A Case Study on Hardware-Accelerated Lagrangian-Eulerian Texture Advection for 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. The implementation allows texture advection 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.

Keywords: vector field visualization, graphics hardware

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], or texture advection [10].

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 an expensive computation, 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 [15], 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 [12] by Nvidia.

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

[†]erlebach@mailer.csit.fsu.edu

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.

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 according to the ordinary differential equation

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

with the particle coordinates $\vec{r}(t)$ and the vector field $\vec{v}(\vec{r},t)$. Conversely, the advection of the particle property (such as color) is achieved with an Eulerian method by replacing the respective entries in a property field. 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.

All information associated with a particle is stored in 2D arrays at the corresponding integer-valued location (i, j). The initial fractional coordinates of the particles are contained in a two-component array $\vec{C}(i, j)$. 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, $\vec{r}(t-\Delta t) = \vec{r}(t) - \Delta t\, \vec{v}(\vec{r}(t), t)$. Based on the array of fractional coordinates, one integration step yields the coordinates $\vec{x}'$ at the previous time step

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

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 componentwise 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

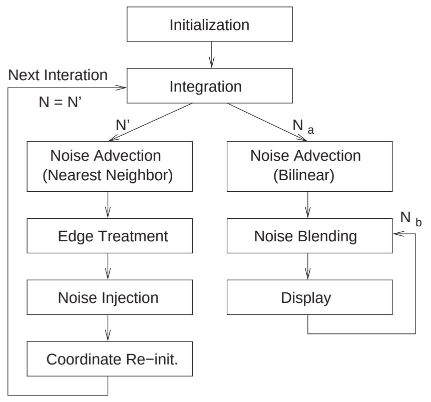

Figure 1: Flowchart of noise advection.

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. Random noise is introduced into the buffer zone as the new information potentially flowing into the physical domain. 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'. 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 the fractional part of the coordinates $\vec{x}'$ originating from Eq. (2). If fractional coordinates were neglected, subcell displacements would be ignored due to nearest-neighbor sampling and the flow would be frozen where the velocity magnitude or the integration step size is too small.

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.

In what follows, we demonstrate single pass noise advection on the ATI Radeon 8500. The Radeon provides two phases in a single

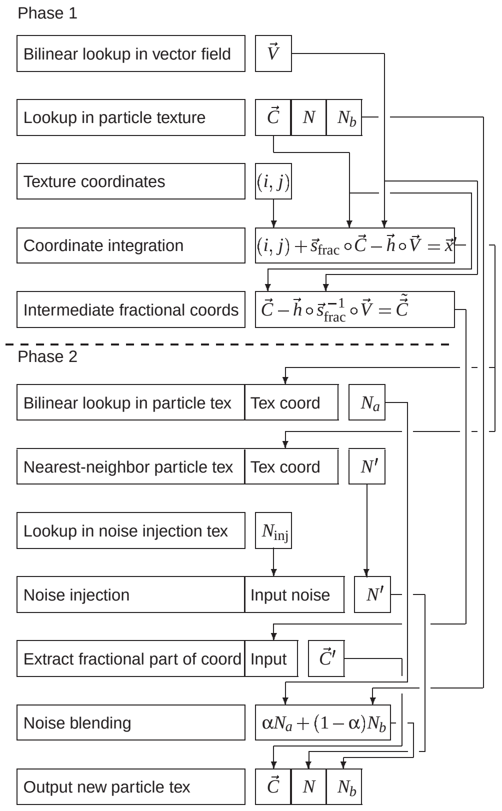
Phase 1
Bilinear lookup in vector field
$\vec{V}$
Lookup in particle texture
$\vec{C}$ N N_b
Texture coordinates
(i,j)
Coordinate integration
$(i,j)+\vec{s}_{\text{frac}}\circ\vec{C}-\vec{h}\circ\vec{V}=\vec{x}'$
Intermediate fractional coords
$\vec{C}-\vec{h}\circ\vec{s}_{\text{frac}}^{-1}\circ\vec{V}=\tilde{\vec{C}}$
Phase 2
Bilinear lookup in particle tex
Tex coord
N_a
Nearest-neighbor particle tex
Tex coord
N'
Lookup in noise injection tex
N_{inj}
Noise injection
Input noise
N'
Extract fractional part of coord
Input
$\vec{C}'$
Noise blending
$\alpha N_a+(1-\alpha)N_b$
Output new particle tex
$\vec{C}$ N N_b

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

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. A single quadrilateral is rendered with the fragment shaders enabled. 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 operation extracts the fractional coordinates from the previously computed intermediate values $\tilde{\vec{C}}$. Subsequently, 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).

