FLOW VISUALIZATION USING MOVING TEXTURES*

Nelson Max
Lawrence Livermore National Laboratory
Livermore, California

Barry Becker
Lawrence Livermore National Laboratory
Livermore, California

SUMMARY

We present a method for visualizing 2D and 3D flows by animating textures on triangles, taking advantage of texture mapping hardware. We discuss the problems when the flow is time-varying, and present solutions.

INTRODUCTION

An intuitive way to visualize a flow is to watch particles or textures move in the flow. The early color table animation of [1] was an example of this technique. More recently, van Wijk [2] has proposed advecting and motion blurring particles by the flow field. The LIC method [3, 4, 5] uses integrals of white noise textures along streamlines, moving the weighting function in the integrals from frame to frame to animate the texture motion. The motion blur of the particles and the directional texture blurring from the LIC integration create anisotropic textures which indicate the flow even in a still frame. However they are computationally intensive, and cannot generate animation in real time. The textured splats of Crawfis [6] use a loop of cycling texture maps with precomputed advecting motion blurred spots, and take advantage of texture mapping hardware. These are composited in back to front order in a local region near each data point, and oriented in the direction of the velocity vector, so that the precomputed advection cycle indicates the flow.

In this paper, we show how texture mapping hardware can produce near-real-time texture motion,

using a polygon grid, and one fixed texture. However, we make no attempt to indicate the flow direction in a still frame. As discussed below, any anisotropic stretching comes from the velocity gradient, not the velocity itself.

The basic idea is to advect the texture by the flow field. In [7] we gave an indication of the wind velocity by advecting the 3D texture coordinates on the polygon vertices of a cloudiness contour surface in a climate simulation. This was slow, because the 3D texture was rendered in software, and because advecting the texture was difficult for time-varying flows. In this paper, we replace the 3D textures by 2D texture maps compatible with hardware rendering, and give techniques for handling time-varying flows more efficiently.

The next section gives our technique for the case of 2D steady flows, and the following one discusses the problems of texture distortion. Then we discuss the problems with extending our method to time-varying flows, and our two solutions. Next we develop compositing methods for visualizing 3D flows. The final section gives our results and conclusions.

TEXTURE ADVECTION FOR STEADY 2D FLOWS

We start with a mathematical definition of texture advection, and then show how it can be approximated by hardware texture-mapped polygon rendering.

Let $F^t(x, y)$ represent the steady flow solution of the differential equation

$$\frac{dF^t(x, y)}{dt} = V\left(F^t(x, y)\right) \tag{1}$$

where $V(x, y)$ is the velocity field being visualized. Thus point P is carried by the flow to the point $F^t(P)$ after a time delay t. The flow F^t satisfies the composition rule

$$F^{s+t}(P) = F^s\left(F^t(P)\right) \tag{2}$$

for both positive and negative s and t. Thus $(F^t)^{-1}(P) = F^{-t}(P)$.

In this paper, we will assume that the initial texture coordinates at $t = 0$ are the same as the (x, y) coordinates of the region R being rendered. In practice, the texture is usually defined in a different (u, v) coordinate system related to (x, y) by translation and scaling, but for simplicity we will ignore the difference.

If $T(x, y)$ is a 2D texture being advected by the flow, then a new texture $T_t(x, y)$ is defined by

$$T_t(x, y) = T\left(\left(F^t\right)^{-1}(x, y)\right) = T\left(F^{-t}(x, y)\right).$$

Thus, to compute T_t at a point P, we go backwards along the streamline through P, to find the point Q such that $F^t(Q) = P$, and then evaluate the texture function T at Q. When animated, this will give the appearance that the initial texture T is being carried along by the flow. By equation (2) above, $F^{-(t + \Delta t)}(P) = F^{-\Delta t}\left(F^{-t}(P)\right)$. Thus the streamlines $F^{-t}(P)$ needed for the texture coordinates can be computed incrementally.

