Hardware-Accelerated Lagrangian-Eulerian Texture Advection for 2D Flow Visualization

Daniel Weiskopf[1] Gordon Erlebacher[2] Matthias Hopf[1] Thomas Ertl[1]

[1]Visualization and Interactive Systems Group, University of Stuttgart, Germany*

[2]School of Computational Science and Information Technology, Florida State University, USA†

Abstract

A hardware-based approach for visualizing unsteady flow fields by means of Lagrangian-Eulerian advection is presented. Both noise-based and dye-based texture advection is supported. The implementation allows the advection of each texture to be performed completely on the graphics hardware in a single-pass rendering. We discuss experiences with the interactive visualization of unsteady flow fields that become possible due to the high visualization speed of the hardware-based approach.

1 Introduction

Traditionally, flow fields are visualized as a collection of streamlines, pathlines, or streaklines that originate from user-specified seed points. The problem of placing seed points for particle tracing at appropriate positions is approached, e.g., by employing spot noise [14], LIC (line integral convolution) [1], texture splats [2], texture advection [10], equally spaced streamlines [13], or flow-guided streamline seeding [15].

In this paper, we present and discuss a hardware-based approach for visualizing unsteady flow fields by means of Lagrangian-Eulerian advection [8, 9]. This approach combines the advantages of the Lagrangian and Eulerian formalisms: A dense collection of particles is integrated backward in time (Lagrangian step), while the color distribution of the image pixels are updated in place (Eulerian step). A common problem of dense representations of vector fields is that they are computationally expensive, especially when a high-resolution domain is used. We demonstrate that texture advection maps rather well to the programmable features of modern consumer graphics hardware: Noise and dye advection can be done in single pass rendering, respectively; therefore, a speed-up of one to two orders of magnitudes compared to an optimized CPU-based approach can be achieved. We discuss how our interactive visualization tool facilitates the understanding of unsteady flows in the context of numerical fluid dynamics.

The hardware approach of this paper is influenced by previous work on hardware-based LIC [5] and texture advection [7]. These early implementations were based on pixel shaders exclusively available on SGI MXE graphics and were limited by quite restrictive sets of operations. Therefore, multiple rendering passes were necessary and accuracy was limited by the resolution of the frame buffer. In [16], we presented a GeForce 3-based texture advection approach that is particularly well-suited for dye advection; a similar approach was independently developed for a fluid visualization demo [11] by Nvidia.

2 Lagrangian-Eulerian Advection

In this section, a brief survey of Lagrangian-Eulerian advection of 2D textures is presented. A more detailed description can be found in [9]. Note that the description in this section partly differs from the algorithm [9] to allow a better mapping to graphics hardware.

In the traditional Lagrangian approach to particle tracing, each single particle can be identified individually and the trajectory of each particle is computed separately according to the ordinary differential equation

$$\frac{\mathrm{d}\vec{r}(t)}{\mathrm{d}t} = \vec{v}(\vec{r}(t),t) \quad , \tag{1}$$

where $\vec{r}(t)$ describes the path of the particle and $\vec{v}(\vec{r},t)$ represents the vector field to be visualized.

*{weiskopf,hopf,ertl}@informatik.uni-stuttgart.de

†erlebach@mailer.csit.fsu.edu

In the Eulerian approach, particles loose their identity and particle properties (such as color) are rather stored in a property field. Positions are only given implicitly as the coordinates of a property field.

The hybrid Lagrangian-Eulerian approach combines the advantages of both the Lagrangian and Eulerian formalisms. Between two successive time steps, coordinates of a dense collection of particles are updated with a Lagrangian scheme whereas the advection of the particle property is achieved with an Eulerian method. At the beginning of each iteration, a new dense collection of particles is chosen and assigned the property computed at the end of the previous iteration. The core of the advection process is thus the composition of two basic operations: coordinate integration and property advection.

