New Techniques for the Scientific Visualization of Three-Dimensional Multi-variate and Vector Fields

BY

Roger Alan Crawfis

B.S. (Purdue University) 1984

M.S. (University of California, Davis) 1989

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Nelson L. Max Chair

[illegible]

[illegible]

Committee in Charge

1995

Acknowledgments

Nelson Max has been instrumental to me throughout this work, as an advisor, a colleague and a friend. Time after time he has helped me through many technical hurdles and motivated me to pursue this degree. A special thanks to Kenneth Joy and Rao Vemuri, my other committee members. There patience to read and provide helpful comments is greatly appreciated.

Carol Hunter deserves credit for her wonderful faith in my abilities and pushing me into the world of research. I have benefited from many many other colleagues at Lawrence Livermore National Laboratory where I have been employed for these last 11 years. Barry Becker, Becky Springmeyer, John Zych, Don Dovey, and Cliff Stein have given me recent advice on my research and academic career.

Many scientists at LLNL have given me their data sets to explore my ideas upon. Thanks to Jerry Potter, Dean Williams, Don Dovey, Steve Pennock, Richard Klein, Bob Corey, Steve Ashby, Elaine Chandler and Tony Ladd for taking the time to get their data to me, explain it to me and provide feedback on these techniques.

Silicon Graphics, Inc. and in particular Robert Brown and Roy Hashimoto deserve thanks for lending me the initial SoOctreeSplat source.

This work was possible due to the generous support of the Computations Directorate, the Livermore Computing Department and the LDRD Office at Lawrence Livermore National Laboratory.

Finally, I need to acknowledge and bless the source of my power and energy: my family and friends. My dear wife Lisa and my three children, Cassandra, Christina and Austin make life a constant joy. Without their love and faith I would lose much of my excitement for research. To my many friends, I thank you for your fellowship. Lastly, I need to thank God, the creator and sustainer of life, who sent his one and only Son to claim me and bring me into fellowship with him. Even though I was as lowly as all earthly beings, God sent his Son to atone for my sins that I may live in peace and harmony with Him. That peace and assurance is a real joy in my life and allows me to accomplish what I need to do.

> So I find this law at work: When I want to do good, evil is right there with me. For in my inner being I delight in God's law; but I see another law at work in the members of my body, waging war against the law of my mind and making me a prisoner of the law of sin at work within my members. What a wretched man I am! Who will rescue me from this body of death? Thanks be to God — through Jesus Christ our Lord!
>
> NIV Romans 7:21-25

Abstract

Volume rendering allows us to represent a density cloud with ideal properties (single scattering, no self-shadowing, etc.). Scientific visualization utilizes this technique by mapping an abstract variable or property in a computer simulation to a synthetic density cloud. This thesis extends volume rendering from its limitation of isotropic density clouds to anisotropic and/or noisy density clouds. Design aspects of these techniques are discussed that aid in the comprehension of scientific information.

Anisotropic volume rendering is used to represent vector based quantities in scientific visualization. Velocity and vorticity in a fluid flow, electric and magnetic waves in an electromagnetic simulation, and blood flow within the body are examples of vector based information within a computer simulation or gathered from instrumentation. Understand these fields can be crucial to understanding the overall physics or physiology. Three techniques for representing three-dimensional vector fields are presented: Line Bundles, Textured Splats and Hair Splats. These techniques are aimed at providing a high-level (qualitative) overview of the flows, offering the user a substantial amount of information with a single image or animation.

Non-homogenous volume rendering is used to represent multiple variables. Computer simulations can typically have over thirty variables, which describe properties

whose understanding are useful to the scientist. Trying to understand each of these separately can be time consuming. Trying to understand any cause and effect relationships between different variables can be impossible. NoiseSplats is introduced to represent two or more properties in a single volume rendering of the data. This technique is also aimed at providing a qualitative overview of the flows.

Table of Contents

Chapter 1

Introduction

Computer simulations are playing an increasing role in understanding the world we live in. With faster and larger supercomputers, more and more complex simulations are migrating from two-dimensions to three-dimensions. Effective algorithms and tools for analyzing the resulting output of these simulations need to be developed. Research into visualizing 3D scalar fields has progressed over the past few years from isocontour surfaces, to direct volume rendering of regularly gridded data, and recently irregularly structured data, but only for a single scalar field. Volume Rendering of density clouds was offered as an effective solution for visualizing a single three-dimensional variable [25, 79, 92]. Many new techniques and extensions for volume rendering a density cloud have been developed [43, 53, 66, 82, 95, 96, 102, 104].

Understanding complex computer simulations requires the ability to comprehend not only a single scalar field, but the relationships between scalar fields as well as non-scalar fields such as vector fields. This thesis presents new techniques for representing vector fields, combined scalar and vector fields, and multiple scalar fields. Two primary driving forces motivates this work. The first is to take advantage of the mind/eye's ability to recognize and assimilate patterns. Natural textures or patterns can be either passive or active, that is, visible but not the eye's primary focus for passive textures, or the primary focus of the eye for active textures. Typical scientific visualizations produced on meshed data sets can lead to patterns resulting from the grid structure rather than the data field. Developing seamless patterns allows us to visualize the overall field and quickly notice irregularities in the texture, while at the same time allowing additional information to be portrayed with the texture.

Volume rendering of large data sets can be powerful and efficient. Unfortunately, volume rendering is currently limited to the representation of ideal isotropic density clouds. Integrating volume rendering with semi-transparent geometry representing other information is difficult when a substantial amount of small objects lie within or penetrate a volume cell. The second goal of this thesis is therefore to extend the state of the art in volume rendering to represent vector and multi-variate data sets.

The thesis is organized to provide background on the two distinct domains that it is involved with: vector field visualization and volume rendering, followed by the presentation of new techniques. First, however, Chapter 2 will present some definitions and background on scientific data that will be useful for subsequent chapters and help to motivate the research. Related work in volume rendering of scalar fields is presented in Chapter 3. Chapter 4 provides a concise background on techniques for representing vector fields. New techniques for representing vector fields are described in Chapters 5 through 7. Chapter 8 presents new techniques for representing multi-variate data. A summary with conclusions is contained in Chapter 9. The Appendices contain information on the object oriented systems that we used to implement our results complete with sample source code.

Portions of this research have been published in citations [21], [22], [23], [19], [18], [64] and [63].

Chapter 2

Scientific Information Background and Definitions

The complexity of today's computer simulations are requiring more and more complicated decomposition of the simulated space. Finite-difference and finite-element calculations require a meshing of the computational space where the solutions are to be solved. Several different mesh structures have been developed over the years for efficiency and flexibility. While we could always resample the solutions onto a mesh more suitable for visualization, these leads to errors or very large data sets. This chapter gives a brief definition of the various mesh types common in computational science with an emphasis on issues relating to scientific visualization. Issues dealing with data variables and time dynamics as applied to visualization are also presented. The material presented in this chapter should not be viewed as exhaustive. A concise treatment is presented here, perhaps more detailed than necessary for the remainder of the thesis.

2.1 Mesh Topologies

Three of these mesh types have a structure or topology such that they can be easily mapped to the integer lattice. Hence, the topology is implicit in the coordinate structure. Regular grids are topologically equivalent to an integer lattice by a simple constant scale factor. They have an equal spacing of data along all coordinate axes. Thus, data is defined at the points $(x0+h \cdot i, y0+h \cdot j, z0+h \cdot k)$ for $0 \le i < nx$, $0 \le j < ny$, $0 \le k < nz$. A total of $nx \cdot ny \cdot nz$ data points and $(nx-1) \cdot (ny-1) \cdot (nz-1)$ cells or zones are defined. The resulting cells are all equally sized cubes. Additionally, it is common to associate a mesh with all equally sized *bricks* as a regular grid. Here, there is are separate step sizes, h_x, h_y and h_z, for each of the coordinate axes. Regular grids are the easiest mesh topology to deal with and advantage can be taken of the constant size and shape of the elements [95, 102].

The second most amenable mesh topology for visualization purposes is the rectilinear mesh. Rectilinear grids are also topologically equivalent to the integer lattice, but by an independent and non-uniform stretching along each axis. They differ from regular grids in that the spacing *h* may vary along an axis. Thus, data is defined at the points $(x[i],y[j],z[k])$ for $0 \le i < nx$, $0 \le j < ny$, $0 \le k < nz$. The coordinate arrays can specify an arbitrary (non-negative) spacing. A total of $nx \cdot ny \cdot nz$ data points and $(nx-1) \cdot (ny-1) \cdot (nz-1)$ cells are defined. The resulting cells are bricks or rectangular hexahedra with possibly varying width, height and depth. All regular grids can be trivially promoted to a rectilinear grid.

Curvilinear grids are also topologically equivalent to the integer lattice. The integer lattice can be thought of as being stretched and bent, with each coordinate point explicitly specified: $(x[i,j,k],y[i,j,k],z[i,j,k])$. The mesh is restricted in that none of the *i-surfaces* (defined for by fixing *i* or taking a curvilinear slice through the mesh) may cross each other. Similarly, the *j-* and *k-surfaces* are restricted. The surfaces may touch at a point or edge, but are prevented from passing through each other. A total of $nx \cdot ny \cdot nz$ data points and $(nx-1) \cdot (ny-1) \cdot (nz-1)$ cells are defined. This mesh type can be difficult to deal with since the cells may have non-planar faces, and may degenerate into a non-six-faced polyhedron, such as a prism, regular pyramid or tetrahedra. Furthermore, there is no a priori traversal or intersection of the mesh with a line or ray, and a back-to-front sort — needed for some volume rendering algorithms — may be impossible. For certain simulations, it may be meaningful to represent the data on the integer lattice (commonly referred to as the ijk-grid), hence demoting the mesh to a regular grid and simplifying the visualization process. Thus, the coordinates of the mesh, $x(i,j,k)$, $y(i,j,k)$, and $z(i,j,k)$ are discarded and the data is represented as $\rho(i,j,k)$ rather than $\rho[x(i,j,k),y(i,j,k),z(i,j,k)]$. All regular and rectilinear grids can be easily promoted to a curvilinear grid by explicitly specifying each vertices' coordinate position.

Unstructured grids, or meshes, may not be topologically equivalent to an integer lattice. A loose definition is a mesh consisting of a collection of polyhedra. Every data point is explicitly defined: $(x[i],y[i],z[i])$ for $i \leq 0 < N$. A list of points, or indices, into these coordinates is given for each polyhedron, specifying its vertices. The connectivity of the faces may either be explicitly given or be implicit in the order of the points for fixed-topology cells. A total of N points and M cells are defined. An unstructured mesh presents many problems for visualization. Non-planar faces and degenerate vertex points are possible. Unstructured meshes may combine 3D polyhedra with 2D polygons (shell elements) and 1D lines (beam elements). Disjoint mesh topologies can occur at sliding interfaces where vertices from one polyhedra may lie on the faces of other polyhedra. Unstructured mesh topologies can be very difficult and possibly impossible to sort in a back-to-front order for volume rendering. Any regular, rectilinear or curvilinear mesh can be expressed as unstructured mesh by explicitly listing the points and topology.

Scattered data does not specify a mesh, only a list of points: $(x[i],y[i],z[i])$ for $0 \leq i < N$. For visualization techniques requiring polyhedra, a mesh can be generated using a Delaunay triangulation [75] or similar technique. The data can also be evaluated or sampled at points other than those specified, by a scattered data interpolation method such as

Hardy's multiquadrics [34] or Shepard's algorithm [27, 81]. Some methods allow for the triangulation to be constrained to a boundary other than the convex hull, avoiding interpolating through unknown regions [75].

Other techniques may combine several of these basic mesh types into one overall mesh. A multigrid mesh consists of a set of possibly overlapping rectilinear or curvilinear meshes. This is useful for domain decomposition techniques where a wing and an airfoil are easily specified as overlapping curvilinear grids. Since they overlap, it is not possible to simply render each mesh separately, but there is a priority that should be followed in the overlapping regions. Adaptive or Multi-resolution meshes consist of a hierarchy of meshes, where a large coarse mesh may have pieces defined on a much finer mesh, which in turn may also be subdivided. This mesh type may offer advantages for visualization. By only rendering the coarser meshes, interactive viewing is made possible. Artifacts can arise however between the bi-linear boundary of a coarse cell and the nonlinear boundary defined by the set of finer gridded cells adjacent to the coarser cell. A final complication can be a mesh with data points where the information is not available. This missing data is usually flagged by using an illegal value for the variable at those locations. Tweedy, et. al. [91] describe a technique for filling in holes left by missing data such that the holes do not grab the user's focus.

2.2 Data Variables

Complex simulations contain many different variables or properties at each grid point. For some disciplines, a core set of variables is stored at each grid point and other, usually more meaningful variables, are derived from this core set. The calculations for these derived variables can constitute a large portion of a visualization system. Generic visualization systems such as AVS or Explorer provide filters — or allow the user to construct their own filters — that act as functions, taking in one or more data variables and outputting derived variables.

Not all variables are scalar variables. In fact, the main set of variables in structural mechanics codes is comprised of a single tensor, single vector and two or three scalar variables. Effectively visualizing this small core of variables would theoretically allow the user or code designer to have a complete snapshot of the simulation. The current set of techniques for representing vector fields are not well suited for providing overviews of the data generated from three-dimensional codes. There are almost no techniques that effectively represent tensor fields. Chapter 4 is devoted to discussing the various techniques currently being used for representing vector fields. Delmarcelle and Hessilink [24] provide a classification technique for representing tensor fields globally, but only show its validity for two-dimensional data sets. This thesis will not address the complex issue of representing tensor fields.

Another complication for visualization of scientific data is the actual location where a data value has meaning. Data can be defined at the nodal points, at the zone "center" or at selected interpolation points within the grid cells. Data defined only at the zone centers usually implies that the simulation code treated the value as a constant over the cell. This can lead to an aliased image since the function is not smooth between cells. Determining the proper representation for an abstract variable can also be a difficult task. For example, the variable percent cloudiness in global climate modeling is more accurately represented as a higher-resolution binary mask indicating completely opaque or completely transparent regions within a cell, rather than as an averaged semi-transparency value for the entire cell.

None of these variables exist in isolation, rather they all work as a whole to model properties of a simulation. A single property of a natural process may actually be represented by several variables. Clouds in a global climate model can be composed of relative humidity, altitude layer, percent cloudiness, liquid water content and downward and upward solar radiation. A user may be specifically interested in certain events that happen when specific relationships between two or more variables are achieved. The correlation or relationships between variables can thus be very important to understand. Few visualization techniques exist for understanding relationships between different variables. Chapter 8 describes a technique for visualizing multivariate data sets, especially those generated from large 3D computer simulations.

2.3 Time

Most natural phenomena are not static. Likewise, many computer simulations calculate data not at a single instance of time, but for a whole series of time steps. This added dimension of time has some of the same characteristics as the spatial dimensions discussed above. Some simulations calculate the data at regular intervals, some with varying time steps. When visualization is a post-process, the data is typically written at regular intervals only. However, newer algorithms — such as those being developed for parallel processing — may even be calculating the data at different times for different data points. For time-dependent visualizations, these issues need to be taken into account. Not only may the variable values change over time, but the mesh may change as a function of time. Structural mechanics codes propagate the mesh vertices in a simulation. If the mesh becomes too distorted a rezoning of the mesh is performed either automatically or under user supervision. Here not only is the mesh changing shape over time, but it can also change topology.

Visualization techniques can make use of time even for a steady-state problem or a single time step of a solution. Animation, especially of flows, is a useful tool, since the eye can readily pick out moving patterns from those that are stationary. Two of the new techniques, Line Bundles and Texture Splats, described in this thesis will take advantage of this.

Chapter 3

Volume Rendering

Volume rendering attempts to mimic the energy absorption and transmission through space, calculating the energy propagated through a density cloud that eventually reaches a viewer's eyes. Many simplified models of this energy propagation have been used in the literature. All offer a compromise between accuracy and efficiency. For scientific visualization purposes a realistic representation is not always what is wanted since the cloud being represented is a synthetic cloud derived from an abstract variable. Interactive rendering rates are usually preferred over physically accurate lighting and energy transmission calculations. However, care must be taken not to introduce misleading artifacts or aliasing effects. This chapter will first present different optical models for integrating a density cloud. Different algorithms for rendering a density cloud will then be surveyed.

3.1 Optical Models

The simplest optical models are either pure absorption of a back light by the cloud or pure addition of energy from the cloud. Pure absorption is similar to what we would get from an X-Ray. Consider a ray passing through a density cloud (Figure 3-1). The ray can be written parametrically in *s* going from 0 to 1. The energy transversing a ray is attenuated by a differential amount at each position. This can be written as the change in the energy or intensity[1], *I*, as:

$$\frac{dI(s)}{ds} = -\tau(s)I(s)$$

where τ*(s)* is the extinction or attenuation coefficient, and *I(s)* is the amount of energy along the ray. Hence, we loose *-dI(s)* of the incoming energy as we progress along the ray from *s* to *s+ds.* The solution to this equation is:

$$I(s) = I_0 e^{-\int_0^s \tau(t)\,dt}$$

where I_0 is the amount of energy entering the cloud at *s=0.* We define the transparency at *s* as:

$$T(s) = e^{-\int_0^s \tau(t)dt}.$$

[1]Energy and intensity in general are different, but related. This thesis uses the terms interchangeably. See Cohen and Wallace [15] or Hall [31] for a more detailed description of the differences.

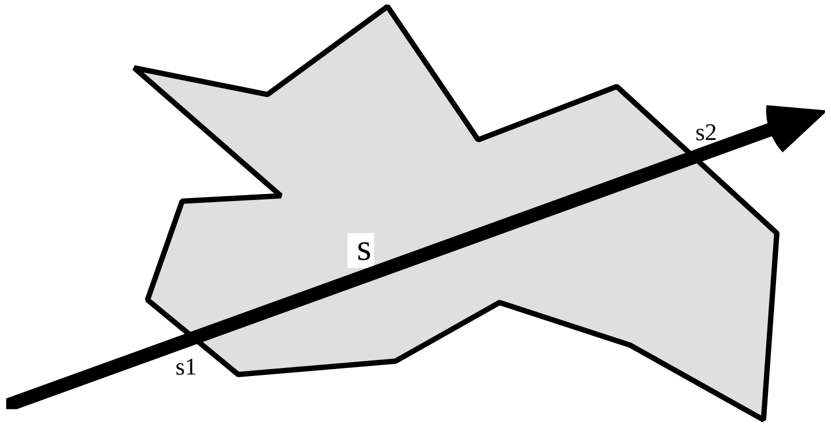

Figure 3-1.

For constant attenuation, $\tau(s) = \tau$, we get $T(s) = e^{-s\tau}$. Max, et. al. [66] show that for linearly varying attenuation, $\tau(s) = as + b$, the transparency $T(s) = e^{-\frac{s}{2}(\tau(s_1)+\tau(s_2))}$, where $\tau(s_1)$ and $\tau(s_2)$ are the attenuations at the entry and exit points of the ray through the volume (see Figure 1).

Alternatively, we can model a fluorescent or self-emitting cloud. Here, no absorption is performed, rather energy is added as we traverse the ray. The change in intensity is thus:

$$\frac{dI(s)}{ds} = g(s)$$

where $g(s)$ is the glow intensity per unit length. The intensity is thus the integral of g. If g is constant, then $I(s) = sg$.

Combining absorption and self-emitting glow gives us the model of Sabella [79]. The density cloud is assumed to consist of many small particles that both occlude and emit light. The differential change in intensity can be written as:

$$\frac{dI(s)}{ds} = g(s) - \tau(s) I(s).$$

By multiplying both sides by the integrating factor $e^{\int_0^s \tau(t)dt}$, we get the differential equation:

$$\frac{d}{ds}\left(e^{\int_0^s \tau(t)dt} I(s) \right) = e^{\int_0^s \tau(t)dt} g(s)$$

which when integrated gives the intensity:

$$I(s) = \int_0^s e^{-\int_u^s \tau(t)dt} g(u)\,du + I_0 e^{-\int_0^s \tau(t)dt}.$$

This equations states that the initial intensity, I_0, is attenuated by the total absorption of the cloud and any self-emitting glow, $g(u)$, is attenuated by the absorption of the remaining particles along the ray.

More complex models can be developed, but require complex numerical calculations for the intensity. Since the intensity is calculated for each projected pixel of each volume element, the cost for calculating it adds substantially to the cost of the overall rendering. Kajiya and von Herzon [42] present the theory for complex light scattering and offer an approximate solution. They also give a simpler two-pass method for accounting for the absorption between the light source and the scattering particle. Max [61] presents a solution for approximating the internal scattering within a volume density. Williams and Max [104] present additional computations for dealing with varying color components across a volume cell.

Traditional animation has relied on creating several layers or painted cells that are stacked sequentially in a box and photographed. Areas that are not changing during a certain sequence only have to be painted once. A second conceptual model for volume rendering is a large stack of very thin semi-transparent layers. This breaks the problem into compositing a set of 2D images with mattes. Porter and Duff [74] describe a language for compositing images in back-to-front order using an opacity at each pixel. The over operator,

$$I_{new} = I_{old} + (1 - \alpha_{old}) I_{layer}$$

$$\alpha_{new} = \alpha_{old} + \alpha_{layer}(1 - \alpha_{old}),$$

from Porter and Duff [74] is commonly used to composite the individual layers. For a three-dimensional volume it is convenient to have the 2D images correspond to 2D slices of the volume data perpendicular to one of its principal axes. An axis must be chosen that is not parallel to the image plane and is usually chosen such that is the coordinate most perpendicular to the screen. This implies that the slices used are view dependent, and changes may occur when the decision as to which set of slices to use changes. Most shear-warp algorithms (Section 3.4) follow this model. Reconstruction and anti-aliasing are critical in constructing the individual slices. If slices are constructed in an extended viewing space, then care must be taken to resample the volume accurately. Cabral, Cam and Foran [9] present a technique to perform the resampling using the three-dimensional hardware texture mapping of the Silicon Graphics' Reality-Engines [2]. Reconstruction and anti-aliasing are also critical in constructing the individual slices.

The calculation of the opacity for each layer can either be chosen aesthetically or using an approximation to the equations above.

3.2 Ray Tracing

Whitted [97] introduced the computer graphics community to ray tracing, a powerful and flexible technique for tracing the path and reflections of light. Ray tracing is conceptually simple and flexible. Viewing rays are traversed from the eye through each pixel and into the scene. As rays intersect a surface, the radiation from the surface is calculated as the resulting pixel's color. The reflected energy from the surface can itself be calculated by tracing rays back to light sources, by tracing rays in the ideal specular reflection direction, or by tracing rays to other radiating surfaces. Many researchers have developed methods to extend ray tracing and make it more efficient [3, 29]. In particular, Sabella [79], Upson and Keeler [92], Levoy [53] were among the first to apply it to volume rendering. For regular grids, two common stepping methods exist. One is to sample at fixed interval's along the ray, interpolating to those data points as we progress down the ray. A interval can be calculated such that the sampling guarantees every cell will be visited by at least one ray. A second approach is to calculate the entry and exist points for each cell. The integrated intensity through the cell can then be calculated and composited into the image. Algorithms for efficiently stepping through a regular grid are described in [1, 84, 86, 105, 106]. Garrity [28] presents an algorithm for tracing through unstructured meshes.

Ray tracing usually proceeds from the eye point into the mesh. This requires that the accumulated opacity be saved as the ray progresses. If the opacity reaches one, implying that no light further along the ray will reach the viewer, then the ray can be terminated early. Levoy [53, 54] uses this technique for calculating contour surfaces. Yagel and Shi [107] illustrate a technique to preprocess the volume data such that opaque surfaces are quickly intersected when ray tracing. Many other techniques [11-14, 48-52, 55, 68, 70, 90, 94, 105, 106] have been developed to accelerate or extend the ray tracing of volume densities.

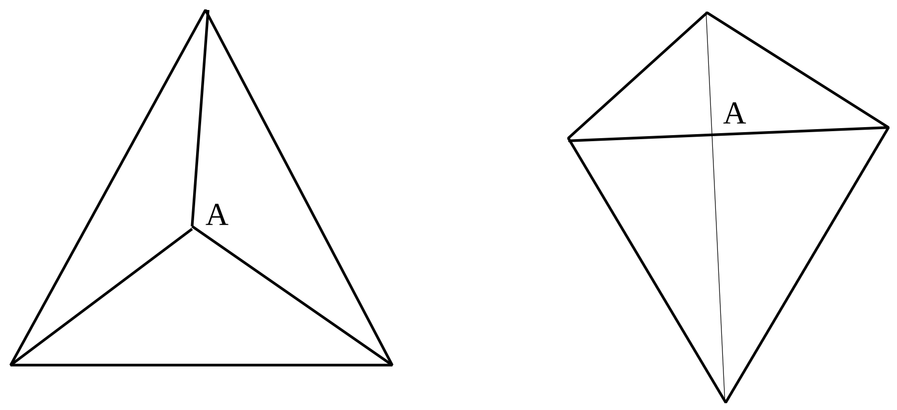

Figure 3-2. Common Tetrahedral Projections

3.3 Cell Projection

Cell projection methods are precisely what their name implies. They project each face of a cell onto the image plane. The faces of a projected polyhedron are either front facing or back facing with respect to the viewer. Max, Hanrahan and Crawfis [66] present a software scan conversion of polyhedra that scan converts the front faces separately from the back faces. The optical density is then integrated at each pixel given the interpolated optical properties and z-depth of the front and back faces at that pixel. Shirley and Tuchman [82] present a similar method that takes advantage of the graphics hardware for scan conversion and interpolation. Their solution first subdivides a mesh into tetrahedron. The projection of each tetrahedron is divided into one to four triangles, each bounded by the projections of the tetrahedron's edges. Figure 3-2 (a) and (b) show two non-degenerate cases (where no vertex projects onto another vertex or edge), requiring three and four triangles respectively. The vertex marked A in each projection corresponds to a viewing ray segment through the tetrahedron, whose length *l* can be computed from the geometry. The optical method used in Shirley and Tuchman [82] is to evaluate the color and opacity once at the "thick" vertex A. The opacities are zero at the other "thin" vertices on the profile. Bilinear interpolation (linear on triangles) in the hardware rendering pipeline is used to interpolate the color and opacity across the triangle, and composite each triangle over the background.

The linear interpolation of the opacity and colors does not accurately model the exponential absorption in the rendering equation. The opacity, $\alpha(x)$, at each pixel is correctly defined as $1 - e^{-\tau l}$. This requires a linear interpolation of the quantity τl, and then an exponential per pixel. Taking the exponential of a number is not commonly available in hardware. Crawfis, Max and Becker [22] used the texture mapping hardware available on high-end graphical workstations for this problem. For the case of constant τ per tetrahedron they put the quantity 1-exp(-u) in a one-dimensional texture table, indexed by u. The texture coordinate u is set to zero at the thin vertices, to τl at the thick vertex, and interpolated by the shading hardware before indexing into the texture table. If τ varies linearly within each tetrahedron, the product τl varies quadratically inside each projected triangle. Quadratic interpolation of texture coordinates was implemented in hardware on the Apollo DN10000VS [45]. Crawfis, Max and Becker [22] use a 2D texture table with τ and l, and put $1\text{-exp}(-\tau l)$ in the table.

Wilhelms and Van Gelder [102] extended Shirley and Tuchman's algorithm to generate polygons for a whole regular grid cell without the need to subdivide it into tetrahedra. This reduces the number of triangles that need to be rendered. Moreover, for parallel projections, they show that the set of polygons is constant for each cell. The polygons of a single cell are calculated once for each redraw. The color attributes of these polygons are updated for each cell and rendered into the frame buffer.

Stein, Becker and Max [87] describe a technique to incorporate the improved color model of Williams and Max [104] into a hardware based cell projection technique. The color at the thick vertex A and at the silhouette vertices are determined accurately, but linear interpolation of the hardware is used to calculate the interior points. Cell projection techniques work on many different mesh topologies. Sorting of the individual cells is required before the scan conversion and compositing. Max [67] describes sorting for several different mesh topologies.

3.4 Shear–Warp

The viewing transformation can be factored into translations of individual data slices, a projection to form a distorted intermediate image, and a 2D warp to produce the final image. This is attractive for volume rendering purposes since the data can be processed in a sequential fashion. Cameron and Undrill [11] develop a parallel projection version of this technique. The technique requires that the data be resampled twice: once in going from data space to the intermediate warped space and once in going from the projected warped space to the final image space. Since the volume data and scanlines in the intermediate warp space are aligned, implementations on SIMD machines have been developed to exploit this. Lacroute and Levoy [46] use run-length encoding of volume data and early ray termination of image space methods to generate near interactive volume rendering rates on a mid-range workstation.

3.5 Fourier Projection Slice Theorem

Frequency domain algorithms attempt to offer fast rendering by first transforming the data by a three-dimensional Fourier Transform. For a 1-D signal, the integral of that signal is equal to the value of its spectrum at the origin. The Fourier projection slice theorem extends this notion to higher dimensions. In particular, for 3D volumes the image obtained by integrating along the viewing rays is equivalent to the inverse transform of the 2D spectrum obtained by taking a slice through the Fourier transformed volume parallel to the viewing plane and passing through the origin. Rotating a three-dimensional volume requires a simple slice plane extraction of the Fourier volume and a 2D IFFT. Rendering times of two seconds are reported for 256^3 volumes [89]. The integration of the volume is limited to the simple model of glow only. An X-Ray type image can be generated by integrating only the optical density and then post-processing the image by exponentiation.

3.6 Splatting

Westover has proposed two methods of using splatting to produce volume rendering. In his first method [96] , the color and opacity filter kernels for each voxel are composited one by one in back to front order (in regard to their center points). In the second [95] , the colors and opacities for all the voxels in a layer are summed into an accumulation buffer, and then composited as a whole into the image. This second method prevents the opacity interactions of the voxels within one layer, and eliminates any possible small glitches from the change in sorting order within a layer during rotation. However, it may introduce larger glitches when the choice of the layer direction (most perpendicular to the viewing direction) changes. Laur and Hanrahan [47] implement Westover's first algorithm with an approximation to each splat by a collection of polygons. Each splat is typically created from fifteen to twenty-one triangles, or a triangle mesh. A gaussian function is used as the reconstruction kernel, but only at the vertices of the triangular mesh. The scan-conversion of the graphics hardware is used to interpolate between these points, yielding a crude approximation for the reconstruction kernel. They use a hierarchical octree representation for interactive viewing. This allowed for a quick coarse representation of the data that evolved into a more accurate representation adaptively. Splats for larger cells are corrected to compensate for their increased optical depth. Proper reconstruction between differing voxel sizes is not accounted for and in general, is a hard problem. With this approach Mach bands are visible at the polygon edges. Individual splats also are visible, because they do not overlap smoothly.

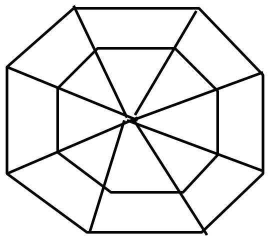

Figure 3-3. Regular polygon used as splat.

In reconstructing the 3D signal, a gaussian function is usually used. An unattractive property of this is its infinite extent. Every splat theoretically contributes to the entire volume density. Some finite extent is usually chosen and the splat is either abruptly cut-off or forced to zero. Max [60] uses an optimal quadratic-spline function with a limited extent for the reconstruction of 2D signals.

The work described in this thesis uses the splatting technique for volume rendering as a foundation. An ideal reconstruction function with a minimal extent is used for scalar reconstruction (see [60] or [21]). Techniques for generating patterns or textures useful for scientific visualization are developed.

Chapter 4

Vector Field Visualization

Many techniques for 3D vector field visualization have been developed over the past several years, with a recent flurry of activity. All of the techniques outlined below can be placed under one of three themes — particle advection, texture generation, or classification/representation techniques.

4.1 Particle Advection Techniques

Experimental fluid dynamicists use a variety of techniques to examine flow fields. Hydrogen bubbles or pH dyes can be injected into a fluid and photographed. Smoke can be introduced into an airflow and observed. The references in [108] contain several other techniques that have been developed over the years. Most of these rely on injecting a foreign substance into the flow and observing effects the flow field has on this substance. Advection is the local change of this foreign substance caused by the flow field, or the transport of this foreign substance due to the flow field. The techniques outlined in this section follow this theme of injecting a foreign substance into the flow field and visualizing it.

4.1.1 Individual Particles and Streamlines

By releasing and advecting abstract particles in a flow field, we can simulate a variety of effects. The particles can be represented as raster points in the simplest of schemes. As these particles are advected, their raster positions are updated and the motion of the particle can be studied. Just using raster points can lead to clutter and a meaningless image when many particles are displayed simultaneously. Reeves [77, 78] introduced structured particle systems for representing fuzzy objects. These were applied to image synthesis to produce fuzzy phenomena such as grass, trees, fire and waterfalls. Sims [83] developed a parallel system for handling particles with hidden surfaces and anti-aliasing and applied these to a two-dimensional vector field. van Wijk [99, 100] uses a particle system with shading and motion blurring of the particles to represent a flow field. Depth of field and hidden particle removal are added to aid in the comprehension of the flow. Max, Crawfis and Grant [64] render particles passing around or through a contour surface to show the relationship between the surface and the flow. Particles fade in as they approach the surface and fade out as they leave it. Particles are also given a life time, with new particles generated as time progresses. A small texture mapped square is stretched in the direction of the projected flow and composited into the image. A texture image of a smooth dot is used for both the intensity and the opacity, hence only motion blurred particles are rendered.