There are two problems with this formulation when the domain of definition for $V(x, y)$ or $T(x, y)$ is limited to a finite region R in which the velocity data or texture is available. First of all, if the streamline $F^{-t}(P)$ leaves the region R, the necessary velocities are not available to continue the integration. One must either extrapolate the known velocities outside R, or continue the streamline as a straight line using the last valid velocity. Fortunately, either of these extrapolation methods will give a correctly moving texture in animation. This is because the visible texture motion at a point P inside R is determined only by the velocity at P, and the extrapolation of the streamline beyond R serves only to determine what texture will be brought in from "off screen".

Second, even if $F^{-t}(P)$ is extended outside R, the texture may not be known there. The standard solution to this is to take $T(x, y)$ to be a periodic function in both x and y, so that it is defined for all (x, y). Most texture mapping hardware is capable of generating this sort of wraparound texture, by using modular arithmetic (or truncation of high order bits) to compute the appropriate texture map address from the x and y values. There are also tools to generate textures which wrap around without apparent seams [8].

To adapt this technique to hardware polygon rendering, the 2D region R is divided up into a regular grid of triangles, and the texture coordinates $F^{-t}(P_i)$ are only computed for the vertices P_i of the grid. During the hardware scan conversion, texturing, and shading process, the texture coordinates at each pixel are interpolated from those at the vertices, and the appropriate texture pixels are accessed. For triangles, the standard bilinear interpolation, which is not rotation invariant, reduces to linear interpolation, which is. For anti-aliasing, the hardware can use the higher order bits of the texture coordinates to weight an average of four adjacent texture map values (or four values in each of the two most-nearly-appropriate-resolution versions of the texture, if MIP mapping [9] is employed.)

TEXTURE DISTORTION

The flow $F^{-t}(P)$ can change the shape of a triangle, so that it becomes long and thin in texture space, as shown in figure 1. In the direction where the triangle is stretched by F^{-t}, the texture will be compressed by F^t. This distortion will not be present if the velocity is constant, so that F^{-t} and F^t are both translations. The distortion instead indicates anisotropies in the derivatives of V. For incompressible 2D flows, stretching

in one direction will be compensated by compression in a perpendicular direction. For compressible flows, there may be stretching in all directions at some positions, and shrinking in all directions at others.
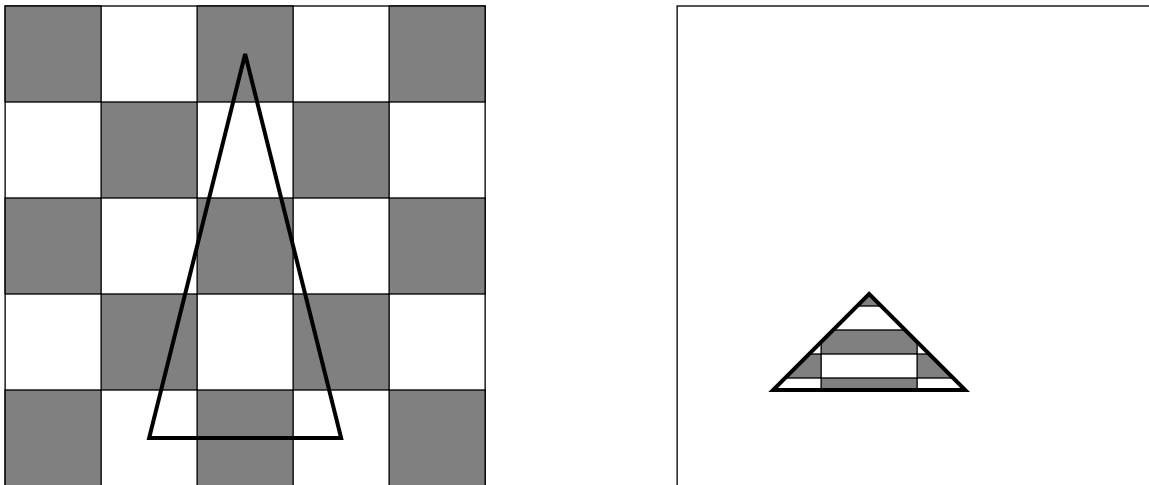

Figure 1. The triangle on the right is mapped to the texture on the left, which ends up being compressed vertically when the triangle is rendered.