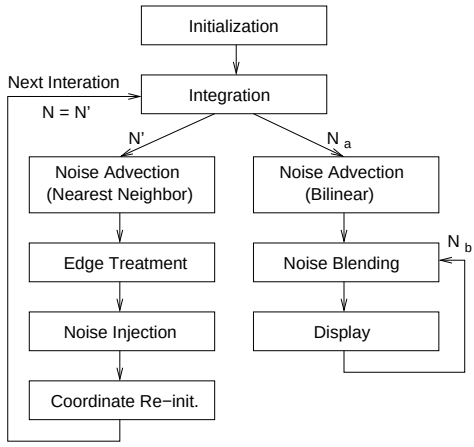

Figure 1: Flowchart of noise advection.

2.1 Noise-Based Advection

All information concerning the particles is stored in 2D arrays at the corresponding integer-valued locations (i, j). Thus, we store the initial coordinates of those particles in a two-component array $\vec{C}(i, j)$. First order (Eulerian) integration of Eq. (1) requires an array $\vec{v}$ for the velocity field at the current time. Similarly to LIC, we choose to advect noise images; four noise arrays N, N', N_a, and N_b, contain respectively the noise to advect, two advected noise images, and the final blended image.

Figure 1 shows a flowchart of the algorithm. We first initialize the coordinate array $\vec{C}(i, j)$ to random values to avoid regular patterns that might otherwise appear during the first several steps of the advection. Note that $\vec{C}(i, j)$ describes only the fractional part of the coordinates—actual coordinates with respect to the grid are $\vec{x}(i, j) = (i, j) + \vec{C}(i, j)$. N is initialized with a two-valued noise function (0 or 1) to ensure maximum contrast.

Carrying out an integration backward in time by a time span $\Delta t > 0$, we can solve Eq. (1) by first order Euler integration,

$$\vec{r}(t - \Delta t) = \vec{r}(t) - \Delta t\, \vec{v}(\vec{r}(t), t) \quad .$$

Based on the array of fractional coordinates, one integration step yields the coordinates $\vec{x}'$ at the previous time step

$$\vec{x}' = (i, j) + \vec{C}(i, j) - \vec{h} \circ \vec{v}(i, j) \quad . \qquad (2)$$

The step size $\vec{h} = (h_x, h_y)$ depends on Δt and on the relationship between the physical size of the problem domain and the cell sizes of the corresponding array. We use the notation "$\circ$" for a component-wise multiplication of two vectors.

After the coordinate integration, the initial noise array N is advected twice to produce two noise arrays N' and N_a. N' is an internal noise array to carry on the advection process and to re-initialize N for the next iteration. To maintain a high contrast in the advected noise and to avoid artificial diffusion, N' is computed by a nearest-neighbor sampling of N, based on the coordinates $\vec{x}'$ for the previous time step. In contrast, N_a serves to create the current animation frame and no longer participates in the noise advection. It is computed by bilinear interpolation in N to reduce spatial aliasing effects. N_a is then blended into the filtered noise texture of the previous time step, N_b, to map the existing temporal correlation along pathline segments to a spatial correlation within a single frame. An exponential temporal filter is implemented as a blending operation, $N_b = (1 - \alpha)N_b + \alpha N_a$.

Before N' can be used in the next iteration, it must undergo a correction to account for edge effects. The need to test for boundary conditions is eliminated by surrounding the actual particle domain with a buffer zone whose minimum width is determined by the maximum velocity in the flow field and the integration step size. Since the advected image contains a two-valued random noise with little or no spatial correlation, we simply store

new random noise in the buffer zone as the new information potentially flowing into the physical domain. At the next iteration, N will contain these values and some of them will be advected to the interior of the physical domain by Eq. (2). Since random noise has no spatial correlation, the advection of the surrounding buffer values into the interior region produces no visible artifacts.

Another correction step counteracts a duplication effect that occurs during the computation of Eq. (2). Effectively, if particles in neighboring cells of N' retrieve their property value from within the same cell of N, this value will be duplicated in the corresponding cells of N'. This effect is undesirable since lower noise frequency reduces the spatial resolution of the features that can be represented. This duplication effect is caused by the discrete nature of our texture sampling and is further reinforced in regions where the flow has a strong positive divergence. To break the undesirable formation of uniform blocks and to maintain a high frequency random noise, we inject a user-specified percentage of noise into N'. Random cells are chosen in N' and their value is inverted. From experience, a fixed percentage of two to three percent of randomly inverted cells provides adequate results over a wide range of flows.

