. The images shown correspond to the following three levels of the error quadtree: 16x16 (top-left), 8x8 (bottom-left) and 4x4 (right) square regions. The error value range is mapped to [0.0,1.0], with 0.0 being the darkest and 1.0 the brightest.*

4. Level-of-Detail Selection

The level-of-detail selection process involves two distinct phases: (1) construction of a quadtree (for level-of-detail errors) as a preprocessing step, and (2) resolution dependent traversal of the quadtree at run-time with user-specified thresholds. Below, we elaborate on each of these stages.

4.1. Error Measures

An 'ideal' error metric for a level-of-detail representation should give a measure of how (in)correct the vector field approximation will be compared to the original field data. The textures produced by the algorithm provide information through unquantifiable visual stimulus. Therefore it is difficult to formally define the 'correctness' of the visualization produced. The fidelity of illustration of the vector field should be measured not from the values of the pixels of the image produced, but from the visual effect that the texture pattern has on the user. We use an error measure which tries to capture the difference between the texture direction at a point, which is the approximated vector direction, and the actual vector field direction at the point.

Consider the texture pattern at a point which is not on the medial streamline, but falls within the stream-patch. For each quad of the quad-strip, the texture direction is parallel to the medial-streamline segment within that quad. So, within each quad, we are approximating the vector field as a field parallel to the medial-streamline. The error can thus be quantified by the angular difference between the directions of the medial streamline and the actual vector field at the sample point.

Since each quadtree node originates a stream-patch that will travel outside the node boundary, the error measure associated with a particular quadtree node should consider all the points that are within the footprint of the stream-patch. To do this, for each quad in the quad-strip, we find the angular difference between the directions of the medial streamline segment and each of the vector field's grid points within the quad. The error for a stream-patch originated from a quadtree node is then defined as the maximum angular difference for the grid points across all quads of that stream-patch. Since the error would depend on the length of the stream-patch (which is user-configurable), we take a conservative approach and calculate the errors assuming large values of length. Note that since taking the maximum angular deviation as the error always keeps the error below the user-specified tolerances, it can be very sensitive to noise. For noisy data, taking a weighted average might prove helpful. The level-of-detail approximation errors for a particular level are computed by constructing stream-patches for all the quadtree nodes for that level and then computing the errors for each stream-patch. The error values for the vector field in figure 7 are shown in figure 2, and those for the vector field in figure 8 are in figure 3. The error values are normalized to the range [0.0,1.0] to make them user friendly.

It can be seen from the images in figure 2 that in any particular level, the regions around the critical points (the three vortices and two saddle points) have the highest error values. The critical points do not need any special handling as they would be represented by stream-patches of the finest level-of-detail allowed by the display resolution (section 4.2). Because of high curvature around critical points, use of thicker stream-patches would have resulted in artifacts due to self-intersections. The errors gradually fall off as we move away from the critical points, and hence would result in progressively coarser level-of-detail stream-patches. Also, errors for any particular region increase across levels. So, a high error threshold will permit a coarse level-of-detail approximation.

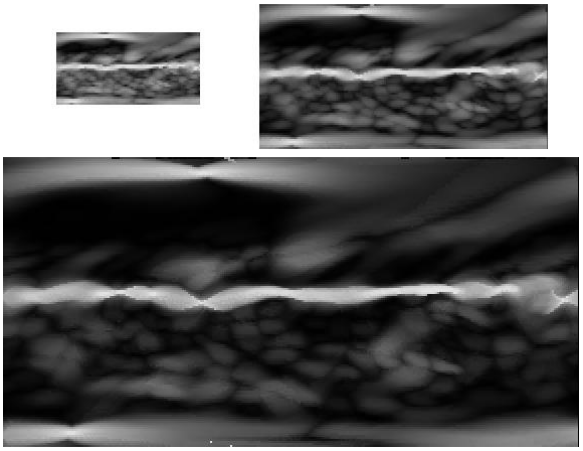


Figure 3: *Multi-level error for the vector field shown in figure 8. The images shown correspond to the levels 8x8 (top-left), 4x4 (top-right) and 2x2 (bottom) of the error quadtree. The error value range is mapped to [0.0,1.0], with 0.0 being the darkest and 1.0 the brightest.*

Similarly, in figure 3, the regions near the vortices have the highest errors. Since no run-time parameters are required for the error calculations, this error quadtree needs to be generated only once for the entire life of the dataset. At run-time, it can be read in along with the dataset.