During the animation of the texture advection, this distortion continues to build up, so that eventually the visualization will become useless. Therefore we periodically restart the texture coordinates back at their original positions in the regular grid. To avoid the sudden jump this would cause in the animation, we gradually fade up the new texture and fade down the old one, according to the weighting curves in figure 2. Each texture starts with weight zero, fades up over the old texture until it alone is present, and then fades
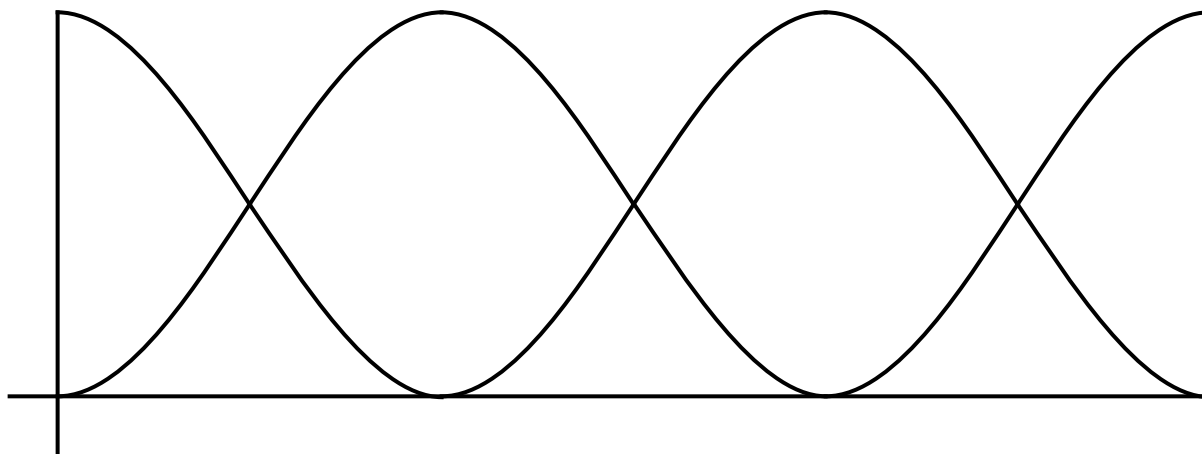

Figure 2. Three cycles of the weighting curves for fading the textures up and down.

down as an even newer texture takes its place. This "cross dissolve" can be done in hardware, using α com-

positing [10]. If the textures are random, and contain an adequate range of spatial frequencies, this cross dissolve will not disturb the perception of continuously flowing motion.

Since each texture is used for only a short time, the distortion does not become extreme. For a steady flow, one cross dissolve cycle ends with the same image at which it began, so an animation loop may be created which can be cycled rapidly and repeatedly on a workstation screen. Similar precomputed loops are possible with the surface particle [2], LIC [3], and textured splat [6] techniques.

TEXTURE ADVECTION FOR UNSTEADY 2D FLOWS

If the velocity V depends on t, the differential equation

$$\frac{dF^t(x, y)}{dt} = V\left(F^t(x, y), t\right) \tag{3}$$

defines a flow which no longer satisfies equation (2). For a fixed initial position Q, the curve $F^t(Q)$ is a particle trace $C(t)$ as in [11], rather than a streamline. To find the texture coordinates for P at time t_0 we need to find the point Q such that $F^{t_0}(Q) = P$. We must go backwards along the particle trace, and thus solve the differential equation

$$\frac{dC(t)}{dt} = V(C(t), t) \tag{4}$$

for the t range $0 \le t \le t_0$, with "final" condition $C(t_0) = P$, and then set $Q = C(0)$. With the change of variables $u = t_0 - t$, this is equivalent to the differential equation

$$\frac{dC(u)}{du} = -V(C(u), t_0 - u) \tag{5}$$

for the u range $0 \le u \le t_0$, with initial condition $C(0) = P$. Then $Q = C(t_0)$.

In the case of unsteady flow, the differential equations (5) for different t_0 are not related and define completely different particle traces, so incremental methods can no longer be used. In [7] we integrated equation (5) anew for each frame time t_0. To find the texture coordinates for frame t_0, we had to access the time varying velocity data for the whole t range $0 \le t \le t_0$, which is very inefficient for large data sets. Here we propose two more practical methods.

