# Applications of Texture Mapping to Volume and Flow Visualization

Nelson Max
Roger Crawfis
Barry Becker

Lawrence Livermore National Laboratory[1]

**Abstract**

We describe six visualization methods which take advantage of hardware polygon scan conversion, texture mapping, and compositing, to give interactive viewing of 3D scalar fields, and motion for 3D flows. For volume rendering, these are splatting of an optimized 3D reconstruction filter, and tetrahedral cell projection using a texture map to provide the exponential per pixel necessary for accurate opacity calculation. For flows, these are the above tetrahedral projection method for rendering the "flow volume" dyed after passing through a dye releasing polygon, "splatting" of cycled anisotropic textures to provide flow direction and motion visualization, splatting motion blurred particles to indicate flow velocity, and advecting a texture directly to show the flow motion. All these techniques are tailored to take advantage of existing graphics pipelines to produce interactive visualization tools.

## 1. Introduction

Scientific visualization is necessary to understand the output of large scale physical computer simulations in mechanics, hydrodynamics, and climate modeling. It is also useful in understanding data reconstructed from measurements like MRI tomography and X-ray crystallography. One often wants to visualize a 2D or 3D scalar, vector, or tensor field, or a steady or unsteady flow. Real time animation is useful for understanding time varying phenomena, and quick interaction is helpful in understanding 3D fields. This demands fast rendering of the images.

In this paper, we survey several rendering techniques we have recently developed, which achieve high speed by taking advantage of texture mapping hardware. Such hardware was originally developed for improved realism in training simulators, and is currently available on several high-end workstations. We expect it to become standard on future workstations, because of its ability to enhance image realism without increasing the complexity of geometric models.

In the following section, we briefly survey the concepts of texture mapping and compositing. Then in section 3, we present two methods which use texture mapping to render 3D scalar volume densities. Section 4 describes four methods which use texture mapping to visualize vector fields or flows.

## 2. Texture mapping

Texture mapping for computer graphics rendering was first done in software by Ed Catmull [1]. It greatly enhances the apparent detail of an image, without increasing the number of graphics primitives like polygons or surface patches. The basic idea is to precompute or scan in a rectangular raster image representing the desired texture. The horizontal and vertical raster coordinates in this image are used as texture parameters. When a primitive is rendered, texture parameters for each image pixel are determined, and used to address the appropriate texture pixels. If the parameters vary smoothly across the surface, the texture appears to be applied to the surface. For example, on polygons, the texture parameters can be specified at the polygon vertices, and bilinearly interpolated in screen space (or in object space for better perspective projection during scan conversion). On triangles, bilinear interpolation is equivalent to linear interpolation. For surface patches, the same parameters used for the surface shape functions can be used as texture parameters.

If the texture is a photograph complete with shading and shadows, it will not appear realistic when mapped to a curved surface. Therefore, the texture map is usually used to specify surface reflectivity, and then shading algorithms are applied.

Appropriate resampling is necessary when applying the texture to a surface. A surface pixel does not usually correspond exactly to a texture pixel, so its texture value should be a weighted average of several nearby pixels in the texture map. Heckbert [2] and Wolberg [3] describe a variety of anti-aliased resampling schemes. Here, I will describe the algorithms implemented in our workstation hardware. The texture parameters are interpolated with extra precision, so that the integer parts determine a texture pixel address, and the fractional parts determine fractional distances to the next adjacent pixel row or column. The fractional parts are then used to compute weights for a bilinear combination of four adjacent texture pixels.

This scheme gives a smooth resampling in the case that the mapped texture pixels are approximately the same size as the image pixels, and a smooth interpolation if the mapped texture pixels are larger than the image pixels. However if the mapped texture pixels are much smaller than the image pixels, some texture pixels which should contribute to the image may be missed entirely, since each image pixel involves at most four texture pixels. Lance Williams' MIP mapping [4] offers a partial solution to this problem. From the original texture map, another map is made at half the resolution, by averaging the pixel values in groups of four. The process is repeated, to make maps of 1/4 resolution, 1/8 resolution, etc. Then when the texture is used on a surface, an appropriate scale map is chosen, and the scheme of the previous paragraph, using a weighted average of four adjacent pixels, is applied. In order to prevent a sudden visible transition between two different resolution versions of the texture, a weighted average of the two closest-scale maps may be used, giving in effect, a weighted average of eight texture map values.