For steady-state flows or a single time step, we can connect a particle's current position to the position it would be advected to for a constant flow. Repeating for several time steps and accumulating the line segments yields a stream line. This is one of the most popular techniques, due to its simplicity and reliance on only a single time steps worth of data. The streamline has the property that it is everywhere tangential to the flow field. Applying the same technique to an unsteady flow, results in a streak line. More formally, a streak line is defined as the collection of particles passing through a common point. For steady state flows, stream lines and streak lines are equivalent. If we release a single particle into a flow and trace the path that it follows, we have a path line. A path line is also equivalent to a stream line for steady state flows. For unsteady flows, it represents a history of the path that the endpoint of the line traversed. Finally, time lines are a tool for observing the velocity magnitude and gradient of a flow. Rather than connecting particles originating from the same position, a line of particles is injected into the flow .

Hin and Post [37] use particles with random walks to simulate the dispersion within a field. Particles are released normally as above, but for the advection, a slight random perturbation or inaccuracy is introduced. Comparing the random walk streamlines to the traditional streamline represents the divergence from the streamline. Several such random walk streamlines need to be positioned around the starting location of the original streamline.

4.1.2 Ribbons and Tubes

If we have two stream lines starting from nearby points, we can draw a line segment between those points and their subsequent advected positions. This will yield a surface that can be rendered. Two starting points in a flow may diverge substantially from each other. Experiment techniques used to understand flows typically use a small ribbon that is assumed to be weightless in regards to the flow. Constructing a ribbon from two streamlines — the edges of the ribbon — will not work, since the streamlines may diverge from each other. A ribbon for scientific visualization purposes therefore needs to preserve its width. Several algorithms for constraining the divergence of the ribbons edges are possible. Three stream lines can be used such that a middle one controls the advected length and the direction that the ribbon twists about is calculated from the stream lines a delta on either side. New starting points for the outer stream lines are used at the edge of the ribbon for each sequential advection calculation. A ribbon illustrates not only the direction of the flow, but also its curl by the amount of twisting in the ribbon. Pagendarm and Walter [72] describe an algorithm to construct the stream ribbon using a single streamline and the curl calculated numerically from the flow field.

We can also calculate a circle a delta away from the stream line that is perpendicular to the flow. Advecting this circle will result in a stream line rendered as a cylindrical tube. The radius of the circle can be constrained in a similar manner as the stream ribbons, or it can be allowed to grow and become elliptical. Schroeder [80] uses a regular polygon rather than a circle to show the local strain or strain-rate that the stream line passes through. He also shows how several scalar variables could be mapped onto this construct which he calls a stream polygon. The color of each edge of the regular polygon can represent a different variable as can the overall radius of the stream polygon.

4.1.3 Stream Surfaces

A stream surface [39] extends the concept of a ribbon by allowing adjacent streamlines to diverge. A triangular mesh is created between the streamlines. As the streamlines diverge, more streamlines are added and a finer triangulation is generated. Streamlines are merged in areas of stagnation. The stream surface is thus everywhere tangent to the flow field (with fine enough accuracy in the subdivision). This is useful for showing the containment of a flow, that is the boundary that a flow will not cross. van Wijk [101] gives an algorithm for constructing stream surfaces implicitly by generating a volume density that can be isocontoured. The accuracy may not be as good as the explicit generation of the stream surface, but the algorithms are much simpler and existing code for isocontours can be used. Ma and Smith [58] study the mixing and dispersion of convection-diffusion problems, by taking a streamline and allowing it not only to propagate forward, but also to disperse by incremental standard deviations. Consecutive circular contours are then connected to form a surface for rendering. The resulting tubes can intersect each other illustrating areas of high mixing.

Flow Volumes and Stream Balls

Max, Becker and Crawfis [62, 65] extend the notion of stream surfaces to a flow volume. A seed polygon which acts as a smoke generator is placed into the flow field under user control. This is a natural technique that is closely related to experimental techniques that release smoke or dye into a flow field. As the vector field passes through the polygon, smoke is propagated forward, sweeping out a volume which is subdivided into tetrahedra. Compression and expansion of the volume due to the flow can be taken into account by adjusting the opacity based on the tetrahedron's volume. As the flow volume expands, they employ an adaptive mesh refinement technique to ensure the curvature of the resulting volume is accurate. The complex topology of the flow volume would require a general sorting method to yield a valid back-to-front sort in order to apply a volume rendering technique. However, for this application, they require that the smoke or dye be a constant color throughout the volume. They show that the resulting integration of the volume density is independent of the order the volume cells are processed for constant colored volume cells. Thus, no sorting is required. They are able to achieve real-time interaction by render the smoke, rather than an entire volume, using graphics hardware for the rendering. They also added additional features that allow the user to watch moving puffs of smoke, control the time propagation of the smoke, and combine opaque geometry with the smoke. A general polyhedra sort [87] can be added and the smoke can be colored by a separate scalar variable.

Meta-balls [71], blobbies [45], or soft-objects [69] are flexible algorithms for modeling fuzzy phenomena. A density volume is created by placing generating points in the volume's space. A basis function is centered at each of these points, contributing to the overall density function throughout the space. Gaussian functions are usually used, centered at each generating point. A continuous function is thus created with the value at any point in space as the sum of the point-centered guassians evaluated at that point. An isocontour surface of the resulting density function can be used to represent objects. Brill et. al. [8] introduced streamballs that use particles or streamlines as metaball location points. The resulting surfaces split gracefully in areas of high divergence and gracefully joined back together in areas of convergence. For large metaballs the results can be misleading since only the centers of the metaballs correspond to the flow path.

4.1.5 Advection Numerics

All of these techniques rely on a procedure to advect a massless particle from one position to another. If we are given the velocity $(\vec{v})$ at the particle's current position $(\vec{p})$, then for a differential time *dt*, the particles new position will be $\vec{p}\,' = \vec{p} + dt\ \vec{v}$. If we desire the particle's position at a time t_1, then an integration of this process is needed to go from the current time to a later time t_1. This is an initial-value ODE problem for which an extensive set of literature exists. The simplest solution is the forward or explicit Euler solution:

$$\vec{p}_{n+1} = \vec{p}_n + \Delta t \vec{v}_n + O(\Delta t^2).$$

More advanced techniques exist [38, 76, 88], of which Fourth Order Runga-Kutta is the preferred method.

These techniques require ascertaining the vector field at intermediate positions. If the calculation of the vector field at these intermediate points is costly, the overall cost of the advection can grow quickly. Since the vector field is defined on a mesh, calculating the vector field at a point interior to a zone requires an interpolation algorithm. Tri-linear interpolation is one of the simplest techniques for regular or rectilinear meshes, but may not be very accurate. Higher order techniques will require more data points. For curvilinear meshes, interpolation can be performed in computational space [26] with a mapping to and from physical space based on the Jacobian. For other mesh types, interpolation with higher order interpolation methods is quite complex with many end cases. Time dynamic fields offer the additional complexity of interpolation of a surface in four dimensions. Storage must also be allocated for each time step needed for the interpolation.

Since two near by particles in a flow can diverge substantially from each other, care must be taken to ensure accuracy. When many particles are being advected, this cost can be quite high. The storage of the particles plus the storage of the vector field at multiple time steps is also a disadvantage to advection techniques. The work in this thesis avoids the use of advection and the resulting storage costs and focuses on a more qualitative and global representation of the flow field.

4.2 Global Techniques via Texture Generation

The techniques outlined above all limit the representation of the vector field to the user selected region, the generation points (points for particles and streamlines; curves for stream surfaces; and polygons for flow volumes), and the volume advected from that region. They offer a more detailed examination of the history of a flow from a particular spot. However, a user trying to get an overall feel for the entire 3D domain, would have to move the generation points through the entire volume (or most of it) in order to ensure that all regions of the field were represented. The techniques outline in this section all try to represent an overview of the entire volume (or large portions of it) by creating a localized pattern of icons. They rely on localized coherence of individual icons to create a seamless texture and hence, usually do not require advection. Several experimental techniques follow this theme. Miniature tufts can be scattered throughout a wind tunnel or across the surface of an airfoil. A viscous oil can be spread across a surface, such that when the flow impinges on the surface it scatters the oil into visual patterns. The reference [108] contain several other examples.

The techniques for vector field representation presented in this thesis follow this theme, and attempt to generate textures or patterns that are meaningful to the human visual system. The small number of references in this section are all relatively recent and suggest that this work is a new thrust area for scientific visualization.

4.3.1 Hedgehogs, Tufts or Arrows

Many visualization systems allow you to place at positions within the flow arrows or vectors which are oriented in the direction of the flow. These work fine for slices of the data, but tend to get very cluttered and exhibit bad aliasing artifacts when examining the entire 3D vector field. These techniques do not really attempt to generate a texture or pattern for the flow, but rather present a quantitative icon at each regularly spaced data point. They are included here for completeness. Most of the work below and in this thesis attempts to show the broad overview that hedgehogs do for 2D slices, but in a clearer less cluttered framework. If the textures are smooth enough and avoid clutter, then they can be applied to 3D flows successfully.

4.3.2 Spot Noise

Van Wijk [98] created a 2D texture in the direction of the vector field that could be mapped to parametric surfaces. Small spots of varying amplitude and size are randomly deposited onto the 2D plane to produce a texture. Different shapes of the spots produce different patterns or textures. For vector field visualization, round spots are stretched in the direction of the 2D vector field before being deposited on the plane. He called this technique "spot noise". It is capable of showing fine detail in texture, while allowing another scalar variable to be represented with changing color. Unfortunately, it is only applied to 2D fields, and extending them to three-dimensional fields would be difficult.

4.3.3 Volume Rendering

Crawfis and Max [20] developed a technique to integrate volume rendering of a scalar field with a line segment representing a flow field. This was accomplished by sampling the data set in back-to-front order on an image space voxel grid. At each point a scalar splat and vector line were drawn. The integration of the optical properties of the density cloud and the anti-aliased line segment was calculated for each pixel. The solution was entirely software based, but the visualizations of time vary winds fields produced successful patterns for understanding these fields. The success of these patterns without the need for advection prompted this research.

4.3.4 Line Integral Convolution

Cabral and Leeds [10] developed an algorithm that they called the Line Integral Convolution or LIC operator. It takes as input an n-dimensional vector field and an n-dimensional image. By using an image consisting of white noise (or band-limited noise), the algorithm will correlate the image in the direction of the vector field. It does this by taking the integral of the image along a local stream line subtended from each point in the output image. The resolution of the input image, vector image and output image were all required to be the same in their implementation. The calculation of the stream lines requires the costly advection calculations described above. Since this is performed at every data point, the technique was quite slow. For 3D data sets, the resulting image still needed to be rendered. The authors chose to render using ray-tracing which also added to the total cost of the algorithm. Forssell [26] extended the LIC operator to work on parametric slices (2D) of a curvilinear grid. The resulting 2D images were used as a texture map for the parametric slice. Stalling and Hege [85] improve the LIC algorithm as presented by Cabral and Leedom to be resolution independent and offer performance enhancements to speed up the running time.

4.4 Classification Techniques

Classification or feature extraction techniques are also relatively new to scientific visualization. Hellman and Hesselink [36] have developed a technique to build stream surfaces to represent the topology of the flow. Globus, et al. [30] developed techniques to identify the critical points of a flow field. Iconic representations can be placed at these points with appropriate streamlines connecting them.

Banks and Singer [5], and Ma and Zheng [59] present techniques that attempt to identify and represent vortex tubes resulting from turbulent flows.

Hanson and Ma [33] developed a technique based on the Frenet frame of a 3D space curve that when mapped to the surface of a sphere can show interesting correlations. The technique is more like a histogram, illustrating the distribution of directions, without illustrating where in the field individual directions occur.

Chapter 5

Line Bundles

Most workstations offer hardware support for drawing line segments, while high-end workstations are capable of drawing several million lines per second. Line Bundles capitalize on this to generate very fast representations of a flow field as an anisotropic volume density cloud, or contour surface with anisotropic reflection. This is a new technique for vector field visualization.

A line bundle is defined to be a collection of line segments all oriented in the same direction and relatively close together. The line segments are placed randomly within a unit cube. A line bundle is then rendered at each (or selected) data point, following the volume rendering technique of splatting, to produce a three-dimensional texture. The term line bundles used here, has no relationship to the term line bundles used abstract mathematics. Line bundles here refer to exactly that, a tight bundle of lines.

5.1 Stochastic Textures

The goal of this technique is the generation of a seamless texture throughout space that can represent the flow field. Stochastic texture generation has been studied by several authors for 2D images [17, 35, 56, 73, 98]. Three-dimensional texture generation has been explored by only a few authors [10, 20, 41, 57].

This technique generates textures for flow visualization by drawing many small line segments in a back-to-front order. Drawing these line segments very closely together will fill in the projected image space with a constant color. By using anti-aliased lines and alpha blending, this fill-in will be reduced, but with enough line segments the colors will saturate the device and still fill in the image. Jittering the hue or saturation of the line segments and blending them together allows for a smoother texture. Jittering of the line segments' color is allowed in all three channels of hue, saturation and value space. Figures 5-1 through 5-4 illustrate the effects of jittering the colors of the line segments to produce anisotropic textures. The amount of jittering is increased for a tornado test data set from Figure 5-1, which has none, to Figure 5-4, which is jittered in the hue to produce random hues. Figure 5-4 also shows a close-up section of the resulting texture.

In addition to the overall color of the line segments, the color and opacity of the head and tail of the segments are modified as follows. The head of the line segment (the end that will point in the vector field direction), is optionally desaturated to give an indicator of the positive direction of the vector field along the line segments (see Figure 5-5). Figure 5-5a also shows the smooth blending of the line segment's end points, and give the clear impression of direction to the right and slightly down. Both the head and the tail of the line segments are made more transparent than the base to allow the line segment to gradually fade into the background. The tail is set to completely fade into the background, while the head is only slightly more transparent than the mid-point. This is similar to a painter gradually lifting his or her brush at the end of a long stroke. Without this blending, harsh edges would be visible at the endpoints and unwanted patterns will appear in the image. The line segments used for the rendering in Figure 5-6 incorporated this smooth blending, while those used for Figure 5-7 did not. These changes to the head and tail of a line segment require at least three points to define them, and hence affect the rendering speed of the overall image. The hardware interpolates the colors along the line segments to produce a gradual change.

The number of lines used is a critical factor in generating a smooth texture. Too few lines do not yield a texture, while too many affect the performance of the rendering and increases the opacity, hiding information behind them. Figure 5-5 shows a single line bundle with varying numbers of line segments. As more lines are added, more of the screen real estate is filled in and an overall texture is generated. A continuous texture will lead to a less distracting view of the vector field. Figures 5-8 through 5-10 show the test tornado data set with varying numbers of lines per line bundle. With careful studying the images produce the same information about the vector field, however, Figures 5-9 and 5-10 gives a quicker, more qualitative view. For time varying data sets, where the individual images may pass by rather rapidly, this is beneficial. Figure 5-11 uses the OpenGL anti-aliased line segments to produce a slightly smoother version of Figure 5-10. Since the line segments in Figure 5-10 are already somewhat anti-aliased through the alpha blending, the difference is not as noticeable.

The overlap of the lines is also critical to achieving a smooth texture. Two user controls are offered to control the overlap — the overall splat size and a vector scale factor. The splat size needs to be large enough such that an overlap of the lines in all directions from the splat data point fills enough of the image space to create a texture. The vector scale can be used to highlight vectors with greater magnitude or to create a more wispy appearance to the image. It is also required for vector fields that have not been normalized for world coordinates. That is, it is needed to scale a vector field from its physical space or units to world coordinates. Figures 5-12 through 5-27 illustrates the effects of varying the splat size and vector scale on a sample tornado data set.

5.2 Back-to-Front Compositing

To create an unambiguous texture in three-dimensions, it is critical that the line bundles are drawn in back to front order. The current implementation uses an octree to store the data values of interest. A back-to-front sorting of this octree is then determined on every redraw. The line bundles are drawn in back-to-front order with a user specified opacity and then blended into the image. Individual lines are not sorted within a bundle. This sorting and back-to-front compositing closely follows the splatting technique for volume rendering of scalar fields. In fact, the C++ Open Inventor class developed for the techniques described in this chapter (as well as the remaining chapters) are derived directly from a set of classes developed for volume rendering of scalar fields using the splatting technique.

Disabling the z-buffering within a bundle allows the line segments to be drawn as if on top of each other to produce the desired texture. This is accomplished by setting the OpenGL z-buffing function to "test but do not set", such that line bundles will not modify the z-buffer, but will examine it and only render to pixels for which the line bundles lie in front. By drawing the line bundles last when other opaque geometry is to be rendered, they are blended into the image appropriately. This is all implemented in hardware on Silicon Graphics workstations equipped with z-buffering hardware.

Varying the opacity allows softer or harder textures. Figures 5-28 through 5-30 show the effects of varying the transparency of the lines from a value of 90% transparent in Figure 5-28 to completely opaque in Figure 5-30. It can be seen that the semi-transparent nature of the line bundles is needed for smooth texture generation. Figure 5-30 has noticeable artifacts that may be disruptive to the visualization process. Figure 5-31 also uses a completely opaque setting for the line segments, but uses anti-aliased lines to lessen the harsh edge effects present in Figure 5-30.

An octree can easily be constructed from a regular grid, so this technique works well for either hierarchical meshes already represented as octrees or for regular grids. Sorting techniques for other mesh types have been developed. Max [67] outlines several algorithms for sorting of volume elements. In particular, special sorting for climate data with a curvilinear height field is developed, as is a sort for wrapping the climate data around the globe. Max also describes algorithms for sorting hierarchical meshes. Williams [103] developed an algorithm to sort unstructured data sets that contain no holes or sliding interfaces. Stein [87] developed an algorithm for handling arbitrary unstructured mesh topologies. Extending line bundles to these topologies would require both changing the number of lines in a line bundle for each data point and handling distribution problems for zones with non-uniform aspect ratios. The number of lines per line bundle needs to change in order to keep the density uniform in areas of smaller or large zones.

5.3 Orientation

The line bundle is centered at the splat data point and oriented in the direction of the vector field at that data point. For efficiency, the line bundles are precomputed by calculating the color jittering and the relative placement of a set of lines and drawing them into an OpenGL display list. All of the lines for this display list are oriented in the positive z-direction. For each redraw, a translation to the splat center and an orientation to the vector field direction need to be calculated before posting the display list. A rotation matrix is calculated to rotate the z-axis to the vector field direction. The lines are stretched in the z-axis by the vector field magnitude and a user specified scaling factor before this rotation. The properly oriented line bundle is then translated into the proper position and composited into the image. This translation is slightly jittered to break-up possible regular patterns, producing a smoother texture.

5.4 Color-Coding

The line bundles discussed so far produce a rather homogenous texture. The colors vary locally, but globally the variation is constant. It is useful to represent either the velocity magnitude or another variable using color encoding. Jittering about a different hue for line bundles representing different scalar values, allows an anisotropic texture that can be color coded. Each splat has an associated color assigned to it by the user. When rendering, a new line bundle is created with the line segments' colors jittered using the splat's color as a base color. This differs from the homogenous line bundles in that there, all of the splats had the same base color. Figures 5-12 through 5-27, Figure 5-32 and Figure 5-33 were generated using the color-coded line bundles, as were several of the application images.

Since each splat can have a different base color, the line bundles can not be precomputed. The line segment positions can be precomputed, but the colors can not. This also implies that the line segments can not be drawn to an optimized OpenGL display list a priori. The homogenous line bundles are therefore faster by as much as a factor of two. The actual improvement in speed with the homogenous line bundles varies depending on the number of lines within a bundle. The more the lines the better the relative performance of the homogenous line bundles will be when compared to the color coded line bundles. Most of this efficiency can be recaptured if we restrict the base colors of the splats to a fixed set. A table of line bundles can be generated with each entry corresponding to a different base color and hence a different scalar variable mapping. With a large enough table, there will be little perceived difference in the images generated from this approach and the less efficient approach of generating each line bundle during rendering. For small tables, sudden changes in color may cause a popping in the image as it is rotated and the sorting order changes. This is due to the relatively large overlap of the line bundles. Of course, the more lines within the bundles the more actual overlap and the more noticeable the popping. The table will also require a fair amount of memory, depending on the number of lines within a bundle. Since each entry will actually be an OpenGL display list, care also has to be taken to avoid allocating all of these limited resources.

5.5 Lighting

Lighting can add an additional depth cue to a visualization. Kajiya [41], Hanson [32], and Banks [4] have developed algorithms for lighting curves in three-space. These techniques work especially well for opaque curves. For many small and fairly transparent line segments the benefit of lighting is not as great. Furthermore, it may introduce distracting artifacts into the desired texture. Lighting was explored with the SoSplatLines class. Figure 5-32 and 5-33 illustrate the test tornado with and without the lighting, respectively. Lighting of curves is not supported in hardware, so the individual colors of each line segment need to be modified. Since different orientations of the line bundles are lighted differently, this prohibits the use of precomputed display lists. The lighting can be calculated as a scale factor for all of the line segment's colors within a single oriented line bundle. For this test, only the ambient and diffuse lighting were applied, with an ambient coefficient of 0.4 and a diffuse coefficient of 0.9.

5.6 Applications

Line bundles have been applied to several data sets and used in a few videos to illustrate flow fields. Figures 5-34 through 5-40 illustrate the application of line bundles to real simulations. In figures 5-34 through 5-36, the line bundles are applied to the flow through an aerogel substrate. Aerogel is a new very lightweight material being used for insulation, packaging, filters, cosmetics and many other applications where materials with every light weight and high surface area are needed. Small spheres are used to represent areas occupied by aerogel in the simulation. In Figure 5-34, velocities near the aerogel particles have been highlighted. Very little color jittering has been applied to produce a moss-like appearance around the aerogel. Figures 5-35 and 5-36 depict areas of high velocity magnitude within the aerogel. Figure 5-36 uses color coded line bundle, where the color represents the velocity magnitude. High velocity magnitudes are represented by magenta, and low velocity magnitudes are represented by yellow.

HEPA (High-Energy Particle Absorption) filters are used in many industrial application to clean or scrub contaminants for the air. Figure 5-37 shows the areas of high velocity magnitude of air flowing through a small piece of a HEPA filter under design. The areas of high-velocity magnitude are represented by reddish line bundles, while areas of low velocity magnitude are represented by blue colored line bundles.

Interstellar collision of shock waves from supernovas are thought to create new stars. Figures 5-38 and 5-39 represent the data from a time dynamic simulation of interstellar cloud collision. Figure 5-38 uses the line bundles to select a specific contour of velocity magnitude. Figure 5-39 depicts a wider range of velocity magnitudes.

Figure 5-40 represents a contour surface of wind velocity magnitude and the directionality of the winds over North America. The line bundles are selected at or near the contour surface to help aid in determining the wind properties.

5.7 Color Images *(Original images are in color)*

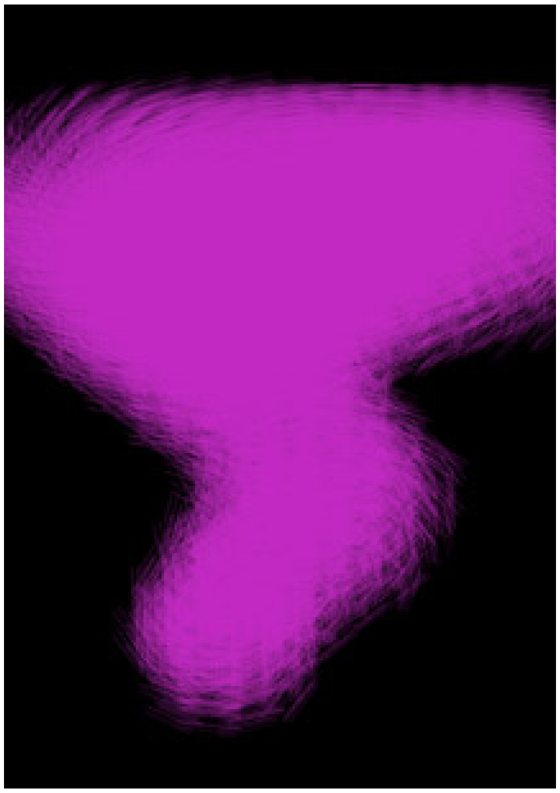
Figure 5-1 No color jittering

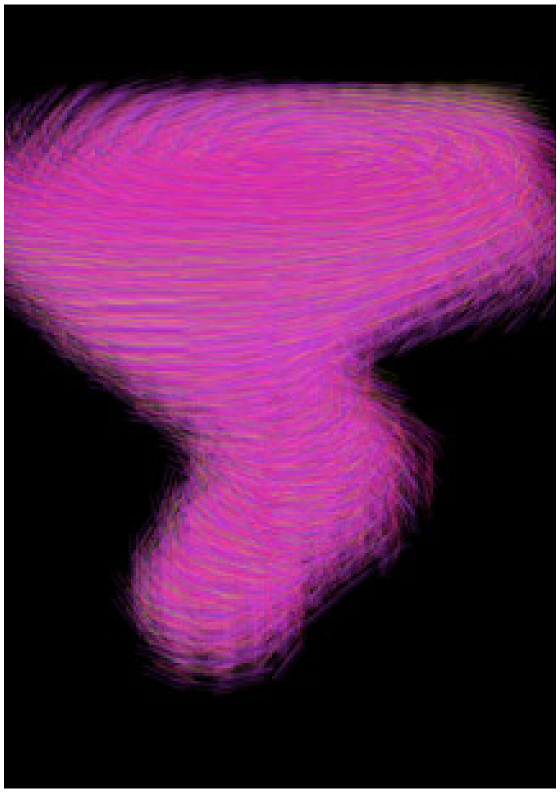
Figure 5-3 HSV Jittering = (0.3, 0.3, 0.3)

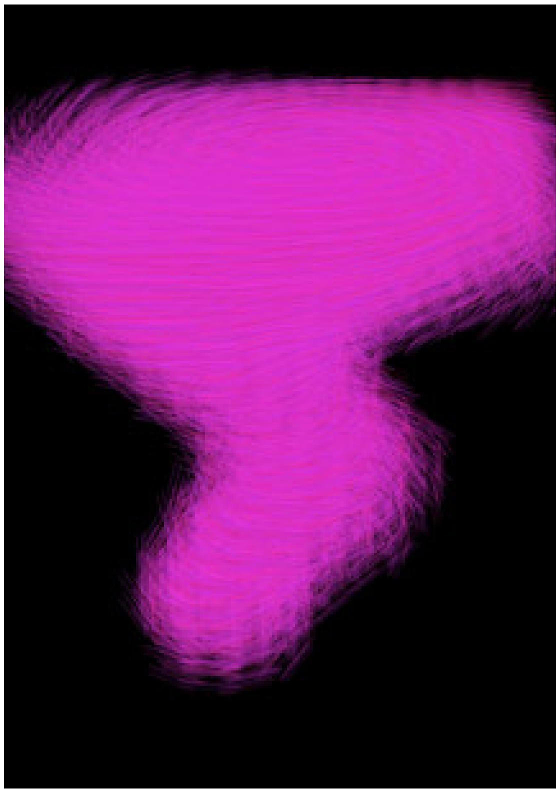
Figure 5-2 HSV Jittering = (0.1, 0.1,0.1)

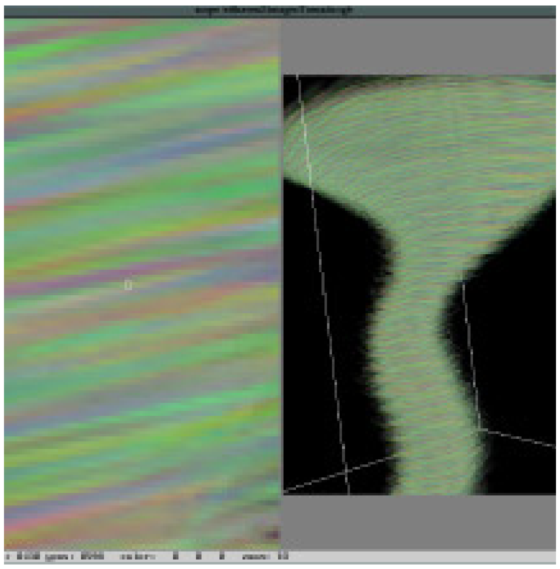
Figure 5-4 Complete jittering of the hue, with a magnification of the resulting texture.