4.2. Resolution dependent level-of-detail selection

In this section we describe the run-time aspects of the level-of-detail selection phase, which requires the user to input a threshold for acceptable error. We shall call the ratio of the vector field resolution to the display resolution the *resolution ratio*, k. Consider, for example, an $x_v \times y_v$ vector field dataset, and suppose our visualization window resolution is $x_w \times y_w$. Let

$$x_v = k \times x_w; \;\; y_v = k \times y_w \tag{1}$$

Note that in an interactive setting, the value of k changes if the user zooms in/out. If the display window has a resolution not smaller than that of the vector field, i.e, $k \leq 1$, the quadtree is traversed in a depth first manner, and stream-patches are rendered for quad blocks satisfying the error condition. In cases the display resolution is smaller, i.e., $k > 1$, the quadtree traversal is performed using resolution dependent tests in addition to the error threshold test. The resolution dependent controls in traversal are motivated by two goals: (1)we want to limit the quadtree traversal to the minimum block size which occupies more than one pixel on the display, and (2)we want to avoid a potential popping effect caused by a changing k which causes the above mentioned minimum block size to go up (or down) by one level.

For the discussion below, let us assume that at a particular instant $m > k > \frac{m}{2}$, where $\frac{m}{2} = 2^i$, $i = \{1, 2, ...\}$. Since $\frac{m}{2} \times \frac{m}{2}$ blocks occupy a display area less than the size of one pixel, we limit the quadtree traversal to the $m \times m$ block level. Now, if the user zooms in, k becomes progressively smaller, and at some point of time we will have $k = \frac{m}{2}$. At this instant, the minimum displayable block size becomes $\frac{m}{2}$. A lot of $m \times m$ blocks (those that do not satisfy the error threshold) will be eligible to change the level-of-detail to $\frac{m}{2} \times \frac{m}{2}$. If allowed to do so, it will result in a popping effect. To avoid this, we allow only a very small number of $m \times m$ blocks to change their level-of-detail to $\frac{m}{2} \times \frac{m}{2}$. As the user keeps zooming in, k continues to decrease, and we gradually allow more and more blocks to change their level-of-detail. If the user continues to zoom in, by the time k becomes equal to $\frac{m}{4}$, all the $m \times m$ blocks will have changed into $\frac{m}{2} \times \frac{m}{2}$ blocks. This gradual change in the level-of-detail is achieved by modifying the error threshold test for $m \times m$ blocks. The user supplied error threshold is scaled to a high value when $k = \frac{m}{2}$. As k decreases, we decrease the scaled threshold, such that by the time $k = \frac{m}{4}$, the threshold reaches its original user supplied value.

5. Hardware Acceleration

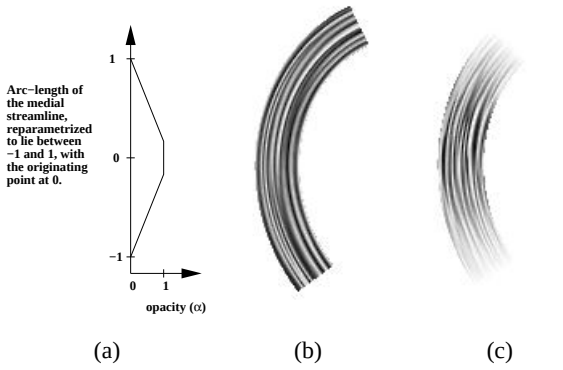
After the quadtree traversal phase has resolved the levels-of-detail for different parts of the vector field, stream-patches are constructed for each region corresponding to its level-of-detail. They are then texture mapped and rendered using graphics hardware (section 5.1). The different sized stream-patches need to be blended together to construct a smooth image (section 5.2). An OpenGL stencil buffer optimization is used to further reduce the number of medial streamlines computed (sec 5.3).

5.1. Resolution Independence

For each quadtree node that the traversal phase returns, a stream-patch is constructed using the medial streamline for that node. The texture coordinates for the stream-patch are derived by constructing a corresponding quad-strip at a random position in the texture space (figure 1(c)). The parameters a (width) and b (height) of the corresponding texture space quad-strip determine the frequency of the texture on the texture mapped stream-patch.

