Hardware-Accelerated Visualization of Time-Varying 2D and 3D Vector Fields by Texture Advection via Programmable Per-Pixel Operations

Daniel Weiskopf Matthias Hopf Thomas Ertl

University of Stuttgart, Institute for Informatics
Visualization and Interactive Systems Group
Breitwiesenstr. 20–22, 70565 Stuttgart, Germany
Email: {weiskopf,hopf,ertl}@informatik.uni-stuttgart.de

Abstract

We present hardware-accelerated texture advection techniques to visualize the motion of particles in steady or time-varying vector fields given on Cartesian grids. We propose an implementation of 2D texture advection which exploits advanced and programmable texture fetch and per-pixel blending operations on an nVidia GeForce 3. For 3D vector field visualization, we present an algorithm for SGI's VPro, based on pixel textures and 3D textures. Moreover, we sketch how 3D texture advection could be implemented on future graphics boards that provide programmable fetch operations for 3D textures. Since all implementations exclusively use graphics hardware without intermediate data transfer to main memory, extremely high frame rates are achieved, e.g., up to 90 frames per second for advecting a calculatory number of one million particles in a 2D flow. The proposed techniques are especially useful for the interactive visualization of vector fields.

1 Introduction

The visualization of 2D and 3D vector fields has been investigated and used in various scientific and engineering disciplines for many years. Typical applications stem from simulations in computational fluid dynamics, calculation of physical vector fields, such as electromagnetic fields or heat flow, or from measurements of actual wind or fluid flows.

As flow visualization has a long tradition, various techniques exist to visually represent steady and unsteady vector fields. Among the standard techniques for flow visualization is the class of methods based on particle tracing, such as pathlines, streamlines, streaklines, or ribbons. The problem of placing seed points for particle tracing at appropriate positions is approached, e.g., by employing spot noise [18, 4], LIC (line integral convolution) [2, 16], texture splats [3], texture advection [12, 11], equally spaced streamlines [17], or flow-guided streamline seeding [19]. These techniques were originally designed to visualize steady flows. Some of these methods were extended to allow for time-varying vector fields, e.g., with respect to LIC [7, 15] or spot noise [5].

All these approaches have in common a dense representation of the vector field. Therefore, they are expensive to compute, especially when using a high-resolution computational domain. The main topic of this paper is to show how texture advection can be simulated and visualized entirely on graphics hardware, thus allowing interactive visualization of high-resolution unsteady vector fields given on Cartesian grids. To achieve this goal, we exploit the programmable graphics pipeline of state-of-the-art low cost graphics boards, such as nVidia's GeForce 3. The visualization of 3D flows is based on SGI's VPro, which supports 3D textures and pixel textures.

In previous work, Heidrich et al. [8] propose the use of graphics hardware for flow visualization by LIC. Based on this approach, Jobard et al. [9] use per-fragment operations to implement texture advection on graphics hardware. Some of their ideas are adopted for the visualization methods of this paper. Therefore, we compare their approach with ours and describe advantages and disadvantages of the different techniques in Sect. 5. Both Heidrich et al.'s and Jobard et al.'s algorithms are restricted to 2D vector fields. In Sect. 4, however, we show how hardware-accelerated visualization of 3D flows can be implemented.

The remaining parts of this paper are organized as follows. In the next section, a mathematical description of the Eulerian approach to particle tracing—the underlying model for texture advection—is presented, including a scheme to numerically solve such a texture advection problem. In Sect. 3, hardware-accelerated 2D flow visualization on a GeForce 3 is described. It is followed by an algorithm for 3D flow visualization based on SGI's VPro. Section 5 contains some results and a discussion of the visualization techniques of this paper. Finally, Sect. 6 concludes with a brief summary and outlook on future work.

2 Eulerian Approach to Particle Tracing

The standard approach to particle tracing is a Lagrangian approach. Here, each single particle can be identified individually, i.e., labeled in some sense, and the properties of each particle are determined depending on time t. In the following discussion of the Lagrangian approach, a particle is identified and labeled by its position $\vec{r}_0 \in U \subset \mathbb{R}^n$ at an initial time t_0. The underlying vector space has dimension n, where n usually is 2 or 3. The computational domain is denoted U. In the case of particle tracing, the properties—such as colors—can be subsumed by a function ϕ which does not change during time for a specific particle, i.e., we have the time-independent property function $\phi_{\text{Lagrangian}}(\vec{r}_0, t) = \phi_{\text{Lagrangian}}(\vec{r}_0)$. The trajectory of a single massless particle is determined by the ordinary differential equation