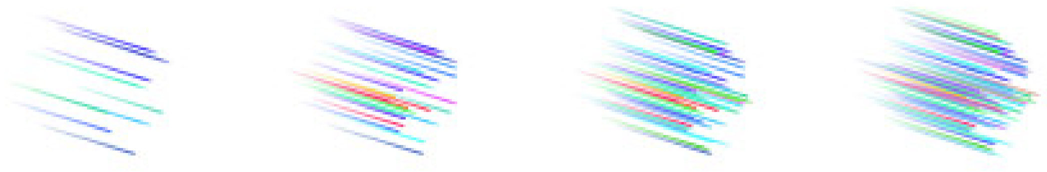

Figure 5-5 a) 10 lines b) 25 lines c) 50 lines d) 75 lines

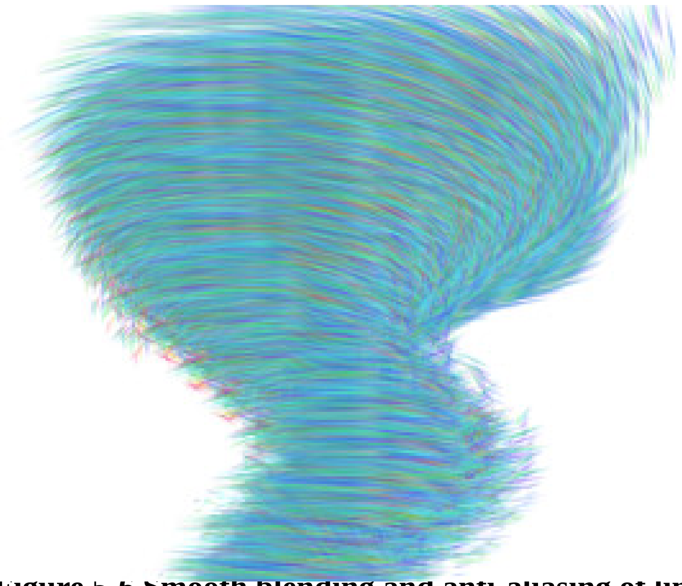

Figure 5-6 Smooth blending and anti-aliasing of lines

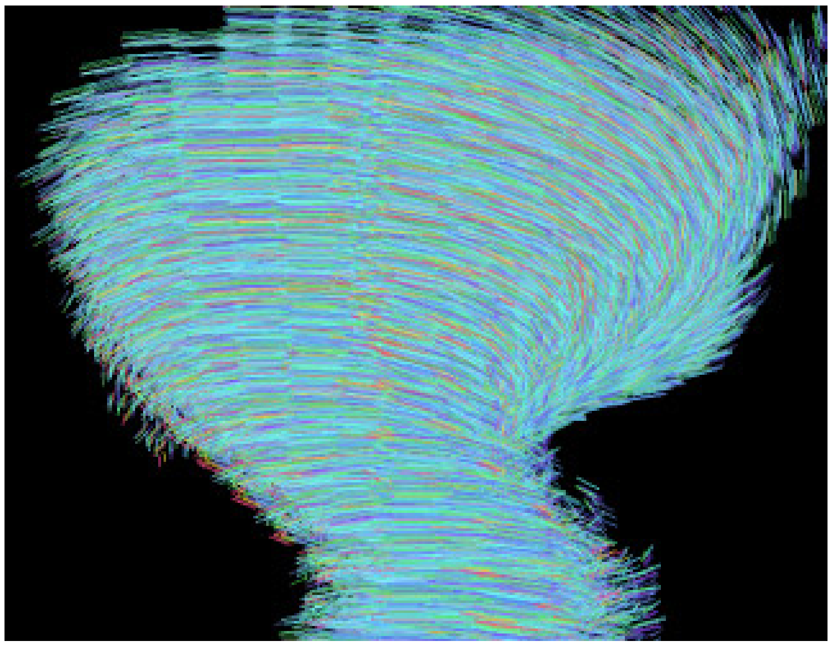

Figure 5-7 Solid opaque lines without anti-aliasing

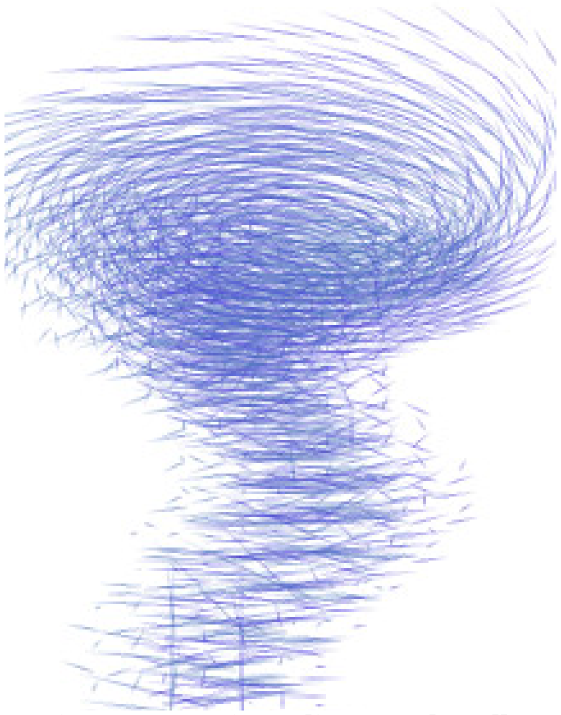
Figure 5-8 Using 2 lines per bundle

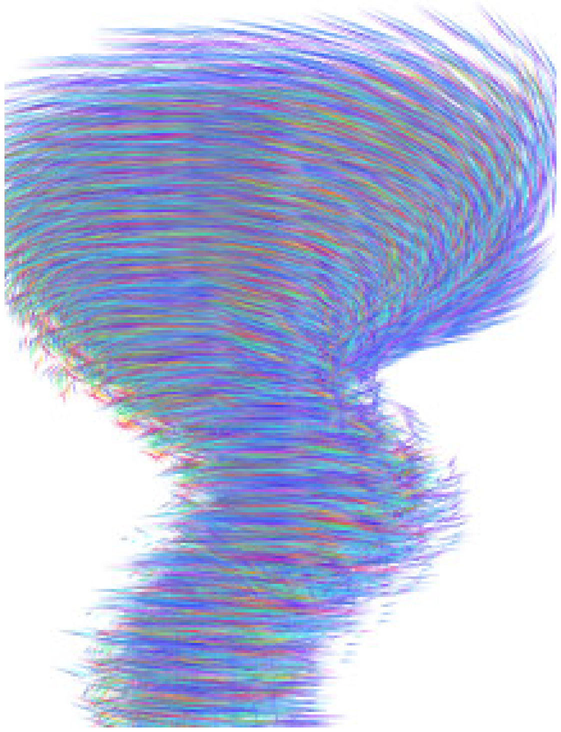
Figure 5-9 Using 25 lines per bundle

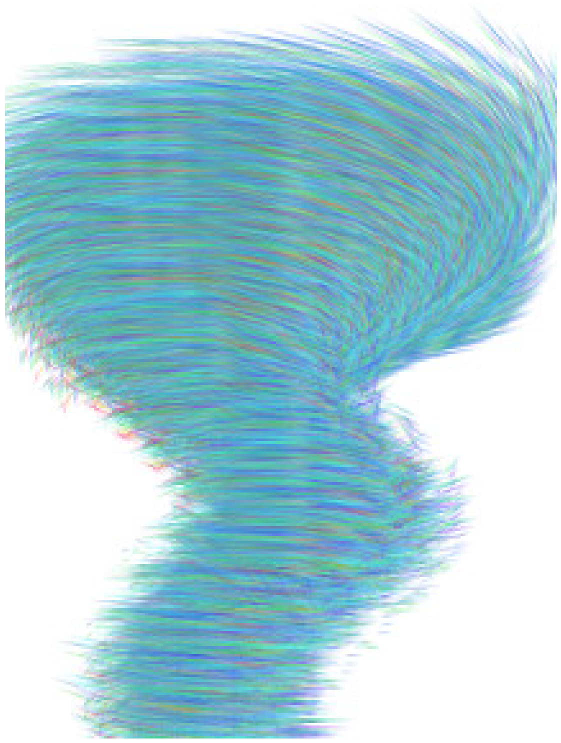
Figure 5-10 Using 50 lines per bundle

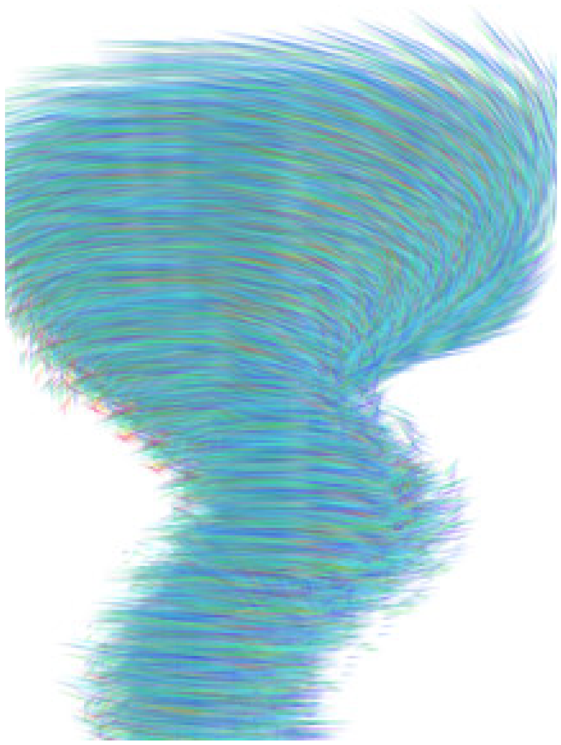
Figure 5-11 50 anti-aliased lines

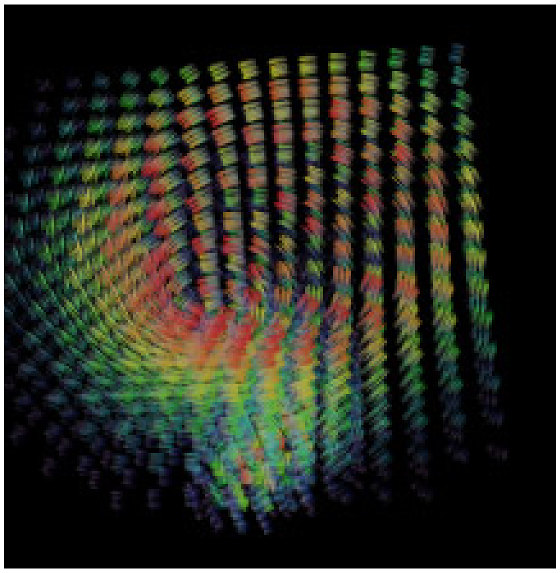
Figure 5-12 Splat size 0.5; VecScale 0.5

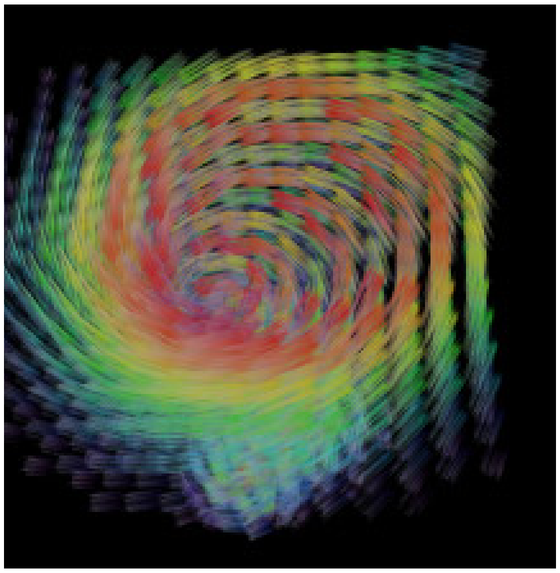
Figure 5-14 Splat size 0.5; VecScale 1.5

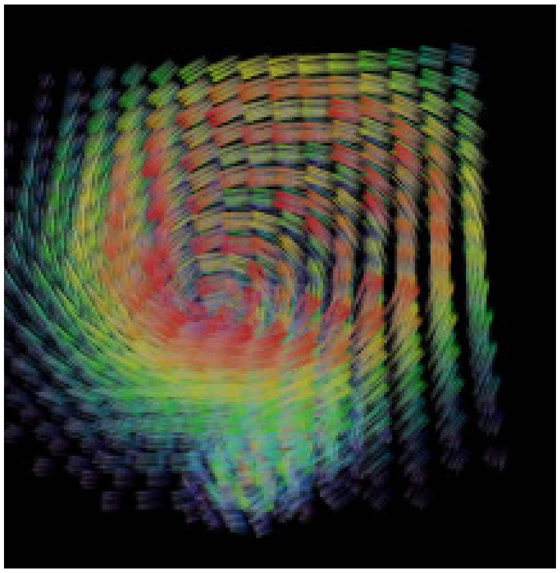
Figure 5-13 Splat Size 0.5; VecScale 1.0

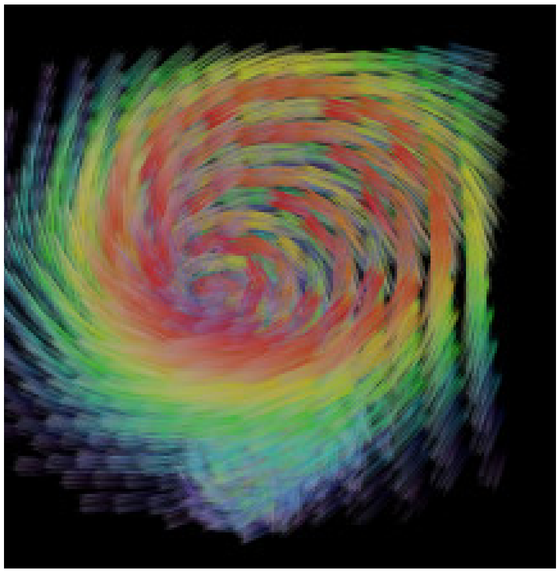
Figure 5-15 Splat size 0.5; VecScale 2.0

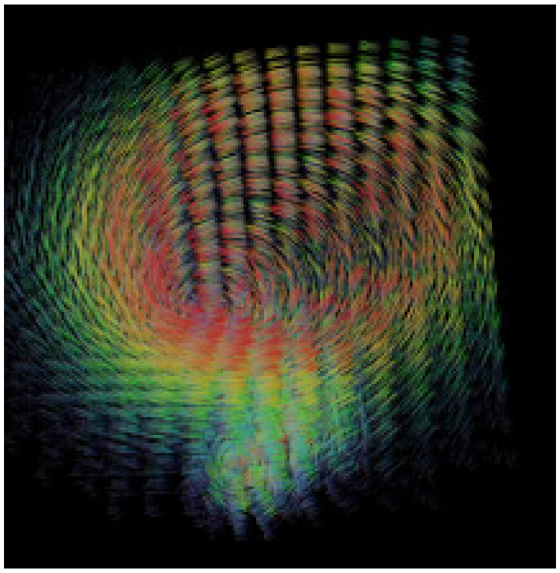
Figure 5-16 Splat size 1.0; VecScale 0.5

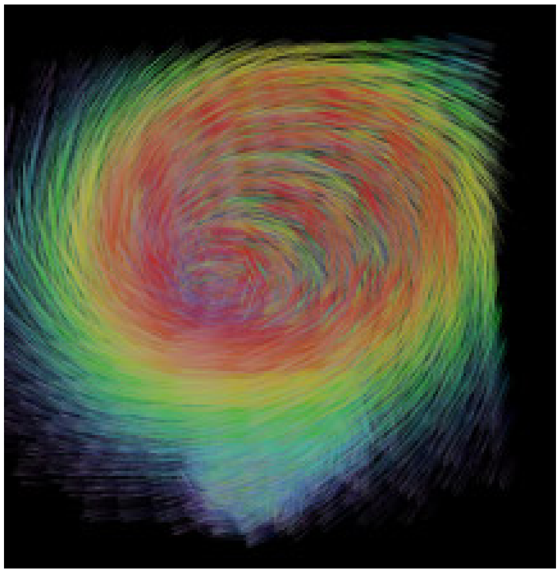
Figure 5-18 Splat size 1.0; VecScale 1.5

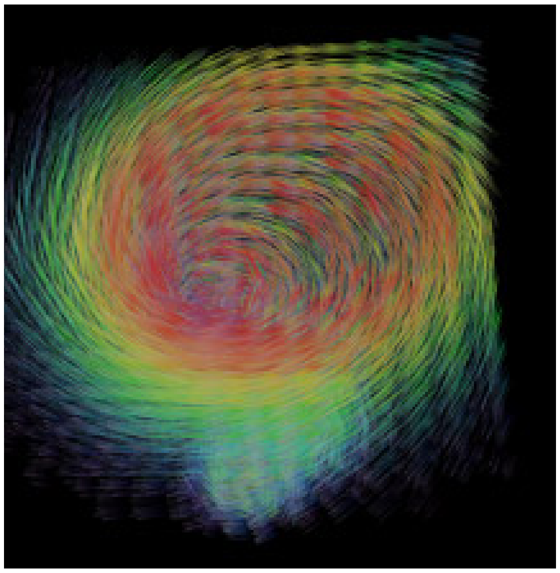
Figure 5-17 Splat size 1.0; VecScale 1.0

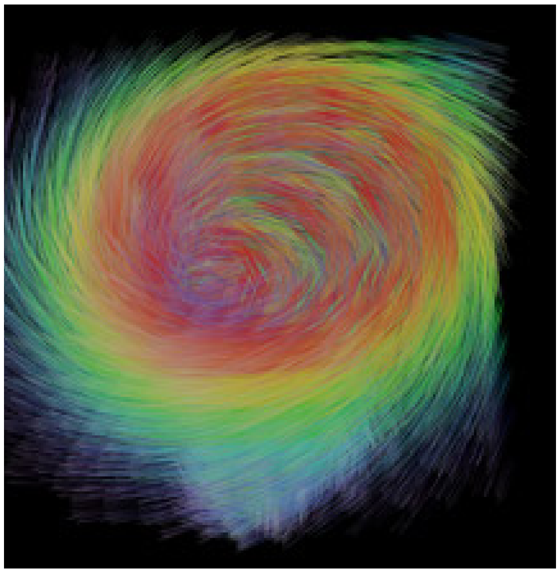
Figure 5-19 Splat size 1.0; VecScale 2.0

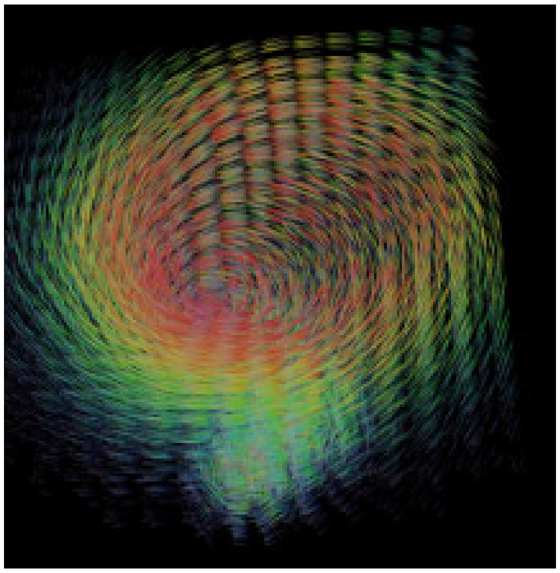
Figure 5-20 Splat size 1.5; VecScale 0.5

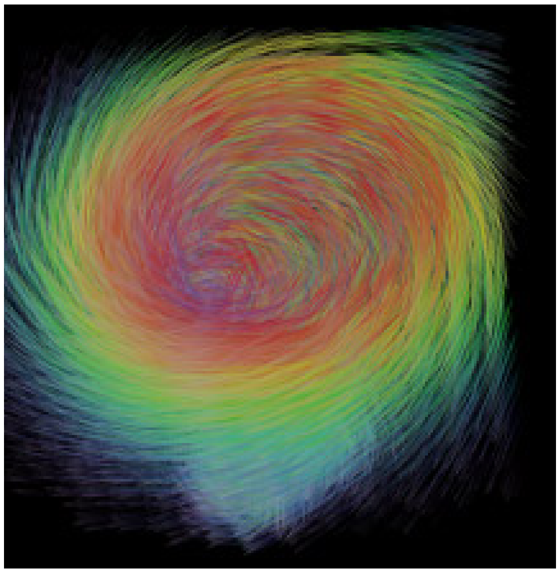
Figure 5-22 Splat size 1.5; VecScale 1.5

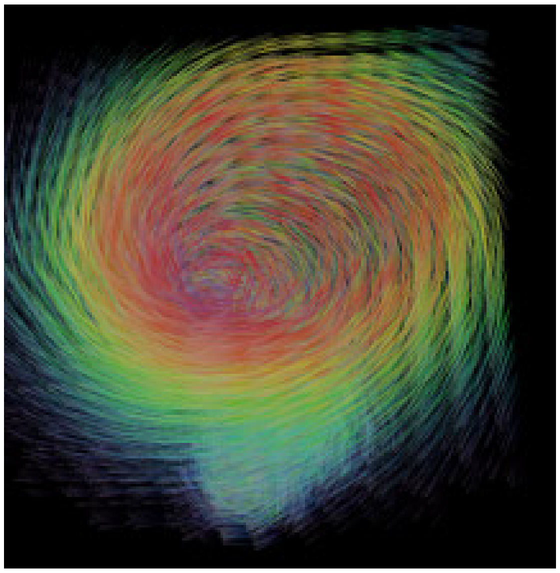
Figure 5-21 Splat size 1.5; VecScale 1.0

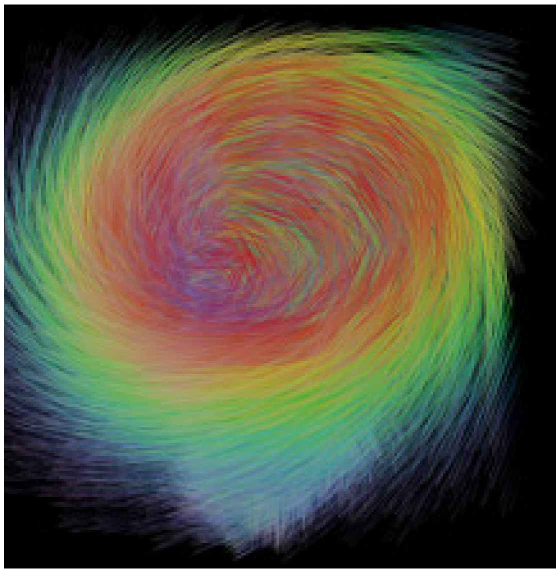
Figure 5-23 Splat size 1.5; VecScale 2.0

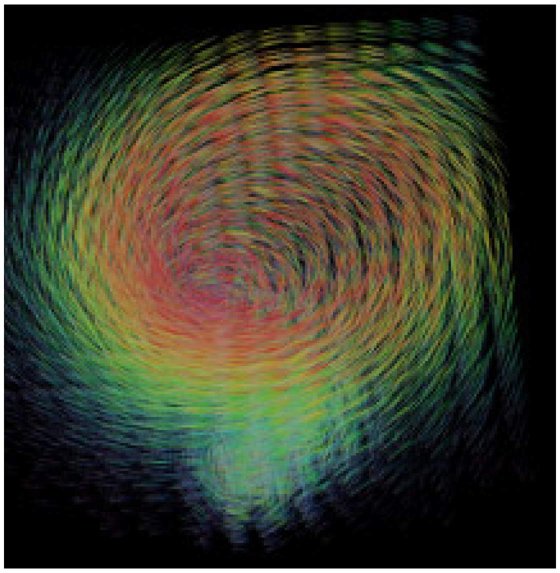
Figure 5-24 Splat size 2.0; VecScale 0.5

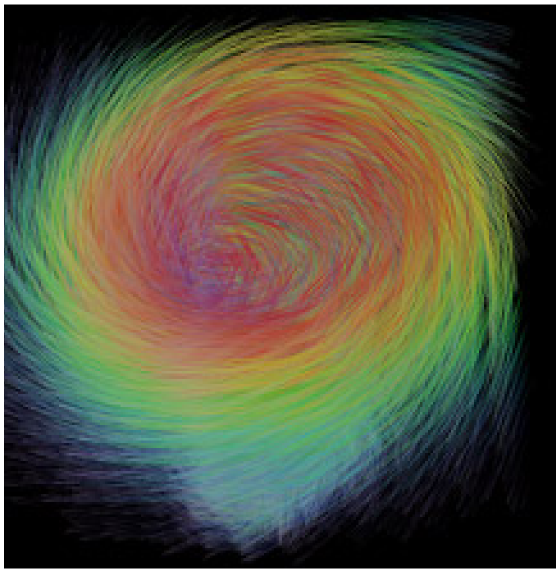
Figure 5-26 Splat size 2.0; VecScale 1.5

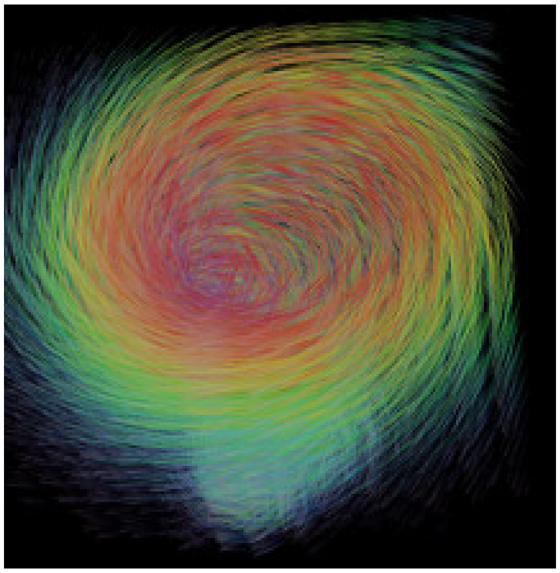
Figure 5-25 Splat size 2.0; VecScale 1.0

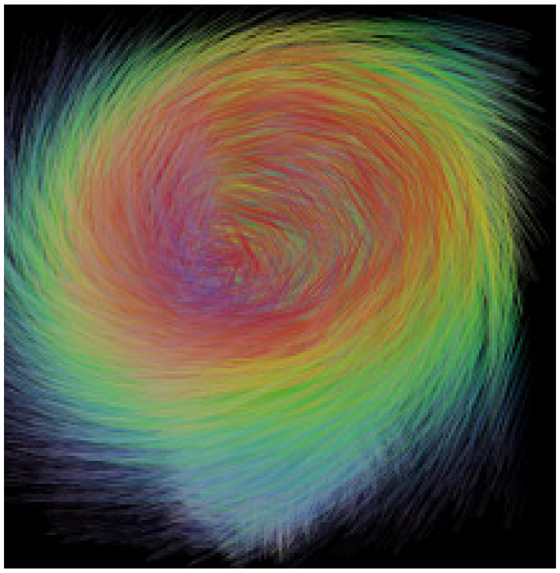
Figure 5-27 Splat size 2.0; VecScale 2.0

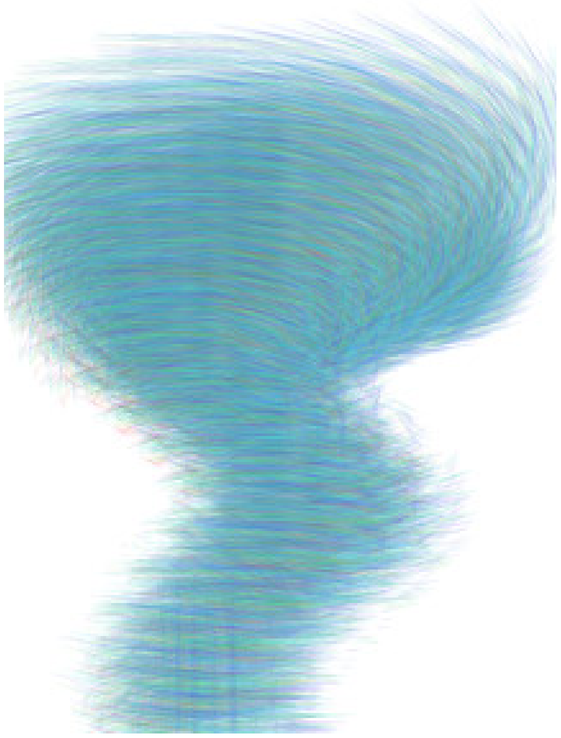
Figure 5-28 Opacity = 0.1

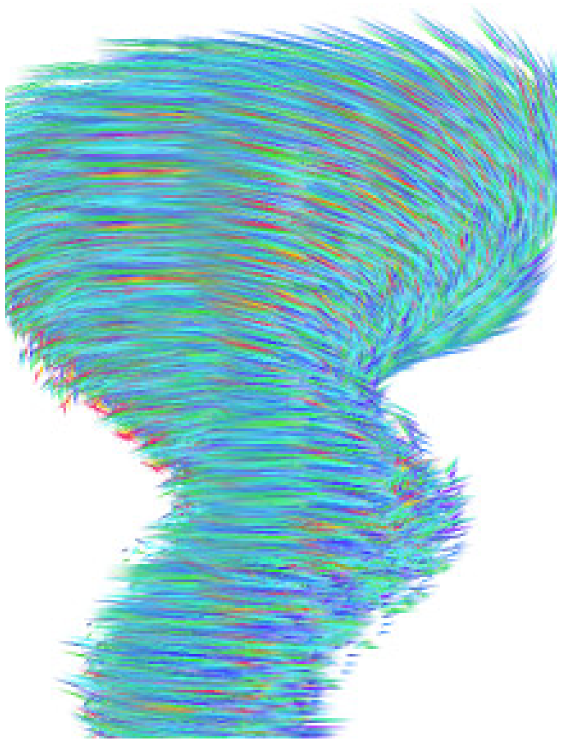
Figure 5-30 Opacity = 1.0

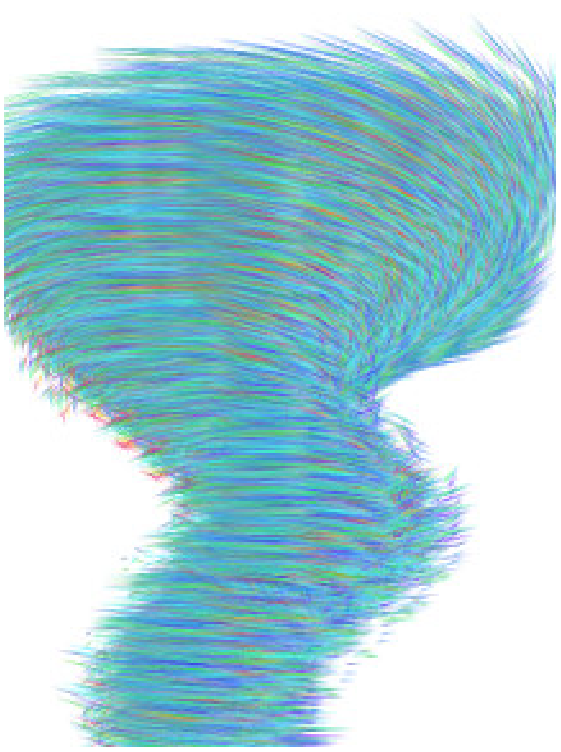
Figure 5-29 Opacity = 0.5

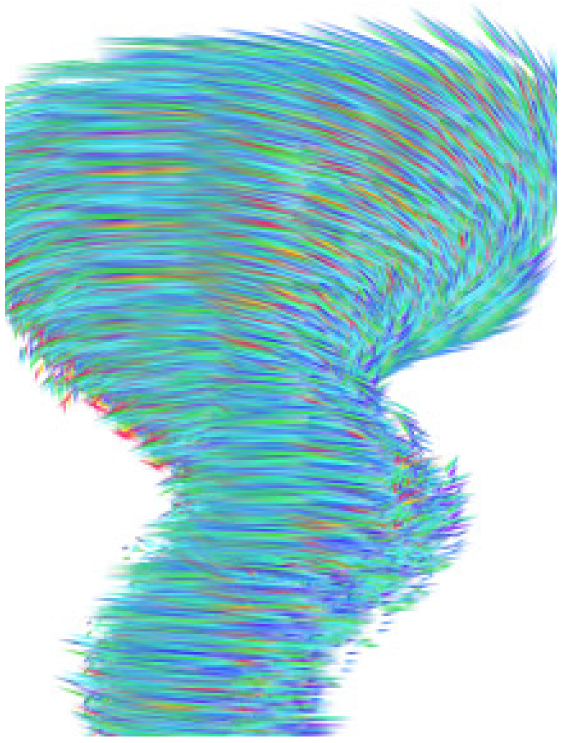
Figure 5-31 Opacity = 1.0 with anti-aliased lines

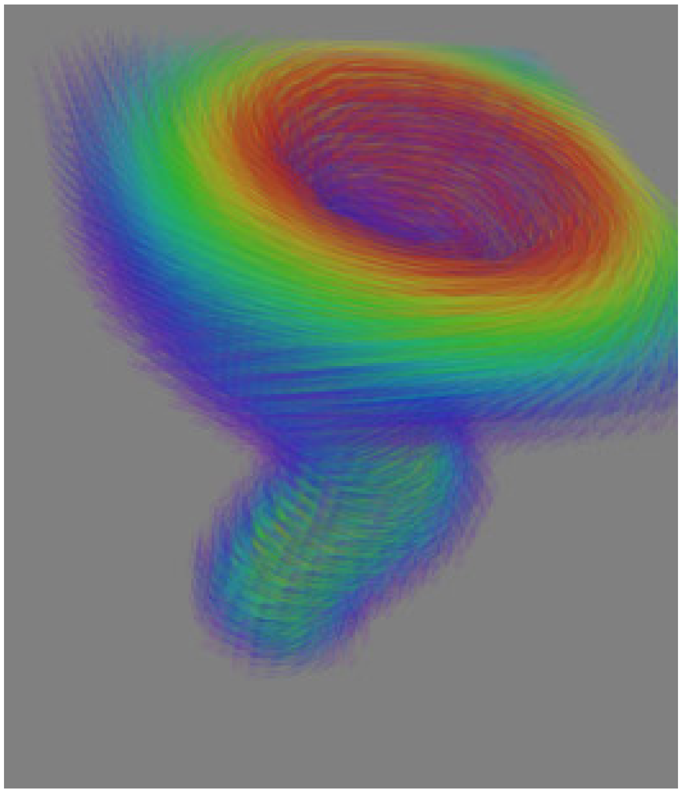
Figure 5-32 Colored Line Bundles representing velocity magnitude.

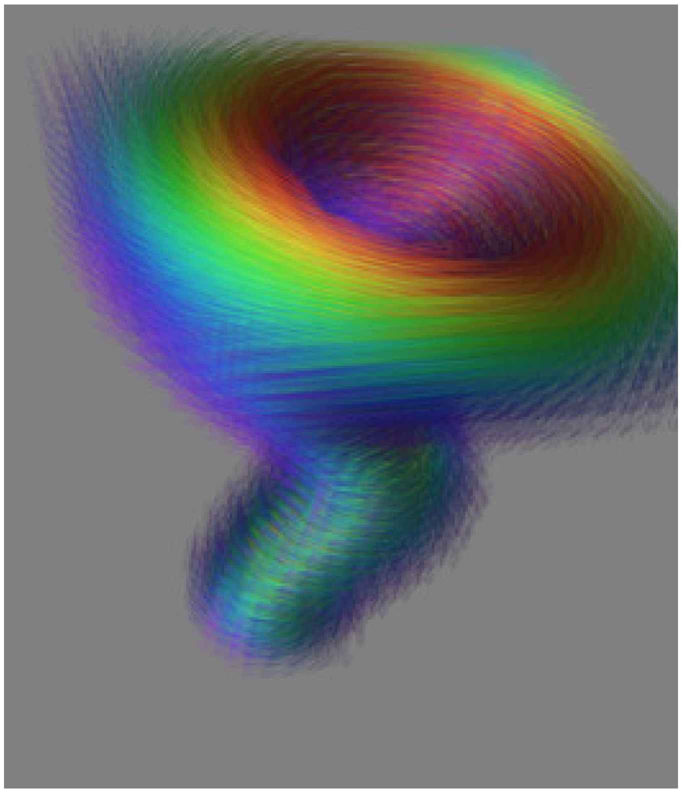
Figure 5-33 Same as Figure 5-32, but with 1D lighting added.