One application for texture mapping is to render complicated shapes like trees, clouds, or people, with a single polygon. This is done by storing both a color and an opacity in the texture map. The opacity $\alpha$ varies from 0 (completely transparent) to 1 (completely opaque). It is used to composite the textured polygon over the background, using one of the following formulae:

$$\text{composite} = \alpha \cdot \text{color} + (1 - \alpha) \cdot \text{background} \quad (1a)$$

$$\text{composite} = \text{color} + (1 - \alpha) \cdot \text{background}. \quad (1b)$$

1. Address: L-301, Lawrence Livermore National Laboratory,
7000 East Avenue, Livermore, CA 9550, USA.
E-mail: max2@llnl.gov, crawfis@llnl.gov, becker1@llnl.gov

See Porter and Duff [5] for a detailed explanation of these and other compositing formulas. In order to get the appropriate transparency effects, the objects on the scene must be sorted and composited in back to front order.

The six methods discussed below all involve compositing semi-transparent objects. They can be combined with other opaque geometric objects in the scene, without involving these opaque objects in the sorting. The opaque objects are rendered first, using a z-buffer to determine their visibility. Then when compositing the semi-transparent objects, their z is compared to the z-buffer value to determine where the compositing should take place, but the z-buffer is not updated.

We used a Silicon Graphics Onyx, with two MIPS 4400 CPUs and a Reality Engine graphics processor, in the work described below. This graphics processor performed in hardware all the texturing and compositing algorithms described above, including fractional precision texture coordinates, MIP mapping, color/opacity texturing, compositing, multiplication of texture map values by separate transparency, color, and shading values interpolated from the polygon vertices, and flexible z-buffer testing/updating options. Similar capabilities are offered on hardware from Kubota, Evans and Sutherland, and other manufactures.

## 3. Volume rendering

The goal of volume rendering is to produce an image of a varying density volume by projecting it on to an image plane. The color and opacity at a point in the volume can be specified as functions of a 3-D scalar field being visualized, and may also involve the gradient of the scalar field, the lighting direction, and the values of the scalar at other distant points. A survey of optical models for volume rendering is given in Max [6]. The basic ray tracing method for volume rendering integrates the color along a ray from the viewpoint passing through each pixel center, and continuing on into the volume. This integration must take account of the accumulating opacity along the ray, and compute

$$I = \int_0^D \text{color}\,(x(s))\,\exp\left(-\int_0^s \text{opacity}\,(x(t))\,dt\right) ds \qquad (2)$$

where $x(s)$ is the point at a distance $s$ from the viewpoint along the ray, $D$ is the distance to the edge of the data volume or to the first completely opaque object, and

$$\exp\left(-\int_0^s \text{opacity}\,(x(t))\,dt\right)$$

is the transparency of the volume between the viewpoint and $x(s)$. See Max [6] for a derivation of equation (2) and the standard efficient algorithms for estimating the integral by sampling along the ray. If the color and opacity values are only determined at the vertices of a volume grid, these values must be interpolated at the sample points on the ray.

An alternative to ray tracing is to project and composite semi-transparent volume elements in back to front order onto the image plane. The basic difference between projection methods and ray tracing methods is in the order of the loops over image pixels and volume data elements. For ray tracing, the outer loop is over the image pixels, and the inner loop is over the data elements along the ray. For projection, the outer loop is over the data elements, and the inner loop is over the image pixels they effect. This section discusses two techniques for using hardware texture mapping in projection methods. The ideal projection method should be mathematically equivalent to the ray tracing integral (2).

### 3.1.1 Splatting

In the first of our hardware assisted projection methods, the data elements are vertices of a regular grid. We use the splatting technique of Westover [7], which considers the continuous volume density as a weighted sum

$$W(x, y, z) = \sum_i \sum_j \sum_k h(x-i, y-j, z-k)\, V(i, j, k) \qquad (3)$$

where $i$, $j$, and $k$ are indices for an integer grid vertex, $V(i, j, k)$ is the data value at that vertex, $W(x, y, z)$ is the interpolated value at a general non-integer point $(x, y, z)$, and $h(u, v, w)$ is the weighting function, sometimes called the reconstruction kernel, describing the influence of each data value in the interpolation. Usually, $h(u, v, w)$ will have small compact support, so only a few terms in the sum (3) have non-zero weights. For trilinear interpolation

$$h(u, v, w) = (1. - |u|)\,(1. - |v|)\,(1. - |w|) \qquad (4)$$

and only 8 non-zero terms are involved.

Westover proposed integrating $h(u, v, w)$ along the viewing direction, to get a function

$$f(u, v) = \int_{-\infty}^{\infty} h(u, v, w)\, dw \qquad (5)$$

of only two variables, representing the influence of a single non-zero data value on the plane. This 2-D projection of the weighting function is called a splat. The splat $f(u, v)$ is stored at high resolution in a texture map. During rendering, the grid vertices are processed from back to front. For each vertex, the values for $f$ at the pixels it influences are retrieved (or interpolated) from the texture map, and multiplied by the color and opacity values for the vertex to get values to use in the compositing equation (la). Westover [7] did this in software, but we did it using the texture mapping and compositing hardware in our workstation.