Since we are essentially rendering textured polygons, the output image can easily be rendered at any resolution. When the window size is the same as the vector field size, that is, the resolution ratio k is unity (equation (1)), the stream-patches have a width equal to the diagonal of the quadtree nodes they correspond to. When k is changed (e.g., when the user interactively zooms in/out, or when the window size is changed), the width of the stream-patches at each level-of-detail is modulated by $\frac{1}{k}$. Simultaneously, a and b need to be changed to reflect the change in k. Otherwise, when we zoom out, the texture shrinks leading to severe alias-

ing (figure 9(b)). Note that we cannot use anti-aliasing techniques like mipmaps as the texture will get blurred and all directional information will be lost. Similarly, for zoom ins, the texture is stretched, and we lose the granularity (figure 10(b)). For high zoom ins, or for high resolution large format displays, the 'constant texture frequency' feature of our algorithm proves very useful. When the display resolution is finer than the vector field resolution ($k < 1$), we are left with sparsely distributed streamlines. But due to the high texture frequency, the final image gives the perception of dense streamlines, which can be considered to be interpolated from the sparse original streamlines.



(a) (b) (c)

Figure 4: *Opacity function: (a) the value of α over the length of the stream-patch, (b) the patch without blending, and (c) the stream-patch after blending.*

5.2. Blending Stream-patches

To ensure that the final image shows no noticeable transition between adjacent simplified regions, a smooth blending of neighboring stream-patches needs to be performed. A uniform blending (averaging) will result in two undesirable properties. Firstly, the resultant image will loose contrast. If many stream-patches are rendered over a pixel, its value tends to the middle of the gray scale range. This is specially unsuitable for our algorithm, as the loss of contrast will be non-uniform across the image due to the non-uniform nature of the level-of-detail decomposition of the vector field. Secondly, the correctness of the final image (up to the user-supplied threshold) will be compromised. This happens because a stream-patch corresponding to a coarser level-of-detail will have a non-zero effect on its neighboring regions, some of which might correspond to finer level-of-details. To prevent coarser level-of-detail stream-patches from affecting the pixel values of nearby finer level-of-detail regions, we use a coarser-level-of-detail to finer-level-of-detail rendering order, combined with the opacity function shown in figure 4. The OpenGL blending function used is glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA). Because we use an opacity value of unity at the central part of the stream-patch, the pixels covered by this part are completely overwritten with texture values of this patch, thus erasing previous values due to coarser level-of-detail stream-patches. Moreover, this opacity function results in a uniform contrast across the image, even though the number of stream-patches drawn and blended over different parts of the image varies a lot. To reduce aliasing patterns, we jitter the advection length of medial streamlines in either direction and adjust the opacity function accordingly.

To ensure a smooth transition between adjacent patches, we also vary the stream-patch opacities in the direction perpendicular to the medial-streamline using a similar opacity function. This is done at a cost of increased rendering time since we add one quad strip on each side of the stream-patch so that the opacities can be varied laterally. In our implementation, the user can turn the lateral opacity variation off for high-frame rate preview quality requirements.

5.3. Reducing Streamline Redundancy

Because stream-patches continue beyond their originating regions, each stream-patch would cover many pixels, albeit with different opacities. From our experiments, we found that a pixel goes to an opacity value of unity with the first few stream-patches that are rendered over it. Thus, if a pixel has been drawn over by a few stream-patches, then we do not need to texture map any more stream-patches for this pixel. We use this idea to reduce the number of stream-patches that need to be rendered, and hence the number of medial streamlines that need to be computed. However, among the many stream-patches that may been rendered over this pixel, only those which are of the same or finer level of detail as this pixel's level-of-detail should be counted.

We avoid doing this 'minimum number of renderings' test in software by using the OpenGL Stencil buffer to keep track of how many stream-patches have been drawn over a pixel. When GL_STENCIL_TEST is enabled, the stencil buffer comparison is performed for each pixel being rendered to. The stencil test is configured using the following OpenGL functions:

- glStencilFunc(GL_ALWAYS, 0, 0): Specifies the comparison function used for the stencil test. For our purpose, every pixel needs to pass the stencil test (GL_ALWAYS).
- glStencilOp(GL_KEEP, GL_INCR, GL_INCR): Sets the actions on the stencil buffer for the following three scenarios: the stencil test fails, stencil test passes but depth test fails, and both tests pass. In our case, the stencil test always passes, so we increment (GL_INCR) the value of the stencil buffer at the pixel being tested by one.