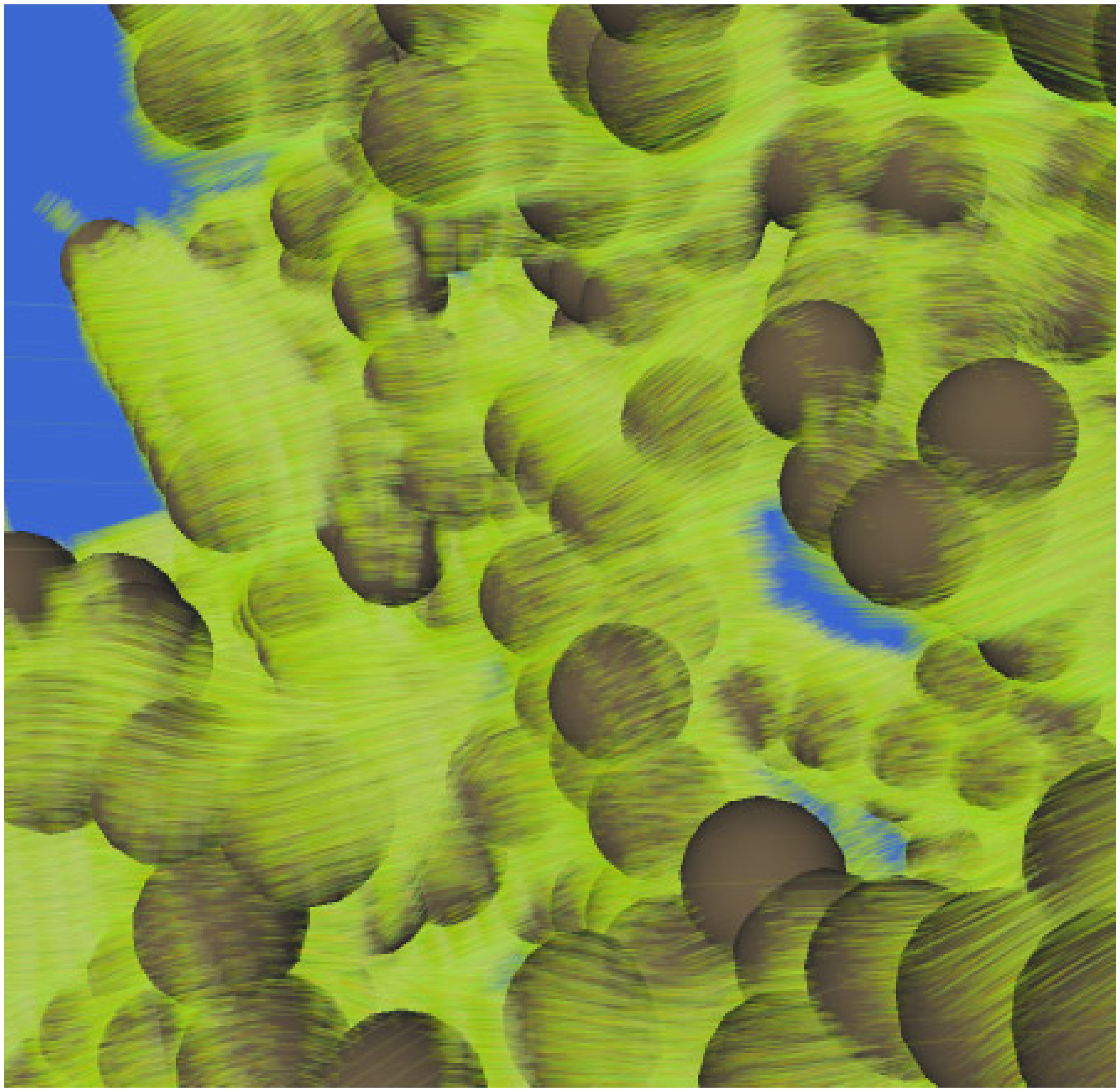
Figure 5-34 Line Bundles near the low velocity areas of Aerogel particles.

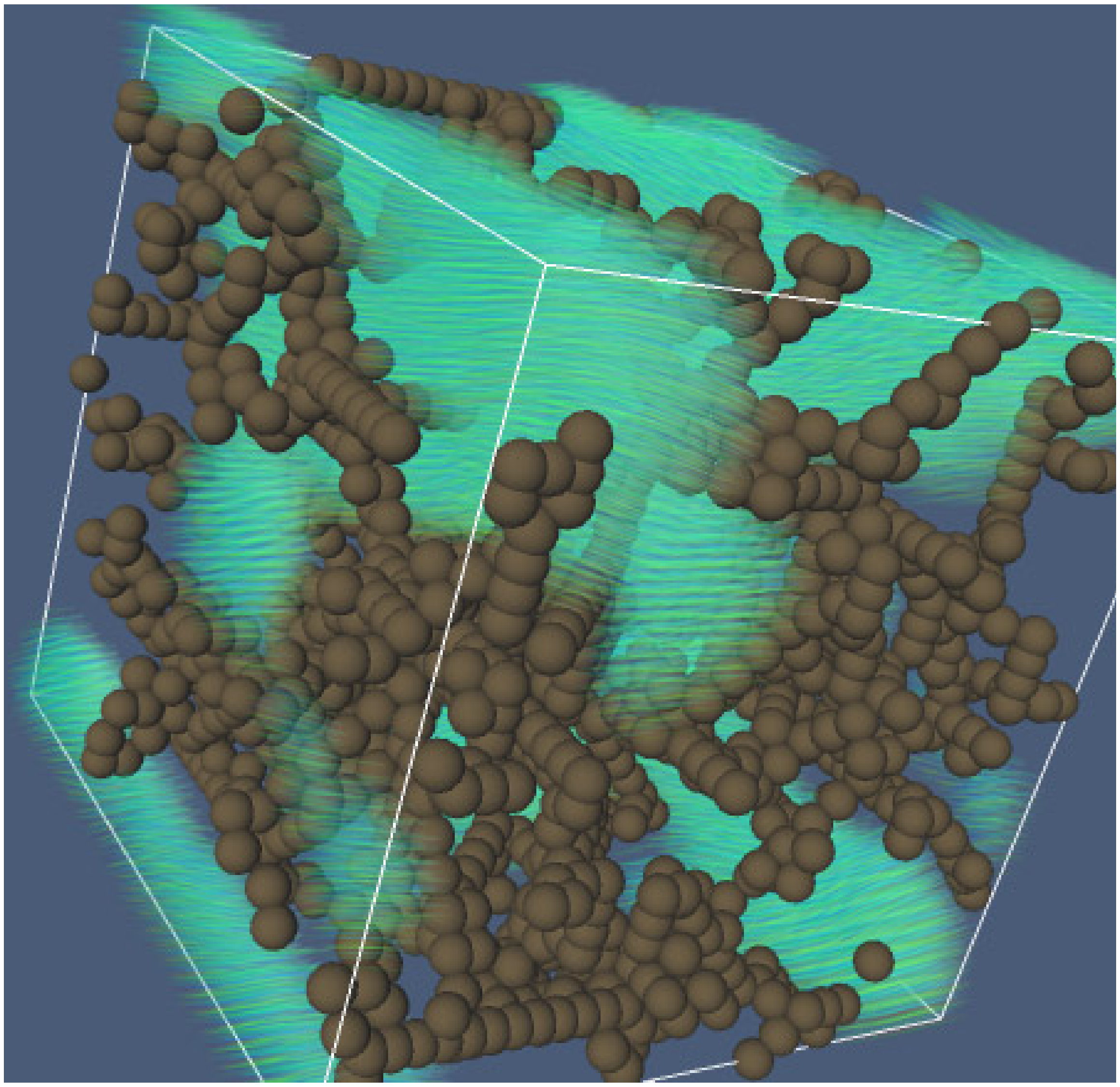

Figure 5-35 Line Bundles through areas of high velocity in aerogel simulation.

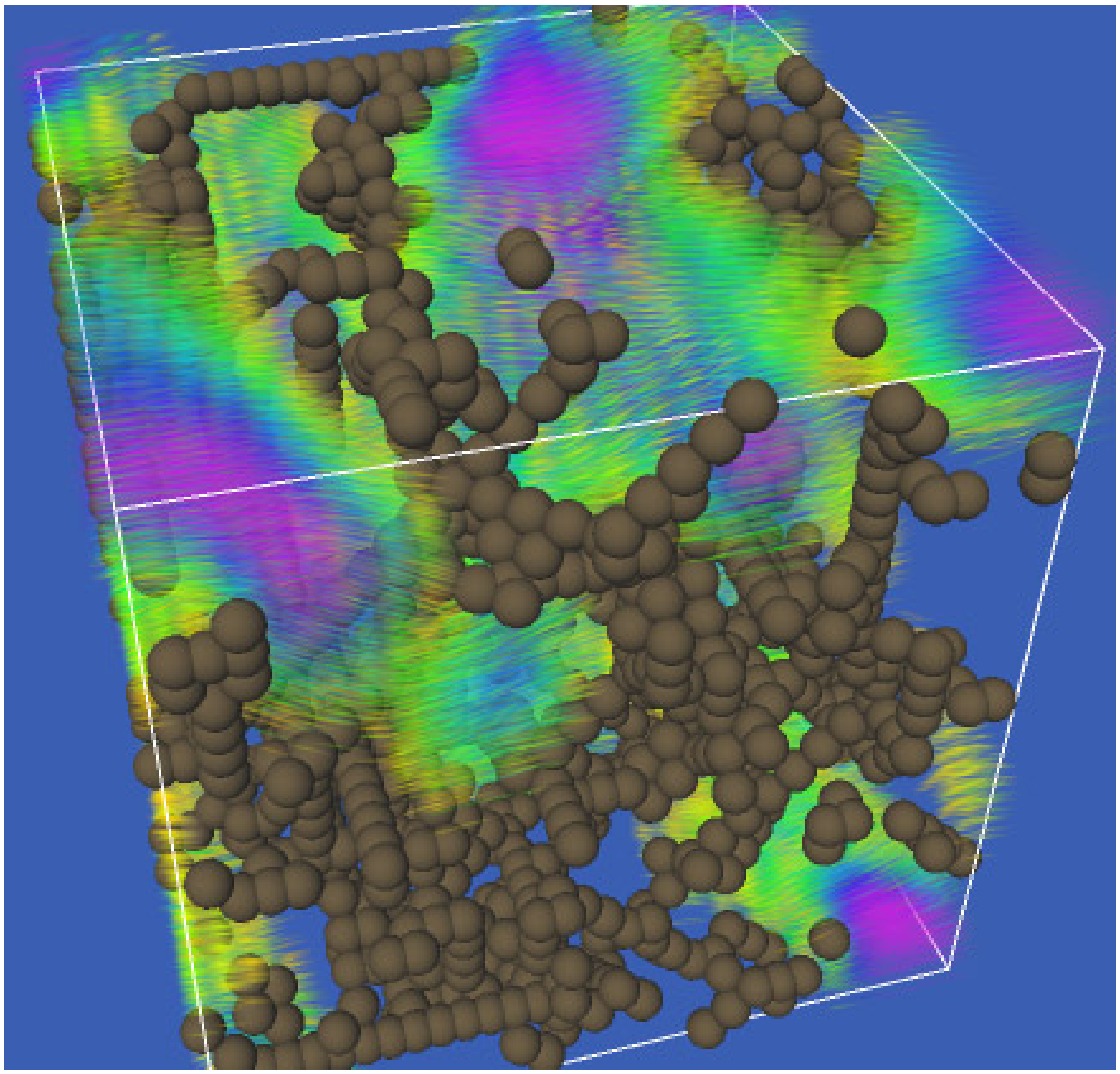
Figure 5-36 Color Line Bundles through areas of high velocity in aerogel simulation.

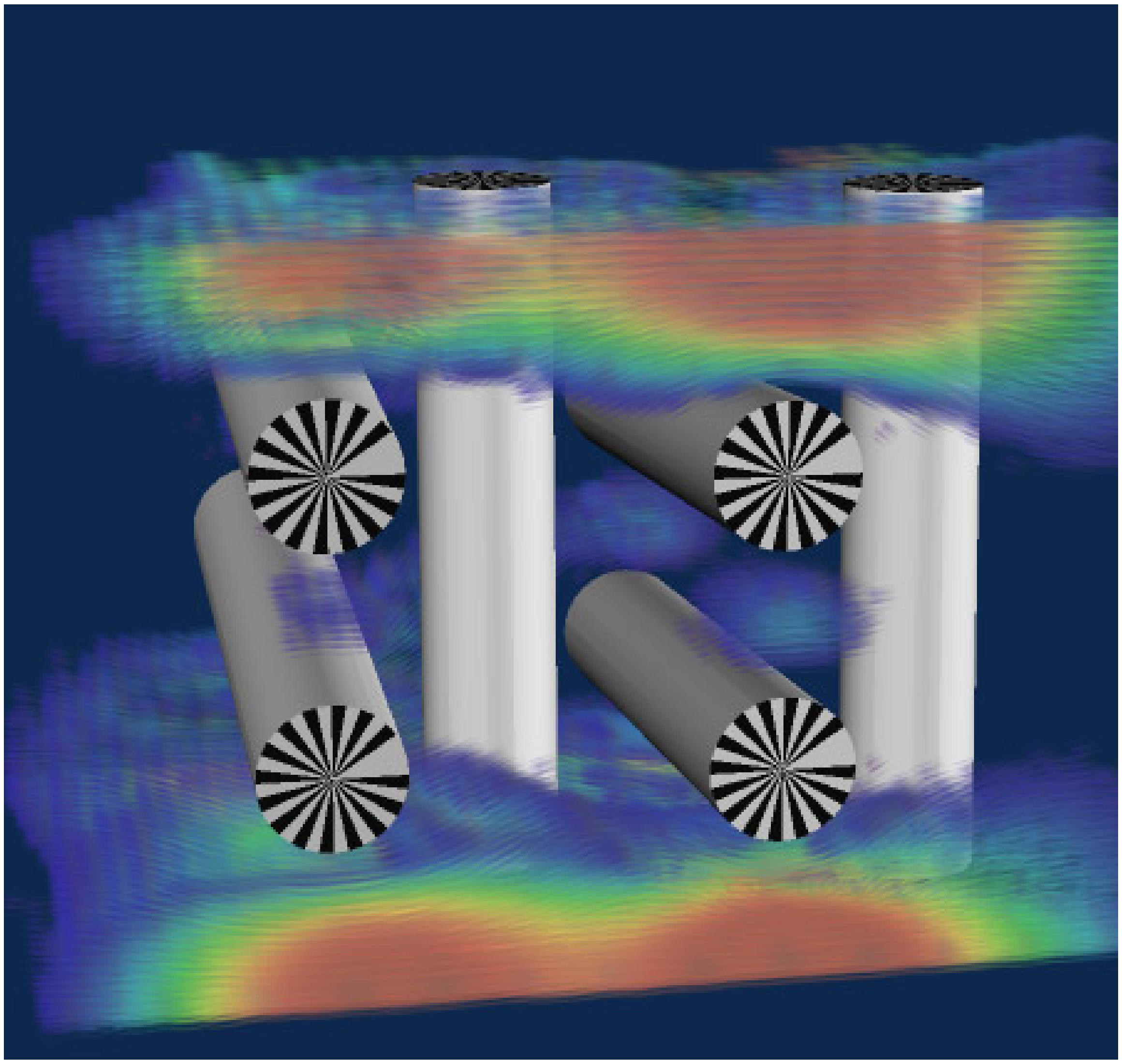

Figure 5-37 Colored line bundles through high areas of velocity of a HEPA (High-Energy Particulant Absorption) filter.

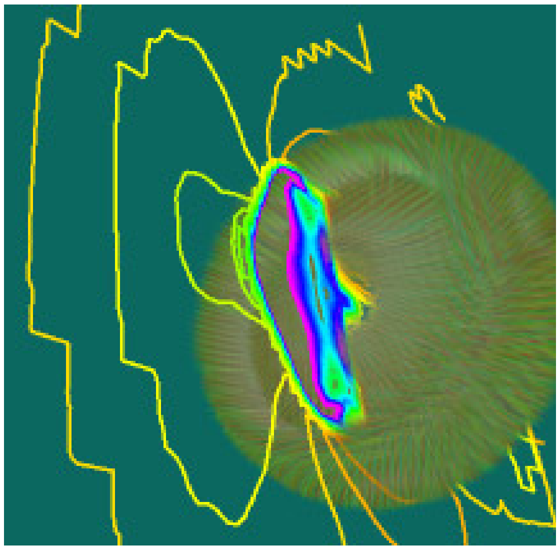

Figure 5-38 Line bundles depicting the velocity of an intersteallr cloud collision.

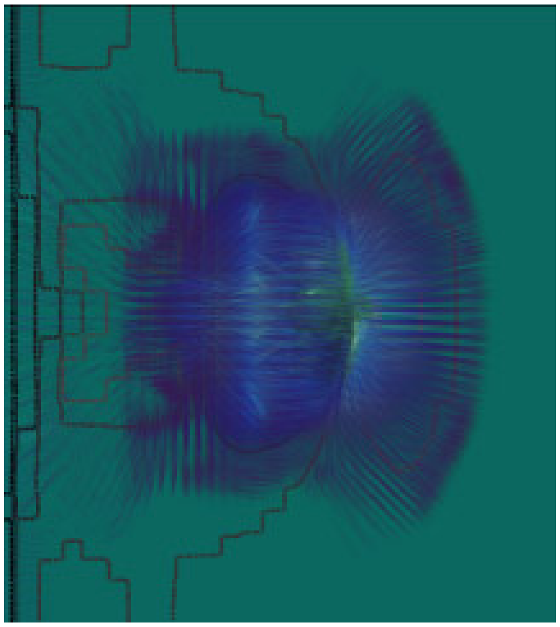

Figure 5-39 Line bundles depicting the velocity of an intersteallr cloud collision.

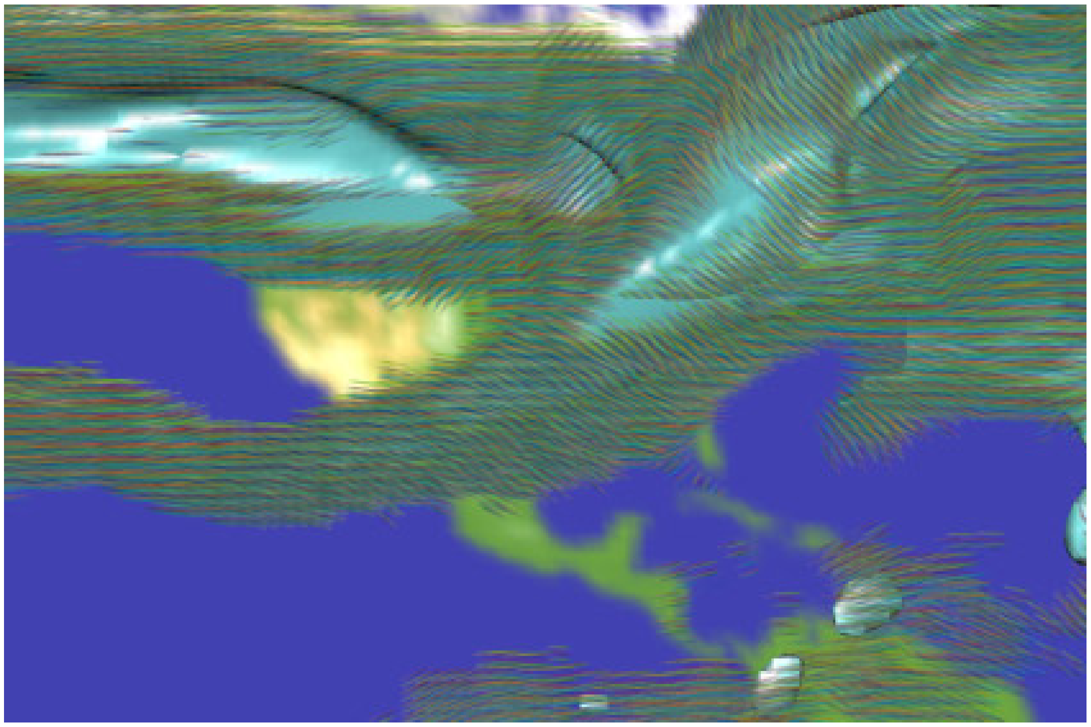

Figure 5-40 Winds over North America. An Isocontour of wind velocity magnitude accompianed with line bundles near the contour surface.

5.8 Open Inventor C++ Class definitions

5.8.1 SoSplatLineBundles.h

Below is the class definition (header) file for the SoSplatLineBundles class:

```
/*****************************************************************************
//  _____________________________________________________________________________
//  ___________                                                ______________
//  ___________     L A W R E N C E   L I V E R M O R E        ______________
//  ___________    N A T I O N A L   L A B O R A T O R Y       ______________
//  _____________________________________________________________________________
//
//
//    NAME
//      SoSplatLineBundles - Volume render by splatting a precomputed bunch of
//                             line segments.
//
//
//    VERSION
//      @(#)SoSplatLineBundles.h version 1.3  created on 95/03/31  at 16:23:27
//
//
//    NOTES
//      1) See the man pages or the implementation prologue for more info.
//
//
*****************************************************************************/
#include "LLNL_Disclaimer.h"


#ifndef  _SO_OSPLATBNL_
#define  _SO_OSPLATBNL_

#include "SoOctreeSplat.h"
#include <Inventor/fields/SoSFShort.h>
#include <Inventor/fields/SoSFBool.h>
#include <Inventor/fields/SoSFFloat.h>
#include <Inventor/fields/SoSFColor.h>
#include <Inventor/fields/SoSFVec3f.h>
#include <GL/gl.h>


class SoSplatLineBundles : public SoOctreeSplat {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatLineBundles);

  public:

    // Fields
    SoSFFloat   MinVecLength;
    SoSFFloat   MaxVecLength;
    SoSFShort   nLines;
    SoSFBool    smoothLines;
    SoSFFloat   VecScale;
    SoSFVec3f    VecJitter;
    SoSFVec3f    HSVJitter;
    SoSFColor    BaseColor;
    SoSFFloat    BaseTrans;

    // Constructor
    SoSplatLineBundles();

    // Destructor
    virtual ~SoSplatLineBundles();

  SoINTERNAL public:
    static void    initClass();

  protected:
    // Implements actions
    virtual void  renderSplat( int index, float x, float y, float z, float ss );
    virtual void  initGLRender( SoGLRenderAction *);

  private:
    SbVec3f  ColorJitterHSV;
    GLuint   GLLineBundleObj;
};

#endif /* _SO_SPLAT_BNL_ */
```

5.8.2 SoSplatLines.h

Below is the class definition (header) file for the SoSplatLines class:

```
/********************************************************************************
//  ____________________________________________________________________________
//  ___________                                        __________________
//  ___________    L A W R E N C E   L I V E R M O R E __________________
//  ___________   N A T I O N A L   L A B O R A T O R Y       __________________
//  ____________________________________________________________________________
//
//
//    NAME
//      SoSplatLines - Volume render by splatting a bunch of line segments.
//
//
//    VERSION
//      @(#)SoSplatLines.h version 1.2  created on 95/03/31  at 16:23:29
//
//
//    NOTES
//      1) See the man pages or the implementation prologue for more info.
//
//
********************************************************************************/
#include "LLNL_Disclaimer.h"


#ifndef  _SO_OSPLATLINES_
#define  _SO_OSPLATLINES_

#include "SoOctreeSplat.h"
#include <Inventor/fields/SoSFFloat.h>
#include <Inventor/fields/SoSFBool.h>
#include <Inventor/fields/SoSFShort.h>
#include <Inventor/fields/SoSFVec3f.h>


class SoSplatLines : public SoOctreeSplat {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatLines);

  public:

    // Fields
    SoSFFloat   MinVecLength;
    SoSFFloat   MaxVecLength;
    SoSFShort   nLines;
    SoSFBool    smoothLines;
    SoSFFloat   VecScale;
    SoSFVec3f    VecJitter;
    SoSFVec3f    HSVJitter;
    SoSFFloat   TransScale;

    // Constructor
    SoSplatLines();

    // Destructor
    virtual ~SoSplatLines();
  SoINTERNAL public:
    static void    initClass();

  protected:
    // Implements actions
    virtual void  renderSplat( int index, float x, float y, float z, float ss );
    virtual void  initGLRender( SoGLRenderAction *);

  private:
    SbVec3f *point1;
    SbVec3f *point2;
    SbColor  *ColorJitter;
};

#endif /* _SO_SPLAT_LINES_ */
```

Chapter 6

Textured Splats

6.1 Optimal Reconstruction Image

Westover [96] introduced splatting as a technique for representing scalar density functions. Laur and Hanrahan [47] extended this technique to octrees and allowed for hardware assisted rendering using an approximation to the splat as a collection of triangles. The premise of this and the Noise Splats techniques is that rather than use a collection of triangles for the reconstruction and representation of the volume density, a texture mapped square can be used. Once an effective texture-map is created for this purpose, slight alterations to the texture can be performed to encode other information.

Crawfis and Max [21, 22] uses a radially symmetric piecewise-cubic function to determine a texture that optimally reconstructs the volume while constrained to a small kernel size.

6.2 Anisotropic Textures

The initial goal of textured splats was to represent both a scalar and a vector field. An alternate goal is to represent only the vector field. These two goals may require totally different textures. This section will focus mainly on textures for representing both scalar and vector fields. Textures for vector field representation can be achieved efficiently for low-end workstations using the Line Bundles techniques discussed in chapter 4 or using this technique as discussed briefly later in this section.

Westover and Laur and Hanrahan proposed using a 3D gaussian for the reconstruction function:

$$h(x,y,z) = \sum_i \sum_j \sum_k \beta e^{-\alpha[(x-i)^2+(y-j)^2(z-k)^2]} .$$

The gaussian has to nice properties: it is radially symmetric and its integral along the viewing direction is itself a gaussian. The gaussian also never reaches zero, only getting arbitrarily close to it. This implies that each data point contributes to the entire resulting image (and in fact, an infinite image). Westover and Laur and Hanrahan simply truncate the gaussian to obtain a reasonable splat size. This can lead to artifacts at the edges of the splats. A better solution is a radially symmetric piecewise cubic function. Crawfis and Max optimized such a function for volume reconstruction. The resulting footprint function — the result of integrating along viewing rays — is used as the "reconstruction texture" for the work described here. This texture is used to volume render a scalar field. Modifications to this texture will allows us to represent vector field data as well. The reconstruction function developed by Crawfis and Max [21] has a non-zero footprint of less than 1.6. For vector field visualization we center this into a texture that logically extends to 2.1. This means that the splat texture centered around a data point will have a half-width of 2.1 for data points 1 unit apart, offering a fair amount of overlap. For the scalar reconstruction though, the texture has zero opacity from a radius of 1.6 to the edges.

The first experiment for embedding vector information into the reconstruction texture was to use tiny spermatozoon-shaped icons. Several of these icons were placed randomly within the texture and windowed to fade to zero at the edges. All of the icons were embedded pointing in the positive u-direction of the u,v coordinate space (i.e. horizontally). Icons that started close to the u=0 or u=1 edge were wrapped about these axes as if the texture were created on a cylinder. Icons that were close to the v=0 or v=1 boundaries were not wrapped, but with the windowing, faded to black near the borders. This wrapping is important for animation of the icons discussed below.

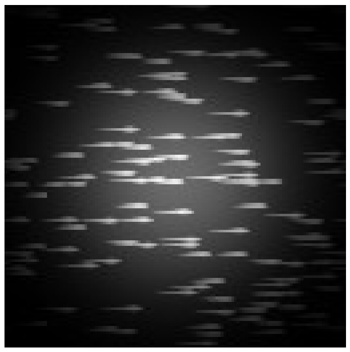

Figure 6-1 Sample Texture Splat with Embedded Vector Icons

The vector icons are embedded into the texture by taking the maximum of the reconstruction texture and the smooth vector icons for the opacity of the texture. A two-component texture, specifying the opacity and intensity, is used with the blend function of OpenGL. The intensity component of the texture is set strictly to the vector icons image. Figure 6-1 shows the opacity channel from a sample splat texture.

6.3 Orienting the Splats

The splats are constrained to be perpendicular to the viewer. For parallel projections, this implies parallel to the viewing plane. For perspective, the normal of the square should coincide with the viewing ray from the center of the splat. This is necessary for proper volume reconstruction of the scalar field. However, the vector field direction at the center of the splat may not lie in the plane of the splat. To resolve this, only the direction of the vector field projected onto the splat's plane is represented. The next section illustrates a technique to compensate for the direction of the vector normal to the splat.

Given the splat and the direction of the vector within that splat, the splat is rotated about its normal such that the embedded texture direction coincides with the projected vector direction. Since the scalar reconstruction portion of the splat is radially symmetric, it is invariant under this rotation and no compromise is made to the scalar rendering. This rotation would be complex if done in world coordinates. Splatting however is accomplished by calculating the inverse rotation and translation matrix for the current view. The splats are defined to be unit squares in the xy-plane. The splats are then translated and scaled into their proper position and projected to the viewing plane for scalar reconstruction. For vector field visualization, the additional step of rotating the splat in the xy-plane is performed before these operations. The vector field is defined in world coordinates. The inverse rotation matrix is used to transform the vector field to the normalized viewing space.

6.4 Foreshortening of the Vector Icons

The section above gives the algorithm for representing the projected direction of the vector field. For interactive settings, this is probably sufficient. However, it does not lead to convincing representations for static views. When viewing a line segment without lighting and shading as it rotates toward the eye and then away, it appears to simply get shorter as it points toward the eye and longer as it becomes perpendicular to the viewing direction. Shortening the splat is not possible while preserving the scalar reconstruction. The individual icons within the splat need to be shortened.

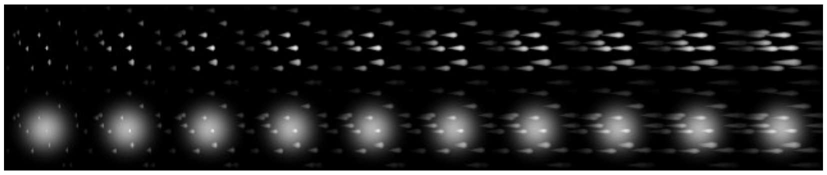

Figure 6-2 Texture Table for foreshortening — Intensity table is on top, Opacity on the bottom.

A table of splats was developed for this purpose. Figure 6-2 shows a series of splats with gradually larger icons. Which splat to use is determined by indexing into the table based on the magnitude of the vector's direction perpendicular to the splat. Since the vector was previously transformed to normalized viewing space, this is simply the z-component of the transformed vector.

6.5 Time Dynamics of the Vector Icons

Animation of the vector icons can greatly enhance the representation of the vector field. This can be accomplished by moving the vector icons through the reconstruction texture in a cyclical fashion. Since the vector icons are embedded into the texture along with the scalar splats, they can not be shifted separately. One solution to this is to simply use two splats for each data point. First render the scalar splat and then render the vector splat on top of it. The texture coordinates can be shifted on the vector splat to cycle the texture using a texture mapping setting to allow the texture to repeat itself for values outside of zero to one. The vector icons however are windowed to allow a smooth transition and avoid harsh edges. For this scheme to work, the windowing would need to be removed. The resulting harsh edges would produce patterns and unwanted artifacts in the rendering.

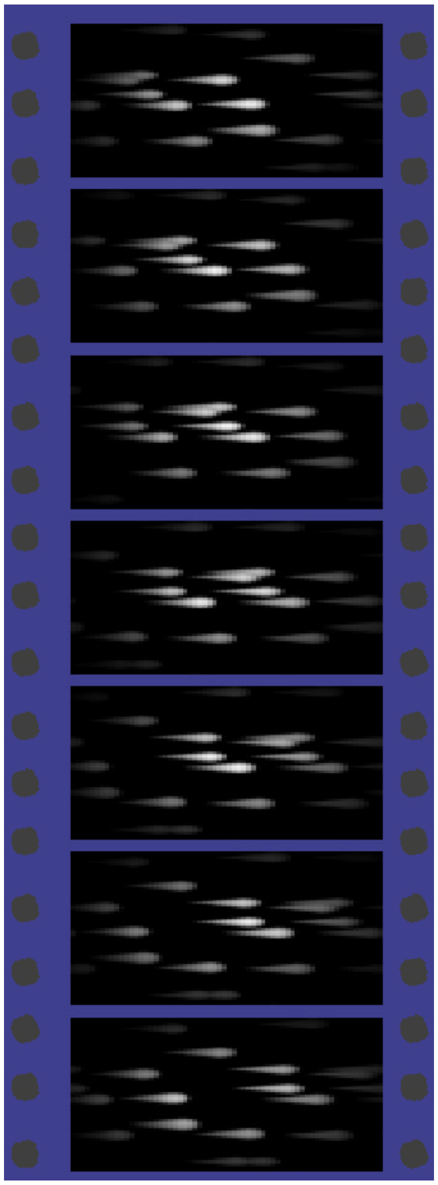

Figure 6-3 Texture Table for Animation - Intensity Component

A better solution is to again create a table of splats with the vector icons cyclically shifted and windowed before being embedded into the reconstruction texture. These precomputed textures can then be selected in sequence to give the vector field representation the appearance of movement. The textures developed for this are represented in Figure 6-3. Sixteen textures were generated to allow a smooth animation. Combined with the ten textures for foreshortening, a total of 160 textures are needed. Each texture is small, only 32 by 32 pixels, with 2-bytes of information per pixel. Figures 6-4 and 6-5 show the 2D table of splat textures organized as foreshortening across the rows and time dynamics down the columns.