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

where $\vec{r}(t)$ describes the path of the particle and $\vec{v}(\vec{r}, t) \in U \subset \mathbb{R}^n$ represents the vector field to be visualized. The time t also parameterizes the pathline of the particle. Therefore, positions at arbitrary times t and t_1 along the trajectory $\vec{r}(t)$ of a particle are related by the integral equation

$$\vec{r}(t) = \vec{r}(t_1) + \int_{t_1}^{t} \vec{v}(\vec{r}(t'), t')\,\mathrm{d}t' \quad . \qquad (1)$$

From a Eulerian point of view, the property ϕ is a function of position $\vec{r}$ and time t, as opposed to a function of time for a fixed particle labeled by $\vec{r}_0$ in the Lagrangian approach. The observer is located at a fixed position $\vec{r}$ and measures changes of ϕ caused by the fact that different particles cross that position.

As the property does not change for a specific particle, we have $\phi(\vec{r}, t) = \phi(\vec{r}_0, t_0)$ in the Eulerian approach if the spacetime positions $(\vec{r}, t)$ and $(\vec{r}_0, t_0)$ belong to the pathline of the same particle, cf. Eq. (1). For a continously differentiable property function, we finally obtain the advection equation

$$\frac{\partial \phi(\vec{r}, t)}{\partial t} + \vec{v}(\vec{r}, t) \cdot \vec{\nabla}\phi(\vec{r}, t) = 0 \quad .$$

Since ϕ is not even continuous in our particle tracing applications, we do not solve the above equation but directly apply Eq. (1) to propagate ϕ through time. All methods based on texture advection have this approach in common, cf., the work by Max and Becker [11], Heidrich et al. [8], or Jobard et al. [9].

To tackle the advection problem on a computer, the initial value problem for the differential equation (1) has to be solved numerically and the property function ϕ has to be discretized. The simplest method to solve an ordinary differential equation is the so-called Euler integration [13], which yields

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

Here, an integration backward in time by a time span $\Delta t > 0$ is carried out. The function ϕ is discretized on a Cartesian grid, both spatially and temporally. We employ the notation $\phi^{\tau}_{\vec{i}}$ to describe a sampling of ϕ at an indexed position $\vec{i} \in \mathbb{N}^n$ and an indexed time τ. We identify $\phi^{\tau}_{\vec{i}} = \phi^{\tau}_{i,j}$ in the 2D case and $\phi^{\tau}_{\vec{i}} = \phi^{\tau}_{i,j,k}$ in the 3D case. Without loss of generality, indexed positions and physical positions are related by $\vec{r} = \Delta r\, \vec{i}$, where Δr is the grid spacing. The same holds for the time $t = \tau \Delta t$. Analogously to $\phi^{\tau}_{\vec{i}}$, the discretized version of the flow field is denoted $\vec{v}^{\tau}_{\vec{i}}$.

Combining a discrete grid for ϕ with Euler integration for particle positions, we obtain the following numerical scheme to propagate a solution from time $\tau - 1$ to τ:

$$\phi^{\tau}_{\vec{i}} = \text{Lookup}(\phi^{\tau-1}; \vec{i} - \Delta s\, \vec{v}^{\tau}_{\vec{i}}) \quad , \qquad (2)$$

with the integration step size $\Delta s = \Delta t / \Delta r$. On the right hand side of this equation, an interpolated lookup in $\phi^{\tau-1}_{\vec{i}}$ has to be performed, since the previous position of a particle, $\vec{i} - \Delta s\, \vec{v}^{\tau}_{\vec{i}}$, is usually not identical to a grid point. In the 2D case bilinear interpolation is employed, in the 3D case trilinear interpolation. By using an integration backward in time, the property field ϕ_i^τ is completely filled for the new time step; in contrast, some elements of this field could be missed if a forward integration from the previous time step was applied. For the implementation of advection on graphics hardware, textures represent the data grids, thus providing hardware support for bi- or trilinear interpolation.

Since all computations in our approach are performed on Cartesian grids, no explicit reference to positions is made. Positions are temporarily computed only in the intermediate step for the interpolated lookup in $\phi_i^{\tau-1}$, but these are not stored in any grid structure. That is an important difference to the texture advection approaches in [11, 8, 9].

In the remaining parts of this paper, we will describe how to implement this numerical scheme on OpenGL-based graphics hardware. Both 2D and 3D cases are discussed.