Before starting to compute the stream-patch for a region, we read the stencil buffer values for all the pixels corresponding to that region. If all the pixels have been rendered to a minimum number of times, we skip the region. If not, we render

the stream-patch and the stencil buffer values of all the pixels which this patch covers are updated by OpenGL. While going through our coarser-to-finer level drawing order (as mentioned in section 5.2), we clear the stencil buffer each time we finish one level. Otherwise, a finer level-of-detail region which has been drawn over by coarser level-of-detail stream-patches might be skipped because each pixel in the region has already satisfied the threshold of minimum number of renderings. This will violate the error-criteria for the region.

The stencil buffer read operation is an expensive one in terms of time. If done for every stream-patch, it would take so much time that we would be better off not using it. For our implementation, we read the stencil-buffer once every few hundred stream-patches. The values are reused till we read in the buffer once again.

6. Results and Discussion

We present the performance and various visual results of our algorithm implemented in C++ using FLTK for the GUI. The timings are taken on a 1.7 GHz Pentium with an nVidia Quadro video card. The results show that the image quality remains reasonable even when error thresholds are increased to achieve high speedups. Moreover, various aspects of the visualizations are interactively configurable, as shown by the different visuals presented in this section.

6.1. Range of Image Quality and Speed

We present results for a simulated dataset of vortices (and saddles) with dimensions of 1000x1000, and for a real 573x288 dataset of ocean winds. The error calculation times for the vortices dataset was 148 seconds, and the ocean dataset required 20 seconds. For each dataset, two images are shown: one with tight error limits, and the other with relaxed bounds. For comparison, the images produced by FastLIC are shown in figures 5,6. A fourth order adaptive Runge-Kutte integration is used, with same parameters for both FastLIC and our algorithm. The medial streamlines in our algorithm were advected to the same length as the convolution length used for FastLIC.

Figures 7 show the results of our level-of-detail algorithm for the vortices dataset rendered for an 1000x1000 display window. Figure 7(a) was generated using a low error threshold in 0.81 seconds, while figure 7(b) was produced using a high error threshold in 0.39 seconds. Compared to FastLIC, we achieve speed-ups of 15-30 depending on the error threshold for level-of-detail selection. Figures 8(a) and (b) are outputs for the ocean dataset, rendered for a display window of same dimensions. A low error threshold was used for figure 8(a); it was relaxed for figure 8(b). The times taken were 0.32 and 0.16 seconds respectively, for speed-ups of 5.9-11.9 compared to FastLIC. There is no visible difference in image quality in either dataset for the low error thresholds.

dataset	*FastLIC*	*LOD*(*low error*)	*LOD*(*high error*)
Vortices	12.32*s*	0.81*s*(15.2)	0.39*s*(31.6)
Ocean	1.9*s*	0.32*s*(5.9)	0.16*s*(11.9)

Table 1: *Timing results for the 1000x1000 Vortices dataset and the 573x288 ocean winds dataset. The timings reported are for FastLIC, and our algorithm for two level-of-detail error thresholds. The speed-ups for the level-of-detail algorithm with respect to FastLIC are shown in parentheses. No post-processing has been done in any of these runs. The images for these runs are in figures 5, 6, 7 and 8.*

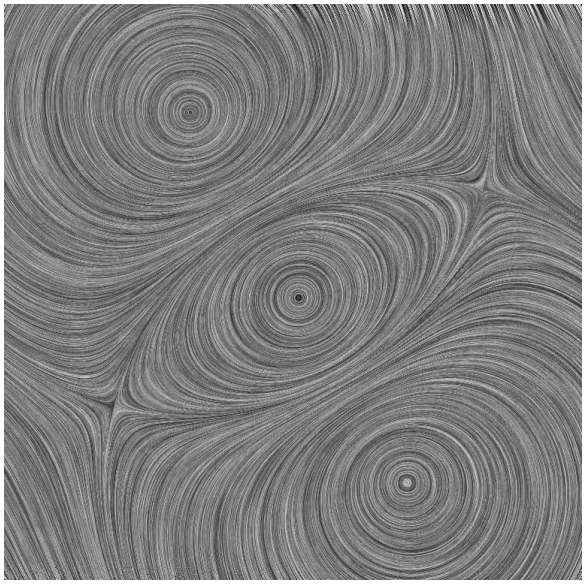


Figure 5: *LIC image of the 1000x1000 vortices dataset in 12.32s, using convolution length of 60*