Finally, the array $\vec{C}(i, j)$ of fractional coordinates is re-initialized to prepare a new collection of particles to be integrated backward in time for the next iteration. Nearest-neighbor sampling implies that a property value can only change if it originates from a different cell. If fractional coordinates were neglected in the next iteration, subcell displacements would be ignored and the flow would be frozen where the velocity magnitude or the integration step size is too small. Therefore, the array $\vec{C}(i, j)$ is set to the fractional part of the coordinates $\vec{x}'$ originating from Eq. (2).

2.2 Dye-Based Advection

The process of dye advection can be emulated by replacing the advected noise texture by a texture with a smooth pattern. A dye is released into the flow and advected with the fluid, giving results analogously to traditional experimental flow dynamics. The noise texture is simply replaced by a texture of uniform background color upon which the dye is deposed. As the high frequency nature of the texture is removed, many of the above correction steps are no longer required and the implementation is greatly simplified: Particle injection into the edge buffer, random particle injection, constant interpolation of the noise texture, coordinate re-initialization, and noise blending can be neglected.

The point of injection can be freely chosen by the user by "painting" into the fluid. Points of finite diameter or lines are possible shapes of dye sources. The dye can be released once and tracked (which approximates the path of a particle), released continuously at a single point (generates a streakline), or pulsated (the dye is turned on intermittently, creating time surfaces). Colored dye is used of distinguish different injection points or times in the resulting patterns.

Dye advection is exclusively based on bilinear lookup in the particle texture of the previous time step. In contrast to noise advection, no nearest-neighbor sampling is needed because dye has a much smaller spatial frequency than noise. Bilinear interpolation computes the true position of particles within a cell and therefore overcomes the problem that subcell displacements are ignored and the flow is frozen where the velocity magnitude or the integration step size is too small. Thus, fractional coordinates can be neglected. As a disadvantage of bilinear interpolation, the dye gradually smears out due to an implicit, artificial "diffusion" process.

3 Advection on Graphics Hardware

The above advection algorithm was originally designed for an optimized CPU-based implementation. For a mapping to the GPU, advection speed and an appropriate accuracy are the two main issues. To achieve the first goal, the number of rendering passes and the number of texture lookups in each pass should be minimized. Therefore, we minimize the number of textures necessary to represent the arrays in the advection process.

Limited accuracy on the GPU is a major issue for most hardware-based algorithms. The main problem is the extremely low resolution of only 8 bits per channel in the frame buffer or texture on consumer market GPUs. This can only be overcome by computing crucial operations with higher resolution, e.g., in the fragment shader unit.

3.1 Noise Advection on the GPU

In what follows, we demonstrate single pass noise advection on the ATI Radeon 8500. The Radeon provides two phases in a single rendering pass, with up to six texture lookups and eight numerical operations in each phase. The two-component array $\vec{C}(i,j)$ and the single-component arrays N and N_b are combined in a single RGBα texture referred to as a particle texture. The intermediate arrays N' and N_a are never stored in a texture, but are only used as temporary variables within the rendering pass. Figure 2 shows the structure of the fragment operations for noise-based advection. The fragments that are processed according to these operations are generated within the rasterization unit of the GPU by rendering a single quadrilateral. The size of the quad is identical to the size of the particle texture. The input texture coordinates provide a one-to-one mapping between the quad and the particle texture so that the final output yields the particle texture for the next iteration.

Phase 1 begins with a bilinear texture lookup in the two-component vector field for the velocity and in the RGBα texture for the particle field. The texture coordinates (i,j) are just the texture coordinates of the quad. Note that texture coordinates lie in the interval $[0,1]$. We therefore scale the corresponding coordinates from Section 2 by the reciprocal of the width or height of the computational domain as symbolized by $\vec{s}_{\text{frac}}$. The coordinate integration implements Eq. (2) for the coordinates $\vec{x}'$. The computation of the intermediate coordinates $\tilde{\vec{C}}$ is similar to the above coordinate integration, except that the overall scaling factor is the component-wise inverse $\vec{s}_{\text{frac}}^{-1}$ and (i,j) is omitted. The "integer term" (i,j) can be neglected since it is removed in the final extraction of the partial coordinates.