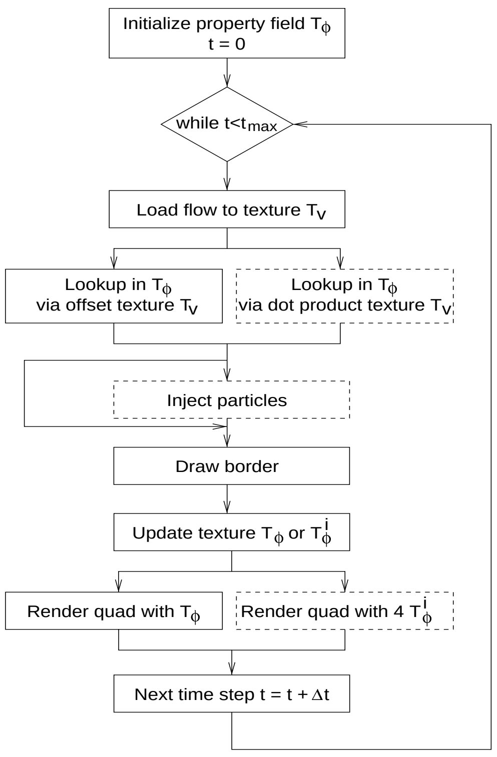
Initialize property field T_ϕ t = 0
while $t<t_{max}$
Load flow to texture T_V
Lookup in T_ϕ via offset texture T_V
Lookup in T_ϕ via dot product texture T_V
Inject particles
Draw border
Update texture T_ϕ or T_ϕ^i
Render quad with T_ϕ
Render quad with 4 T_ϕ^i
Next time step $t = t + \Delta t$

Figure 1: Structure of the 2D texture advection algorithm.

3 Hardware-Based 2D Texture Advection

Our implementation of 2D texture advection heavily relies on programmable per-pixel operations provided by nVidia's GeForce 3. The simulation and rendering of texture advection is based on standard OpenGL 1.2, with the following extensions [10] being included: texture shaders and register combiners for the actual advection process; rectangular textures, which relax the power-of-two dimensions requirements for 2D textures; and pbuffers for hardware-supported off-screen rendering.

3.1 Basic Advection

The basic idea of the algorithm is as follows. The property field ϕ_i^τ is represented by a 2D image that is normally held in a 2D texture and, only during intermediate calculations, in the framebuffer. The computational domain is determined by the extend of the image and has to be rectangular, since textures are rectangular as well. The number of components in that property texture depends on the type of application; normally, we use three RGB values per grid point. The vector field $\vec{v}_i^\tau$ itself is stored in a two-component 2D texture T_v.

The core of the advection algorithm uses a dependent texture lookup in order to shift the particles along the vector field. Figure 1 shows a schematic diagram of the advection process; default operations are contained within solid lines, optional operation in dashed lines. First, the texture T_ϕ holding the property field ϕ_i^τ is set to its initial values; simulation time t and the corresponding value for τ are set to zero. The following steps are repeated while increasing the time by Δt for each new cycle.

In this loop, the property field for the new time step τ is computed according to Eq. (2). Two different methods—offset textures and dependent dot product textures—can be used to implement the bilinear lookup in the property texture for the previous time step. These two methods will be explained shortly. The new property grid is first rendered into the framebuffer and then copied back into the texture T_ϕ, thus replacing the information on the previous time step. Copying from framebuffer to texture memory takes place only on the graphics board and is not slowed down by a limited bandwidth between graphics subsystem and main memory. This last step may even be removed completely, as soon as direct rendering into texture memory is possible, which has already been implemented for Direct 3D.[1] Pbuffers can be used for all these computational steps. In this way, the computational parts are completely independent of visual representation on the screen; in particular, a visual window smaller than the computational domain may be used.

For a steady flow, the simulation process is essentially identical; only the texture T_v is initialized with the vector field data once. Therefore, a recurrent transfer of texture data from main memory to the graphics subsystem is not necessary.

Offset textures and dependent dot product textures are part of the texture shader extension that provides programmable texture mapping within the rasterization stage. A sequence of texture shaders allows a very flexible mechanism for mapping sets of texture coordinates to the actual texture.

An offset texture program transforms signed (ds, dt) components of a previous texture unit by a 2×2 floating point matrix and then uses the result to offset the texture coordinates of the current stage for a 2D texture fetch operation. Figure 2 shows a diagram of the offset texture process. This process contains two texture fetch operations. The first step comprises a lookup in an offset texture based on texture coordinates (s_0, t_0). An offset texture—also called DSDT texture—is a special type of 2D texture with two components describing the distortion of coordinates in the successive texture fetch from an RGB texture. A third, optional component can be supplied to determine a scaling of the final RGB colors. In the second step, a dependent texture lookup in a standard RGB texture is performed. Here, another pair of texture coordinates (s_1, t_1) is modified by the previously fetched offset values. Each component of the final RGB triplet is scaled by a factor $M = k_{\text{scale}}\, mag + k_{\text{bias}}$, where mag is given by the offset texture and k_{scale} and k_{bias} are parameters constant during the whole texture program.