There are some problems with the method described above. One often wishes to rotate the data volume, so that the resulting motion parallax gives visual cues about the 3D distribution of the volume density. In fact, interactive rotation was the original motivation for the splatting technique. However, the standard trilinear weights in equation (4) are not rotationally symmetric, so a separate integration as in equation (5) would be needed for each new orientation, and the orientations even vary within a single frame in the case of perspective projection. Therefore a rotationally invariant weighting function is desired.

Westover proposed using a gaussian function

$$h(u, v, w) = \exp\left(-(u^2 + v^2 + w^2)/\sigma^2\right)$$

which is rotationally symmetric, and has the simple integral

$$f(u, v) = \int_{-\infty}^{\infty} h(u, v, w)\, dw$$
$$= \sigma\sqrt{\pi}\, \exp\left(-(u^2 + v^2)/\sigma^2\right) \qquad (6)$$

This function does not have compact support, so it must be truncated to make a reasonable-sized texture. Laur and Hanrahan [8] approximated the gaussian (6) by a piecewise linear function, whose effect on the image could be produced by compositing a collection of triangles with linearly varying color and opacity. This could be done with standard scan conversion and compositing hardware, without the need for texture mapping. However Mach bands are visible at the triangle edges.

When rendering a volume as a composition of splats, they should blend together so that the individual splats are not visible. In an ideal situation, the sum (3) should be constant if all the data values $V(i, j, k)$ are equal to 1., *i. e.*

$$W(x, y, z) = \sum_i \sum_j \sum_k h(x-i, y-j, z-k) = 1 \qquad (7)$$

This will be the case if the trilinear interpolation weighting (4) is used, but can never be the case for a rotationally symmetric splat. Nevertheless, there are splats of small finite support which are superior to gaussians in this regard. In [9], we derived a piecewise cubic function

$$h(r) = h\left(\sqrt{u^2 + v^2 + w^2}\right)$$

which was optimized to make the sum (7) as constant as possible. It is

$$h(r) = \begin{cases} 0.557526 - 1.157743r^2 + 0.671033r^3 & 0 \le r \le s \\ 0.067599\,(t-r)^2 + 0.282474\,(t-r)^3 & s \le r \le t \\ 0. & t \le r \end{cases}$$

with $s = 0.889392$ and $t = 1.556228$, and the sum (7) deviates from 1.0 by only 0.25%. Figure 1 shows a slowly varying scalar function volume rendered with the integral (5) of this splat in the texture map. For each data point in back to front order, a small square polygon was oriented perpendicular to the viewing ray, and composited into the image using hardware texture mapping.

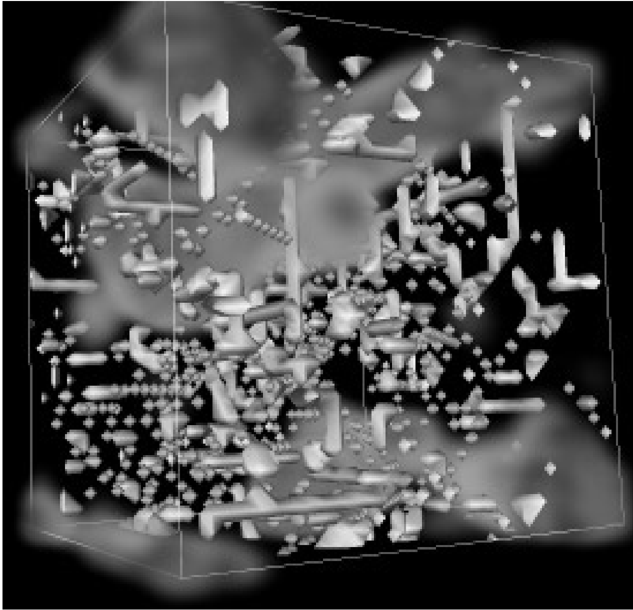

Figure 1. Air flow velocity through an aerogel, indicated by splats composited using z-buffer comparisons over opaque objects representing the aerogel.

The volume rendering appears smooth in this case. However if the vertex data represents point samples of a function with a discontinuity, individual splats may be visible near the discontinuity surface. This is similar to the aliasing that occurs when point samples are taken of pixel centers near an intensity discontinuity in a 2D image. Once this is done, no reconstruction kernel can eliminate the aliasing. Instead, the image must be appropriately filtered before sampling to eliminate high frequencies, so that each sample data value is a weighted integral of intensities near the pixel center. The same sort of filtering should be used when taking 3D samples to be used in 3D reconstruction. Sometimes this filtering comes automatically when the data are observed. For example, data samples from X-ray crystallography come from inverse Fourier transforms of measured diffraction intensities, and can easily be band limited, and tomographic densities determined by CAT or MRI scans are averages determined by the limited spatial resolution of the detectors and of the mathematical reconstruction. In other cases, when the density function is defined mathematically or from a geometric model, filtering must be done by integration around each sample point.