The first method is to derive a differential equation for the flow $G^t(x, y) = \left(F^t\right)^{-1}(x, y)$. This flow maps a point P to the texture coordinate point Q needed at frame time t, that is, the point with $F^t(Q) = P$. Thus we have

$$F^t\left(G^t\left(P \right) \right) = P. \tag{6}$$

Let G^t_x and G^t_y be the x and y components of the vector-valued function $G^t(x, y)$, and similarly let F^t_x and F^t_y be the components of F^t. Then by differentiating the components of equation (6) with respect to t by the chain rule, we get the pair of equations

$$\frac{\partial F^t_x}{\partial t} + \frac{\partial F^t_x}{\partial x}\frac{\partial G^t_x}{\partial t} + \frac{\partial F^t_x}{\partial y}\frac{\partial G^t_y}{\partial t} = 0,$$

$$\frac{\partial F^t_y}{\partial t} + \frac{\partial F^t_y}{\partial x}\frac{\partial G^t_x}{\partial t} + \frac{\partial F^t_y}{\partial y}\frac{\partial G^t_y}{\partial t} = 0.$$

Now by equation (3), $\dfrac{\partial F^t_x}{\partial t} = V_x$ and $\dfrac{\partial F^t_y}{\partial t} = V_y$, where V_x and V_y are the components of the velocity field at position $F^t\left(G^t\left(P \right) \right) = P$ and time t. Therefore we have

$$M\begin{pmatrix} \dfrac{\partial G^t_x}{\partial t} \\ \dfrac{\partial G^t_y}{\partial t} \end{pmatrix} = \begin{pmatrix} -V_x \\ -V_y \end{pmatrix}$$

where M is the Jacobian matrix for the flow $F^t(x, y)$:

$$M = \begin{bmatrix} \dfrac{\partial F^t_x}{\partial x} & \dfrac{\partial F^t_x}{\partial y} \\ \dfrac{\partial F^t_y}{\partial x} & \dfrac{\partial F^t_y}{\partial y} \end{bmatrix}.$$

Thus

$$\begin{pmatrix} \dfrac{\partial G^t_x}{\partial t} \\ \dfrac{\partial G^t_y}{\partial t} \end{pmatrix} = M^{-1}\begin{pmatrix} -V_x \\ -V_y \end{pmatrix}.$$

But since $G^t(x, y) = \left(F^t \right)^{-1}(x, y)$, the matrix M^{-1} is the Jacobian matrix J for $G^t(x, y)$:

$$J = \begin{bmatrix} \dfrac{\partial G^t_x}{\partial x} & \dfrac{\partial G^t_x}{\partial y} \\ \dfrac{\partial G^t_y}{\partial x} & \dfrac{\partial G^t_y}{\partial y} \end{bmatrix}.$$

Thus $G^t(x, y)$ satisfies the partial differential equations:

$$\frac{\partial G_x^t(x, y)}{\partial t} = -\frac{\partial G_x^t(x, y)}{\partial x}V_x - \frac{\partial G_x^t(x, y)}{\partial y}V_y$$

$$\frac{\partial G_y^t(x, y)}{\partial t} = -\frac{\partial G_y^t(x, y)}{\partial x}V_x - \frac{\partial G_y^t(x, y)}{\partial y}V_y. \tag{7}$$

These differential equations essentially say that the flow $G^t(x, y)$ is determined from the negative of the velocity V, as transformed into the texture coordinate system appropriate for $t = 0$, so they determine the texture flow necessary to give the desired apparent velocity at time t. The initial condition for G^t at $t = 0$ is that $G^0(P) = P$, that is, G^0 is the identity map. Equations (7) can be integrated incrementally in time by Euler's method. If $G^t(P_i)$ is known at time t for all vertices on a regular grid, the partials in the Jacobian matrix $J(P_i)$ can be estimated from central differences between the G^t values at adjacent grid vertices. (For vertices at the boundary of R, one-sided differences must be used.) Then, using the current velocity $V(G^t(P_i), t)$, increments $\Delta G_x = \frac{\partial G_x^t}{\partial t}\Delta t$ and $\Delta G_y = \frac{\partial G_y^t}{\partial t}\Delta t$ are found for the components of G^t. If necessary, Δt can be a fraction of the time step between frames, and/or the vertex grid used for solving equations (7) can be finer than the triangle grid used in rendering the texture, in order to make the solution more accurate.