Phase 2 starts with two dependent texture lookups in the particle texture, each based on the previously computed texture coordinates $\vec{x}'$. The advected noise texture N' results from a nearest-neighbor sampling, whereas N_a results from a bilinear interpolation in the old noise image. In a third texture fetch operation, a fixed noise image N_{inj} is obtained. Based on the values in N_{inj}, the values of some texels in N' are flipped to inject additional noise. To prevent regular patterns, the fixed texture N_{inj} is shifted across the quad by randomly changing the texture coordinates from frame to frame.

The following step extracts the fractional coordinates from the previously computed intermediate values $\tilde{\vec{C}}$. The problem is that the available fragment operations do not allow to extract the fractional or the integer part of a number. Therefore, we use another dependent texture lookup to extract fractional parts of a number. The basic idea is to exploit the repeat functionality for texture coordinates to map values to their fractional parts that are in the range $[0,1]$. The original numbers serve as texture coordinates for the dependent texture lookup and an identity operation is represented by the texture for a range of values from $[0,1]$. Larger and smaller texture coordinates are mapped into $[0,1]$ by the texture repeat. Since both x and y components are extracted simultaneously, a 2D texture is required. This texture has a size of 256×256, which corresponds to the maximum resolution of the fractional coordinates stored in the particle texture.

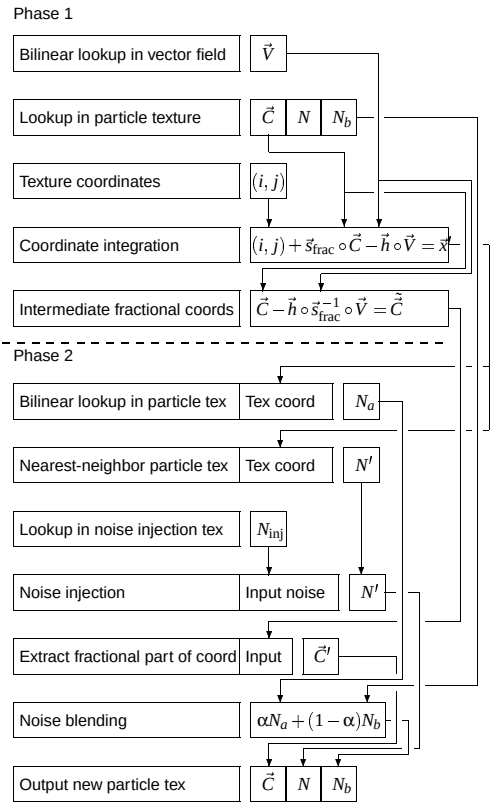

Figure 2: Structure of the fragment operations for hardware-based noise advection.

After the fractional coordinates are extracted, the advected noise N_a is blended with N_b, the filtered noise of the previous iteration. Finally, edge correction is implemented in a separate step by rendering randomly shifted noise textures into the buffer zones. Note that some of the numerical operations within a single box in the flowchart have to be split into several consecutive operations in the actual fragment shader implementation (e.g, a sum of three terms has to be split into two summations).

3.2 Dye Advection on the GPU

As another application of texture advection, the process of dye advection can be emulated by replacing the advected noise texture by a smooth dye texture. In Section 2.2, we described that many of the above correction steps are no longer required and that the implementation is greatly simplified: Particle injection into the edge buffer, random particle injection, nearest-neighbor lookup in the noise texture, fractional coordinates, and noise blending can be neglected. Only the core advection step with bilinear interpolation is required. User-specified dye injection is implemented by rendering the dye sources into the texture, after the actual advection part was performed.