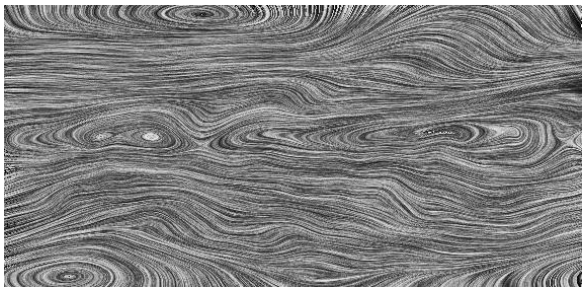
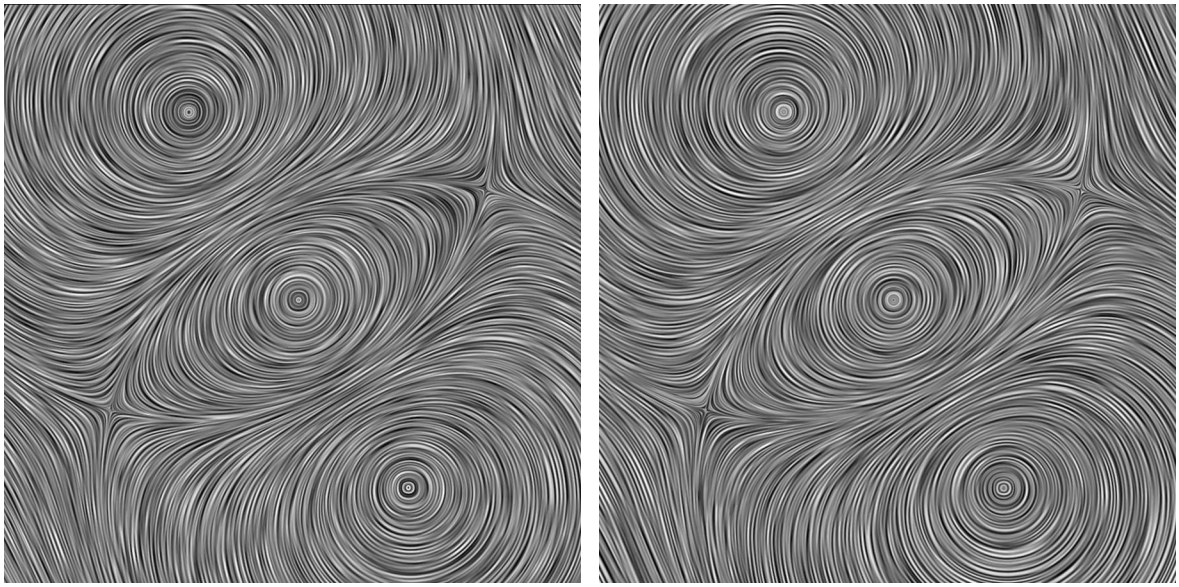


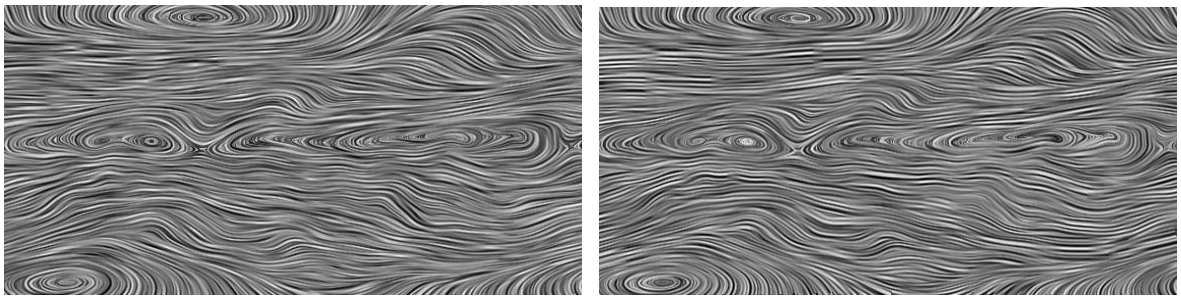
Figure 6: *LIC image of the 573x288 ocean wind dataset in 1.9s, using convolution length 30.*

For the higher thresholds (figure 7(b) and 8(b)), the differences are very minute and not readily noticeable.