The vertex grid spacing will affect the accuracy of the finite difference approximations to the partial derivatives like $\frac{\partial G_y^t}{\partial x}$. This accuracy is critical, because small errors in these partials will cause errors in position in the next frame, which may compound the errors in the partials, and cause them to grow exponentially from frame to frame. Here again, it is useful to fade out the current advected texture and fade in a new texture whose coordinates are reinitialized to the identity map, so that the integration errors cannot accumulate for too long.

The second method for handling unsteady flows is to move the triangle vertices by the flow $F^t(x, y)$, keeping their texture coordinates constant. This advects the texture directly, by moving the triangles, and carrying the texture along with them. To do this, we incrementally integrate equation (3), and no partial derivative estimates are needed for a Jacobian. However we again have a problem at the edges of the region R. The boundary vertices may move inside R, leaving gaps at the edges, or may move outside, causing too much texture to be rendered. The excess rendering is easily prevented by clipping all triangles to the boundary of R. The gaps can be eliminated by creating extra guard polygons around the edges of R, widening it to a larger region S. Whenever any vertex on the boundary of S crosses into R, a new row of guard polygons is added to the affected side of S. Again it is useful to integrate only over a limited time interval before reinitializing the texture coordinates, to avoid creating too many extra polygons.

FLOWS IN 3D

In three dimensions, one could advect 3D texture coordinates, but 3D texturing is not widely available.

We have instead used 2D textures on parallel section planes. We made the textured planes semi-transparent, and composited them from back to front using the α compositing hardware in our workstation. (This is how 3D texture mapping is usually implemented in hardware.) For the methods which change only the texture coordinates, we used the 2D projection of the velocity onto the section plane. For the method which moves the triangle vertices, we used the true 3D velocity, allowing the section surfaces to warp out of planarity.

Combining the compositing for the cross-dissolve of figure 2 with the compositing of the separate texture planes can lead to problems in the accumulated opacity. Given two objects with opacities α_1 and α_2, the resulting opacity from compositing both objects is $\alpha_1 + \alpha_2 - \alpha_1\alpha_2$. (See [10] or multiply the transparencies.) Suppose $f_1(t)$ and $f_2(t)$ are the two weighting curves shown in figure 2, with $f_1 + f_2 = 1$, and α is the desired section plane opacity. If we just take the two component opacities to be $\alpha_1 = \alpha f_1$ and $\alpha_2 = \alpha f_2$, the result is a composite opacity

$$\alpha_C \;=\; \alpha f_1 + \alpha f_2 - \alpha^2 f_1 f_2 \;=\; \alpha - \alpha f_1 f_2$$

The unwanted last term causes a periodic pulsation in α_C.

A solution is to use exponentials, which have better multiplicative properties. Define an "optical depth" $l = -\ln(1 - \alpha)$, so that $\alpha = 1 - e^{-l}$, and let $\alpha_1 = 1 - e^{-f_1 l}$ and $\alpha_2 = 1 - e^{-f_2 l}$. The resulting composite opacity is then

$$
\begin{aligned}
\alpha_C \;&=\; \alpha_1 + \alpha_2 - \alpha_1 \alpha_2 \\
&=\; 1 - e^{-f_1 l} + 1 - e^{-f_2 l} - \left(1 - e^{-f_1 l}\right)\left(1 - e^{-f_2 l}\right) \\
&=\; 1 - e^{-(f_1 + f_2)\, l} \;=\; 1 - e^{-l} \;=\; \alpha
\end{aligned}
$$

as desired.

Another problem with compositing texture planes of constant transparency is that the frontmost planes will eventually obscure the ones to the rear if the data volume gets large. One solution is to use variable-transparency textures, so that some regions of the texture are completely transparent. Another is to specify the transparency on triangle vertices using a separate scalar data variable which can select out regions of interest where the texture motion should be visible. In [7] we used percent cloudiness contour surfaces to specify the location of the advecting software-rendered texture. With our new hardware based technique, this cloudiness variable is used to specify the vertex transparency, and produces similar realism in much less time.