3.3 Complete Visualization Process

The complete visualization process is as follows. First, the noise and the dye textures are advected, each in a single rendering pass. Here, we render directly into a texture. Second, the just updated textures serve as input to the actual rendering pass: The α channel of the noise texture is replicated to RGB colors (giving a gray-scale noise image) and blended with the colored dye texture. For an unsteady flow, the vector field is transferred from main memory to texture memory before each iteration. Otherwise, there is no difference between the implementation of unsteady and steady flows.

Since the blending of noise and dye textures requires only a simple fragment blending operation, more advanced postprocessing steps can also be included in the final rendering part. A useful postprocessing operation modulates the pixel intensity with the magnitude of the velocity. In this way, rather uninteresting regions of the flow that exhibit only slow motions are faded out in the noise texture by reducing their brightness; the important parts of the flow are emphasized. This velocity mask is implemented by introducing a third texture lookup which retrieves the vector field texture. The red and green channels of the flow texture hold the v_x and v_y components of the velocity. We assume that the blue channel stores a weighting factor that depends on the magnitude of the velocity; this weight has to be computed by the CPU before the flow field is downloaded to the GPU. In the final rendering pass, the brightness of the noise texture can be scaled by this weighting factor.

As another postprocessing operation, the representation of the flow can be superimposed over a background image that provides additional context. For the implementation, another texture that represents the background has to be fetched and blended with the noise and dye textures.

4 Implementation

We chose DirectX 8.1 [3] for our implementation. One of the advantages of DirectX is that advanced fragment operations are configured in the form of so-called pixel shader programs. Pixel shaders target a vendor-independent programming environment (instead of vendor-specific OpenGL extensions). Another advantage is a rather simple, assembler-like nature of pixel shader programs, as opposed to more difficult and complex OpenGL extensions. Moreover, the DirectX SDK provides an interactive environment for testing pixel shader code, which facilitates debugging of fragment code. A very important advantage of DirectX is its better support by some vendors' graphics drivers, both with respect to stability and performance. The main reason for this is the much bigger market for DirectX products than for OpenGL products. The transfer between frame buffer and texture memory is indispensable for the advection algorithm and is an example of better support by DirectX. On the ATI Radeon 8500, only DirectX allows direct rendering into a texture and thus makes a transfer of texture data completely superfluous.

A major problem with DirectX is the restriction to Windows because many visualization environments are based on Unix/Linux. OpenGL (without vendor-specific extensions) promotes platform-independent software development and very often the resulting code can be included into existing visualization tools.

Table 1: Performance measurements in frames per second.

	Noise Advection		Noise, Dye & Mask	
Particle Size	1024^2	1024^2	1024^2	1024^2
Flow Size	256^2	1024^2	256^2	1024^2
GPU (steady)	24.8	23.8	20.5	19.9
GPU (unsteady)	24.3	19.4	20.3	16.7
CPU-based	0.8	0.8	NA	NA

For our hardware implementation we chose the ATI Radeon 8500 because it is the only available GPU that could process noise advection in a single pass. Its two phases with six texture lookups and eight numerical operations each can accommodate quite complex algorithms on the GPU. One problem with the Radeon 8500 is that fragment operations are limited to an accuracy of 16 bits per channel in the interval $[-8,8]$, i.e., only 12 bits in $[0,1]$. The limitation to 12 bits in $[0,1]$ is a major problem for texture advection because we have only a 2 bit subtexel accuracy when a typical $1024^2 = 2^{10} \times 2^{10}$ particle texture is addressed. (Note that texture coordinates lie in $[0,1]$). This subtexel accuracy is not appropriate for texture advection and causes noticeable visual artifacts. Fortunately, the accuracy of dependent texture lookups can be improved by using a larger range of values for (x,y) texture coordinates (e.g., $(x,y) \in [0,8]$ with 15 bit accuracy) and a "perspective" division by a constant z value (e.g, $z = 8$ in this example). In this way, the subtexel accuracy can be increased to 5 bits, which no longer causes visual artifacts.