Smooth animation however implies rendering rates of ten frames per second or better. For large data sets, this may not be possible. However, sixteen frames from a static view can be rendered and saved. These frames can then be played back in a continuous cycle to give a dynamic representation of the flow field.

Even though the textures are designed for a phase speed of one sixteenth, a faster or slower speed can be specified. The user can control the speed of the animation by specifying a phase speed for each class instantiation. A phase speed of 0.0625 or one sixteen is the default setting. Hence, sixteen frames will generate a cycle. Speeds that are not simple multiples of 1/16 or fractions of 1/16 — 1/32, 1/48, etc. — may move non-uniformly.

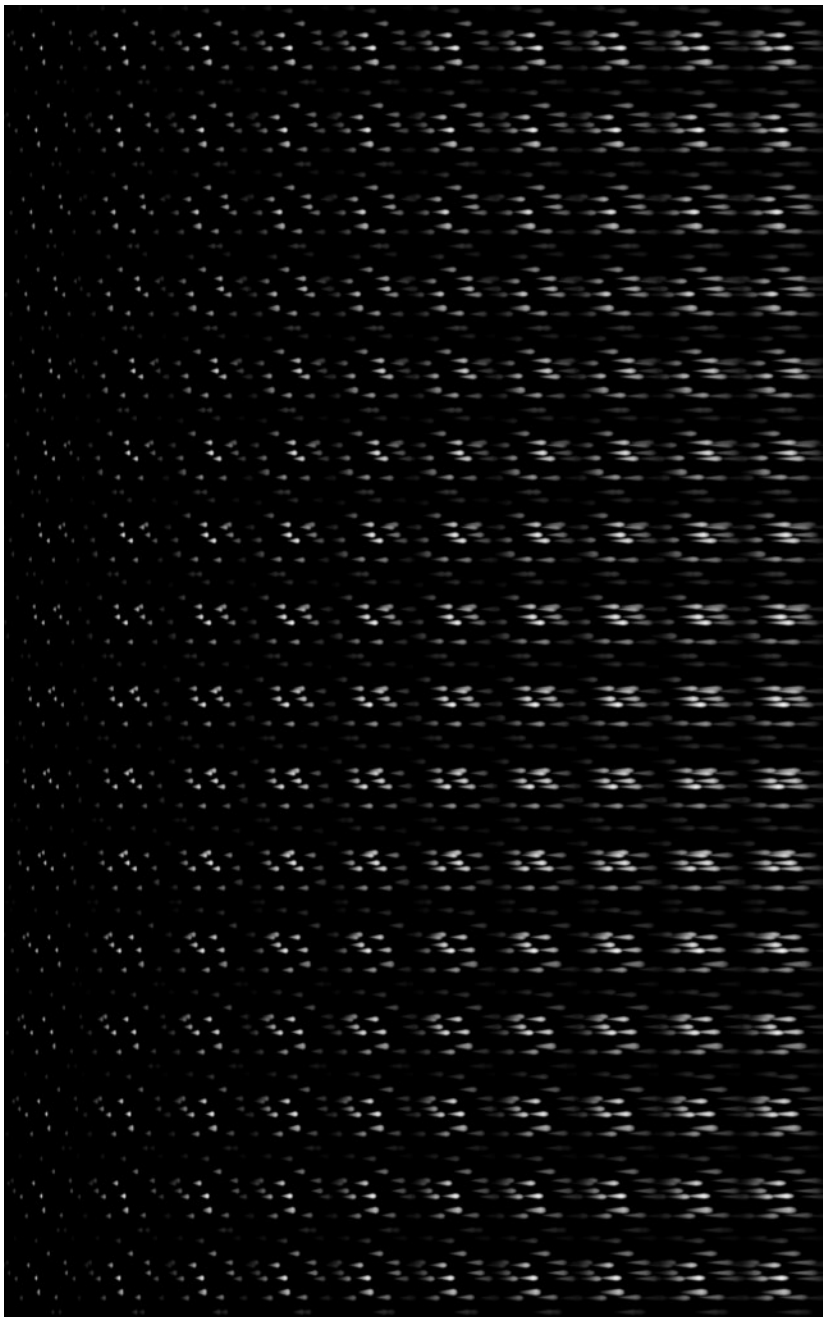

Figure 6-4 160 Splats - Intensity Component of Textures

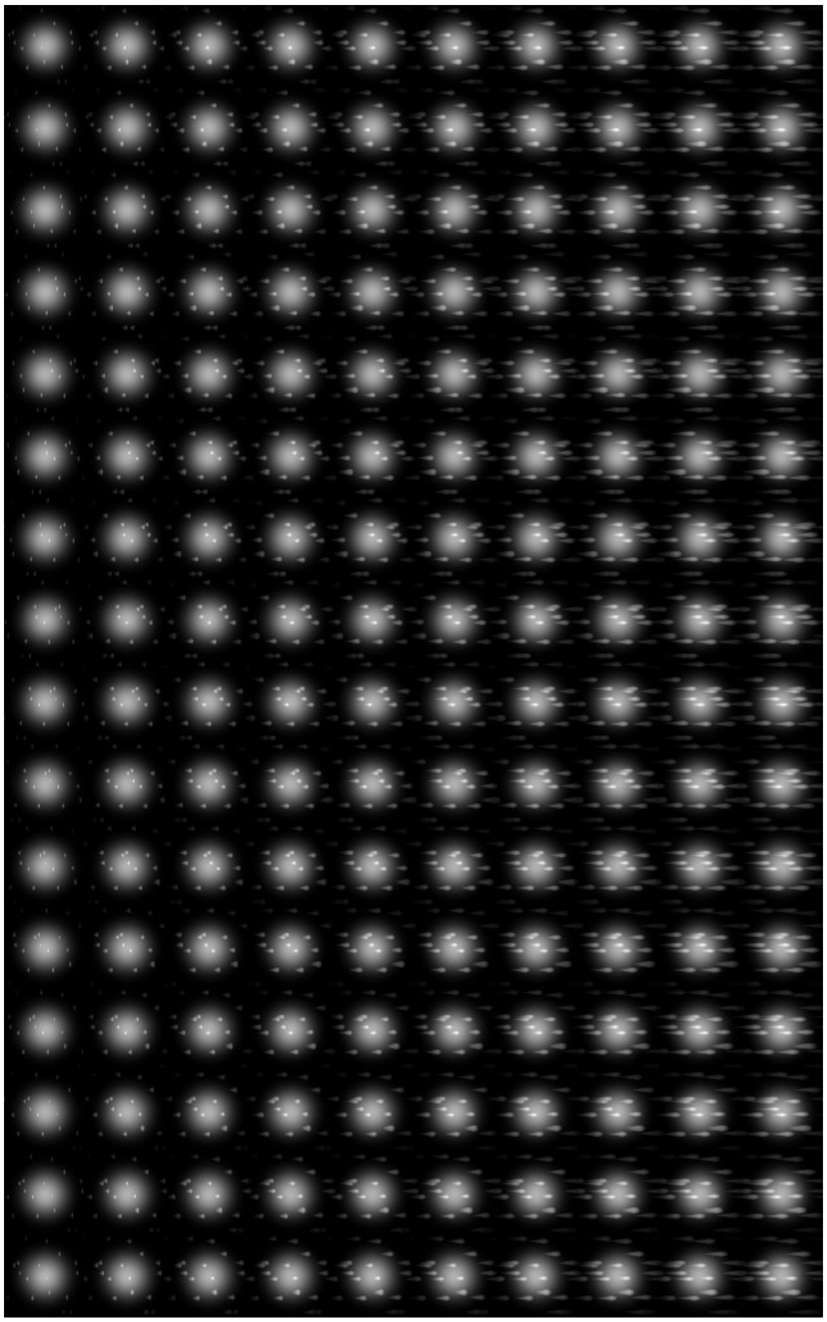

Figure 6-5 160 Splats - Opacity Component of Textures

The vector icons discussed so far do not represent the vector field's magnitude, only its direction. By giving each splat a separate phase speed, a rudimentary representation of the magnitude is generated. This is accomplished by determining the magnitude of the vector at each splat and using it to determine a scale factor for the global phase speed. Areas with very slow velocity will appear stagnate. Areas with very high velocities will move faster. Care must be taken in the scaling to not exceed a maximum threshold. Otherwise, the areas of high velocity could appear stagnant. If the scaling factor is sixteen and the global phase speed is one sixteenth, the resulting phase speed for the splat would be one and the exact same splat would be selected for each frame. This would obviously yield a static representation for this area. Constraining the calculated phase speed for each splat to be lower than about one fourth produces acceptable results. Of course, the phase speed is also required to be non-negative.

This arbitrary phase speed for each splat has the unfortunate property that a quick 16 frame movie loop can not be generated for continuous playback. All of the splats would need to return to their starting positions at the same time. This time can be arbitrarily large for fractional phase speeds. To maintain the dynamic playback, a time, t, that is a multiple of 16 can be selected and the textures can be forced to cycle back at this time by either restricting the phase speeds to an even divisor of this time, or adjusting the current table entry selection such that it forms a cycle. Of course only phase speeds that are a multiple of $1/t$ can be handled with this solution. Speeds lower than $1/t$ can be made to be stagnant.

A final benefit of the table of cycled splats is the ability to create better textures. When all data points use the exact same splat or splats from the same row, subtle patterns in the vector icon's position can be apparent. When each splat has a different phase speed, this will rapidly change. For cases where a global phase speed is used, a solution needs to be developed. A simple solution would be to randomly pick which row of splats to use in the table of cycled splats. This will produce better textures for static views, but will not work with the time dynamics, since we need the splat used for a data point to cycle in a continuous fashion. Since the splats are drawn in a different order depending on the view, the random numbers can be different for each redraw even if the random number generator is reseeded. What is required is an initial random starting row selection within the table for each splat, but then a consistent selection — down the column — for each subsequent redraw. To accomplish this, we use a unique identifier for each splat, the location within the compressed octree hierarchy. This identifier is used to seed the random number generator for each splat. A consistent number is then returned for a data point on each redraw. This number is used to index into the table of splats for the initial rendering by selecting a starting row. Subsequent renderings calculate an index based on this number and the phase speed. Since each splat has a unique identifier with which to seed the random number generator, a random selection into the splat table is generated for each redraw. Alternatively, a random starting location could be calculated and stored for each splat. Since there may be millions of splat points, a trade-off of CPU time versus memory favors using additional CPU time.

6.6 Vector Icon Colors

The textures consist of only the intensity and the opacity channels. OpenGL allows a useful texture mapping operation with these that can separate the vector icons from the reconstruction footprint. Two colors are used with the intensity and opacity of the texture maps: the color or colors of the polygon and a specified texture color. After the color, C_p, and opacity, α_p, of the polygon for a certain projected pixel is calculated, the intensity, I_t, and opacity, α_t, of the texture are used to calculate the final color. The equation used is:

$$I = C_p (1-I_t) + C_t I_t$$
$$\alpha = \alpha_t \alpha_p$$

where, C_t is a defined texture color. A two-component texture is used to represent intensity and opacity (Figures 6-4 and 6-5). Hence, the texture does not have a color. The texture color above is constant for the entire scan-conversion of the polygon, but may be changed between polygons.

The texture color can thus be used to specify the magnitude of the vector field for each splat. Care must be taken such that the color scheme used for the mapping of the vector icons does not cause confusion with the color scheme used for the scalar field of the splats. The texture color can also be used to represent a different variable or to provide a greater depth cue. Figure 6-6 maps the texture color to the altitude height, allowing another indicator of the where the flow resides in three-space.

6.7 Representing Only Vector Fields

Representing only a single vector field in three-dimensions can be beneficial. New textures could be developed to achieve this. These textures should maintain the useful benefits of foreshortening and animation. Rather than creating new textures, the existing textures can be remolded for this purpose by several simple steps. First, use the intensity channel of the textures defined above for both the opacity and the intensity. Secondly, switch to the modulate texture mapping function in OpenGL [7]. This option will simply multiply the polygon's colors by the intensity channel of the texture and the polygon's opacity by the opacity of the texture. Hence, only the vector icons are drawn, but they are drawn in the polygon's color. Finally, the splat can optionally be further stretched in the vector field direction to produce a more brush-like appearance. The wind velocities over North America are represented in Figure 6-7. Since a scalar function is not being represented, the color for each splat, and hence the vector icons, can be used to represent the velocity magnitude. This generates an image of an anisotropic volume density cloud, where both the opacity and the color can be used to represent the flow magnitude.

The amount of stretching can also be controlled individually for each splat by the vector magnitude. With this stretching and a fixed phase speed, the perceived phase speed will be greater for the splats with greater stretching. Splats with greater velocity magnitude will be stretched longer, but the texture will still cycle through this longer distance in the same amount of time to produce a faster motion. This stretching is possible, since the splat size is not constrained by the scalar reconstruction.

6.8 Performance / Implementation Issues

The first step towards generating the textured splats was to extend the SoOctreeSplat class used by the Explorer product from SGI to use texture mapped squares rather than points or polygons for the scalar volume rendering. To this end, the class was rewritten into a base class and to allow new subclasses which could perform the rendering using different algorithms. The initial SoOctreeSplat class was split into the base class of the same name and subclasses: SoSplatConstant, SoSplatLinear, SoSplatGaussion to render the volumes as constant colored squares, a regular polygon consisting of triangles from the center point, and Laur and Hanrahan gaussian triangular meshes, respectively. The class, SoSplatTexture, was generated to perform more accurate and efficient volume rendering on machines with hardware support for texture-mapping. A few key member functions are all that need to be redefined for a new class and representation. Three of these are mainly boilerplate member functions required of any new Inventor class: the constructor, the destructor and the initClass methods. The renderSplat member function is called to render a single splat. By rewriting this virtual function for each new class, new representations are possible. Since new representations may require different graphics parameters or states, two additional routines are providing for initializing and restoring the graphics state before and after each redraw: initGLRender and restoreGLRender respectively. Expensive operations that should only be done once per redraw can also be placed within these two routines. The loading of the texture maps and the creation of OpenGL display lists are performed once for a class instantiation (or once for all class instantiations) on the first request to render the class. The renderSplat member function will simply orient the splat in the proper direction and post the precomputed OpenGL display list to composite the splat into the image.

Three classes have been developed for representing vector fields with texture splats: SoSplatVectors, SoSplatVecCloud and SoSplatVandS. SoSplatVectors and SoSplatVecCloud are used for representing only a vector field. SoSplatVandS can be used to represent a scalar and a vector field. The SoSplatVectors class renders the vectors only as described above using the modulate texture mapping function. Thus, only the vector icons are drawn as in Figure 6-8. The SoSplatVecCloud class actually represents the velocity magnitude scalar field and the directionality of the vector field. All of the vector icons are the same user specified color, while the scalar component of the splats represents the velocity magnitude. It is mainly used to represent a volume rendering of the vector magnitude with embedded direction indicators within it. Figure 6-9 shows the VecCloud class applied to the tornado test data. It lacks the vector scaling parameter of the SoSplatVectors, but adds a color member for specifying the vector icon's color.

The class SoSplatVandS represents both a vector field and a scalar field, hence V(ector)andS(calar). An additional color for each splat data point is specified for the vector icons. The texture color is then set to that splat's vector color each time a splat is rendered. Figure 6-6 used the percent cloudiness variable of a global wind climate simulation color-coded using a gray scale color scheme for the scalar volume rendering, while the direction of the wind velocities were encoded within the textures. A separate scalar variable, the altitude of the splat data point was used to map colors to the vector icons. Red icons represent winds near the surface of the earth, while blue icons represent winds in the upper atmosphere. An inverse hot to cold color table is used in between, going from red to yellow to green to cyan and finally to blue.

6.9 Applications

Textured splats have been applied to several different data sets and used in parts of several videos. Figures 6-6 and 6-7 represent the textured splats applied to a global climate simulation. The simulation used over a million data points and variables were calculated for each hour of a thirty day simulation. The quarter of a million zone over North America are represented in Figures 6-6 and 6-7. Figure 6-6 uses a gray scale to represent the areas where percent cloudiness is high — greater than twenty percent cloud occlusion. Vector icons color-coded by the altitude are embedded into the volume rendering to indicate the wind velocity direction through the clouds. In Figure 6-7, an anisotropic volume rendering of the wind velocity illustrates not only areas of high and low wind velocity magnitude, but also the actual wind directions. A cold-to-hot (blue to cyan to green to yellow to red to white) color mapping is used to represent the changes in the velocity magnitude, with red and white mapped to the highest velocities and blue to lower velocities. Very low velocities are completely transparent. The colored plane in the back represents the out-going long-wave radiation leaving the atmosphere, an important property of climate simulations. The plane is positioned however, at the location of the earth's surface.

Figure 6-10 shows the textured splats applied to a simulation of air flow through a simulated aerogel material. Aerogel is described in Section 5.6. The yellow surface fragments are used to represent volumes occupied by aerogel in the simulation. The color represents the velocity magnitude, where high velocity magnitudes are represented by magenta, and low velocity magnitudes are represented by green and yellow. This image illustrates the ability of textured splats to represent a scalar field effectively, while also integrating in vector field information.

HEPA (High-Energy Particle Absorption) filters are used in many industrial application to clean or scrub contaminants for the air. Figure 5-37 shows the areas of high velocity magnitude of air flowing through a small piece of a HEPA filter under design. The areas of high-velocity magnitude are represented by color-coding the volume reconstruction of the velocity magnitude from red for high velocities to blue for lower velocities. The air flow direction is embedded into the volume rendering using small black vector icons in the texture splats.

Figure 6-12 represents data from a simulation of a shock wave impinging on an interstellar cloud. The high velocities at the shock front and cloud intersection are represented. When animated with a sixteen frame loop, a recirculating pattern can be seen just behind the cloud/shock interaction (behind the green isosurface of pressure).

6.10 Color Images

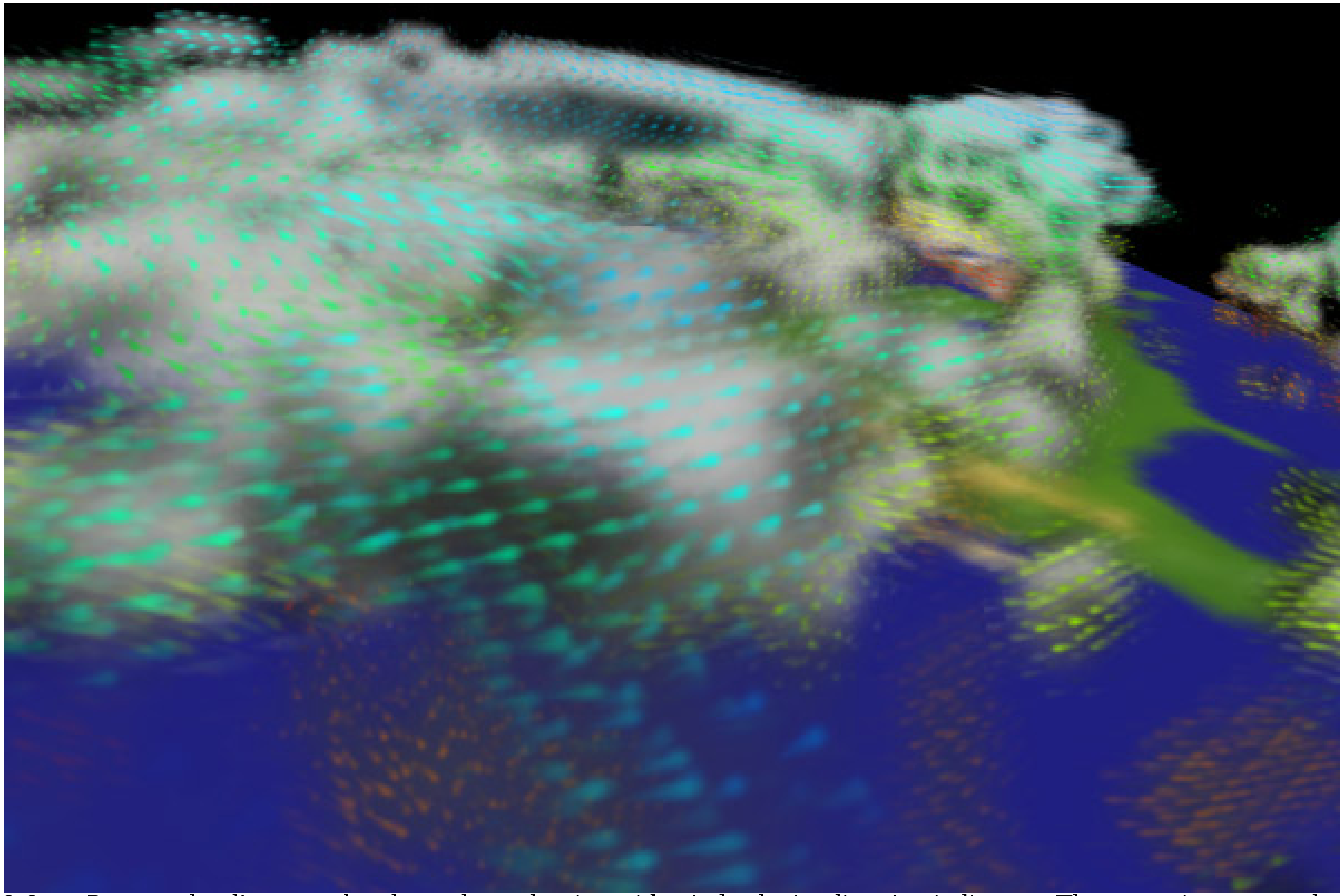

Figure 6-6 Percent cloudiness rendered as volume density, with wind velocity direction indicators. The vector icons are color-coded by altitude

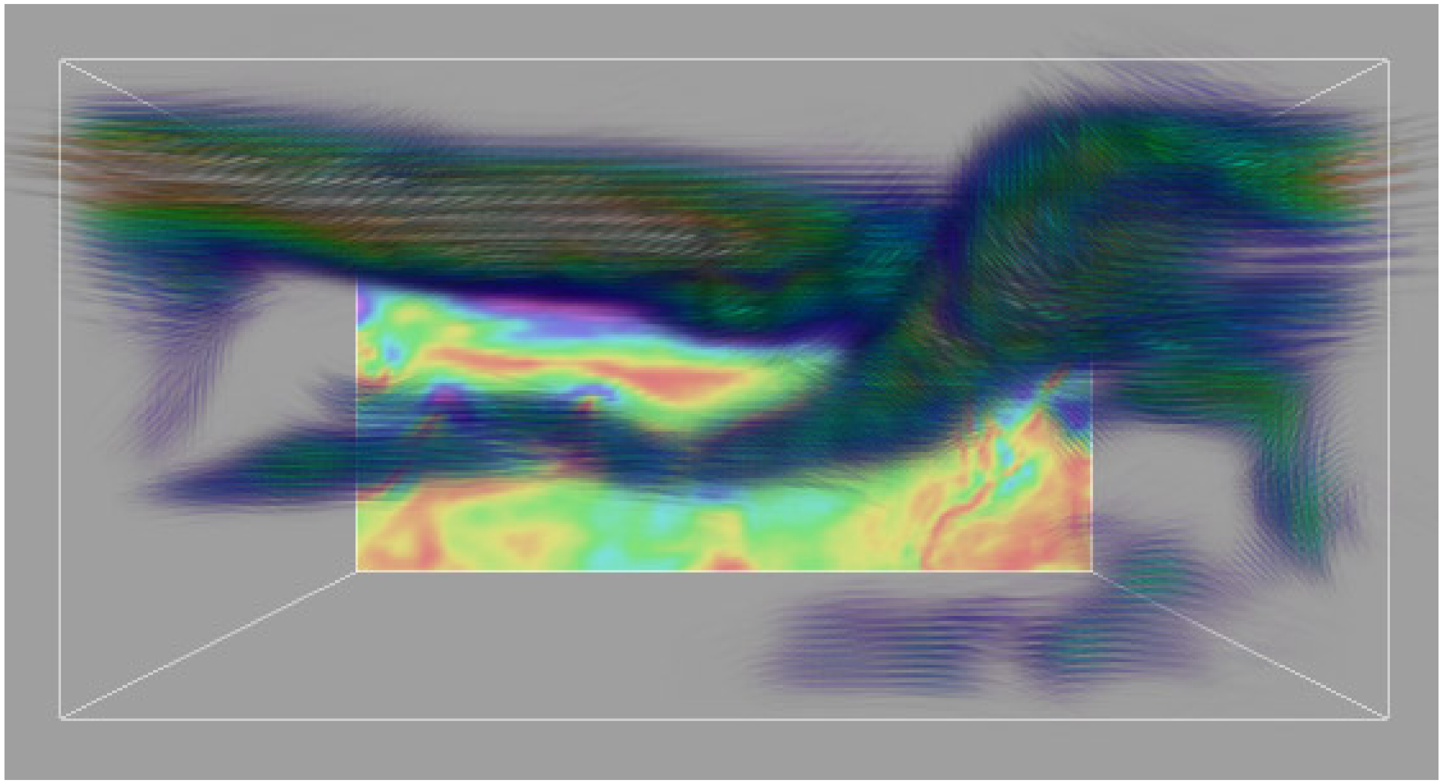

Figure 6-7 Wind velocities over North America are represented using the Textured Splats. Higher wind velocity magnitudes are red or white. The outgoing long-wave readiation is depicted on the bottom plane.

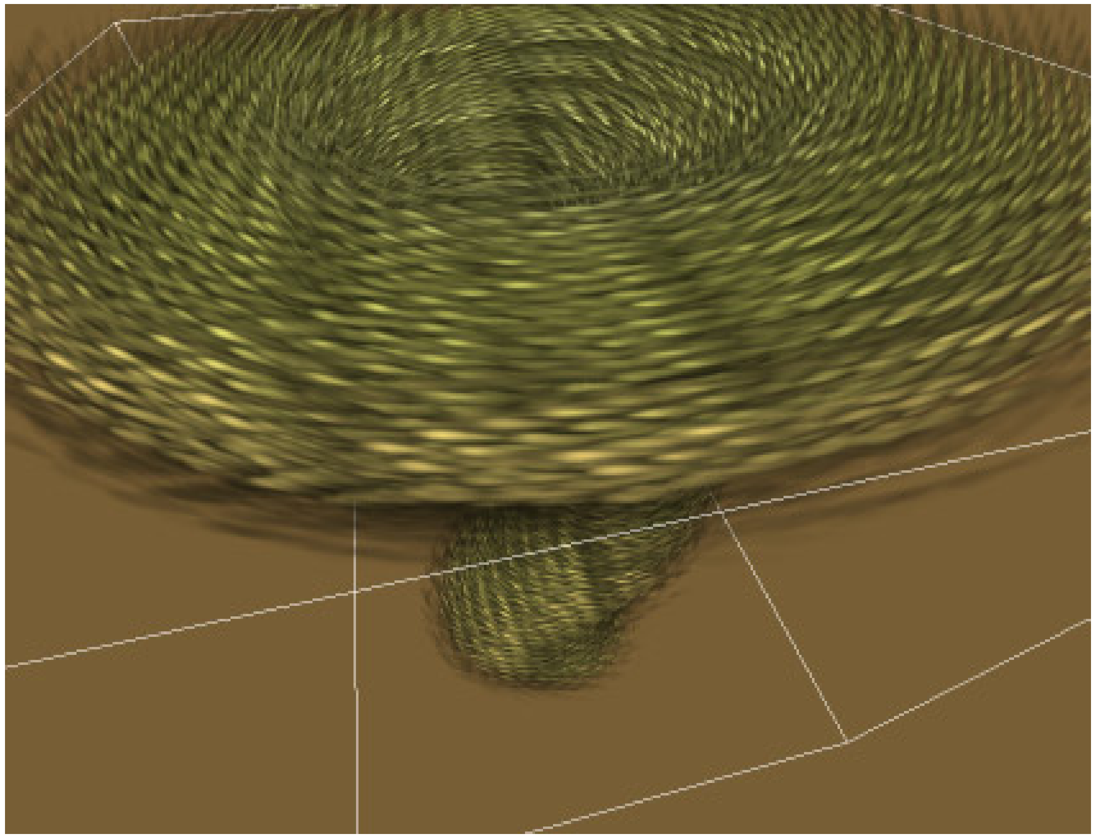

Figure 6-8	Test tornado rendered using the textured splats class SoSplatVectors.

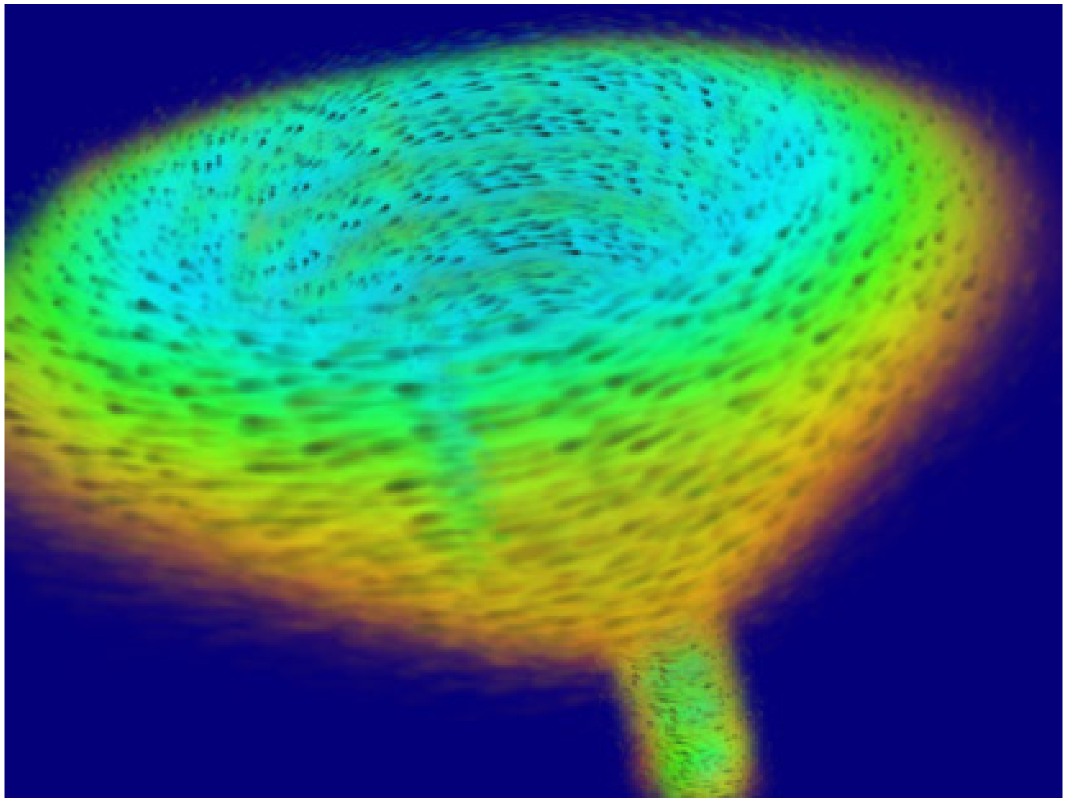
Figure 6-9	Test Tornado rendered using the textured splats class SoSplatVecCloud. The volume density cloud is the velocity magnitude.