Offset texture programs have a one-to-one correspondence to the iteration equation (2). The property field $\phi^{\tau}_{\vec{i}}$ corresponds to a standard RGB texture (if ϕ has three components), the flow field $\vec{v}^{\tau}_{\vec{i}}$ corresponds to the offset texture, $(-\Delta s)$ can be represented by a uniform scaling via a 2×2 matrix. The texture coordinates (s_0, t_0) and (s_1, t_1) are identical and reflect the indexed position $\vec{i}$. The computational domain is implemented by rendering a single quadrilateral with the texture coordinates set in a way to represent the corner points of the domain. By using offset texture programs, the basic advection process is implemented in a single pass. Then, this part of the framebuffer has to be copied back into T_ϕ, so that incremental integration can be continued in the next time step. Note that all calculations for the shifted, dependent texture lookup are done in floating point accuracy internally on the graphics chip and thus accuracy is not restricted to the resolution of color channels in the frame buffer.

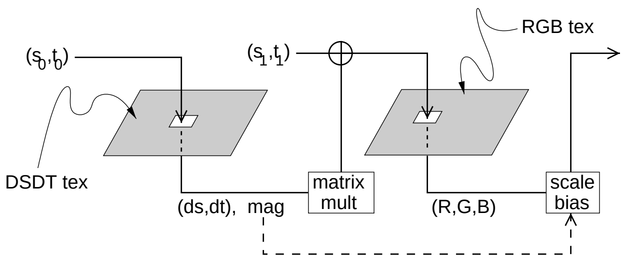

Figure 2: Dependent texture lookup via offset texture.

In addition, we propose another, alternative approach to this basic advection calculation based on dot product textures. Once again, we exploit a dependent texture lookup. However, it consists of three steps this time. In the first step, a signed RGB texture containing the 2D flow field is accessed, where texture coordinates are chosen the same way as above. The RGB texture contains the (v_x, v_y) vector in the first two components, the blue value is set to 1. In the second step, the first coordinate u_x for a dependent texture lookup is computed via a dot product. Here, the texture coordinates are set to $(-\Delta s, 0, x_{\text{domain}})$, with x_{domain} corresponding to the x coordinate of the computational domain. After having executed the dot product, the coordinate

$$\begin{aligned} u_x &= (-\Delta s, 0, x_{\text{domain}}) \cdot (v_x, v_y, 1) \\ &= x_{\text{domain}} - \Delta s\, v_x \end{aligned}$$

is exactly the x component of the previous particle position. In the third step, another dot product is computed to obtain u_y. Here, the texture coordinates have to be set to $(0, -\Delta s, y_{\text{domain}})$. Still in the third texture program stage, the new coordinates (u_x, u_y) are used for a dependent texture lookup in the 2D RGB texture T_ϕ. This approach is normally not useful for 2D visualization because it needs one more texture stage than the offset-texture approach. However, the 3D extension of product textures will

[1] We decided not to use Direct 3D in order to develop a platform-independent code.

become interesting for 3D flow visualization as described in Sect. 4.1.

As we are working with fixed-point arithmetics in texture memory and the framebuffer for both approaches, we have to assume that $\vec{v}_{\vec{i}}^{\tau} \in [-m, m)^n$, for some extreme value m. So we can fully utilize the value resolution of the vector field texture by stretching $\vec{v}$ to the used texture domain. Then the integration step size Δs has to be adapted to compensate this scaling during the iteration step, Eq. (2).

Finally, the property field for the new time step is actually drawn to screen by rendering a quadrilateral equipped with the texture T_ϕ.

3.2 Tracing Distinguishable Structures

Texture advection can readily be applied to the visualization by the motion of larger, distinguishable structures through the flow. A prominent visualization technique for time-varying flows are streaklines. These can be generated by setting values of the property field $\phi_{\vec{i}}^{\tau}$ at a specified starting position of the streakline for each time step. Steaklines provide an intuitive understanding of the flow structure as they resemble dye advection used in classical, experimental fluid mechanics. Dye injection was also used, e.g., by Shen et al. [14] and Jobard et al. [9].

Analogously, texture advection allows for streamlines by injecting dye while the vector field is held constant in time. Pathlines do not easily fit into our advection approach because older positions of a particle are discarded during the incremental update of the property field.

However, the concept of "short pathlines" can be implemented without any additional rendering pass. Here, not the current property field T_ϕ is used to texture the quadrilateral in the final output. Rather the property fields for the latest n_{path} time steps are added, yielding moving, short segments of pathlines. Up to four textures can be fetched on a GeForce 3 in a single rendering pass; theses textures are then blended via register combiners. This methods yields LIC with a limited kernel.