The main goal of our hardware-based approach is high advection and rendering speed. Table 1 compares performance measurements for the GPU-based implementation (on ATI Radeon 8500, Athlon 1.2 GHz CPU, Windows 2000) with those for a CPU-based implementation on the same machine. Included are the numbers for mere noise advection (single pass advection) and for advection of noise and dye images (two advection passes) with subsequent velocity masking. A final rendering to a 1024^2 window is included in all tests. The performance measurements indicate that download of vector field data from main memory to texture memory via AGP bus (line "GPU (unsteady)") takes only a small portion of the overall rendering time even for large data sets. For many applications, the flow field is medium-sized and allows typical overall frame rates of 15–25 fps. The cost of the software implementation is dominated by the advection of the noise texture and is quite independent of the size of the flow field data; a transfer of flow data to texture memory is not necessary.

5 Results

Figure 3 and Color Plates 4 and 5 show example visualizations of an unsteady flow field. Here, the velocity field is produced by the interaction of a planar shock with a longitudinal vortex (in axisymmetric geometry) [4]. The figures show a single time step from a time series of an unsteady flow simulation. The complete data set has 200 time steps and a spatial resolution of 256×151. In Figure 3, only noise advection is employed. The noise blending factor is $\alpha = 0.05$. The rather uninteresting regions of the flow are faded out by reducing their brightness via velocity masking. Therefore, one clearly sees the primary shock, secondary shock and the slip lines [6]. Color Plates 4 and 5 additionally include the advection of colored dye which was released into the flow by the user. In Color Plate 5, velocity masking is applied to enhance important parts of the flow. In Color Plate 4, no velocity masking is applied.

Other examples with high-resolution images, videos, and a demo program can be found on our project web page *http://wwwvis.informatik.uni-stuttgart.de/~weiskopf/advection*.

In our experience, interactive visualization is particularly useful for exploring unsteady flow fields. Still images of advected noise can already display both the direction of the flow (by means of short path segments) and the magnitude of the velocity (the images are smeared out in regions of high velocity). An animated sequence can additionally visualize the orientation of the flow. The combination of noise-based and dye-based advection further improves the understanding of the flow. The dense representation by a noise texture provides information on all parts of the flow and thus prevents the omission of potentially important features. Guided by this information, the interactive system allows the user to explore specific, interesting regions by tracing dye. The user can freely choose the source

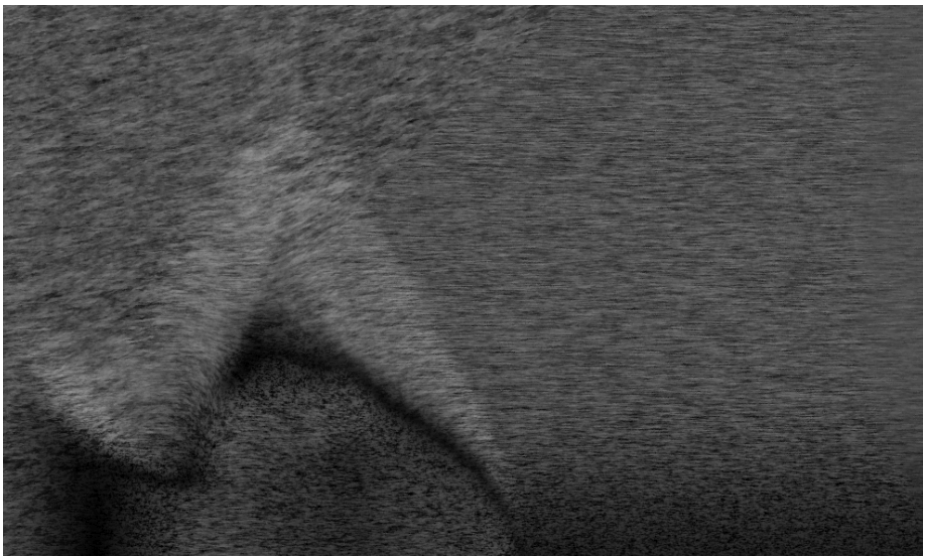

Figure 3: One frame from the interaction of a shock with a longitudinal vortex.

of dye injection by literally "painting" into the fluid. Different shapes and sizes of sources and differently colored dye are supported. The dye can be released once and tracked (which approximates the path of a particle), released continuously at a single point (which generates a streakline), or freely placed into the flow. Moreover, pulsated dye injection leads to time surfaces; colored dye is used to distinguish different injection points or times in the resulting patterns.