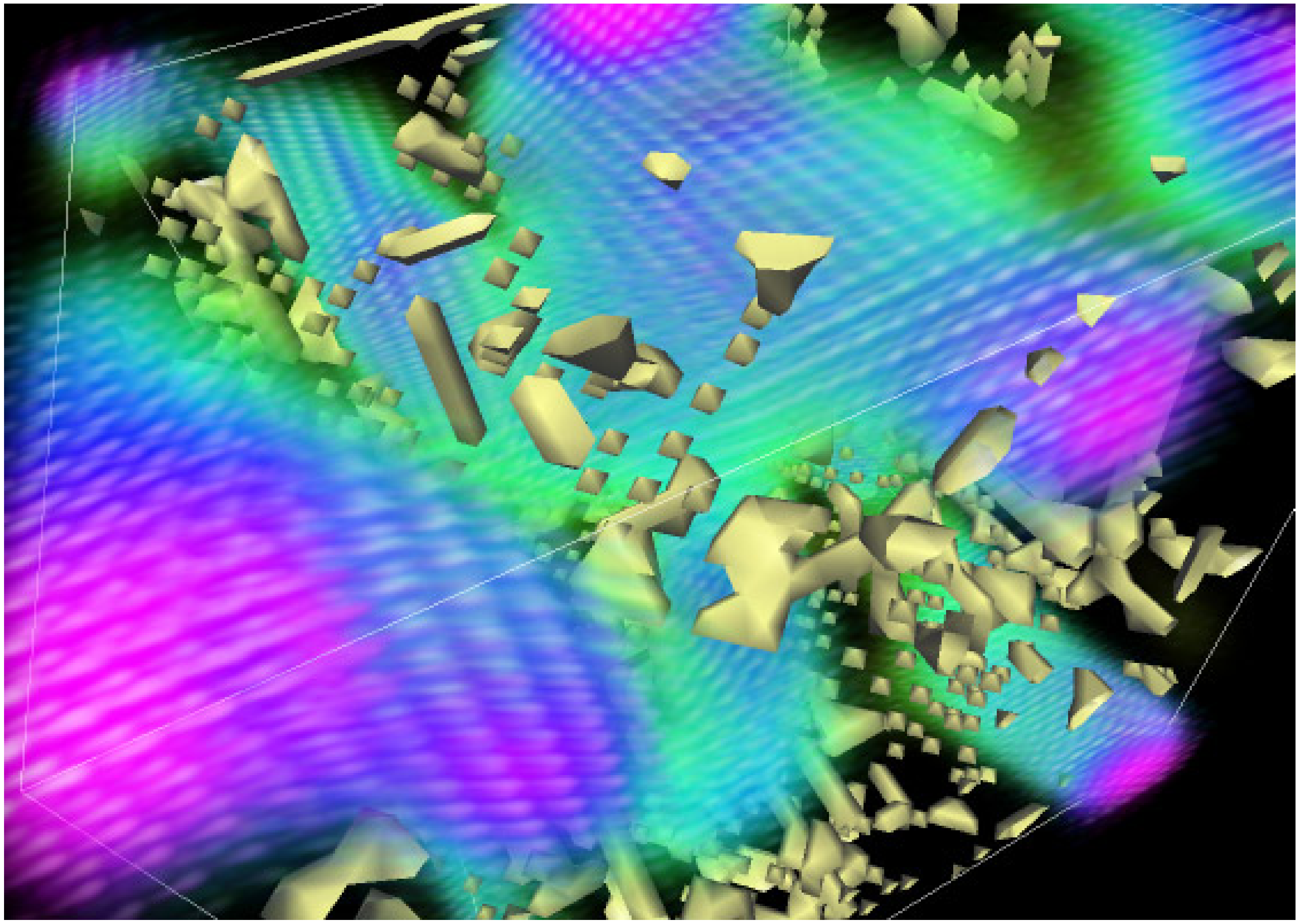

Figure 6-10 The flow through a simulated aerogel material is represented using the textured splats.

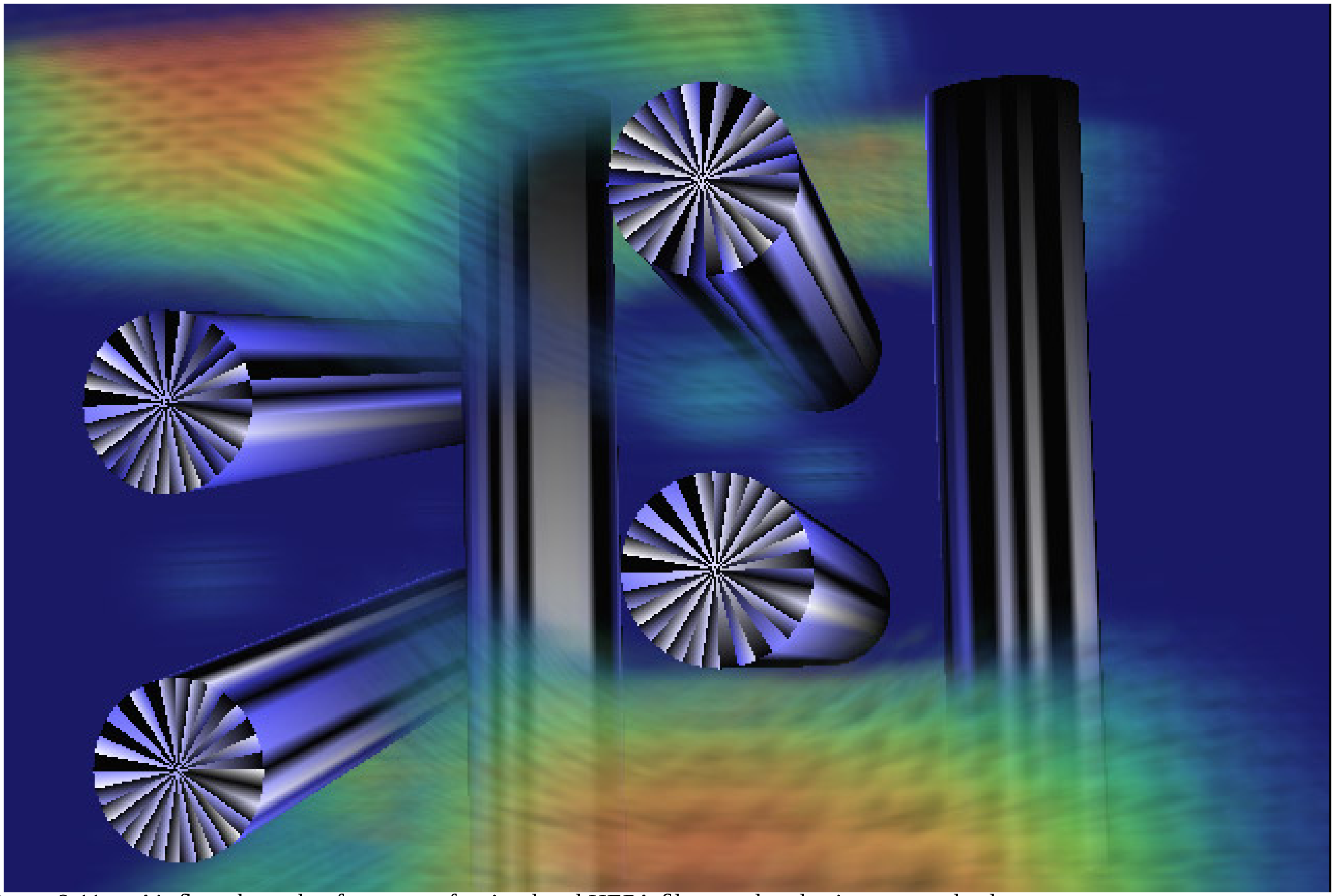
Figure 6-11 Air flow through a fragment of a simulated HEPA filter rendered using textured splats.

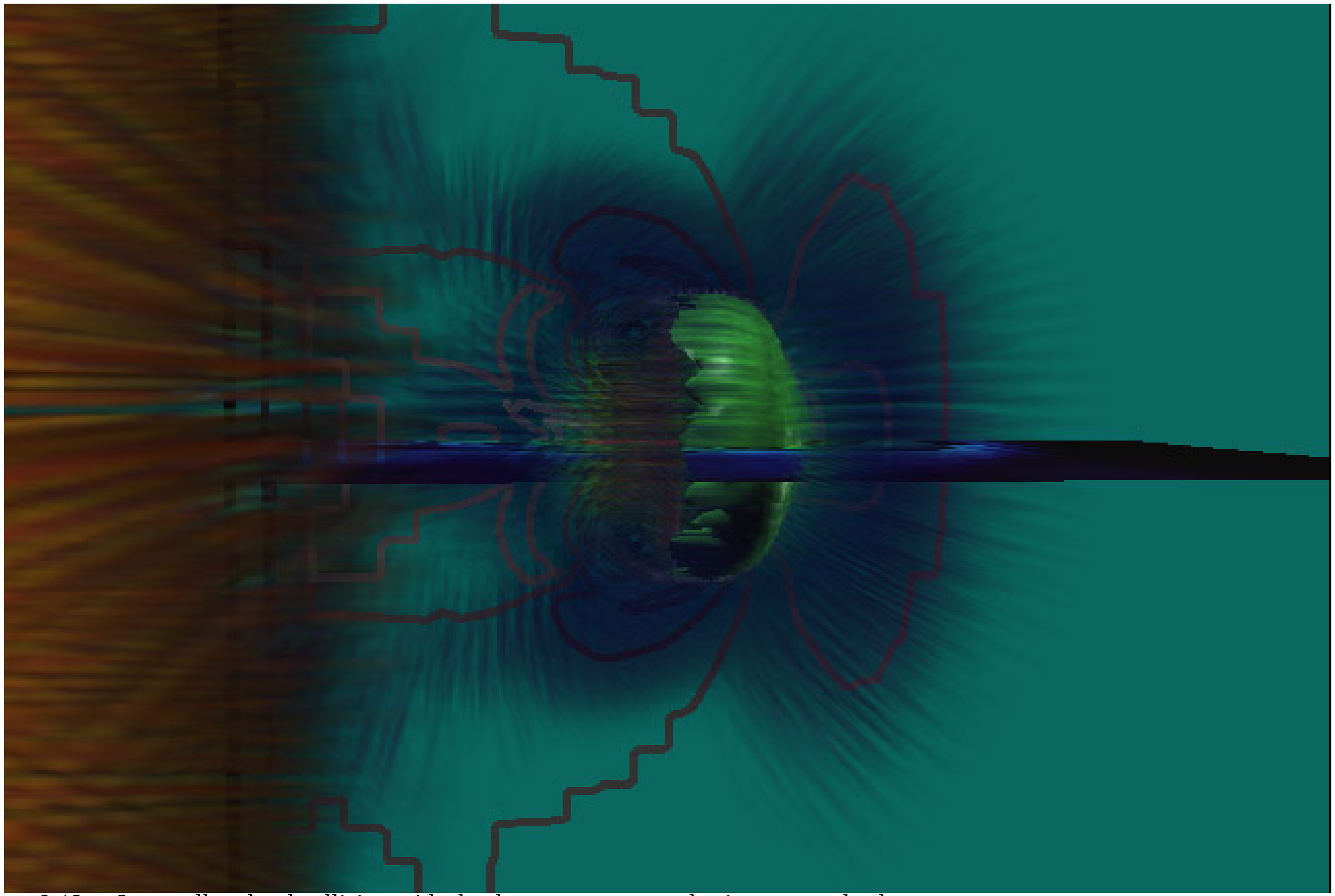
Figure 6-12 Interstellar cloud collision with shock wave represented using textured splats.

6.11 Open Inventor C++ class definitions

6.10.1 SoSplatVectors.h

Below is the class definition (header) file for the SoSplatLineBundles class:

```
/*******************************************************************************
//  ____________________________________________________________________________
//  ___________                                                ________________
//  ___________      L A W R E N C E   L I V E R M O R E      ________________
//  ___________     N A T I O N A L   L A B O R A T O R Y     ________________
//  ____________________________________________________________________________
//
//
//    NAME
//   SoSplatVectors - Volume render of a vector field by splatting texture
//                      mapped squares. An anisotropic texture is used.
//
//
//    VERSION
//   @(#)SoSplatVectors.h version 1.4  created on 95/03/31  at 16:23:26
//
//
//    NOTES
//   1) See the man pages or the implementation prologue for more info.
//
//
*******************************************************************************/
#include "LLNL_Disclaimer.h"


#ifndef  _SO_OSPLATVEC_
#define  _SO_OSPLATVEC_

#include "SoSplatTexture.h"
#include <GL/gl.h>


class SoSplatVectors : public SoSplatTexture {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatVectors);

  public:

    // Fields
    SoSFFloat   VecScale;
    SoSFFloat   MinVecLength;
    SoSFFloat   PhaseSpeed;

    // Constructor
    SoSplatVectors();

    // Destructor
    virtual ~SoSplatVectors();

  SoINTERNAL public:
    static void           initClass();
  protected:
    // Implements actions
    virtual void  renderSplat( int index, float x, float y, float z, float ss );
    virtual void  initGLRender( SoGLRenderAction *);

    float phase_shift;
    SbMatrix  invmat;

  private:
    static GLuint  VecSplats;

    static int            textureLoaded;
};

#endif /* _SO_SPLAT_VEC_ */
```

6.10.2 SoSplatVecCloud.h

Below is the class definition (header) file for the SoSplatLineBundles class:

```
/*******************************************************************************
//  ___________________________________________________________________________
//  ___________                                               _________________
//  ___________      L A W R E N C E   L I V E R M O R E      _________________
//  ___________     N A T I O N A L   L A B O R A T O R Y     _________________
//  ___________________________________________________________________________
//
//
//    NAME
//   SoSplatVecCloud - Volume render a vector field by splatting
//                    anisotropic textured squares. The magnitude
//                    is reconstructed using Textured Splats.
//
//
//    VERSION
//   %Z%%M% version %I%  created on %E%  at %U%
//
//
//    NOTES
//   1) See the man pages or the implementation prologue for more info.
//
//
*******************************************************************************/
#include "LLNL_Disclaimer.h"


#ifndef  _SO_OSPLATVecCloud_
#define  _SO_OSPLATVecCloud_

#include "SoSplatVectors.h"
#include <Inventor/fields/SoSFColor.h>
#include <GL/gl.h>


class SoSplatVecCloud : public SoSplatVectors {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatVecCloud);

  public:

    // Fields
    SoSFColor  VecColor;

    // Constructor
    SoSplatVecCloud();

    // Destructor  - was private??? - should be virtual???
    virtual ~SoSplatVecCloud();

  SoINTERNAL public:
    static void           initClass();

  protected:
    // Implements actions
    virtual void  initGLRender( SoGLRenderAction *);
    virtual void  renderSplat( int index, float x, float y, float z, float ss );

  private:
    static GLuint  CloudSplats;

    static int            textureLoaded;
};

#endif /* _SO_SPLAT_VaS_ */
```

6.10.3 SoSplatVandS.h

Below is the class definition (header) file for the SoSplatLineBundles class:

```
/******************************************************************************

//  ____________________________________________________________________________
//  ___________                                                 ________________
//  ___________      L A W R E N C E   L I V E R M O R E        ________________
//  ___________     N A T I O N A L   L A B O R A T O R Y       ________________
//  ____________________________________________________________________________
//
//
//    NAME
//   SoSplatVandS - Volume render a scalar and a vector field using
//                 Textured Splats.
//
//
//    VERSION
//   @(#)SoSplatVandS.h version 1.2  created on 95/06/06  at 08:41:52
//
//
//    NOTES
//   1) See the man pages or the implementation prologue for more info.
//
//
******************************************************************************/

#include "LLNL_Disclaimer.h"

#ifndef  _SO_OSPLATVaS_
#define  _SO_OSPLATVaS_

#include "SoSplatVectors.h"
#include <GL/gl.h>


class SoSplatVandS : public SoSplatVectors {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatVandS);

  public:

    // Fields

    // Constructor
    SoSplatVandS();

    // Destructor  - was private??? - should be virtual???
    virtual ~SoSplatVandS();

  SoINTERNAL public:
    static void           initClass();

  protected:
    // Implements actions
    virtual void  renderSplat( int index, float x, float y, float z, float ss
);
    virtual void  initGLRender( SoGLRenderAction *);
  private:
      static GLuint       Splats;
      static int   textureLoaded;
};

#endif /* _SO_SPLAT_VaS_ */
```

Chapter 7

Hair Splats

Watching the leaves of a tree, the grass or a person's hair blow in the wind can give an indicator of the intensity and direction of the wind. Experimental fluid flow techniques similarly rely on tufts or oils spread across surfaces to visualize the flows on or around the surface. The goal of Hair Splats is to represent the flows on or around arbitrary surfaces. They have been extended to represent two vector fields or to show the velocity and the change in the velocity.

7.1 Previous Work on Rendering Hair and Fur

Several authors have attempted to model and render hair and grass for image synthesis purposes. Kajiya and von Herzon [41] developed a model for rendering teddy bear fur. They developed and used anisotropic volume textures, which they called texels. Watanabe [93] describes techniques to render hair using connected prisms. They also describe physically based models for the bending and shaping of the hair. Reeves' [77, 78] particle systems have been used to generate a variety of fuzzy phenomena such as grass, trees and fire.

7.2 Hair Splats

Hair splats are similar to line bundles and are more volume oriented than surface oriented. How they are applied to different visualization tasks can vary greatly, as will be discussed later. For each splat data point, a collection of curves is constructed and rendered into the image. Each curve is constructed as a set of line segments. Two vector fields are used to define the orientation of each line segment within a curve. A typical application would be to specify a surface normal or gradient as the direction that the curve should start with and a velocity field as the direction that the curve should end with.

The C++ Inventor class for Hair Splats, SoSplatHairs, is derived from the splatting class SoOctreeSplat. The C++ class allows a very flexible, if not cumbersome, specification of the curves based on these two vector fields. The number of line segments used to construct the curves can be specified by the user. Then for each of these segments, a "stiffness" factor and a "percent normal" factor are specified. The direction of each segment is calculated as a linear interpolation from the "normal" direction to the "vector field" direction using the "percent normal" as an interpolation factor. The equation for this is represented as C++ code:

```
direction = PercentNorm[ns] * unitgrad
          + (1.0-PercentNorm[ns]) * vecfield;
```

Direction, unitgrad and *vecfield* are all 3-component vectors represented using the Open Inventor class, *SbVec3f*. The multiplication and addition are performed on a component-wise basis. *PercentNorm* is the user specified amount that the direction should follow in the "normal" direction, with *ns* selecting the current line segment within the curve. The *unitgrad* and *vecfield* vectors are the vector quantities specified at each splat point, normalized. *Unitgrad* is referred to as the "normal" direction, while *vecfield* is the vector field direction.

Each segment may have a different length according to the "stiffness" variable specified by the user. The curve for each hair starts at a random location within the bounding box of the splat. The next point on the curve is then determined by moving from the current location in the direction calculated above, *direction,* an amount specified by the segment's stiffness. The use of the two variables, *PercentNorm* and *Stiffness,* can lead to very abrupt bends or nice smooth curves. The "stiffness" variable, *Stiffness*[*ns*], is not constrained and may have values greater than one. This is useful for the last few segments of hairs that are oriented in the direction of the flow. By specifying large values for the stiffness, along with a *PercentNorm* value of zero, long wispy hairs can be constructed in the direction of the vector field.

Several of these curves are generated for each splat data point. Each starting location is randomly placed (or jittered) around the data point by an amount specified by the user. The amount of jittering can by specified individually for the x, y and z coordinates. The user can also specify the number of hairs to be drawn for each splat. Like the line bundles, several hairs are needed to form a continuous texture. Figure 7-1 represents the test tornado data set with a single hair rendered at each splat data point. The hairs are made to look as if growing from the isocontour surface of the tornado magnitude as explained in Section 7.3. Figure 7-2 renders several hairs per splat point, producing a smooth texture across the surface.

Several colors for each hair can be specified by the user. The color of each splat data point is specified individually. Along with these colors, a global root color and vector head color are specified. For each vertex, the amount of the root color, splat color and vector head color is then specified. All colors are assumed to be in HSV space. For each vertex a jittering amount for hue, saturation and value is also specified. Similarly, a transparency is specified at each vertex. The color for each vertex of the curve defining each hair is then the sum of the these colors weighted by their specified amounts:

```
HSVColor[ns] = RootHSVColor.getValue() * RootBlend[ns]
             + VectorColor.getValue() * VecBlend[ns]
             + SplatColor   * SplatBlend[ns]
             + RandJitter[ns];
```

The final colors are clamped in the saturation and value components. The hue component is allowed to wrap around the color wheel.

7.3 Hairy Surfaces

One of the goals of the Hair Splats is to represent the relationship between a flow and a scalar variable by showing the flow impinging on a contour surface of the scalar variable. The "normal" direction is specified by the gradient of the scalar field. By selecting only the data points close to a constant value of the scalar field and drawing the contour surface associated with this value, hairs around the surface will be drawn. If the selected data points lie far enough from the contour surface, they will appear to float in space, separated from the surface. To alleviate this, the roots of the hairs can be pushed back into the surface using the gradient of the scalar field along with the scalar field and contour values. The initial starting point of the hair is moved along the negative gradient direction to push it into the surface by a user specified amount. Selecting data points close to a contour surface requires the difference between the scalar field at the starting point of the hair and the contour value. The method of selection is similar to Levoy's shading model [53] for volume rendering. The difference between the contour value, *c*, and the scalar field interpolated to the random starting point — which specifies how far the point is away from the surface — requires that the scalar field also be specified for the hair splats. In particular, it requires that the scalar field be known at enough data points for interpolation. The selection criteria is then:

$$\text{if } \frac{|f - \vec{x} \cdot \quad f - c|}{\| \quad f \|} < width \text{ then draw the splat,}$$

where *f* is the data point value associated with the splat point, *x*, and *c* is the contour value.

Figures 7-1 and 7-2 have a contour surface of the velocity magnitude of the tornado data set, with hairs "growing" from the surface and then bending to follow the velocity direction. The scalar field in thus the velocity magnitude and the gradient of this is used for the normal direction. Figure 7-2 has several hairs per splat with a long vector field direction segment.

7.4 Time Evolving Glyphs

By setting the "normal" vector field to the vector field of a time varying data set and the "vector" field to the next time step of the vector field, a smooth representation of the changing flows can be seen. In addition to the visualization of the flow field at a single time step, the bending of the hairs, or lack of bending, can represent the change in velocity direction from one time step to another. In areas where the flow direction does not change rapidly in time, straight hairs will produce a smooth texture similar to the line bundles. In areas where the flow direction substantially changes in time, the hairs will produce bent glyphs resulting in a different texture from the areas of constant directionality.

Care must be taken to avoid destroying the textures representing the directional texture for the current time step. Figure 7-4 represents the wind velocity over the Western Pacific at two different time steps eight hours apart. The splat color was set to the velocity magnitude of the winds in the first time step. The root color was set to black and the vector color was set to a unsaturated pink. Thus, the hairs start out black and in the direction of the winds at the first time step, gradually bend (if winds change) using the splat color and end up in the direction of the winds in the second time step, colored as a light pink. One hair per data point (selected based on the magnitude of the wind velocity at the first time step) was used with completely opaque hairs. Figure 7-5 is the same data sets, but with eighteen hairs per data point which are fairly transparent.

The use of HSV color space leads to some problems when we want the ends of the hairs to either smoothly desaturate (go to white) or to smoothly go to black. The pink vector heads in Figure 7-4 became pink since the specification of white in HSV space still requires a value for the hue. This constant hue is weighted into the calculations described above to change the hue of the hair as it goes from the root to the splat hue to the head. For white and black endpoints, RGB color space calculations or a method of specifying no hue is needed. The HSV space is still useful for jittering the splat color mainly in the hue component.

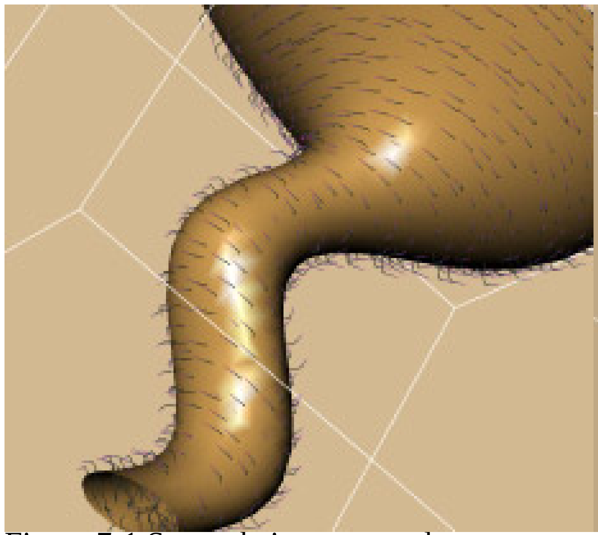
Figure 7-1 Sparse hairs on tornado contour surface.

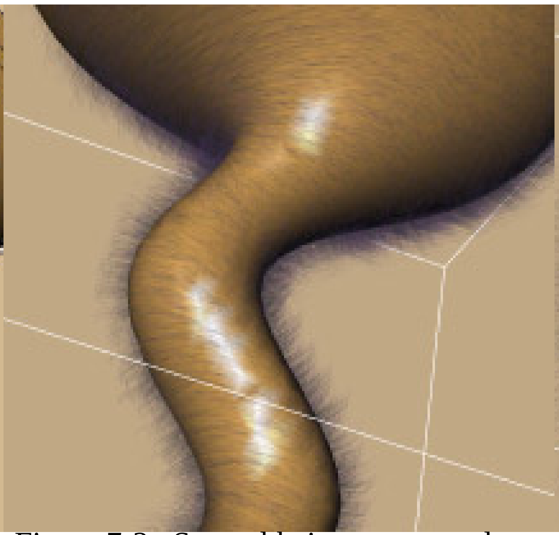
Figure 7-2 Several hairs near tornado surface.

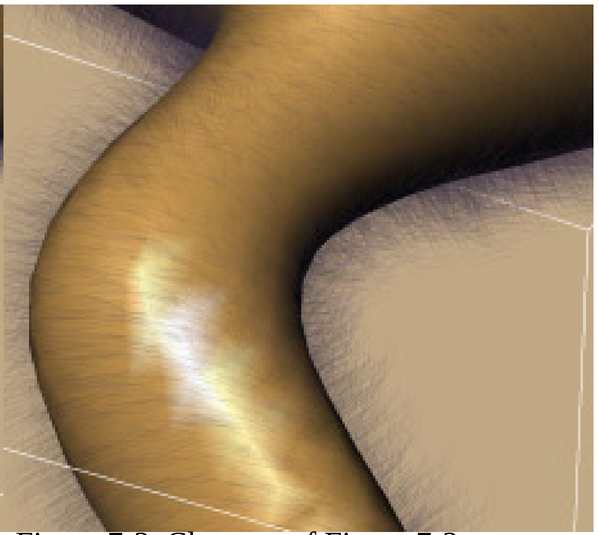
Figure 7-3 Close-up of Figure 7-2.

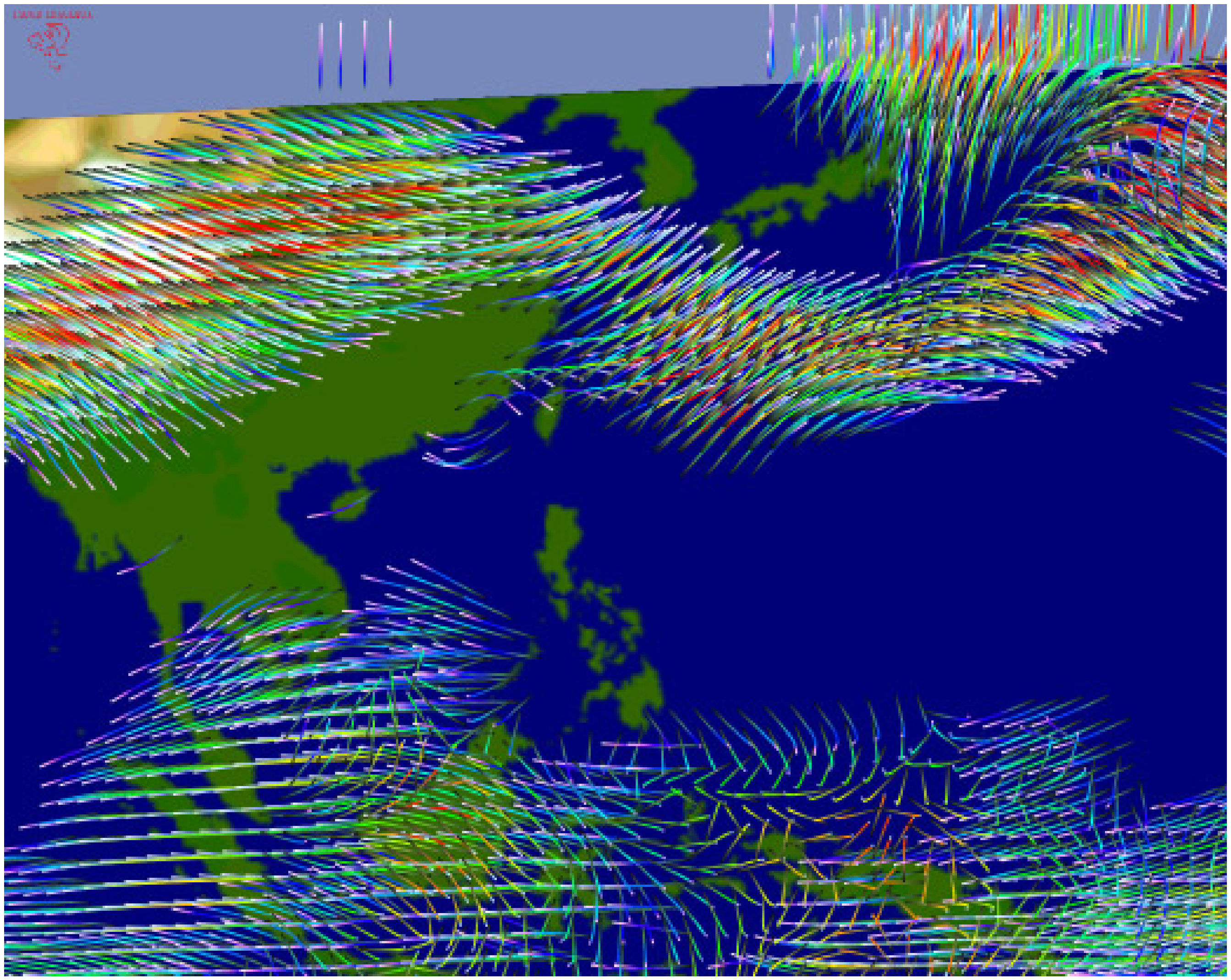
Figure 7-4 Hairs used to represent two time steps of the winds over Indonesia.

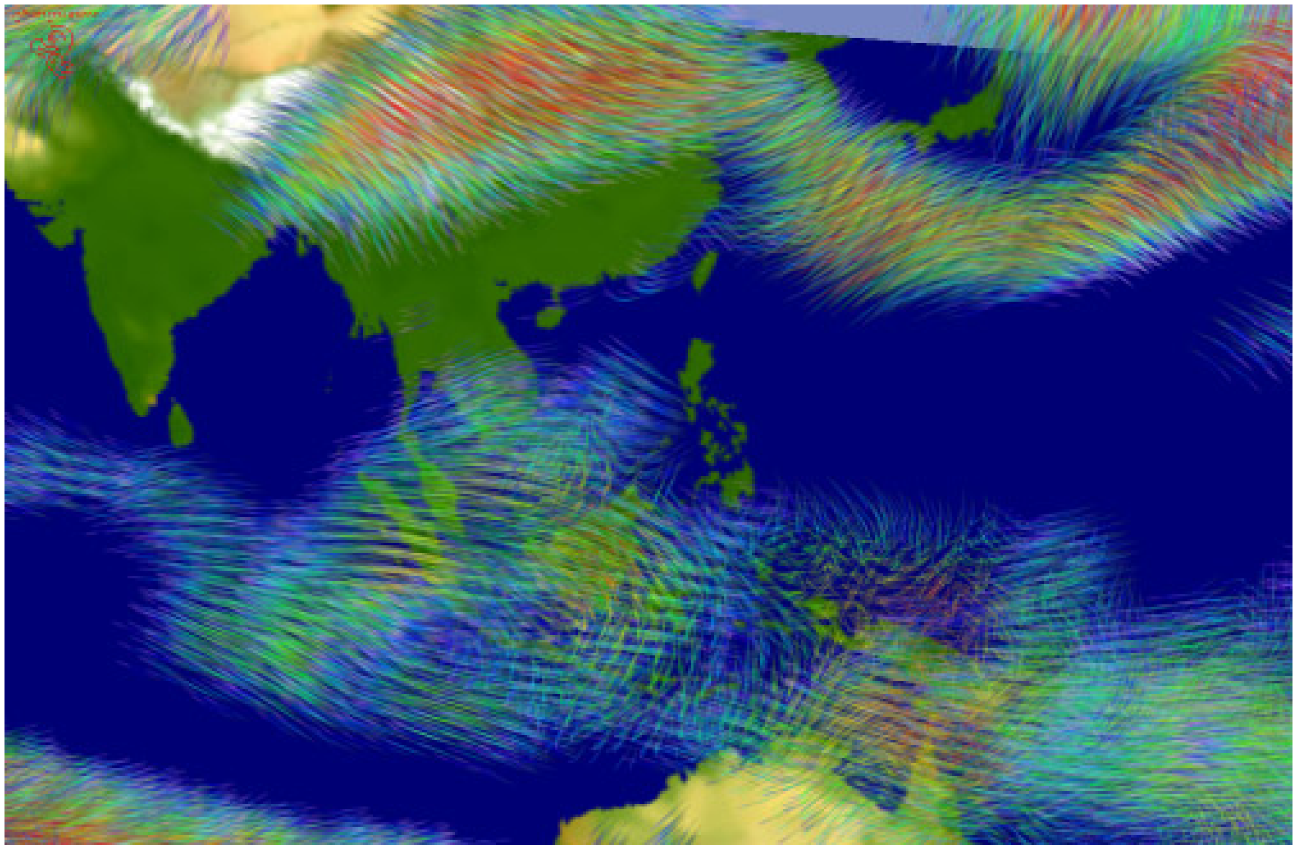

Figure 7-5 Several semi-transparent hairs representing the wind over Indonesia at two time steps.