(a) (b)

Figure 7: *Vortices dataset using: (a)low error threshold in 0.81s, (b)high error threshold in 0.39s. The image quality remains good even for high error thresholds.*



(a) (b)

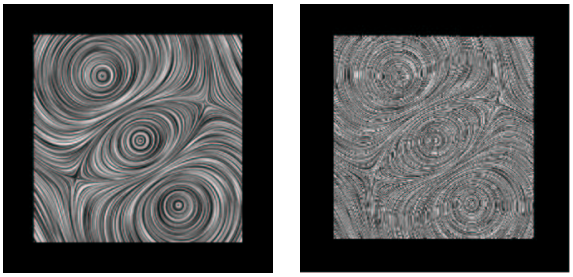
Figure 8: *Ocean dataset using: (a)low error threshold in 0.32s, (b)high error threshold in 0.19s. The image quality remains good even for high error thresholds.*

6.2. Interactive Exploration

The level-of-detail features allow users to visualize the vector field at a wide range of resolutions. The algorithm automatically adjusts the display parameters to render an unaliased image of a large dataset for display on a small screen. Figures 9 (a) and (b) show the results for visualizations on low resolution displays, with and without resolution adjusted parameters. At the other extreme, the vector data can be rendered at very high resolutions when displayed on large format graphics displays, or when viewed at high zoom factors. The hardware texture mapping allows us to change resolutions at interactive frame rates, thus allowing the user to freely zoom into and out of the dataset. Figure 10(a) shows the central vortex of the vortices dataset at a magnification factor of 20x. The texture mapping parameters are changed dynamically, so that the texture does not get stretched. For comparison, figure 10(b) shows the same rendering when the texture parameters are not adjusted.

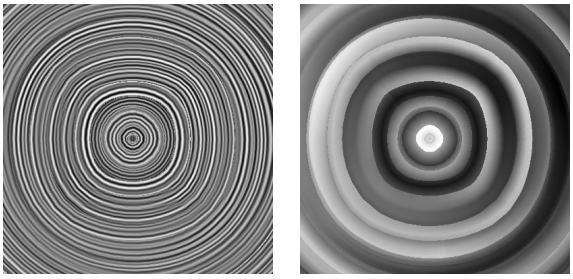
6.3. Scalar Variable Information

Information about a scalar variable defined on the vector field can be superimposed on the directional representation of the vector field. One technique for that is to modulate the texture frequency of the final image based on the local values of the scalar. Kiu[13] and Banks applied this scheme to LIC images by using noise textures of different frequencies.



(a) (b)

Figure 9: *Zoom out: The vortices dataset rendered for a display window one-fourth its size: (a)The textures coordinates are adjusted to prevent aliasing, and the finest displayable level-of-detail is adjusted to match display resolution. (b)the image gets aliased if scaled down without adjusting the texture parameters.*



(a) (b)

Figure 10: *Zoom in: The central vortex of the dataset in figure 7 is shown at a magnification factor of 20x: (a)The texture coordinates are adjusted automatically to compensate for the magnification factor, thus maintaining the original texture frequency, (b)the texture loses granularity if the coordinates are not adjusted.*

We follow a different approach of using multiple textures to achieve the same goal.

We start with an ordered set of precomputed textures, in which some texture property varies monotonically from one extreme of the set to the other. For the example presented in figure 11, the property is the length of the LIC directional pattern. That is, textures of short LIC patterns (produced by small convolution lengths) are at one end of the set and those with long patterns (large convolution lengths) are at the other. For each quad in a stream-patch, different textures are selected as the source for texture mapping based on the value of the scalar, much like mipmaps are used depending on screen area. However, use of different textures in adjacent quads of the quad-strip destroys the directional continuity of the texture mapped strip. As a way around this, in a process analogous to blending two adjacent levels of mipmaps,