In addition to the obvious advantages of interactive visualization, another aspect of the visualization pipeline benefits from the high advection speeds of the hardware approach. A significant amount of time can be spent in earlier steps of the pipeline without affecting the interactive character of the complete visualization system. Therefore, reading considerable amounts of data from disk for unsteady flows, filtering this data, and transferring it to the graphics board is possible.

6 Conclusion

We have presented an interactive visualization tool for exploring unsteady flow fields by texture advection. The mapping and rendering components of the visualization pipeline are completely located on the graphics hardware. Therefore, data transfer between main memory and GPU is heavily reduced and an extremely high advection speed is achieved at acceptable numerical accuracy. In this way, an interactive exploration of unsteady flows is well supported. Our experiences with DirectX 8.1 as a software basis for hardware-based graphics show that DirectX in its current version can be suitable for interactive visualization applications on Windows PCs. In a future project, we plan to use hardware-based 2D advection to visualize flows on non-planar (curvilinear) hypersurfaces, e.g., for data given on aircraft wings or on the surface of an automobile. Moreover, we will investigate how noise advection can be extended to 3D, similarly to 3D dye advection [16] and 3D LIC [12].

Acknowledgments

We would like to thank Joe Kniss for the valuable tip to improve the accuracy for dependent texture lookups on the Radeon 8500.

References

[1] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *SIG-*

GRAPH 1993 Conference Proceedings, pages 263–272, 1993.

[2] R. A. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *IEEE Visualization '93 Proceedings*, pages 261–267, 1993.

[3] *DirectX*. Web Site: http://www.microsoft.com/directx.

[4] G. Erlebacher, M. Y. Hussaini, and C.-W. Shu. Interaction of a shock with a longitudinal vortex. *J. Fluid Mech.*, 337:129–153, 1997.

[5] W. Heidrich, R. Westermann, H.-P. Seidel, and T. Ertl. Application of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics*, pages 127–134, 1999.

[6] M. Y. Hussaini, G. Erlebacher, and B. Jobard. Real-time visualization of unsteady vector fields. In *40th AIAA Aerospace Sciences Meeting*, 2002.

[7] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Hardware-accelerated texture advection for unsteady flow visualization. In *IEEE Visualization 2000 Proceedings*, pages 155–162, 2000.

[8] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Lagrangian-Eulerian advection for unsteady flow visualization. In *IEEE Visualization 2001 Proceedings*, pages 53–60, 2001.

[9] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Lagrangian-Eulerian advection of noise and dye textures for unsteady flow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):211–222, 2002.

[10] N. Max, R. Crawfis, and D. Williams. Visualizing wind velocities by advecting cloud textures. In *IEEE Visualization '92 Proceedings*, pages 171–178, 1992.

[11] Nvidia. OpenGL fluid visualization. Web Site: http://developer.nvidia.com/view.asp?IO=ogl_fluid_viz.

[12] C. Rezk-Salama, P. Hastreiter, C. Teitzel, and T. Ertl. Interactive exploration of volume line integral convolution based on 3D-texture mapping. In *IEEE Visualization '99 Proceedings*, pages 233–240, 1999.

[13] G. Turk and D. Banks. Image-guided streamline placement. In *SIGGRAPH 1996 Conference Proceedings*, pages 453–460, 1996.

[14] J. J. van Wijk. Spot noise-texture synthesis for data visualization. In *SIGGRPAPH 1991 Conference Proceedings*, pages 309–318, 1991.

[15] V. Verma, D. Kao, and A. Pang. A flow-guided streamline seeding strategy. In *IEEE Visualization 2000 Proceedings*, pages 163–170, 2000.

[16] D. Weiskopf, M. Hopf, and T. Ertl. Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *VMV '01 Proceedings*, pages 439–446. infix, 2001.