Note that streaklines, streamlines, and pathlines are identical for steady flows.

3.3 Noise-Based Approach

Another technique based on texture advection applies noise textures to visualize a vector field. The essential difference to the previous dye-injection techniques is the use of a noise texture as initial input for the property field T_ϕ. Therefore, a noise-based approach can easily be incorporated in the advection algorithm.

This approach is investigated in detail by Jobard et al. [9]. They propose several extensions to basic texture advection in order to improve image quality and spatial and temporal correlation. First, the use "edge correction" to allow for a continuous inflow of noise at the borders of the computational domain. Secondly, they propose "noise injection" to maintain a constant noise frequency even for a flow with positive flow divergence. Thirdly, they implement "noise blending" to enhance spatial correlation along a pathline segment. These extensions could be integrated in our advection approach similarly. In particular, noise injection could be implemented without any additional rendering pass, since a noise texture can be accessed in another texture stage and XORed via register combiners.

3.4 Random Particle Injection

Another variation of texture advection employs a randomly distributed injection of particles, thus combining the ideas of the previous two techniques. These particles are added by drawing randomly distributed and colored points into the property field during each iteration step, similarly to dye injection. By increasing the "inflow" and adapting the size of new particles, one can gradually adjust the visualization from being rather dye-based to being rather noise-based, i.e., from a sparse to a dense representation of the vector field.

Due to a continuous inflow of additional particles, the image tends to converge to a white image in the long term limit. Therefore, a continuous, large-scale dissipation of particles has to be employed. Its implementation is described in the following subsection.

3.5 Continuous Distribution of Outflow

In Sect. 3.1, a third, optional component *mag* for offset textures and a corresponding scaling of final RGB colors by a factor $M = k_{\text{scale}}\, mag + k_{\text{bias}}$ is mentioned. By setting $k_{\text{bias}} = 0$, $mag = 1$, and choosing a value less than one for k_{scale}, a uniform dissipation of particles can be implemented.

However, the *mag* component can be used in an even more flexible way. By adjusting this part of the offset texture, a gradual and locally controlled dissipation can be impressed. This feature can be used to mimic arbitrarily shaped outflow regions inside the computational domain or to fade out less important parts of the flow field, such as regions with small velocity. Once again, this variant of texture advection comes without any additional rendering pass.

4 Extension to 3D Texture Advection

4.1 Advection Based on Texture Shaders

Unfortunately, 3D textures are not yet supported on GeForce graphics boards in hardware, although they are part of standard OpenGL 1.2. Nevertheless, corresponding operations and adapted texture shading operations (`NV_texture_shader2`) have already been specified by nVidia. Since we expect this functionality to be available soon, we give a brief sketch of how the 2D algorithm from the previous section could be adapted to 3D flows and what extensions would be required for doing so.

First, both the computational domain as well as the vector field have to be extended from 2D to 3D. This is trivial for the vector field; just a 3D texture with three components has to be employed instead of a 2D texture. The extension to a 3D computational domain is easy, as long as the property field is stored as 3D texture. As the framebuffer is only 2D, the intermediate results have to be stored as 2D data. Here, we adopt the standard approach of equidistant parallel slices to sample the whole volume. The new property field contained in a slice is written back into the complete 3D texture by using `glCopyTexSubImage3D`, thus allowing an incremental update of this 3D property field. Other than in 2D flow advection, two versions of the property texture have to be stored to separate side effects during the iteration of Eq. (2).

Unfortunately, offset textures are restricted to 2D. In order to adapt this approach to 3D flow visualization, 3D offset textures would be required. Another approach could be based on 3D dot product textures, analogously to the description of dot product texture programs in Sect. 3. The basic algorithm could be as follows.

In the first texture shader stage, the data for $(v_x, y_y, 1)$ is fetched from a 3D texture. This texture is identical to the flow field, except for the blue component set to 1. The second stage fetches data for $(v_z, 0, 1)$ from another modified flow data set. In the next two stages, dot products are calculated for shifted texture coordinates u_x and u_y, based on the flow data from the first stage. The texture coordinates are $(-\Delta s, 0, x_{\text{domain}})$ and $(0, -\Delta s, y_{\text{domain}})$, respectively. Another dot product operation during the fifth stage can calculate u_z by using texture coordinates $(-\Delta s, 0, z_{\text{domain}})$ and flow data from stage two. Still in the fifth texture program stage, the new coordinates (u_x, u_v, u_z) are used for a dependent texture lookup in the 3D texture holding the property field for the previous time step. Unfortunately, the current GeForce 3 does only support up to four texture stages. Therefore, this algorithm would work only on improved chip sets with at least five texture stages. Since the gaming industry demands an increased number of texture stages, we are confident that this deficit will be overcome soon.