Another problem with splatting is that it does not exactly correspond to the integral (2). The back to front compositing means that the contributions from equation (5) to the colored intensity for a single splat will not contain opacity effects from that splat or any others that have previously been composited. The transparency factor

$$\exp\left(-\int_0^s \text{opacity}\,(x\,(t)\,)\,dt\right)$$

accumulated by the compositing process only includes effects from splats that are subsequently composited over the current one. This may not be noticeable in a single image, but when a volume is rotated, the sorting order may change, causing the image to visibly jump, particularly if different colored splats are adjacent. Westover [10] suggested a way to avoid much of this jumping. Among the three possible coordinate plane orientations in the $(i, j, k)$ lattice, the one most perpendicular to the viewing direction is chosen. Splats are used to sum (rather that composite) both the color and opacity in each lattice plane in this orientation, and then these resampled planes are composited as a whole, from back to front. This eliminates all mutual opacity effects between splats in the same plane. However, at some point during a rotation, the selection of the coordinate plane most perpendicular to the viewing direction will change, and a much stronger jump could result.

This modification could not be implemented easily on our hardware, since the hardware compositing only uses data coming down the graphics pipeline. Thus each color/opacity plane could be produced in hardware, but it would then need to be composited in software, or loaded into the texture map memory as a texture on one large polygon. Neither of these alternatives are fast, so we chose to composite each splat separately as in Westover [7].

### 3.2 Polyhedron compositing

Instead of compositing splats for the data points, an alternative is to composite polyhedra joining the data points. If the polyhedra can be correctly sorted in back to front order, equation (2) can be integrated separately along the viewing ray segments in each polyhedron. Garrity [11] has shown how to trace a ray through a collection of polyhedral cells provided that one knows which cell, if any, is on the other side of every face in the current cell.

This topological information is also useful for doing a global back to front sort on the cells. In [12] we considered the directed graph whose edges correspond to the cell faces, and are directed from the cell on the viewpoint side of the face plane to the other cell on the side facing away from the viewpoint. If the data volume is convex, with no holes or concavities, a topological sort of this graph (see Knuth [13]) produces a back to front sort of the cells if one is possible, or determines that it is impossible, in time $O(n)$, where $n$ is the number of cells and faces. Edelsbrunner [14] has shown that for a Delaunay triangulation, this sort will always succeed. If data is available at irregularly spaced points, with no preferred meshing into cells, the Delaunay triangulation is thus a good choice, and is also preferred because the resulting tetrahedra have good shape properties.

For volumes with holes or concavities, Williams [15] has supplemented this topological sort with a separate sort on the cells which have free faces facing towards the viewpoint, but his method is not guaranteed to give the correct answer. Stein *et al.* [16] give a general sort which requires no topological information and is always correct, but it takes time $O(n^2)$. Max [17] gives special sorts for some restricted geometries.

For polyhedral environments, polyhedron compositing [12] is potentially more efficient than ray tracing [11], because if the inner loop is over the cells, the scan conversion of the cells can take advantage of vertical and horizontal coherence. Max *et al.* [12] show how to do this for a convex cell by scan converting the front faces into one z-buffer, and the back faces into another. Lucas [18] and Max *et al.* [19] show how a global z-buffer eliminates the need for the back face buffer, provided there are no holes in the volume.

After scan conversion into the front and back z-buffers, one must calculate for each affected pixel the integral (2) along the ray segment inside the cell. If $\text{color}(x) = C\rho(x)$, and $\text{opacity}(x) = \tau\rho(x)$, for a volume density $\rho(x)$ of particles whose color $C$ and opacity $\tau$ are constant within the cell, then the integral (2) reduces to

$$
\begin{aligned}
I &= \int_0^D C\rho(x(s))\exp\left(-\int_0^s \tau\,\rho(x(t))\,dt\right)ds \\
&= -\frac{C}{\tau}\int_0^D (-\tau\,\rho(x(s)))\exp\left(-\int_0^s \tau\,\rho(x(t))\,dt\right)ds \\
&= -\frac{C}{\tau}\int_0^s \frac{d}{ds}\exp\left(-\int_0^s \tau\,\rho(x(t))\,dt\right)ds \\
&= -\frac{C}{\tau}\exp\left(-\int_0^s \tau\,\rho(x(t))\,dt\right)\Bigg|_{s=0}^{s=D} \\
&= \frac{C}{\tau}\left(1.-\exp\left(-\int_0^D \tau\,\rho(x(t))\,dt\right)\right) \qquad (8)
\end{aligned}
$$

The ratio $C/\tau$ can be interpreted as the surface glow color of the particles. (See [6].) If $\rho(x)$ is trilinearly interpolated from the cell vertices, as in a 3D version of Gouraud shading, with the viewing direction being one of the interpolation directions, then $\rho(x(s)))$ will vary linearly in $s$, so that the integral in (8) reduces to

$$I = \frac{C}{\tau}(1.-T) \qquad (9)$$

with the transparency

$$T = \exp\left(-\frac{D}{2}(\rho(x(0)) + \rho(x(D)))\right) \qquad (10)$$

where $x(0)$ and $x(D)$ are the entry and exit points of the viewing ray for the cell. Equation (9) is can be used in the standard compositing formula (1b) with opacity $\alpha = 1 - T$, and object color $C/\tau$.