As another application of texture 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 analogous to those found in traditional flow experiments. Since 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, 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.

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.

4 Implementation and Experiences

We chose DirectX 8.1 [3] for our implementation. One of the advantages of DirectX is that advanced operations in the transform and lighting or the fragment stages 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.

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.

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)	23.8	23.3	19.7	19.3
GPU (unsteady)	20.1	3.9	15.3	3.7
CPU-based	0.8	0.8	NA	NA

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 (as explained in the following section). A final rendering to a 1024^2 window is included in all tests. The performance measurements clearly indicate that download of vector field data from main memory to texture memory (line "GPU (unsteady)") is the major bottleneck of the hardware-based implementation. However, for many applications, the flow field is medium-sized and allows typical overall frame rates of 15–20 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 Applications

In cooperation with U. Rist from the Institute of Aerodynamics and Gasdynamics at the University of Stuttgart, the advection system of this paper was used to visualize data from direct numerical simulations of transition from the laminar to the turbulent flow state. Research of laminar-turbulent transition is of practical interest because skin friction at a wall and hence fuel consumption can be reduced by delaying laminar-turbulent transition. As an example, Figure 3 shows a single time step of a simulation of a transitional boundary-layer structure [11]. The complete data set has 60 time steps and a spatial resolution of 119×481. Both images in Figure 3 show the simultaneous advection of a gray-scale noise image and colored dye. The noise blending factor is chosen $\alpha = 0.05$. In the top image, noise advection is applied as described previously. In the bottom image, the rather uninteresting regions of the flow are faded out in the noise texture by reducing their brightness during the final rendering pass. This masking is based on the magnitude of the flow velocity. This example demonstrates that a 2D hypersurface is quite useful to explore even a 3D simulation data set.

Figure 4 shows results of another application. Here, the velocity field produced by the interaction of a planar shock with a longitudinal vortex (in axisymmetric geometry) [4] is visualized by noise advection with masking turned on. One clearly sees the primary shock, secondary shock and the slip lines [6]. The data set has 200 time steps and a spatial resolution of 256×151.

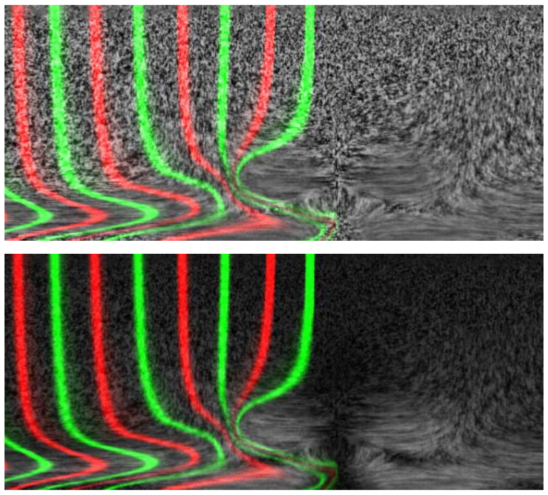

Figure 3: Visualization of a transitional boundary-layer structure by noise and dye advection.

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 orientation 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 direction 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 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 [15] and 3D LIC [13].

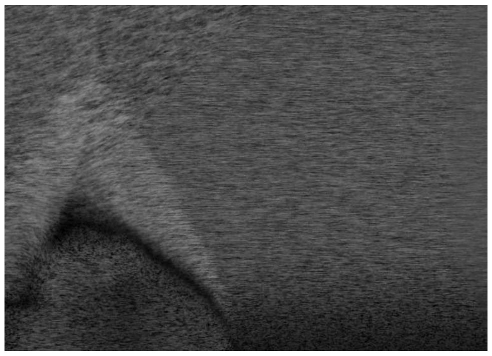

Figure 4: One frame from the interaction of a shock with a longitudinal vortex.

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 *SIGGRAPH 1993 Conference*, pages 263–272, 1993.

[2] R. A. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *Visualization '93*, 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 *Visualization 2000*, pages 155–162, 2000.

[8] B. Jobard, G. Erlebacher, and M. Y. Hussaini. Lagrangian-Eulerian advection for unsteady flow visualization. In *Visualization 2001*, 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, accepted for publication*, 2002.

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

[11] D. G. W. Meyer, U. Rist, V. Borodulin, V. Gaponenko, Y. Kachanov, Q. Lian, and C. Le. Late-stage transitional boundary-layer structures. direct numerical simulation and experiment. In H. Fasel and W. Saric, editors, *Laminar-Turbulent Transition, IUTAM Symposium Sedona*, pages 167–172. Springer, 1999.

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

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

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

[15] 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.