Texture-based volume rendering [1] can be easily combined with the 3D texture advection approach in order to render the property fields. Even more advanced texture-based volume rendering techniques [6] could be applied to achieve a high image quality on that kind of consumer graphics hardware.

4.2 Advection Based on Pixel Textures

A variant of the above algorithm can be implemented on SGI's Octane 2 workstations with VPro graphics pipes in order to demonstrate that texture advection is not restricted to nVidia chip sets. SGI's system does not support texture shaders but the similar, yet not as powerful concept of pixel textures. This technique allows RGB color values in the framebuffer to be interpreted as 3D texture coordinates for a trilinearly interpolating texture lookup.

As pixel textures can only be applied on imaging operations, the temporarily computed positions have to be explicitly written into the framebuffer, before using them as texture coordinates in the pixel texture lookup step. Unfortunately, standard texture modulation operations are not as flexible as texture shaders or register combiners, thus the framebuffer pixels representing the position information have to be created by rendering two quads. For the first quad, the colors $(0,0,z)$, $(1,0,z)$, $(0,1,z)$, and $(1,1,z)$ are assigned to the four vertices, with z being the current slice number scaled to $[0,1)$. Using Gouraud shading, this effectively leads to framebuffer colors that would create an identity mapping for ϕ. While drawing this first quad, the vector field texture is applied, using GL_ADD modulation. The looked-up texture values are first scaled by $2\frac{m\Delta s}{w}$ and biased by $-\frac{m\Delta s}{w}$ in order to lift the texture color values to the intended vector range; w denotes the size of the vector field texture in one direction. The post-filter scale and bias operation is, again, an SGI specific OpenGL extension. As the values are clamped before modulation, no signed adds are possible. Therefore, a second quad has to be drawn, this time without added primary color, but scaled and biased with negated values and blended into the framebuffer with GL_FUNC_REVERSE_SUBTRACT.

Now the framebuffer holds the positions for the next time step, encoded in color values. The lookup can be performed by copying the region onto itself with the pixel texture enabled. Finally, the slice is to be copied back into its 3D texture as with the previous approach.

5 Results and Discussion

Figures 3–5 show examples of 2D flow visualization based on the algorithms from Sect. 3. In all examples, the size of the computational domain is 1024^2 and the size of the vector field is 512^2. Figure 3 shows the visualization of a 2D circular flow via streaklines. Red dye is permanently injected at three fixed positions. Despite using only Euler integration, the resulting streaklines are (almost) closed to circles. With the chosen time step, a complete circle needs approximately 620 integration steps. The dye gradually smears out due to an implicit diffusion process. This issue of texture advection is caused by bilinear interpolation in the property texture of the previous time step.

This problem can be overcome either by noise injection, or by adding new, randomly distributed particles, as shown in Figure 4. A uniform dissipation absorbs this constant inflow of particles and spatial correlation is enhanced by using short pathlines.

Figure 5 shows the visualization of a 2D flow around a rod by using randomly injected particles and short pathlines. In a spherical region around the center, a continuous distribution of particle outflow with gradually changing degree of dissipation is employed.

These three examples were generated on a Windows PC with an Athlon 650 MHz CPU and a GeForce 3 board. In all examples, the time for one iteration of texture advection, including the update of the property texture T_ϕ, is 11 msec, corresponding to a frame rate of 90 fps. For the complete visualization process, including the final display on a 800^2 window, 37 fps are achieved. The rendering speed is reduced to 21 fps if the unsteady vector field is transferred from main memory to the graphics subsystem in every integration step.

Figure 6 shows an example of 3D flow visualization by using randomly injected particles. Both the 3D cylindrical vector field and the computational domain have dimensions 128^3; texture-based volume rendering uses approximately 380 slices. The implementation runs on an SGI Octane 2 with R12000 CPU (400 MHz) and VPro (V8). The computation of texture advection runs at 9 fps, the complete visualization process, including volume rendering, runs at 4 fps on a 320^2 window. We find that the injection of randomly distributed particles is especially useful in 3D flow visualization, as the density of the vector field representation can be gradually adjusted to achieve an appropriate, semi-transparent rendering.

Note that our methods for 2D and 3D texture advection greatly benefit from time-dependent visualization in interactive environments because the motion of particles can be fully recognized only here but not in static images.