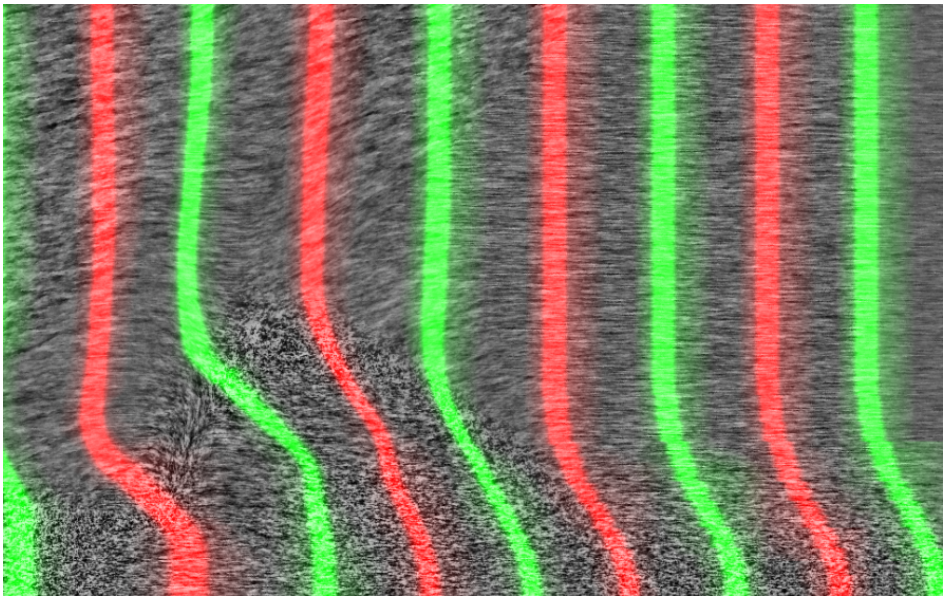

Figure 4: Visualization of the interaction of a planar shock with a longitudinal vortex by noise and dye advection. The intensity of the gray-scale image is not changed.

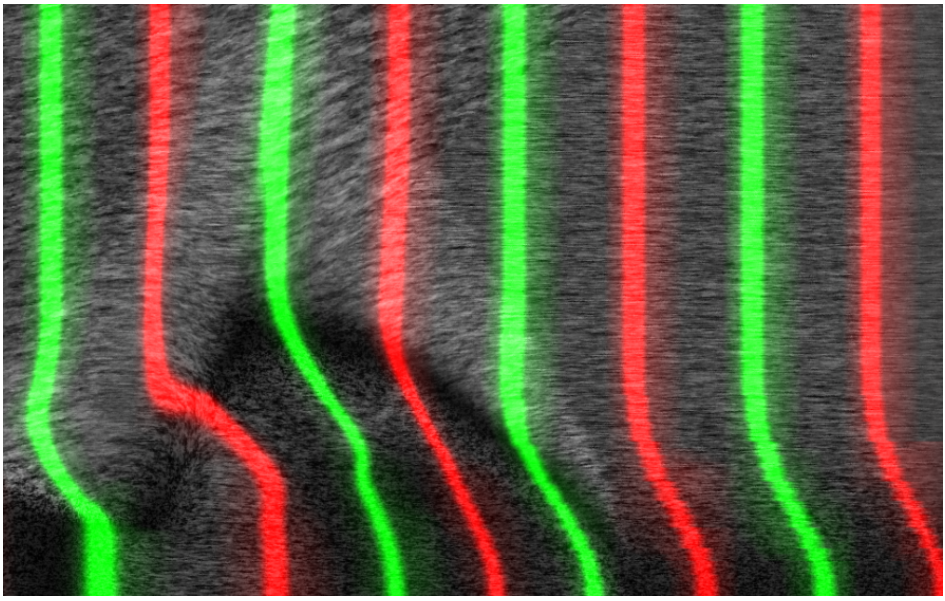

Figure 5: Visualization of the interaction of a planar shock with a longitudinal vortex by noise and dye advection. The intensity of the gray-scale image is weighted by the magnitude of the velocity to enhance the important structures in the flow.