The remaining two directions for the trilinear interpolation are along scan lines, and vertically between scan lines, so $\rho(x(0))$ and $\rho(x(D))$ can be bilinearly interpolated across the front and rear faces of the cell by standard scan conversion hardware. However, an exponential per pixel is still required for equation (10).

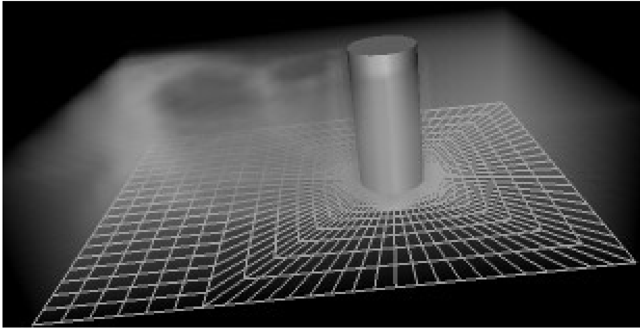

Figure 2. Z component of vorticity for water flow past a cylinder, rendered by compositing tetrahedra, using z-buffer comparison with the cylinder.

Here, finally, is where texture mapping can help. We divide the screen projection of a cell into polygons bounded by projected cell edges. Each polygon is the projection of a prismatic region of the cell bounded on the front by a single front-facing cell face, and on the rear by a single back facing cell face. The quantities $u = D$ and $v = (\rho(x(0)) + \rho(x(D)))/2$ then vary linearly across each such polygon. They are specified at the polygon vertices, and linearly interpolated as texture parameters by the scan conversion and texture mapping hardware. Then, in the texture table, we put the opacity $\alpha = 1. - T = 1. - \exp(-u\,v)$, which is used by the texture compositing hardware. Thus equations (9) and (10) are compatible with the graphics hardware pipeline.

The only software component of this scheme is the subdivision into polygons of the cell's projection. Wilhelms and van Gelder [20] give a line sweep algorithm for doing this for a general cell. In our implementation, we divided all cells into tetrahedra, and used the simpler scheme of Shirley and Tuchman [21] to subdivide the projected tetrahedra into triangles. In their hardware rendering, Shirley and Tuchman linearly interpolated $\alpha$ itself across the tetrahedra instead of doing the exponential per pixel. As demonstrated in [16] and [22], this approximation can lead to Mach bands, while our texture mapping scheme gives smooth images. Figure 2 shows a finite element model, where the color indicates the z component of the vorticity, produced by this technique.

In applications like this where the color is specified separately at each vertex, equation (9) is not valid, because the color $C$ is not constant across the viewing ray segment in a cell. Williams and Max [23] show how to calculate the integral in equation (8) when the color $C(x)$ and the particle density $\rho(x)$ both vary linearly along the ray. There are too many parameters in this calculation to be used as texture table indices, so the color must be calculated in software. To produce figure 2, we used the correct color only at the vertices of the screen projection triangles, and let the hardware interpolate it across the triangles. Only one vertex in the screen triangulation of each tetrahedron's projection corresponds to a ray segment for which color integration is required. The others, along the profile of the projection, correspond to single tetrahedron vertices where the color is already known.

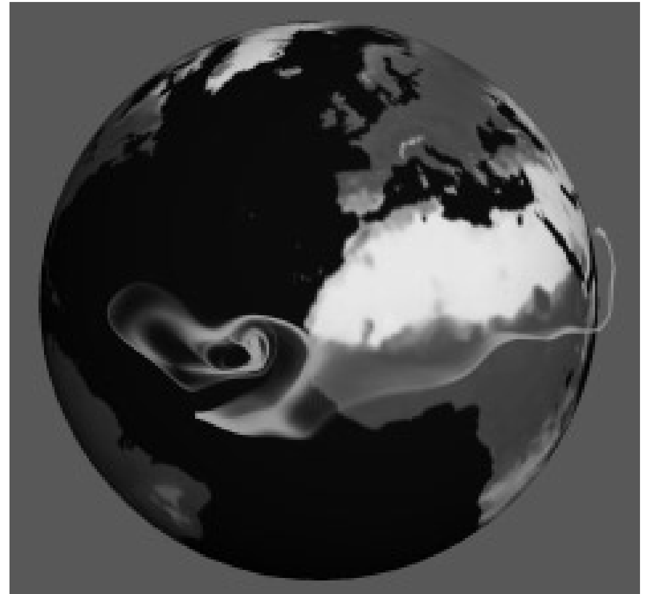

Figure 3. Flow volume for winds in a global climate simulation.