IMPLEMENTATION AND RESULTS

The different types of moving textures discussed were implemented as a class hierarchy in C++. Inven-

tor [12] quadmeshes were used to represent texture layers. An Iris Explorer module was then constructed in order to make use of color maps and data readers.

Figure 3 shows what happens when the vertices themselves are advected. The whole surface distorts, even in the direction perpendicular to the plane. In Figure 4 the texture coordinates are advected backwards while the vertices are held fixed. This gives the impression of motion in the direction of flow. Unfortunately the texture distorts too much over a long period of time. Also the texture vertices may move outside the defined domain. A solution to the first problem is to fade in a second texture with the texture coordinates reset to their original positions. The resulting cross dissolve is shown in Figure 5. The opacity for each texture is computed using exponentials, as discussed above, so there is no distracting variation in the overall intensity during animation. To avoid the problem of having to sample outside the domain, we used the inverse flow G^t for the texture coordinates, as explained above, while keeping the vertices fixed (Figure 6). This method also gives bad results over time if we do not periodically fade in a new unadvected texture as shown figure 7. Figure 8 illustrates how flow moves through particles of aerogel, a material with very low density which is a good thermal insulator. Figure 9 shows a frame from an animation of global wind data on a spherical mesh. The opaque regions represent high percent cloudiness. Although the vector field is static, the texture (but not the colors) appear to move in the direction of flow. Figures 10 and 11 depict steady flow near a high density contour in an interstellar cloud collision simulation (data courtesey of Richard Klein). Figure 10 has moving vertices, while figure 11 has moving texture coordinates. The color indicates density. A frame from an animation of unsteady wind data over Indonesia on a curvilinear mesh is shown in Figure 12. Percent cloudiness is mapped to color and opacity.

We ran our software on an SGI Onyx supporting hardware texture mapping. For a 32 by 32 slice of a volume (as in the aerogel example) we were able to achieve about four frames per second. To rotate a complete 50x40x10 volume, like the one shown in Figure 9, about 15 seconds was required.

REFERENCES

1. Shoup, Richard: *Color Table Animation*. Computer Graphics Vol. 13, No. 2 (August 1979) pp. 8 - 13

2. van Wijk, Jarke: *Flow Visualization With Surface Particles*. IEEE Computer Graphics and Applications, Vol. 13, No. 4 (July 1993) pp. 18 - 24.

3. Cabral, Brian; and Leedom, Lieth: *Imaging Vector Fields Using Line Integral Convolution*. Computer Graphics Proceedings, Annual Conference Series (1993) pp. 263 - 270.

4. Forssell, Lisa: *Visualizing Flow over Curvilinear Grid Surfaces using Line Integral Convolution*. Proceedings of IEEE Visualization '94, pp. 240 - 247.

5. Stalling, Detlev; and Hege, Hans-Christian: *Fast and Resolution Independent Line Integral Convolution.* ACM Computer Graphics Proceedings, Annual Conference Series, 1995, pp. 249 - 256.

6. Crawfis, Roger; and Max, Nelson: *Texture Splats for 3D Scalar and Vector Field Visualization.* Proceedings of IEEE Visualization '93, pp. 261 - 265.

7. Max, Nelson; Crawfis, Roger; and Williams, Dean: *Visualizing Wind Velocities by Advecting Cloud Textures.* Proceedings of IEEE Visualization '92, pp. 179 - 184.

8. Heeger, David; and Bergen, James: *Pyramid-Based Texture Analysis and Synthesis.* ACM Computer Graphics Proceedings, Annual Conference Series, 1995, pp. 229 - 238.

9. Williams, Lance: *Pyramidal Parametrics.* Computer Graphics Vol. 17 No. 3 (July 1983) pp. 1 - 11.

10. Porter Tom; and Duff, Tom*: Compositing Digital Images.* Computer Graphics Vol. 18, No. 4 (July 1984) pp. 253 - 259.

11. Lane, David: *UFAT - A Particle Tracer for Time-Dependent Flow Fields.* Proceedings of IEEE Visualization '94, pp. 257 - 264.