An advantage of our 2D texture advection algorithm is an extremely high simulation and rendering speed. This method benefits from a very simple approach, which allows to advect a texture by drawing a single quadrilateral in only a single pass. Further visualization features, such as short pathlines or continously varied dissipation of particles comes without any additional rendering pass. Transferring intermediate data back into a texture is fast, since all operations take place entirely on the graphics board. The limited depth of a color channel does not restrain the accuracy in the calculation of particle positions because only difference vectors are stored as textures; new absolute coordinates are computed and used only within the texture stages of the rendering pipeline, where all computations are based on floating-points.

Conversely to previous work, such as by Heidrich et al. [8] or Jobard et al. [9], we use a completely Eulerian approach and implementation. At no part of the rendering pipeline, particle positions are explicitly stored in the framebuffer or a texture; our approach needs only textures for computational domain itself and the flow field. Moreover, we neglect some of the image enhancement steps proposed by Jobard et al. [9]. Therefore, the implementation is very simple and requires only one real rendering pass, apart from reading results back to texture memory. In this way, e.g., we achieve 90 fps for advecting a 1024^2 texture as opposed to 2 fps for 256^2 texture with Jobard et al.'s implementation (on an Octane 1 with EMXI graphics).

For the implementation of 3D texture advection, pixel textures and 3D textures on SGI's VPro are used. Here, one indirection step writing particle positions into the framebuffer is required because pixel textures can only be applied during image copy processes. Hence, the accuracy of computation of particle positions is an issue. With a color depth of 12 bits per channel available, sufficient accuracy is achieved for moderately sized volumes, such as 128^3.

6 Conclusion and Future Work

We have presented very fast, hardware-based, and simple-to-implement methods to visualize 2D and 3D unsteady vector fields. We have developed an algorithm for 2D flows, based on consumer graphics hardware, and for 3D flows, based on SGI's VPro. We think that these techniques are especially useful for interactive ad hoc visualization of flows.

In future work, we will focus on extensions of the 3D approach—in particular, on the next generation of consumer graphics hardware.

Acknowledgments

We would like to thank Klaus Engel, Martin Kraus, and Manfred Weiler for fruitful discussions and help with the implementation.

References

[1] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Symposium on Volume Visualization*, pp. 91–98, 1994.

[2] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *SIGGRAPH 1993 Conference*, pp. 263–272, 1993.

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

[4] W. C. de Leeuw and R. van Liere. Enhanced spot noise for vector field visualization. In *Visualization '95*, pp. 359–366, 1995.

[5] W. C. de Leeuw and R. van Liere. Spotting structure in complex time dependent flow. Technical Report SEN-R9823, CWI, Sept. 30, 1998.

[6] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Workshop on Graphics Hardware*, 2001.

[7] 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, June 1995.

[8] 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*, pp. 127–134, 1999.

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

[10] M. J. Kilgard, editor. *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, 2001.

[11] N. Max and B. Becker. Flow visualization using moving textures. In D. C. Banks, T. W. Crockett, and S. Kathy, editors, *ICASE/LaRC Symposium on Visualizing Time Varying Data*, pp. 77–87, 1996.

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

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1994.

[14] H.-W. Shen, C. R. Johnson, and K.-L. Ma. Visualizing vector fields using line integral convolution and dye advection. In *Symposium on Volume Visualization*, pp. 63–70, 1996.

[15] 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):98–108, June 1998.

[16] D. Stalling and H.-C. Hege. Fast and resolution independent line integral convolution. In *SIGGRAPH 1995 Conference*, pp. 249–256, 1995.

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

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

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

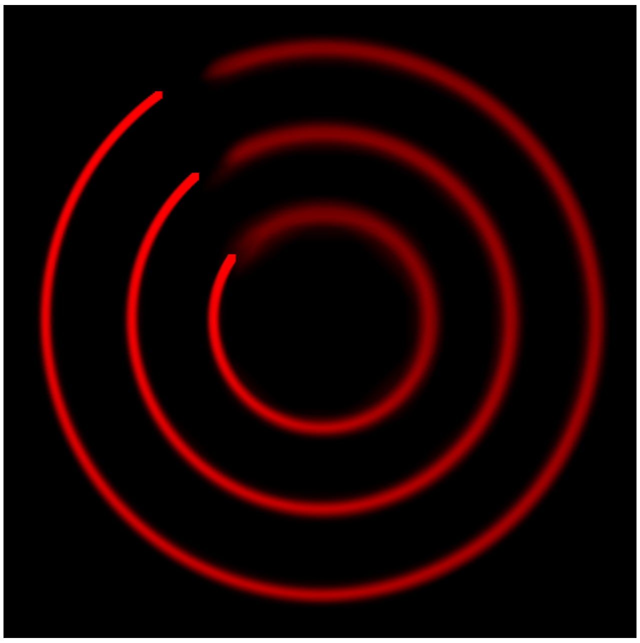

Figure 3: Visualization of a 2D circular flow $(v_x, v_y) = (y, -x)$. Red dye is permanently injected at three fixed positions, yielding a visualization via streaklines.