## 4. Flow visualization

We now describe four techniques for flow visualization which use texture mapping hardware. The goal is to produce animations which indicate the flow velocity.

### 4.1 Flow volumes

The first technique simulates the dye or smoke used to visualize flows in physical experiments, and takes advantage of the tetrahedron projection schemes just explained in section 3.2. A dye generating polygon is interactively positioned and sized by the user, and is automatically oriented perpendicular to the flow. The fluid which flows through this polygon is colored by the dye. The "flow volume" to be colored is adaptively subdivided into tetrahedra, which are rendered by the tetrahedra projection scheme above.

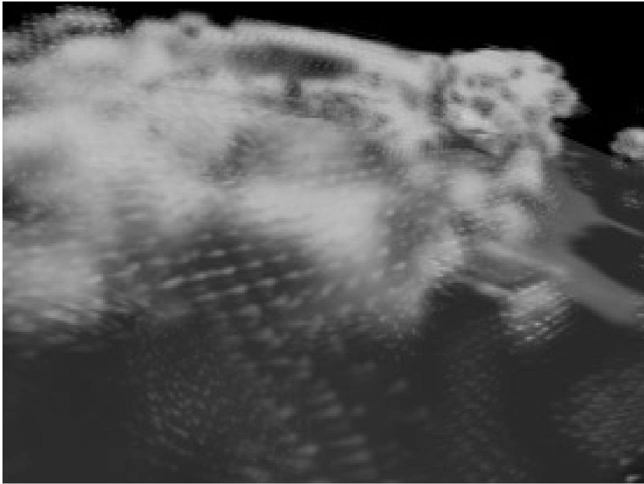

Figure 4. Wind velocity in global climate simulation, indicated by textures splats whose scalar component indicated percent cloudiness.

For a steady flow, a new layer of tetrahedra is added to the flow volume for each time step. In [22] we give details of the adaptive subdivision of each layer into tetrahedra and also show that if all the tetrahedra have the same dye color C/τ, the final color produced is independent of the order in which they are composited, even if the dye concentration ρ varies per tetrahedron. We thus eliminated the sorting step, and the whole process runs in real time. For compressible flows, we adjusted the concentration ρ to account for any change in volume during the flow.

Even without sorting, we could insure that the dyed regions were correctly obscured by opaque geometry, by first rendering the opaque objects into the z-buffer. Figure 3 shows our flow volume technique, applied to the wind velocity in a climate simulation. This figure was generated using an instantaneous wind velocity, assumed to be constant during the flow. Becker, Lane and Max [24] have recently extended this technique to time varying flows.

### 4.2 Textured splats

The splats of section 3.1 were generalized by Crawfis and Max [9], to indicate velocity direction, by using an anisotropic texture. The texture has streaks resembling motion blurred particles which grow brighter in the direction of motion, towards the right in the texture map. As with the scalar splats, these vector splats are rendered with small texture mapped squares, perpendicular to the viewing direction. But now these squares are oriented so that the streaks point in the projected vector direction. By taking advantage of the flexible arithmetical combination of polygon vertex color and opacity with texture map color and opacity available in our workstation hardware, we were able to render both a scalar variable and a vector variable in a single splat. (See [9] for details.)

In addition, we were able to make the texture move in real time to animate the flow. We used a cycle of separate texture maps. In each successive frame, the motion blurred particles in the texture moved farther to the right, and perhaps re-entered at the left. Each frame in the animation accessed the appropriate map in the texture cycle, so that the texture moved continuously in the flow. Even if there were too many splats to render in real time, we could rapidly accumulate the cycle of frames in the workstation memory, and then view them as an infinite loop. Figure 4 shows an example of this technique.

The use of textures for splatting is a very powerful and flexible concept. The texture used can be arbitrarily changed depending on the purpose. We are currently investigating techniques for representing multivariate data sets using the textured splats.

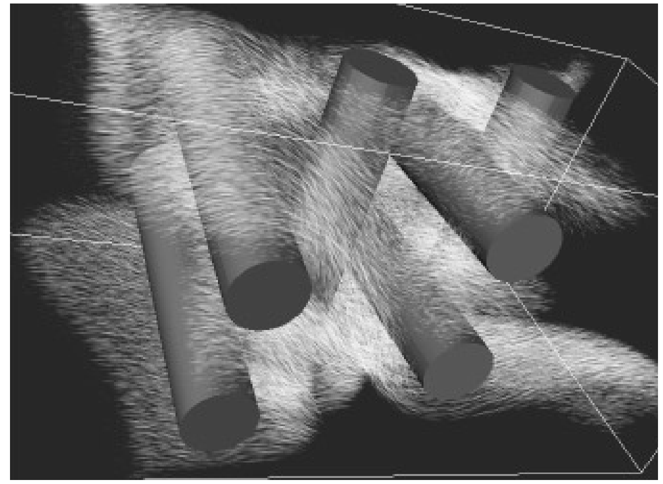

Figure 5. Spot noise indicating air flow through an air filter.