12. Wernecke, Josie: *The Inventor Mentor.* Addison -Wesley Publ. Co., Inc., 1994.
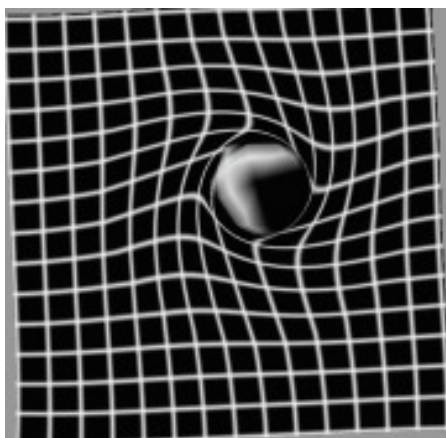
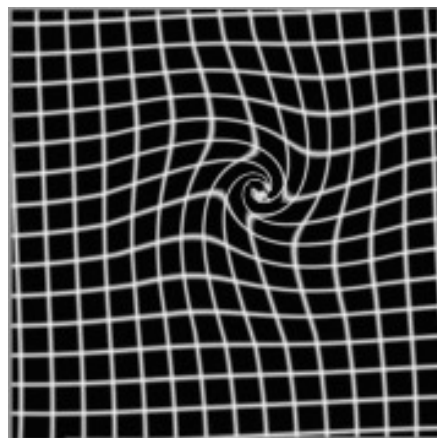Figure 3. Actual vertices are advected in 3D.

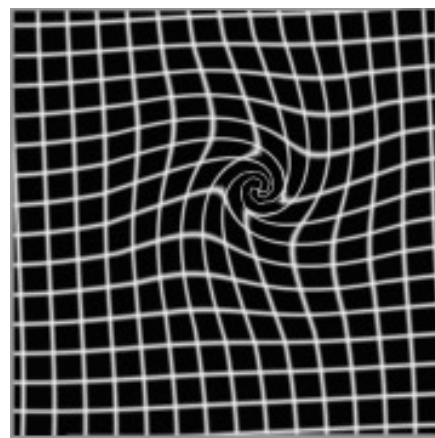Figure 4. Texture coordinates are advected backwards.

Figure 6. Texture coordinates are advected using vectors transformed by the local jacobian matrix, while vertices are held fixed.

Figure 5. Same method as figure 4, but with a new texture fading in as soon as the other becomes too distorted.
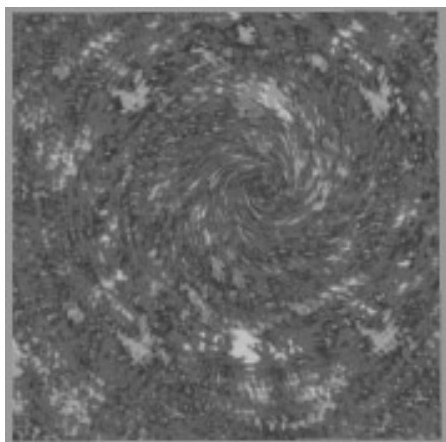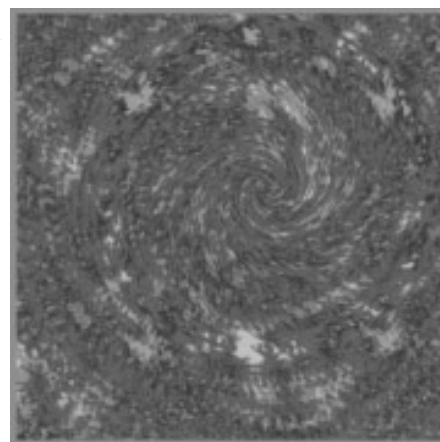

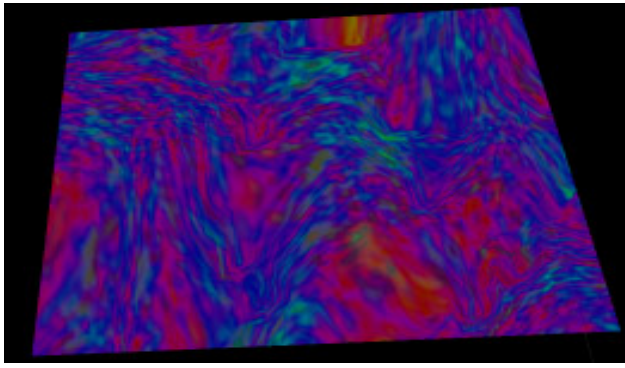Figure 7. Same method as figure 6, but with a new texture fading in as soon as the other becomes too distorted.