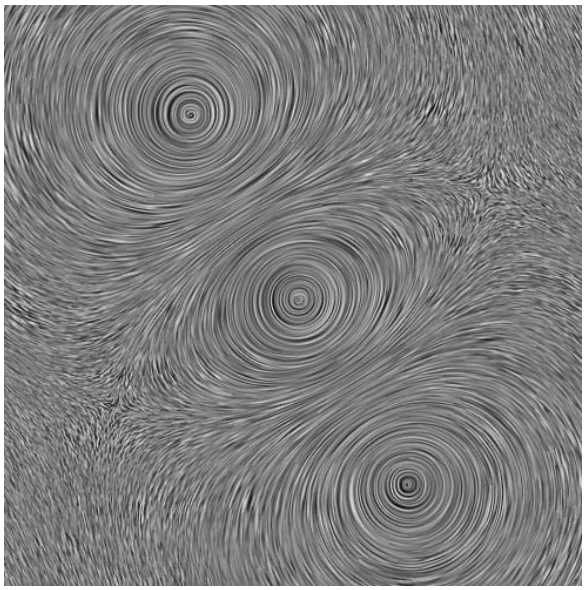
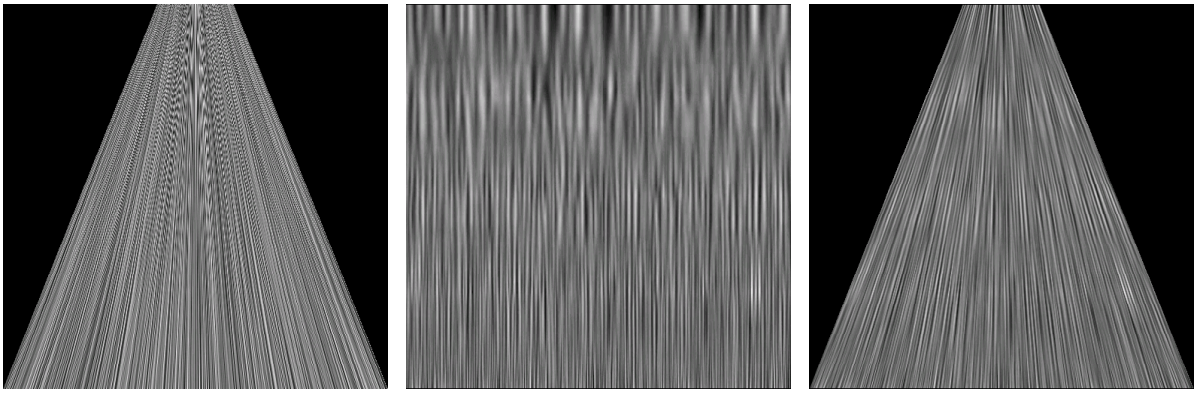


Figure 11: *Illustration of use of multiple textures to show a scalar variable on the vector field, velocity in this example. The velocity is high in parts around the vortices, and is low at the lower left and upper right corners.*

each quad is rendered twice with textures adjacent in the ordered set of texture bank. The opacities of the two textures rendered are weighted so that the resultant texture smoothly varies as the scalar value changes. The net effect is a texture property which varies smoothly relative to a scalar value. The detail of the scalar representation in a particular image is limited to the level-of-detail approximation that is used for generating the image. In some situations it may be desired to control the error in the scalar value illustration. Then the quadtree of errors would need to be constructed using both the angle error (section 4.1) and the error in the scalar value. Figure 11 is generated using the multi-texturing method to show the magnitude of velocity of the vortices dataset. The short patterns indicate lower velocity, whereas the LIC pattern is long in areas of high velocity.

We can also apply the multiple texture technique to preserve constant texture frequency for vector fields on grids which are not regular. Using the method presented by Forsell[4], a vector field representation is generated on a regular grid. This image is then texture mapped onto the actual surface in physical space. During this mapping, the regular grid cells used for computation are transformed to different sizes in physical space, which can result in the stretching or pinching of the texture. To prevent this, we use textures of varying frequencies for each quad subject to the area the quad occupies in physical space. Figure 12(a) shows the result when a usual constant frequency texture is mapped on a grid in which the cells grow progressively smaller from bot-



(a) (b) (c)

Figure 12: *Multiple texture rendering to show antialiasing for a grid which has small cells at the top and large ones at the bottom: (a)constant frequency image mapped to the grid, showing aliasing, (b)multi-frequency texture generated using multiple textures, (c)the texture in (b) mapped to the grid, without aliasing.*

tom to top, resulting in aliasing. Figure 12(b) is the image generated by our algorithm using multiple frequency textures subject to the grid cell area. The texture frequency decreases from bottom to top, so that when it is mapped to the grid (figure 12(c)), there is no aliasing.

6.4. Streamline Textures

The level-of-detail technique can be applied with a variety of textures to get diverse visualizations. If we use a sparse texture, it gives the output an appearance of streamline representation. This is a popular method of flow visualization [14],[15], [16].

Figures 13(a-c) are generated using a stroke-like texture to give the image a hand drawn feeling. The stroke in the texture is oriented by making the tail wider than the head. This adds directional information to the image. Unlike previous textures, this texture has an alpha component which is non-zero only over the oriented stroke. Figure 13(a) is generated by constraining the level-of-detail approximation to a single level-of-detail. This creates an uniform distribution of streamlines. The stencil buffer option (section 5.3) is turned on to limit the streamlines from crowding one another. Figures 13(b-c) have been rendered using the level-of-detail approximations. Since the level-of-detail is finer near the saddle points and the vortices, more streamlines are drawn near these parts. Due to the relative lack of streamlines in the parts of the vector field which have low errors, the more complex parts of the vector field stand out to the viewer.