### 4.3 Spot noise

Van Wijk [25, 26] has used small motion blurred particles to visualize flows on contour surfaces. The particles were represented as ellipses with their long axes oriented along the direction of flow. They were composited in software. In [27], we used hardware texture mapping to render and composite these ellipses. A basic texture was defined on a square, to give a blurred circular spot. Then, instead of texturing a square oriented normal to the viewing ray, we used a rectangle, whose long side was oriented along the flow velocity vector. This turned the projected spot into a stretched ellipse, whose long axis increased with increasing projected velocity. The particles were advected by the flow, so that they indicated the flow velocity in animation. By advecting and rendering only the particles near a contour surface, we were able to produce real time animation. Figure 5 shows particles near a surface of constant velocity magnitude, from a simulation of air flow through a filter.

### 4.4 Texture advection

The previous two methods involved compositing separate textured splats for each data point or particle. We can also advect the texture on a continuous mesh of polygons. I will describe the method below for 2D flows, but in three dimensions, several meshed sheets of semitransparent textured polygons can be composited on top of each other to visualize a 3D flow.

For steady 2D flows, the simplest way to achieve texture advection is to keep the positions of the polygon vertices constant, and change their texture coordinates from frame to frame. The appropriate texture coordinates for a vertex $P$ at time $t$ are found by tracking a stream line through $P$ backwards to find $Q = Q(P, t)$, the point at time 0 which will move to $P$ at time $t$. This can be done incrementally, one frame at a time. The texture pattern should be periodic, so that streamlines of arbitrary length can still determine good texture map addresses.

For time varying flows, it is more difficult to define the point $Q$, because the analogue to the backwards stream line, a backwards particle trace, maybe completely different for each frame. In [28], we give two ways to advect the texture for an unsteady flow. For the first method, we derive a partial differential equation for $Q(P, t)$, and show how this equation can be solved incrementally from frame to frame. Another method is to simply move the mesh vertices themselves along the forward particle traces, which easily integrate incrementally from frame to frame. Care must be taken to clip polygons as they leave the region to be rendered, and to create any necessary new polygons at places where the flow enters the region. Figure 6 was produced by the first of these two methods.

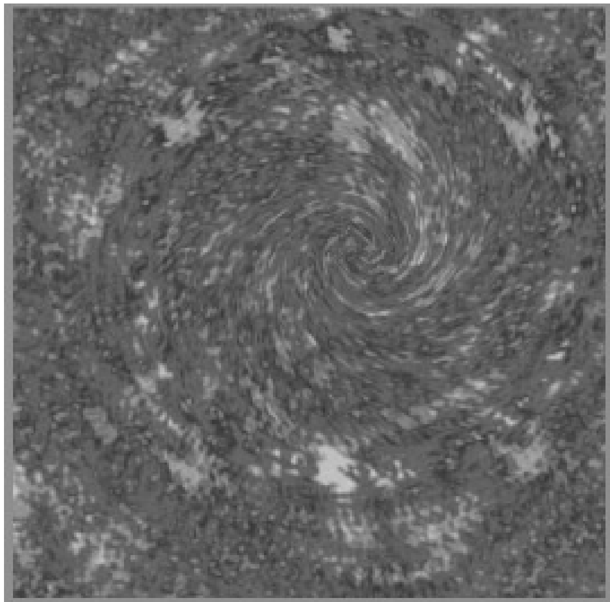

Figure 6. Texture advection indicating flow in a simulated tornado.

**Acknowledgments**

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48, with specific support from an internal LDRD grant. The sorting for figure 2 used an algorithm of Cliff Stein. The text of this paper was initially typed by Fran Faria.

**References.**

[1] Edwin Catmull, “A Subdivision for Computer Display of Curved Surfaces,” Ph. D. dissertation, University of Utah (1974).

[2] Paul Heckbert, “Survey of Texture Mapping,” IEEE Computer Graphics & Applications, Vol. 6 N. 11 (Nov. 1986) pp. 56 - 67.

[3] George Wolberg “Digital Image Warping,” IEEE Computer Society Press, Los Alamitos, CA (1990).

[4] Lance Williams “Pyramidal Parametrics,” Computer Graphics Vol. 12 No. 4 (August 1978) pp. 270 - 274.