Figure 8. Method of figure 6 applied to a **steady** flow moving through particles of aerogel and using a colored texture.
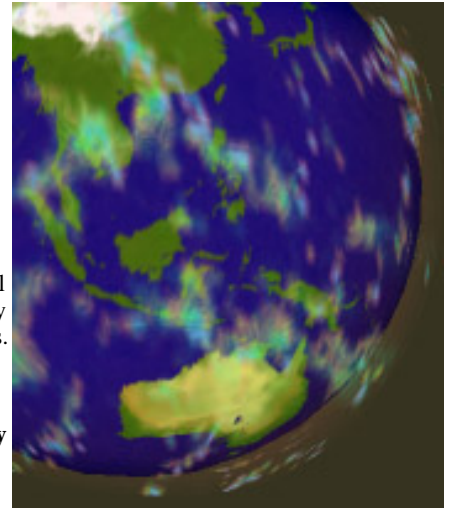
Figure 9. Method of figure 5 applied to a **steady** flow depicting wind data on a spherical mesh. Color and opacity from percent cloudiness.

Figure 11. Method of figure 6 applied to a **steady** flow representing a field from an interstellar cloud collision simulation.
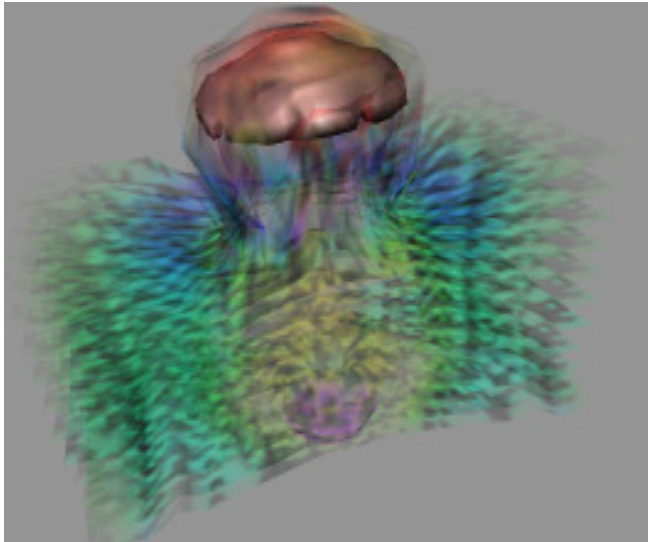

Figure 10. Several layers of textures advected using the method of figure 3. The layers are colored by density and move near a high density solid contour surface.
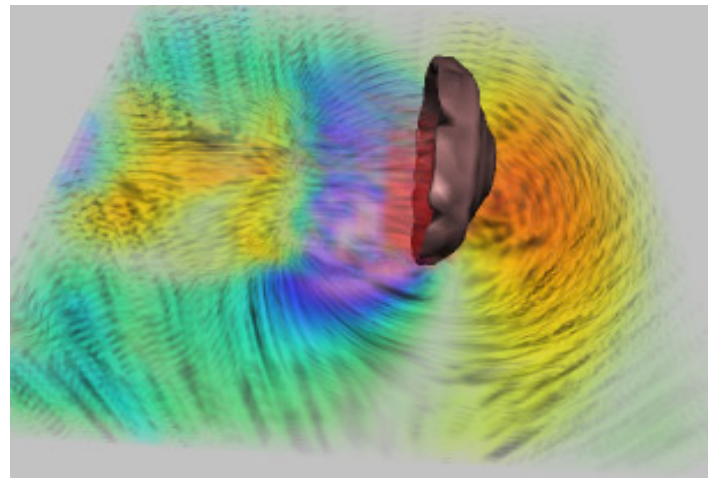

Figure 12. Method of figure 7 applied to an **unsteady** flow representing global climate data. Color and opacity indicate percent cloudiness. Both the winds and percent cloudiness vary in time.