7.6 Source Code

7.6.1 SoSplatHairs.h

```
/*******************************************************************************
//  ______________________________________________________________________
//  ___________                                             __________________
//  ___________      L A W R E N C E   L I V E R M O R E     __________________
//  ___________     N A T I O N A L   L A B O R A T O R Y    __________________
//  ______________________________________________________________________
//
//
//    NAME
//      SoSplatHairs - Splatting of curves constructed from two vector
//                     fields. Can be used to construct hairy surfaces.
//
//
//    VERSION
//   @(#)SoSplatHairs.h version 1.5  created on 95/06/06  at 13:03:31
//
//
//    NOTES
//   1) See the man pages or the implementation prologue for more info.
//
//
*******************************************************************************/
#include "LLNL_Disclaimer.h"


#ifndef  _SO_OSPLATHAIR_
#define  _SO_OSPLATHAIR_

#include <Inventor/fields/SoMFUShort.h>
#include <Inventor/fields/SoSFColor.h>
#include <Inventor/fields/SoSFVec3f.h>
#include <Inventor/fields/SoMFVec3f.h>
#include "SoOctreeSplat.h"

static const int MAXSEGMENTS = 100;
static const int MAXLINES = 300;


class SoSplatHairs : public SoOctreeSplat {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatHairs);

  public:

    // Fields
    SoSFFloat   MinVecLength;
    SoSFFloat   MaxVecLength;
    SoSFShort   nLines;
    SoSFShort   nSegments;
    SoMFFloat      Stiffness;
    SoMFFloat      PercentNorm;
    SoSFFloat   VecScale;
    SoSFVec3f      VecJitter;
    SoSFColor      RootHSVColor;
    SoSFColor      VectorColor;
    SoMFFloat   hairTrans;
    SoMFVec3f      HSVJitter;
    SoMFFloat      RootBlend;
    SoMFFloat      VecBlend;
    SoMFFloat      SplatBlend;
    SoSFFloat   Contour;
    SoSFFloat   Width;

    // Constructor
    SoSplatHairs();

    // Destructor
    virtual ~SoSplatHairs();

  SoINTERNAL public:
    static void           initClass();

  protected:
    // Implements actions
    virtual void  renderSplat( int index, float x, float y, float z, float ss );
    virtual void  initGLRender( SoGLRenderAction *);

  private:
    SbVec3f point[MAXLINES];
};
#endif /* _SO_SPLAT_HAIR_ */
```

Chapter 8

Multivariate Volume Rendering

8.1 Background on Multivariate Representations

Much research has been explored on representing multi-dimensional data such as that resulting from census data [44]. The emphasis with these previous techniques has concentrated on the analysis of scattered data with many independent variables. However, few techniques have been explored for representing several continuous fields defined on data grids. A common approach for two-dimensional data sets is to color-code one variable using a hue-ramp and another using a saturation or value-ramp [44]. The resulting color is then defined by taking the hue from the first variable-mapping and the saturation (or value) from the second while holding the value (or saturation) constant. Similar schemes have been explored using a separate red-ramp, green-ramp and blue-ramp for three different variables. For two-dimensional data sets, texture mapping techniques can be applied to a three-dimensional surface plot of a variable. Crawfis [17] illustrates several techniques for representing 2D multivariate data using texture mapping.

The algorithms described in this section attempt to represent several variables defined on a regular three-dimensional mesh. Volume rendering is extended to yield noisy-looking density clouds. The amount of noise within a cloud is controlled to represent a separate variable from the variable defining the density cloud. This approach is a new technique which we have called **Noise Splats**.

8.2 Adding Noise to the reconstruction texture

The basic approach we take is to extend the Textured Splats technique described in Chapter 6. Rather than use an anisotropic texture embedded within the reconstruction texture, we embed some noise or dots within the texture. By defining a table of textures with increasing number of dots, we can control the amount of noise within the volume by using a second independent variable. Several different algorithms for generating noise are described in the sections below.

8.2.1 Modulate texture with white noise

What we would like is a reconstruction function, that rather than being smooth, contains a substantial amount of high-frequency noise, while still preserving the overall kernel shape. Stated differently, we require the mean of the reconstruction function of the noisy splats to be equal to the reconstruction function of Chapter 6 within a small neighborhood. By adding white noise with a mean of zero and amplitude $\omega * h(r)$ to the reconstruction function scaled by $(1-\omega)$, we produce the desired result. This produces a noisy reconstruction function similar to that shown in Figure 8-1. By varying the amplitude ω, different amounts of noise can be embedded into the texture.

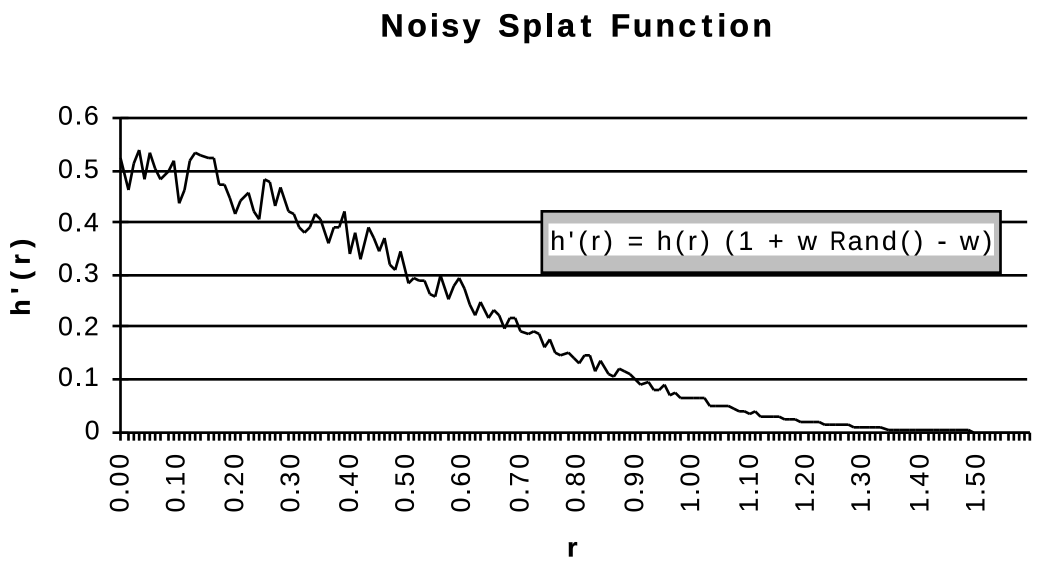

Figure 8-1 Noisy Splat Function

The resulting splats, called Noise Splats, need to maintain the overall property of the reconstruction function. Figure 8-2 shows a series of splats with increasing noise amplitudes. Since the noise has a mean of zero, the overall intensity is preserved across the splats. Reconstruction of two constant functions with these noisy splats should still produce a volume rendering with a constant local-mean intensity and a constant amount of variance. Figure 8-3 represents a single layer of splats with a constant amplitude appropriately composited together. The image is smooth, but noisy, yielding a faithful representation for the volume rendering and allowing the noise to represent another variable.

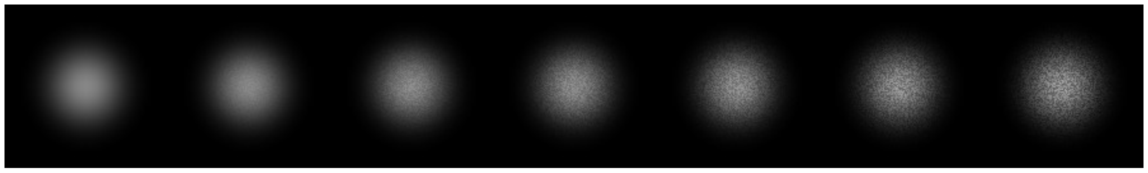

Figure 8-2 Noise Splats with varying amplitude in the noise.

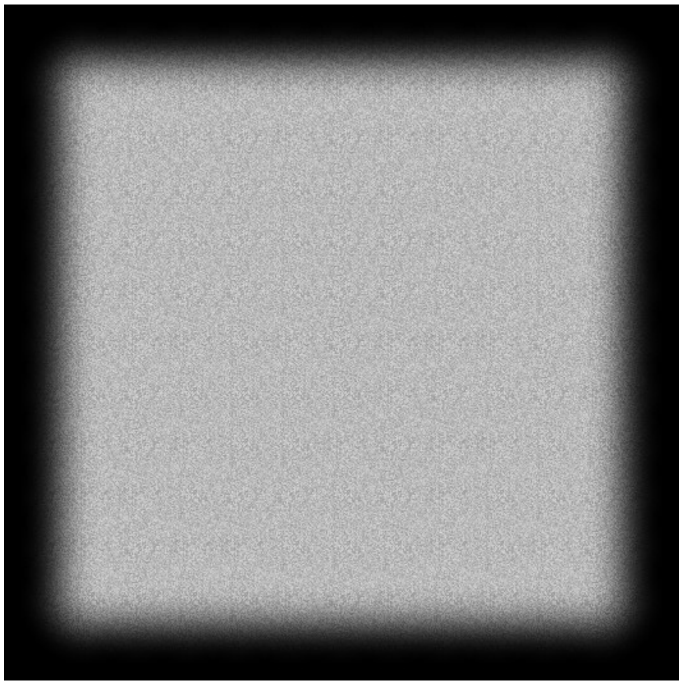
Figure 8-3 A single layer of Noise Splats composited together

The Noise Splats are rendered in a similar fashion to the Texture Splats described in Chapter 6 for representing only a vector field. The modulate blend function is used within the OpenGL [6, 7] graphics system. For each splat point, a splat color and an amount of noise is specified. The amount of noise varies between zero and one. A splat is selected from a precomputed table of splats with increasing noise amplitude, using the amount of noise at the splat point as a selector. A value of zero will select a splat with no noise, while a value of one will select the splat in the table with the highest noise amplitude. The maximum amplitude of the splats within the table can be specified by the user as part of the class specification. The size of the texture table (how many splats there are) can also be specified by the user. The textures are computed on the fly the first time the data is rendered and anytime the data or the class settings (the maximum amplitude or the number of textures) are changed.

Figure 8-4 illustrates a volume rendering of the tornado data set using the Noise Splats. The x-velocity component is mapped to the noise amplitude, with a larger positive value increasing the strength of the noise. Figure 8-5 has two functions being represented, one a gravitation test function and the other a summation of sine functions. The gravity function is represented by the colored semi-transparent volume density cloud, while the sinusoidal function is represented by varying the amplitutde of the noise.

8.2.2 Blend texture with random points

The modulated noise described above leads to very subtle texture differences in the image, making it difficult to discern the changing functional values. Another approach is to place small dots within the texture of a different color than the volume. The OpenGL [7] blend operation allows a texture to specify the varying opacity needed for the reconstruction function along with an intensity mask to specify either the polygon color or a specified color, or an amount to blend between the two. Thus we can add dots to the intensity component of the texture maps to select a constant user specified color rather than the splat color. A set of splats were created with an increasing number of points or dots in the texture. Figure 8-6 shows such a series of splats without the reconstruction function. This is exactly what the intensity mask would look like in the OpenGL texture map. The opacity component of the texture would hold the reconstruction function as in the Textured Splats.

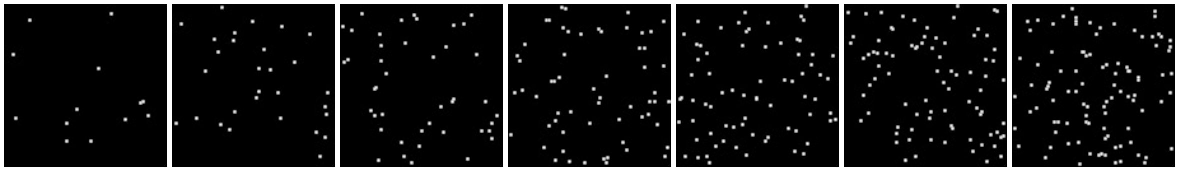
Figure 8-6 Series of random point distributions with an increasing number of points

Figure 8-7 shows a composite image of constant data points using a single splat. Regular patterns are clearly evident in the composited image. Since the points are random, even worse patterns are possible and have been noticeable. One solution to avoiding these patterns is to create a set of splats with the same number of dots and randomly select which splat to use. Figure 8-8 shows the same compositing as in Figure 8-7, but using a set of three different splats all with the same number of random points. This greatly reduces the regular patterns noticeable in Figure 8-7. Unfortunately, there is no guarantee that the points in one or several of these splats will not bunch together, leading to serious artifacts in the imagery. For this reason, several near optimal point distributions will be examined in the next few sections.

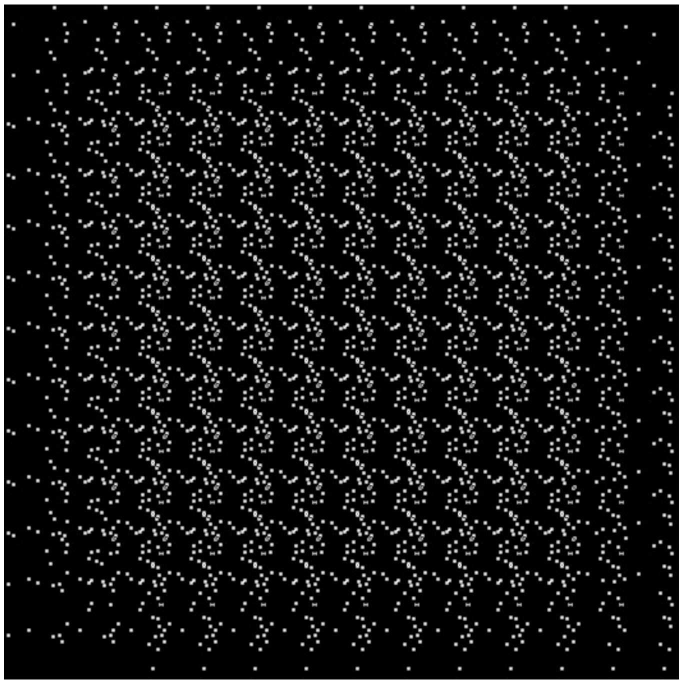

Figure 8-7 Composite of splats using a single splat with random points.

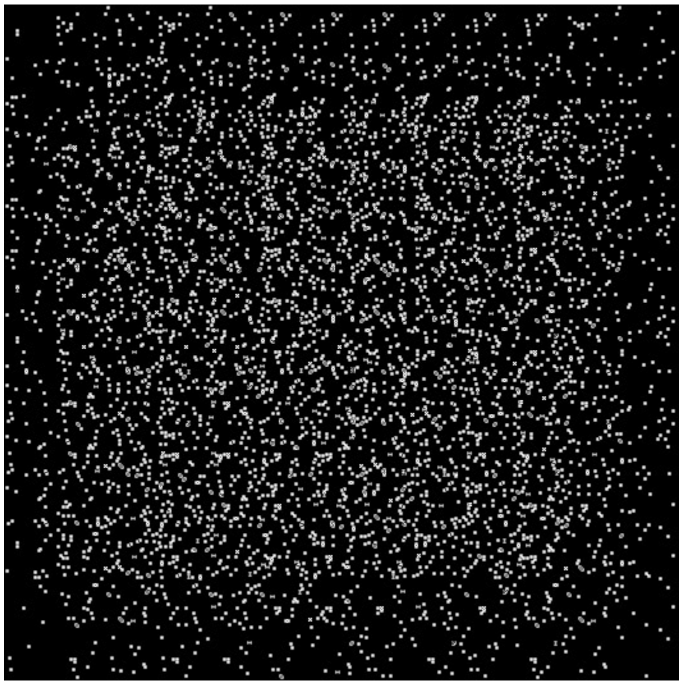

Figure 8-8 Three splats with random points composited together.

Figure 8-9 illustrates white dots within a volume rendering of the tornado data set. The x-velocity component is mapped to the number of dots, hence more dots are apparent when this component is a high positive number. Figure 8-10 has two functions being represented, one a gravitation test function and the other a summation of sine functions. The gravity function is represented by the colored semi-transparent volume density cloud, while the sinusoidal function is represented using a varying dot density.

8.2.3 Poisson disc distribution

With individual points, we need a uniformly random looking pattern to avoid false patterns, but also need the frequency or density of points to be meaningful. A strictly random or pseudo-random set of points will not produce this. A first experiment at producing random pattern tried this and while somewhat successful, led to cases where the data could not be interpreted accurately. A Poisson disc distribution insures that the data points do not clump together. This can be accomplished by requiring the distance between a point and all previous points to be greater than some threshold. The threshold gradually gets smaller, starting with a wide distribution of points and then gradually increasing the point density.

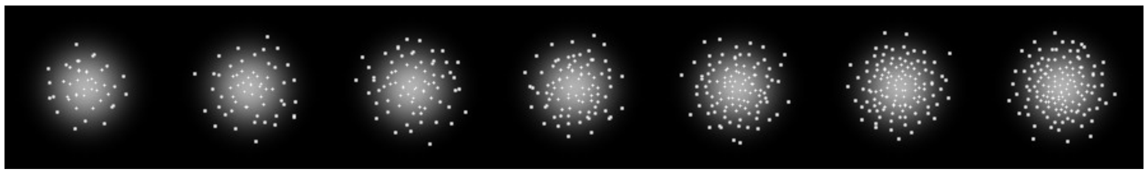
Figure 8-11 Weighted Poisson point distribution embedded in the reconstruction function

For the purposes of splatting, we need the resulting image from several composited splats to produce a density of points that is meaningful. Much like the reconstruction functions for textured splats sum up close to a constant for areas with a constant functional value, the noise distribution needs to be fairly uniform in areas with a constant functional value. Unless the overlap is such that the same number of splats overlap everywhere (i.e., the splat size is a whole multiple of one half), the Poisson disc distribution will not produce the proper distribution when the splats are composited. To overcome this, a weighted Poisson disc distribution has been developed. The basic premise is to have a higher point density near the center of the splat, that gradually fades to zero. The summation of the expected densities from splat points composited on top of each other should then be fairly constant. Since the reconstruction function sums to one, the same function can be used in the weighting. The threshold in the Poisson disc distribution calculation is changed to be inversely proportional to the reconstruction function. Hence, the probability of adding points near the center is greater than at the edges of the splat texture. A series of splats with an increasing number of points is represented in Figure 8-11. The resulting composite of splats with the same number of points is represented in Figure 8-12. Three splats are calculated for each number of points to avoid regular patterns. Figure 8-13 shows the composited splats without the reconstruction function.

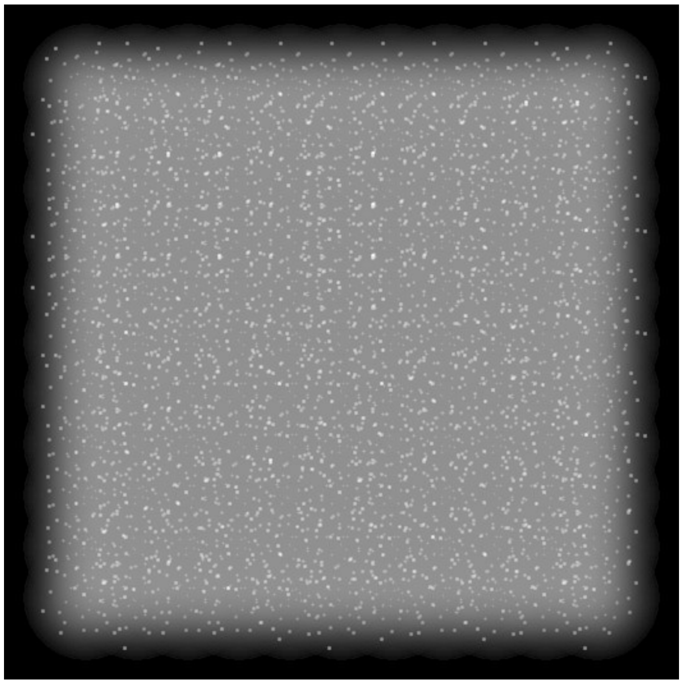

Figure 8-12 The Poisson splats composited together.

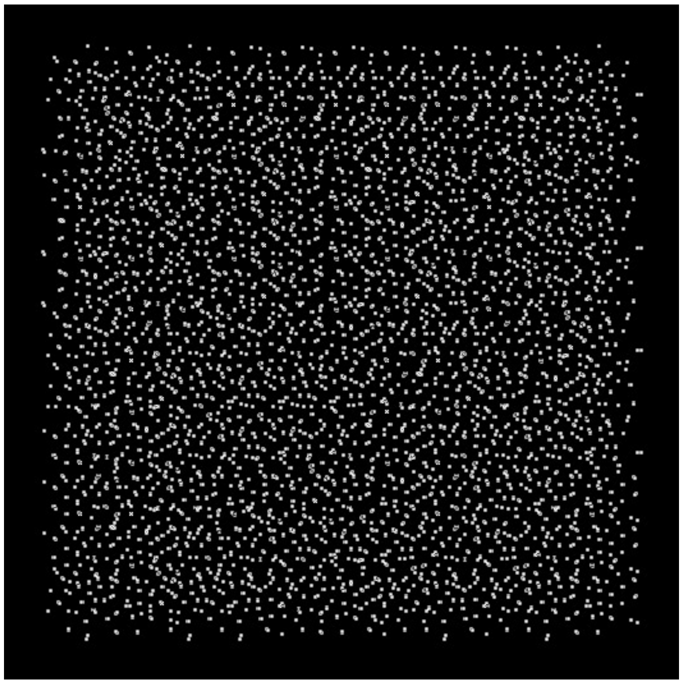

Figure 8-13 The Poisson splats composited together showing just the points.

While this technique generates near optimal distributions of the points, it is computationally expensive. The next two sections describe alternative schemes for generating point distributions such that the point density matches the reconstruction function.

8.2.4 Jittered Dithering Distributions

Mitchell [69] presented an algorithm for quickly generating a uniform distribution based on the Floyd-Stienberg dithering algorithm. His algorithm used a regular 2D grid with sixteen times as many grid cells as the desired number of points. A candidate data point is considered for each grid cell. The data point is determined by randomly jittering around the center of the cell. Each cell is then visited in turn in a left-to-right and top-to-bottom fashion. A *potential* is calculated using the function:

$$D_{i,j} = (4D_{i-1,j} + D_{i-1,j-1} + 2D_{i,j-1} + D_{i+1,j-1}) / 8 \ + \ R$$

where,

$$R = \frac{1}{16} \pm \frac{1}{64}\, random() \, .$$

If $D_{i,j}$ is greater than the threshold of one-half, the point at cell (i,j) is selected and the $D_{i,j}$ potential is reduced by one. Thus, about 1/16th of the data points are selected.

To generate a weighted point density, two possible modifications to this algorithm were explored. The first was to adjust the threshold from a constant value of one-half to a value inversely proportional to the reconstruction function. Thus, it takes a higher potential to generate a point away from the center of the splat. A series of the point distributions calculated using this technique is illustrated in Figure 8-14. The composite of one of these over a 10 by 10 set of grid points is represented in Figure 8-15. Regular patterns are clearly visible in the splats and the resulting composited image.

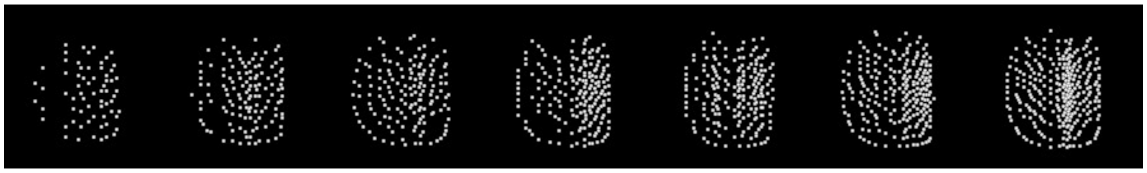

Figure 8-14 Weighted Jittered Dithering Point Distributions

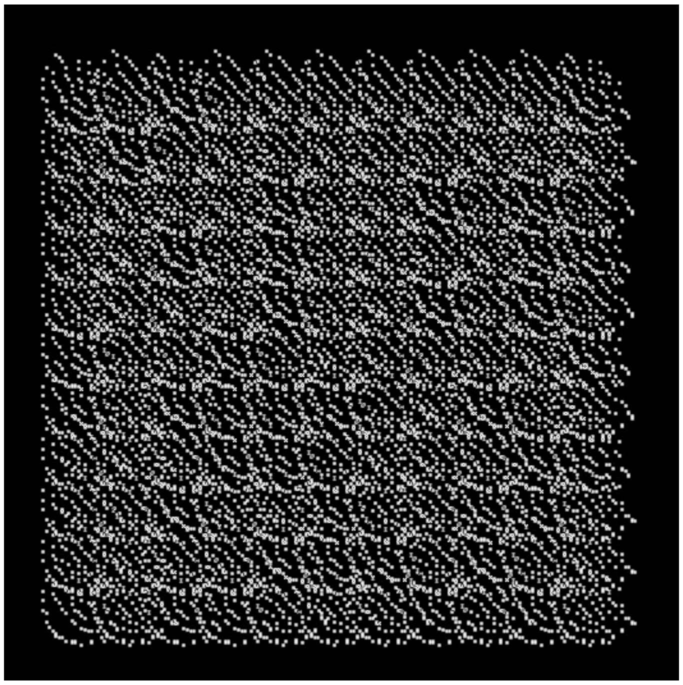

Figure 8-15 The Jittered Point Splats (changing the threshold) composited together

The second approach to modifying the uniform dithering technique to produce weighted point density distributions is to keep the threshold constant, but change the incrementally added potential *R* to be proportional to the reconstruction function. The series of splats resulting from this approach is represented in Figure 8-16 and the composited image in Figure 8-17.

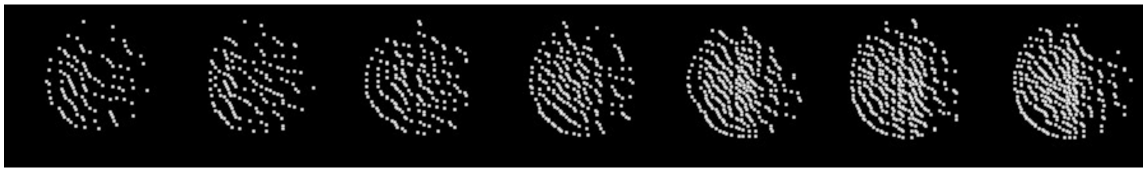

Figure 8-16 The Jittered Point Splats based on *R*

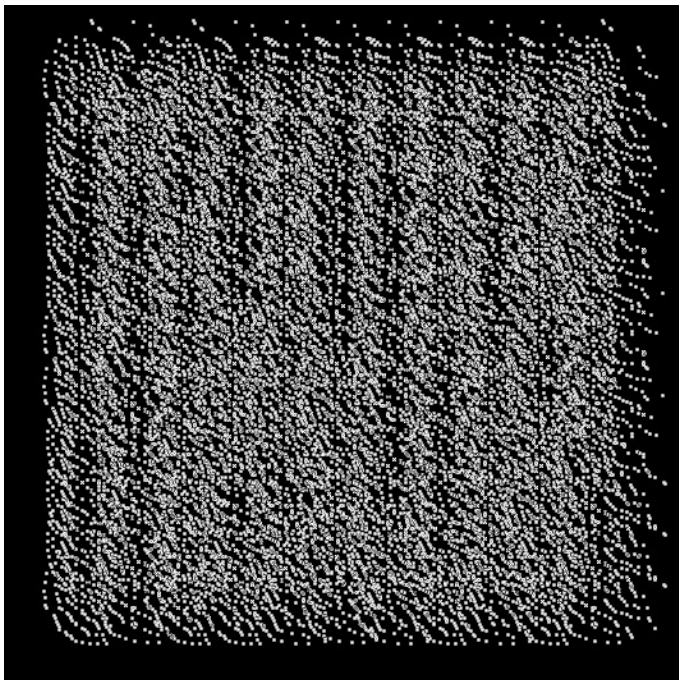

Figure 8-17 The Jittered Point Splats (changing *R*) composited together

The utility of this technique for generating weighted point distributions is further hindered in the lack of precise control over the number of points deposited into the texture. The actually number of points deposited in the splats has differed greatly from the expected number of points.

8.2.5 Stretched Uniform Distributions

Cook [16, 40] presented a technique to generate weighted point densities in one-dimension. Since the reconstruction function is radially symmetric, we can generate a uniform point denisty and pull the points radially into the center. The function that "pulls" the points into the center is proportional to the reconstruction function. To determine this, recall that we want a point density to be proportional to the reconstruction function. Hence, we have

$h(r)$ = point density

and

$h\, da$ = expected number of points in a unit area.

If we then look at an infinitesimal ring around the center of the splat, where the reconstruction function has a value of *h,* then the expected number of dots within that ring is:

$h\, dring$ = expected number of points in infinitesimal ring.

Now, the area of the ring is $2\pi r dr$, and hence the number of points per infinitesimal ring is:

$h\, 2\pi r\, dr$ = expected number of points in infinitesimal ring.

The number of points expected within an infinitesimal ring for a uniform point distribution would be:

$k\, 2\pi\rho\, d\rho$ = expected number of points in infinitesimal ring in "flat land".

Since the goal is to map a ring in "flat land" to the raised surface of the reconstruction function, we want the point densities of these two infinitesimal rings to be equal:

$$k\, 2\pi\rho\, d\rho = h\, 2\pi r\, dr.$$

Integrating both sides, yields:

$$\int_0^{\rho_0} k\rho d\rho = \int_0^{r_0} h(r) r dr,$$

$$k\frac{\rho_0^2}{2} = \int_0^{r_0} h(r) r dr.$$

Solving for ρ_0, leads to the mapping function

$$\rho_0 = f(r_0) = \sqrt{\frac{2}{k}\int_0^{r_0} h(r) r dr}.$$

The inverse of this function is needed to map from ρ to r. If the reconstruction function is a gaussian, then this can be solved analytically. Let

$$h(r) = e^{-\frac{r^2}{2}}.$$

Then

$$\int_0^{r_0} h(r) r\,dr = \int_0^{r_0} e^{-\frac{r^2}{2}} r\,dr$$

$$= 1 - e^{-\frac{r_0^2}{2}}$$

and we have

$$\rho = f(r) = \sqrt{\frac{2}{k}\left(1 - e^{-\frac{r^2}{2}}\right)}.$$

Solving for the inverse function yields:

$$r = f^{-1}(\rho) = \pm\sqrt{\ln\left(\frac{4}{4 - 4k\rho^2 + k^2\rho^4}\right)}.$$

We will always want the non-negative value of this mapping.

The algorithm is then to generate random points on the unit square, calculate the distance from the center of the square and using the distance as ρ in the mapping above to pull the point towards the center. Figure 8-18 illustrates a series of splats with an increasing number of points. The resulting composition from splats with the same number of points is shown in Figure 8-19. Three separate splats were randomly used in the compositing.

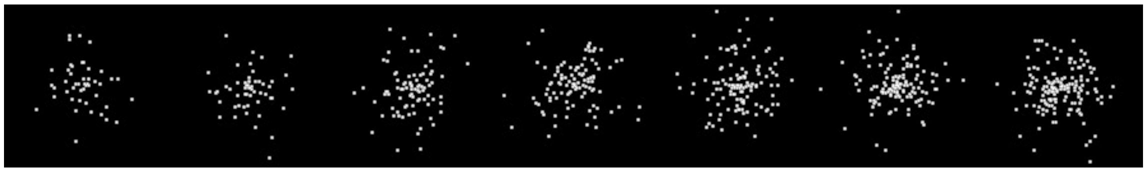

Figure 8-18 Stretched Point Splats with an increasing number of points.

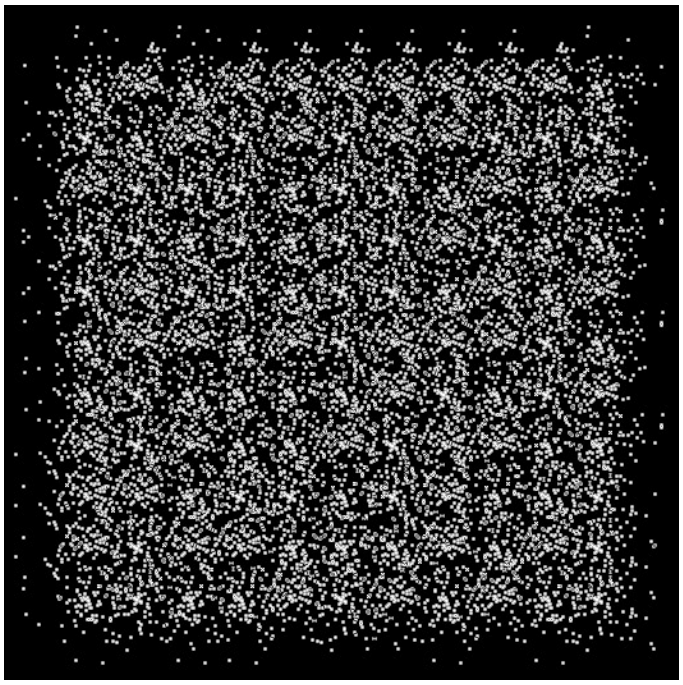

Figure 8-19 The Stretched Point Splats composited together.

Since random points were chosen rather than a Poisson disc distribution, some slight regular patterns are visible in the composited image. Starting with a more uniform distribution or using more than three splats in the compositing should alleviate some of this.