[5] Thomas Porter and Tom Duff, “Compositing Digital Images,” Computer Graphics Vol. 18 No. 3 (July 1984) pp. 253 - 359.

[6] Nelson Max, “Optical Models for Direct Volume Rendering,” to appear in IEEE Transactions on Visualization and Computer Graphics Vol. 1. No. 2 (1995)

[7] Lee Westover, “Interactive Volume Rendering”, Proceedings of the Chapel Hill Workshop on Volume Rendering, ACM, New York,(1989) pp. 9 - 16.

[8] David Laur and Pat Hanrahan, “Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering,” Computer Graphics Vol. 25 No. 4 (July 1991) pp. 285 - 288.

[9] Roger Crawfis and Nelson Max, “Texture Splats for 3D Scalar and Vector Field Visualization,” Proceedings, Visualization ’93, IEEE Computer Society Press, Los Alamitos, CA (1993) pp. 261 - 266.

[10] Lee Westover, “Footprint evaluation for volume rendering,” Computer Graphics Vol. 24 No. 4 (Aug. 1990) pp. 367 - 376.

[11] Michael Garrity, “Ray Tracing Irregular Volume Data,” Computer Graphics Vol. 24 No. 5 (November 1990) pp. 35 - 40.

[12] Nelson Max, Pat Hanrahan, and Roger Crawfis “Area and volume coherence for efficient visualization of 3D scalar functions,” Computer Graphics Vol. 24 No. 5 (1990) pp. 27 - 33.

[13] Donald Knuth, “The Art of Computer Programming, Volume 1: Fundamental Algorithms,” 2nd Edition, Addison Wesley, Reading Mass. (1973)

[14] Herbert Edelsbrunner “An Acyclicity Theorem in Cell Complexes in *d* Dimensions,” Proceedings of the ACM Symposium on Computational Geometry (1989) pp. 145 - 151.

[15] Peter Williams, “Visibility Ordering Meshed Polyhedra,” ACM Transactions on Graphics, Vol. 11 No. 2 (April 1992) pp. 103 - 126.

[16] Cliff Stein, Barry Becker, and Nelson Max, “Sorting and Hardware Assisted Rendering for Volume Visualization,” Proceedings, 1994 Symposium on Volume Visualization, ACM Press (1994) pp. 83 - 89.

[17] Nelson Max, “Sorting for Polyhedron Compositing”, in “Focus on Scientific Visualization” Hagen H., Müller H. and Nielson G. (eds) Springer Verlag, Berlin, (1993) pp 259-268.

[18] Bruce Lucas, “A Scientific Visualization Renderer”, Proceedings of Visualization ’92, IEEE Computer Society Press, Los Alamitos CA, (1992) pp 227 - 234.

[19] Nelson Max, “New Techniques in 3D Scalar and Vector Field Visualization,” in “Computer Graphics and Applications,” S. Y. Shin and T. L. Kunii, editors, World Scientific, Singapore (1993) pp. 301 - 315

[20] Allan Van Gelder and Jane Wilhelms, “Rapid Exploration of Curvilinear Grids Using Direct Volume Rendering (extended abstract) Proceedings of Visualization ’92, IEEE Computer Society Press, Los Alamitos CA, (1993) pp. 70 - 77.

[21] Peter Shirley and Allan Tuchman “A Polygonal Approximation to Direct Scalar Volume Rendering,” Computer Graphics Vol. 24 No. 5 (November 1990) pp. 63 - 70.

[22] Nelson Max, Barry Becker, and Roger Crawfis, “Flow Volumes for Interactive Vector Filed Visualization,” Proceedings, Visualization ’93, IEEE Computer Society Press, Los Alamitos, CA (1993) pp. 19 - 24.

[23] Peter Williams and Nelson Max, “A volume density optical model,” Proceedings - 1992 Workshop on Volume Visualization, Boston, October 1992, ACM Order No. 429922, pp. 61-68.

[24] Barry Becker, David Lane, and Nelson Max, “Unsteady Flow Volumes,” submitted to IEEE Visualization ’95.

[25] Jarke J. van Wijk, “Spot Noise: Texture Synthesis for Data Visualization,” Computer Graphics Vol. 25 No. 4 (July 1991) pp. 309 - 318.

[26] Jarke J. van Wijk, “Flow Visualization with Surface Particles,” IEEE Computer Graphics and Applications, Vol. 13 No. 4 (July 1993) pp. 18 - 24.

[27] Nelson Max, Roger Crawfis, and Charles Grant, “Visualizing 3D Velocity Fields Near Contour Surfaces,” Proceedings, Visualization ’94, IEEE Computer Society Press, Los Alamitos, CA (1994) pp. 248 - 255.

[28] Nelson Max and Barry Becker, “Flow Visualization using Moving Textures,” submitted to the ICASE/LaRC Symposium on Visualizing Time-Varying Data (1995).