6.5. Animation

Animation of the vector field images is achieved by translating the texture coordinates of each stream-patch from one frame to another. The only additional constraint is that a cyclic texture be used for texture mapping the stream-patches. For example, the texture in figure 1(c) needs to be cyclic along the vertical direction. For each time step, the image is rendered by vertically shifting (downwards) the texture coordinates compared to the previous frame. Since the texture is cyclic, if the texture coordinates move past the bottom edge of the texture, they reappear at the top edge. To show different speeds, the texture coordinates for each stream-patch are moved by an amount proportional to the velocity of the seed-point of the patch.

7. Conclusion and Future Work

Using a level-of-detail framework, we are able to reduce the computation times for a dense visualization of vector fields. Coupled with hardware acceleration, the algorithm generates high quality visualizations at interactive rates for large datasets and large displays. The resolution independence and user-controlled image quality features make this algorithm extremely useful for vector data exploration, which till now is being done using probing techniques like streamlines, particle advection etc.

Acknowledgement

This work was supported by The Ohio State University Research Foundation Seed Grant, and in part by NSF Grant ACR-0118915 and NASA Grant NCC 2-1261. We would like to thank Ravi Samtaney and Zhanping Liu for providing the test data sets and Dr. Roger Crawfis and Dr. David Kao for useful comments.

(a) (b) (c)

Figure 13: *Stroke-like textures used to generate streamline representation: (a) Uniform placement of streamlines using constant level-of-detail, (b),(c) streamlines generated using the error quadtree traversal. The level-of-detail algorithm generates a non-uniform distribution of streamlines, giving more detail to higher error regions and vice versa.*

References

1. B. Cabral and C. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH 93*, pages 263–270. ACM SIGGRAPH, 1993.
2. J. van Wijk. Spot noise: Texture synthesis for data visualization. *Computer Graphics*, 25(4):309–318, 1991.
3. D. Stalling and H.-C. Hege. Fast and resolution independent line integral convolution. In *Proceedings of SIGGRAPH '95*, pages 249–256. ACM SIGGRAPH, 1995.
4. L.K. Forssell and S.D. Cohen. Using line integral convolution for flow visualization: Curvilinear grids, variable-speed animation, and unsteady flows. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):133–141, 1995.
5. A. Okada and D. L. Kao. Enhanced line integral convolution with flow feature detection. In *Proceedings of IS&T/SPIE Electronic Imaging '97*, pages 206–217, 1997.
6. 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), 1998.
7. V. Verma, D. Kao, and A. Pang. Plic: Bridging the gap between streamlines and lic. In *Proceedings of Visualization '99*, pages 341–348. IEEE Computer Society Press, 1999.
8. B. Jobard and W. Lefer. Unsteady flow visualization by animating evenly-spaced streamlines. *Computer Graphics Forum (Proceedings of Eurographics 2000)*, 19(3), 2000.
9. H. Garcke, T. Preußer, M. Rumpf, A. Telea, U. Weikard, and J. van Wijk. A continuous clustering method for vector fields. In *Proceedings of Visualization '00*, pages 351–358. IEEE Computer Society Press, 2000.
10. X. Tricoche, G. Scheuermann, and H. Hagen. Continuous topology simplification of planar vector fields. In *Proceedings of Visualization '01*, pages 159–166. IEEE Computer Society Press, 2001.
11. B. Cabral and C. Leedom. Highly parallel vector visualization using line integral convolution. In *Proceedings of Seventh Siam Conference On Parallel Processing for Scientific Computing*, pages 803–807, 1995.
12. J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.
13. M.-H. Kiu and D. Banks. Multi-frequency noise for LIC. In *Proceedings of Visualization '96*, pages 121–126. IEEE Computer Society Press, 1996.
14. G. Turk and D. Banks. Image-guided streamline placement. In *Proceedings of SIGGRAPH '96*, pages 453–460. ACM SIGGRAPH, 1996.
15. B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. In *Proceedings of the eight Eurographics Workshop on visualization in scientific computing*, pages 57–66, 1997.
16. R. Wegenkittl and E. Gröller. Fast oriented line integral convolution for vector field visualization via the internet. In *Proceedings of Visualization '97*, pages 309–316, 1997.