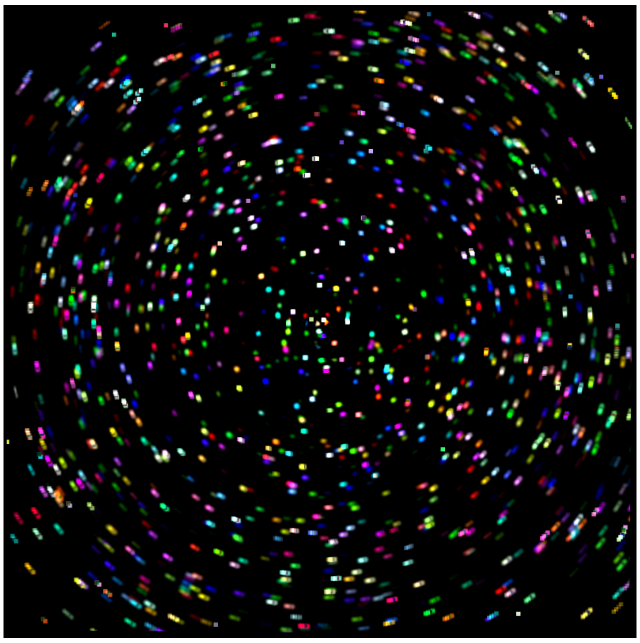

Figure 4: Visualization of a 2D circular flow, based on randomly injected particles. Short pathlines comprising the last four integration steps are used to enhance spatial correlation. A uniform dissipation absorbs the constant inflow of particles.

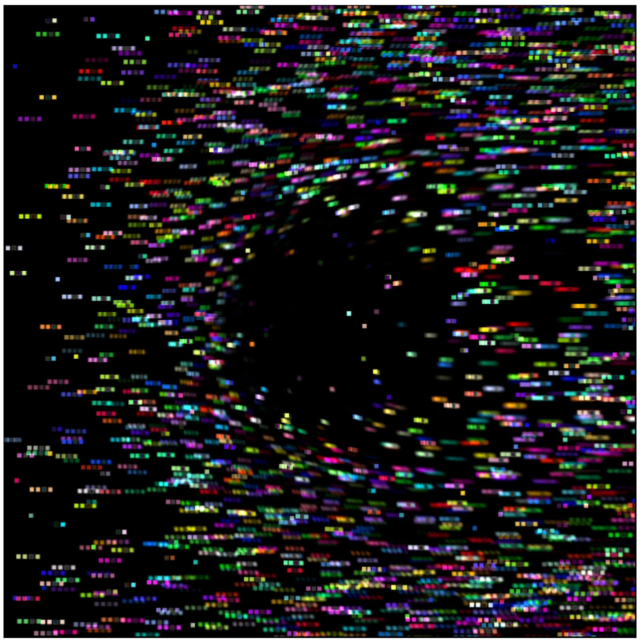

Figure 5: Visualization of a 2D flow around a rod by using randomly injected particles and short pathlines. In a spherical region around the center, a continuous distribution of particle outflow with gradually varying degree of dissipation is employed.

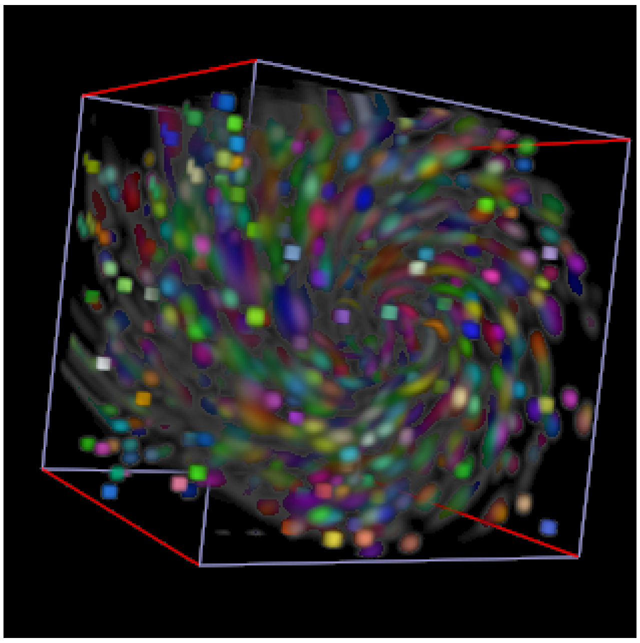

Figure 6: Visualization of a 3D cylindrical flow, based on randomly injected particles.