Figures 8-20 and 8-21 illustrate this technique on the tornado and gravity/sine functions.

8.3 Animating the Noise Splats

Calculating a few (3-5) splats with the same amount of noise is useful in avoiding regular patterns as shown by the composited images in the previous section. By cycling through this set of splats, the noise is animated, providing a stronger and dynamic representation of the variable mapped to the noise. A 2D table of splats is created with increasing amounts of noise down the columns. Each column is created independently using different sets of random numbers. By randomly selecting a column, the noise is animated. For small to mid-sized data sets that can be rendered interactively, this dynamic "sparkling" of the noise is also achieved interactively.

8.4 More than two independent variables

So far in this paper, we have only represented two variables with the noise splats. This is a very useful start for aiding the understanding of many complex simulations. It is also useful to be able to examine the relationships between three or more variables. The algorithms described in this chapter can be extended to include at least one more variable. The blending operations described above have all used a constant color for the noise. By varying the color of the noise from splat point to splat point a third variable can be represented. Figure 8-22 illustrates the tornado using the jittered point distribution as in Figure 8-21, but here the color of the points are controlled by the *z*-coordinate. A mapping from black to red to yellow to white is used as *x* goes from the bottom of the tornado to the top.

By combining the textures in Chapter 6 for representing scalar and vector field with the splat textures developed in this chapter, two independent scalar fields and one vector field can be represented. The vector icons in the texture must be of a different scale than the noise to avoid losing the vector icons into the noise. The same time dynamics from both the vector splats and the noise splats can be combined into the same textures. Thus a three-dimensional table of splats is needed if animation and vector foreshortening are desired. The axes of the table correspond to the vector foreshortening, the amount of noise and the time dynamics. Figure 8-23 illustrates a sample splat from this approach.

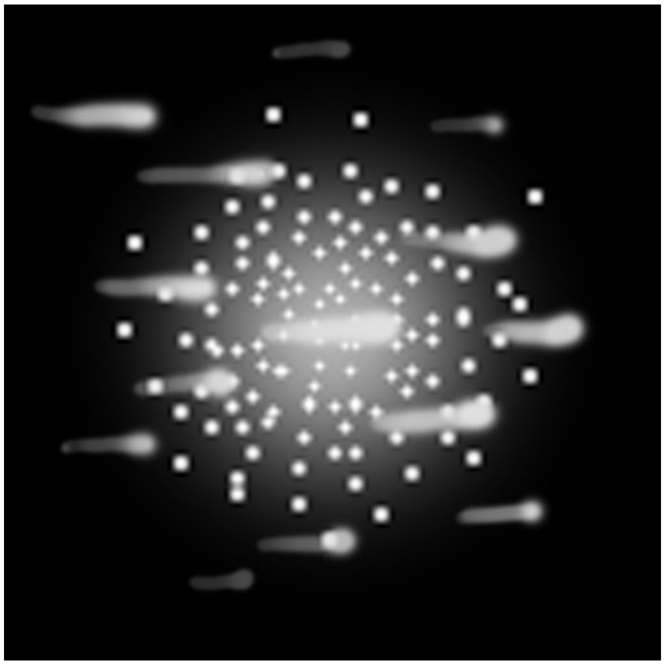

Figure 8-23 Combining the Textured Splats and the Noise Splats.

8.5 Color Images

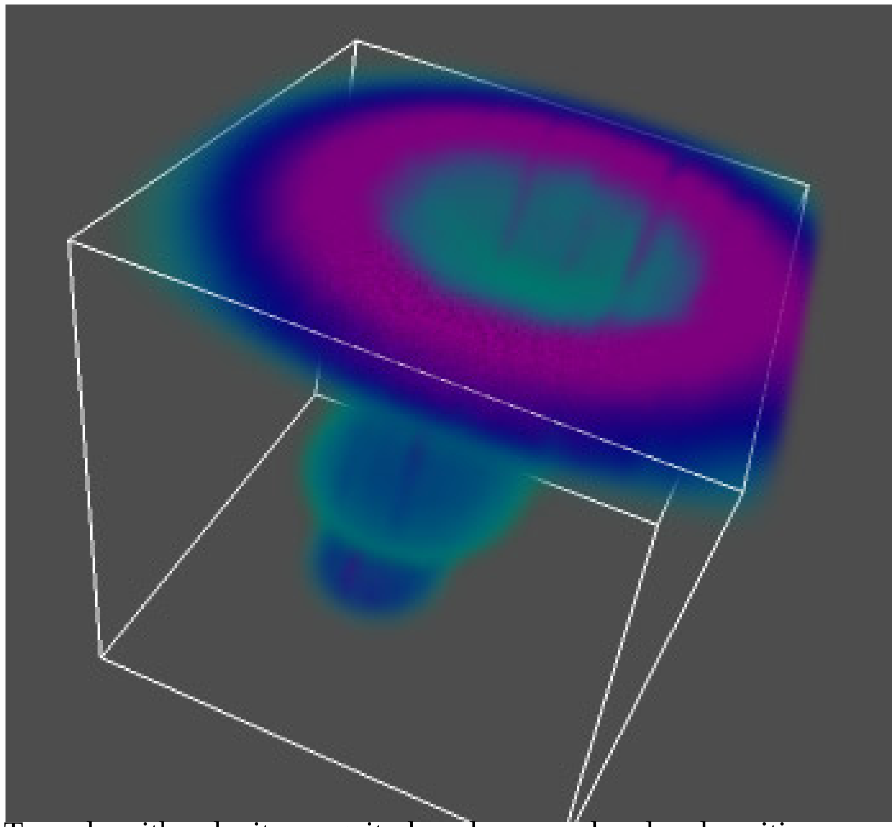

Figure 8-4 Tornado with velocity magnitude volume rendered and positive x-component shown using modulated noise

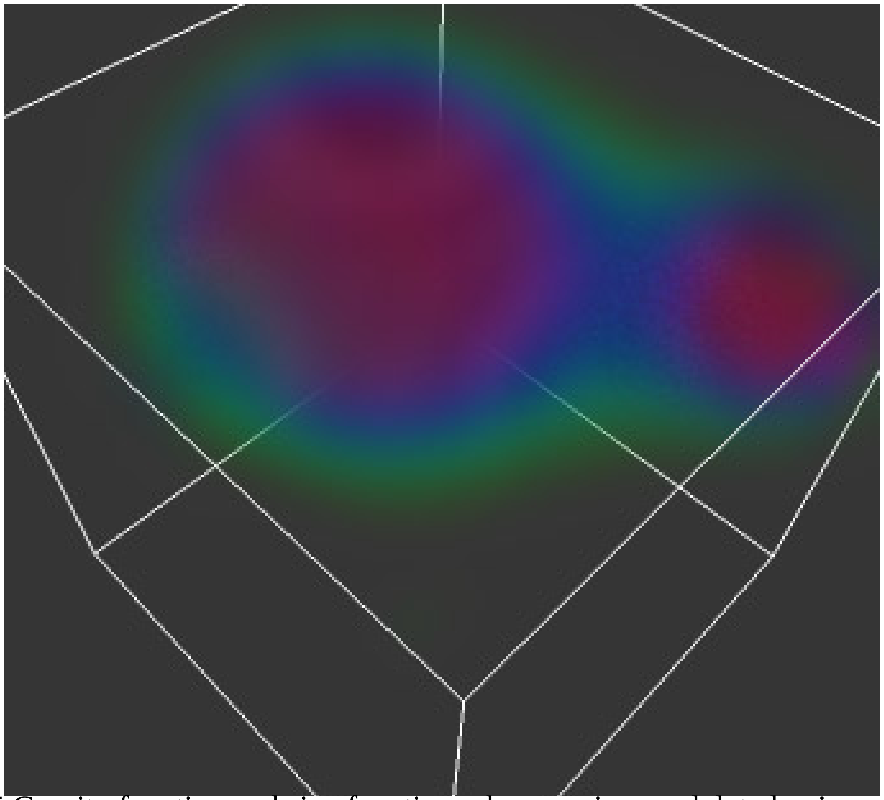

Figure 8-5 Gravity function and sine functions shown using modulated noise.

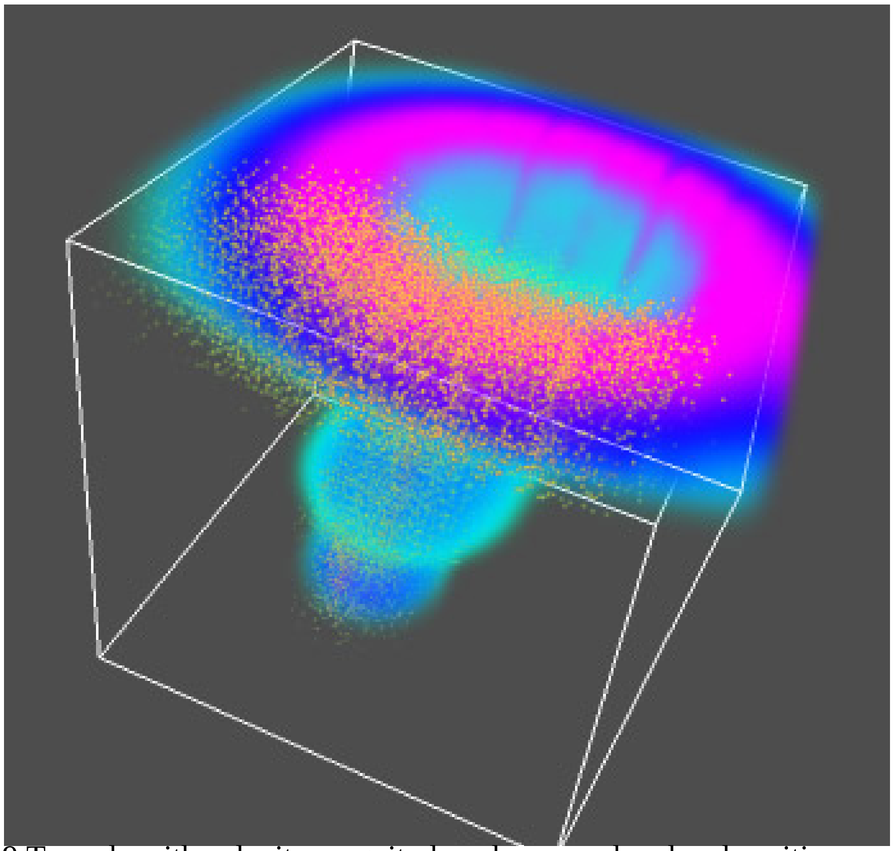
Figure 8-9 Tornado with velocity magnitude volume rendered and positive x-component shown using random point distributions

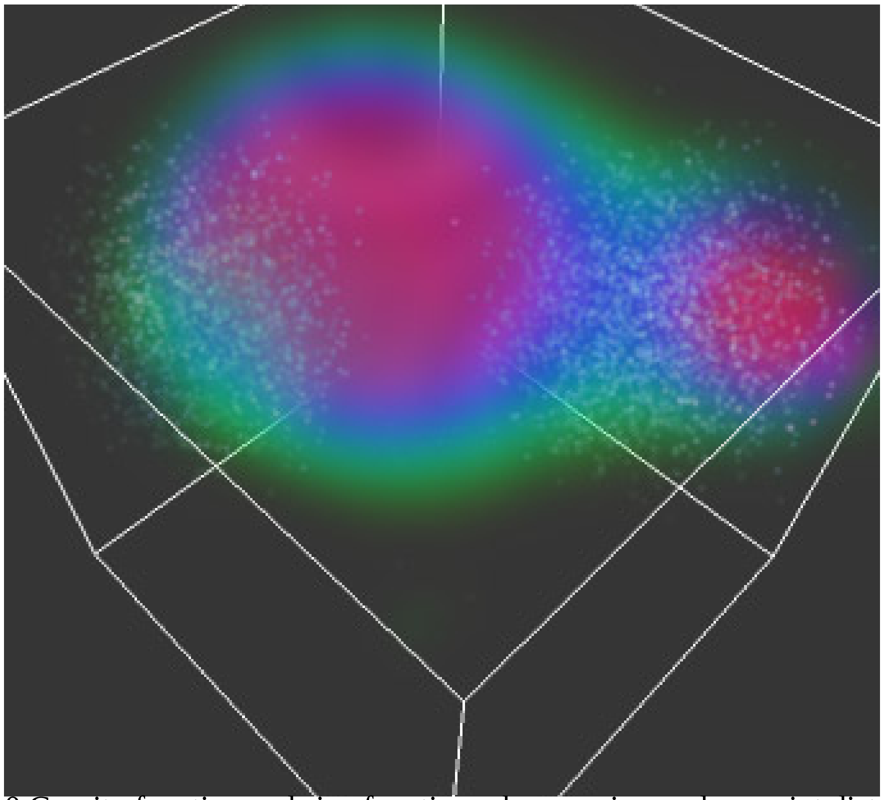
Figure 8-10 Gravity function and sine functions shown usingrandom point distributions.

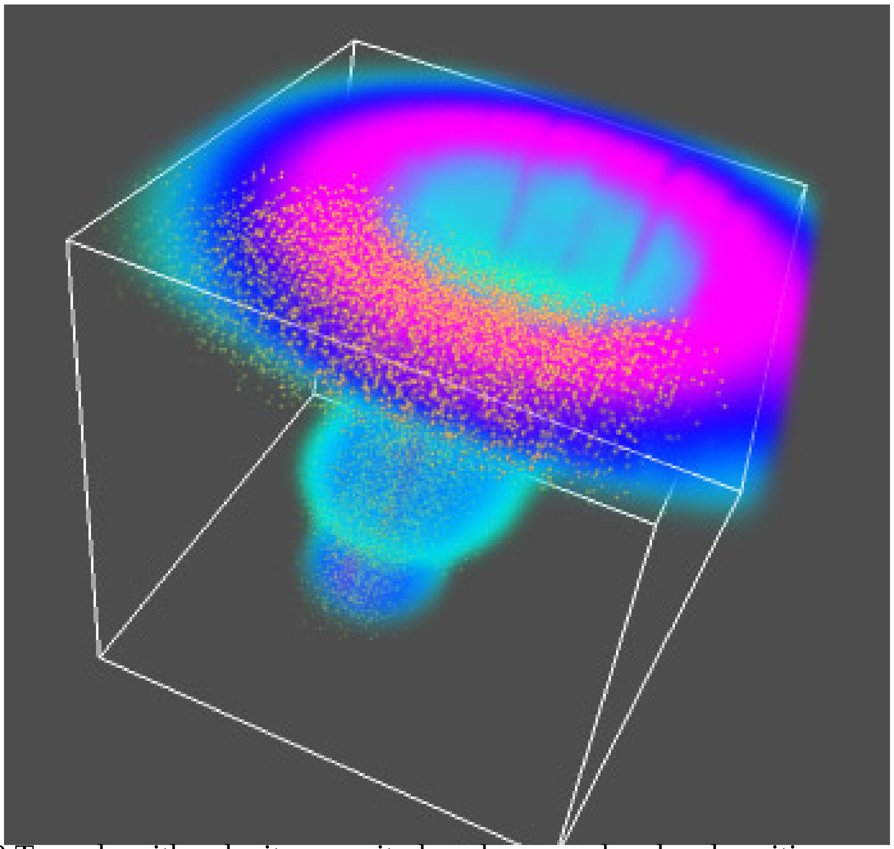
Figure 8-20 Tornado with velocity magnitude volume rendered and positive x-component shown using stretched point distributions

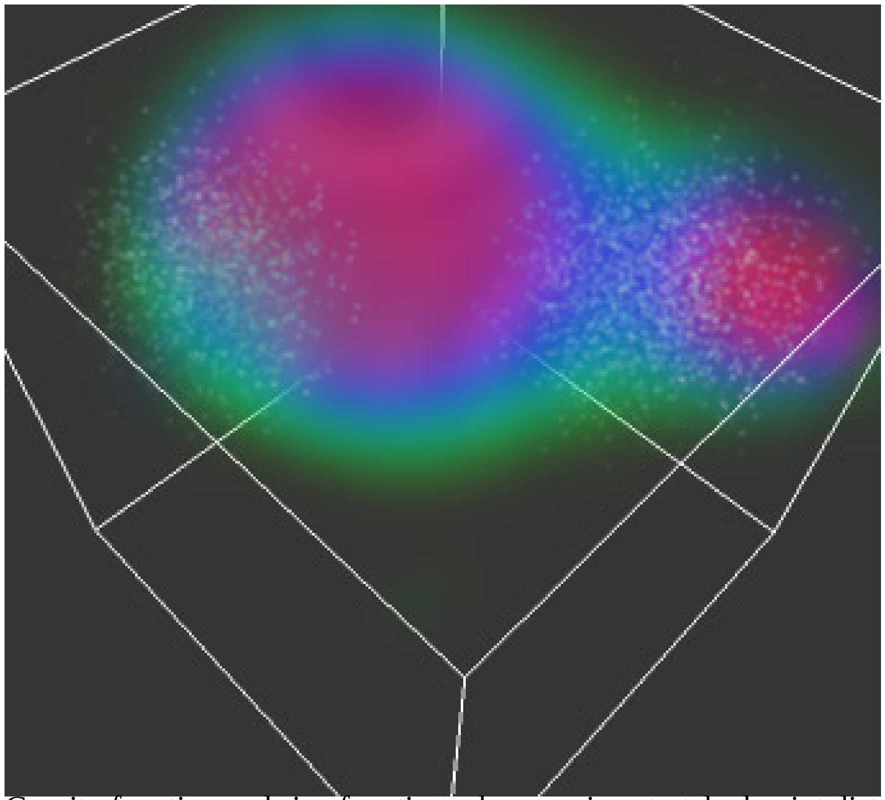
Figure 8-21 Gravity function and sine functions shown using stretched point distributions.

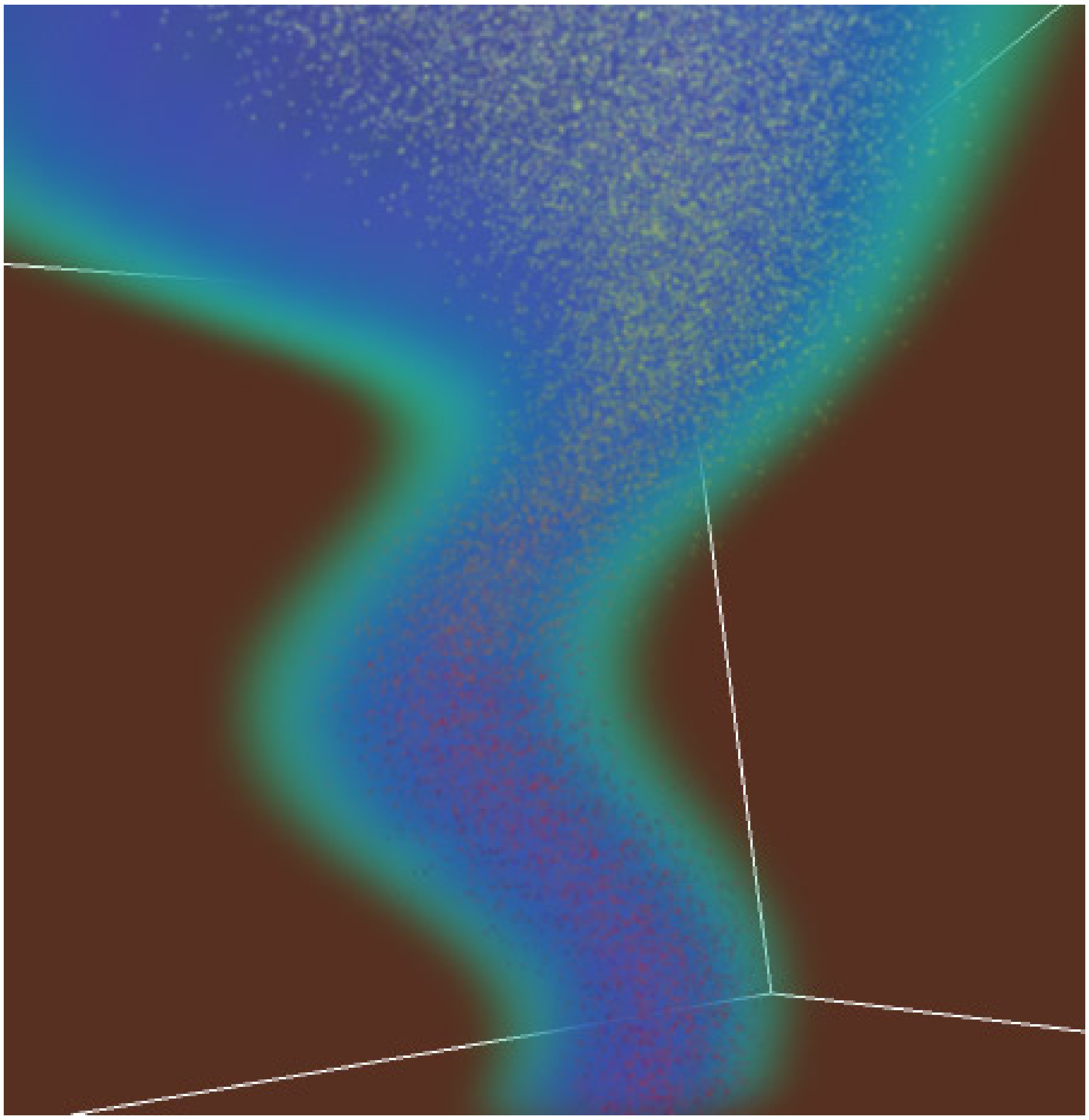
Figure 8-31 The tornado data set with increasing noise on the x-component. The noise is colored according to the z-coordinate.

8.6 Open Inventor C++ Class definitions

8.6.1 SoSplatNoise.h

```
/*****************************************************************************
//  ______________________________________________________________________
//  ___________                                         __________________
//  ___________    L A W R E N C E   L I V E R M O R E  __________________
//  ___________   N A T I O N A L   L A B O R A T O R Y __________________
//  ______________________________________________________________________
//
//
//    NAME
//      SoSplatNoise - Volume render of a vector field by splatting texture
//                     mapped squares. An anisotropic texture is used.
//
//
//    VERSION
//      @(#)SoSplatNoise.h version 1.2  created on 95/06/06  at 13:03:56
//
//
//    NOTES
//      1) See the man pages or the implementation prologue for more info.
//
//
*****************************************************************************/
#include "LLNL_Disclaimer.h"


#ifndef  _SO_OSplatNoise_
#define  _SO_OSplatNoise_

#include "SoSplatTexture.h"
#include <Inventor/fields/SoSFFloat.h>
#include <Inventor/fields/SoSFColor.h>
#include <Inventor/fields/SoSFULong.h>
#include <GL/gl.h>


class SoSplatNoise : public SoSplatTexture {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatNoise);

  public:

    // Fields
    SoSFFloat   Granularity;
    SoSFColor   NoiseColor;
    SoMFFloat   AmountNoise;
    SoSFULong    numTextures;
    SoSFFloat    MaxAmplitude;

    // Constructor
    SoSplatNoise();

    // Destructor
    virtual ~SoSplatNoise();

  SoINTERNAL public:
    static void    initClass();

  protected:
    // Implements actions
    virtual void  renderSplat( int index, float x, float y, float z, float ss );
    virtual void  initGLRender( SoGLRenderAction *);

  private:
    static GLuint  NoiseSplats;

    static int     textureLoaded;

    double  optsplat( double r );
};

#endif /* _SO_SPLAT_NOISE_ */
```

8.6.2 SoSplatRandom.h

```
/*******************************************************************************
//  _______________________________________________________________________
//  ___________                                            __________________
//  ___________    L A W R E N C E   L I V E R M O R E     __________________
//  ___________   N A T I O N A L   L A B O R A T O R Y    __________________
//  _______________________________________________________________________
//
//
//    NAME
//      SoSplatRandom - Volume render of two scalar fields by splatting texture
//                      mapped squares. An noisy texture is used with an
//                 increasing number of random points in the texture.
//
//
//    VERSION
//      @(#)SoSplatRandom.h version 1.1  created on 95/06/06  at 13:03:32
//
//
//    NOTES
//      1) See the man pages or the implementation prologue for more info.
//
//
*******************************************************************************/
#include "LLNL_Disclaimer.h"


#ifndef  _SO_OSplatRandom_
#define  _SO_OSplatRandom_

#include "SoSplatTexture.h"
#include <Inventor/fields/SoSFFloat.h>
#include <Inventor/fields/SoSFColor.h>
#include <Inventor/fields/SoSFULong.h>
#include <GL/gl.h>


class SoSplatRandom : public SoSplatTexture {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatRandom);

  public:

    // Fields
    SoSFFloat   Granularity;
    SoSFColor   NoiseColor;
    SoMFFloat   AmountNoise;
    SoSFULong    numTextures;
    SoSFULong    numPoints;

    // Constructor
    SoSplatRandom();

    // Destructor
    virtual ~SoSplatRandom();

  SoINTERNAL public:
    static void    initClass();
  protected:
    // Implements actions
    virtual void  renderSplat( int index, float x, float y, float z, float ss );
    virtual void  initGLRender( SoGLRenderAction *);

  private:
    static GLuint  NoiseSplats;

    static int     textureLoaded;

    double  optsplat( double r );
};

#endif /* _SO_SPLAT_RANDOM_ */
```

8.6.3 SoSplatPoints.h

```
/*****************************************************************************
//  _______________________________________________________________________
//  ___________                                           __________________
//  ___________    L A W R E N C E   L I V E R M O R E    __________________
//  ___________   N A T I O N A L   L A B O R A T O R Y   __________________
//  _______________________________________________________________________
//
//
//    NAME
//      SoSplatPoints - Volume render of two scalar fields by splatting texture
//                      mapped squares. An noisy texture is used with an
//                 increasing number of points in the textures. The points
//                 are added in a wieghted fashion to produce a uniform
//                 distribution as the splats overlap each other.
//
//
//    VERSION
//      @(#)SoSplatPoints.h version 1.1  created on 95/06/06  at 13:03:31
//
//
//    NOTES
//      1) See the man pages or the implementation prologue for more info.
//
//
*****************************************************************************/
#include "LLNL_Disclaimer.h"


#ifndef  _SO_OSplatPoints_
#define  _SO_OSplatPoints_

#include "SoSplatTexture.h"
#include <Inventor/fields/SoSFFloat.h>
#include <Inventor/fields/SoSFColor.h>
#include <Inventor/fields/SoSFULong.h>
#include <GL/gl.h>


class SoSplatPoints : public SoSplatTexture {

    // Define typeId and stuff
    SO_NODE_HEADER(SoSplatPoints);

  public:

    // Fields
    SoSFFloat   Granularity;
    SoSFColor   NoiseColor;
    SoMFFloat   AmountNoise;
    SoSFULong    numTextures;
    SoSFULong    numPoints;

    // Constructor
    SoSplatPoints();

    // Destructor
    virtual ~SoSplatPoints();

  SoINTERNAL public:
    static void    initClass();

  protected:
    // Implements actions
    virtual void  renderSplat( int index, float x, float y, float z, float ss );
    virtual void  initGLRender( SoGLRenderAction *);

  private:
    static GLuint  NoiseSplats;

    static int     textureLoaded;

    double  optsplat( double r );
};

#endif /* _SO_SPLAT_POINTS_ */
```

Chapter 9

Summary

Four new techniques have been developed for the exploration of complex three-dimensional simulations. It should be noted that the goal of these techniques is not to provide quantitative information, rather they aid in the search process of large terabyte data sets. Understanding the vast amounts of data generated from large supercomputers requires several different visualization tools. One of the first steps is usually to browse through the data — in space, time, and variable space — and determine where further investigation may be the most fruitful. The techniques described in this study offer a global qualitative representation of a single vector field, combined scalar and vector fields, multiple vector fields and multiple scalar fields.

These browsing techniques offer new methods for examining multivariate data. The concept of integrating vector and multivariate information within volume rendering for scientific visualization is a new thrust area. These techniques offer a solid approach to this area. The Line Bundles technique offers a fast and efficient method for generating "etched" surfaces, showing not only isocontours of a vector field's magnitude, but also of its directionality. They can also be used to represent entire volumes in three-dimensional space and allow for the representation of time-varying data without costly advection algorithms.

The Textured Splats technique allows for animation of the vector field and offers the ability to represent an additional scalar variable. The use of a high-end workstation's capability to quickly render objects with texture maps is maximized. Real-time rates are possible for small data sets, with just a small degradation in performance over the representation of a single scalar field.

The Hair Splats extend the Line Bundles from straight line segments to curves specified by two vector fields. They are useful for representing the flow across a surface, showing the change in a time-varying vector field or showing the relationship between two vector fields.

For multiple scalar fields, the idea of textured splats has been extended to encode a separate variable into the textures using generated noise. The relationship between two variables can be explored in three-dimensions, with again, a very modest performance penalty over the representation of a single scalar field.

This research continues to lead to many new ideas — simple enhancements or extensions to the techniques presented in this thesis, as well as new approaches for this thrust area. The Line Bundles needs to be extended to non-regular grid topologies. The use of shadowing in the Line Bundles and the Hair Splats would aid in their representation. Scaling issues are inherit in all of the techniques. As we zoom in or out of the data space, the line density of the line bundles and hairs, or texture granularity of the texture splats and noise splats changes. The ability to zoom into interesting areas of the line bundles, while preserving the appearance of the density of lines is a further topic of exploration.

The texture splats offers a general base for much further exploration. There is no limit as to the type of texture that can be used in the splatting techniques. Only a very few textures have been developed here. Many new and interesting textures are still to be developed for possibly new and exciting purposes.

Bibliography

[1] Adelson, S.J. and C.D. Hansen. *Fast Stereoscopic Images with Ray-Traced Volume Rendering.* in *1994 Symposium on Volume Visualization*. 1994. Washington, D.C.: ACM Press.

[2] Akeley, K. *RealityEngine Graphics.* in *SIGGRAPH '93*. 1993. Anaheim, California: ACM SIGGRAPH.

[3] Avila, R.S., L.M. Sobierajski, and A.E. Kaufman. *Towards a Comprehensive Volume Visualization System*. in *Visualization '92*. 1992. Boston, Massachusetts: IEEE Computer Society Press.

[4] Banks, D.C. *Illumination in Diverse Codimensions.* in *SIGGRAPH 94*. 1994. Orlando, Florida: ACM SIGGRAPH.

[5] Banks, D.C. and B.A. Singer. *Vortex Tubes in Turbulent Flows: Identification, Representation, Reconstruction.* in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[6] Board, O.A.R., *OpenGL Reference Manual*. Release 1 ed. 1992, Reading, MA: Addison-Wesley. 388.

[7] Board, O.A.R., *et al.*, *OpenGL Programming Guide*. Release 1 ed. 1993, Reading, MA: Addison-Wesley. 516.

[8] Brill, M., *et al. Streamball Techniques for Flow Visualization.* in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[9] Cabral, B., N. Cam, and J. Foran. *Accelerated Volume Rendering and Tomographics Reconstruction Using Texture Mapping Hardware.* in *1994 Symposium on Volume Visualization*. 1994. Washington, D.C.: ACM Press.

[10] Cabral, B. and C. Leedom. *Imaging Vector Fields Using Line Integral Convolution*. in *Computer Graphics.* August 1993.

[11] Cameron, G.G. and P.E. Undrill. *Rendering Volumetric Medical Image Data on a SIMD Architecture Computer*. in *Third Eurographics Workshop on Rendering.* May 1992. Bristol, UK.

[12] Challinger, J., *Parallel Volume Rendering on a Shared-Memory Multiprocessor*

Tech. Report UCSC-CRL-91-23, . 1991, University of California, Santa Cruz.

[13] Chapman, J., T.W. Calvert, and J. Dill. *Spatio-Temporal Coherence in Ray Tracing.* in *Proceedings of Graphics Interface '91.* June 1991. Calgary, Alberta: Canadian Information Processing Society.

[14] Chapman, J., T.W. Calvert, and J. Dill. *Exploiting Temporal Coherence in Ray Tracing.* in *Proceedings of Graphics Interface '90.* May 1990. Toronto, Ontario: Canadian Information Processing Society.

[15] Cohen, M.F. and J.R. Wallace, *Radiosity and Realistic Image Synthesis*. 1993, Boston: Academic Press Professional. 381.

[16] Cook, R.L., *Stochastic Sampling in Computer Graphics.* ACM Transactions on Graphics, Jan. 1986. **5**(1): p. 51-72.

[17] Crawfis, R. and M. Allison. *A Scientific Visualization Synthesizer*. in *Visualization '91*. 1991. San Diego, CA: IEEE Computer Society Press.

[18] Crawfis, R., *et al.*, *Advanced Techniques for Scientific Visualization*, in *SIGGRAPH '94 Course Notes CD-ROM*, S. Keith, Editor. 1994, ACM SIGGRAPH: New York.

[19] Crawfis, R., *et al.*, *Advanced Techniques for Scientific Visualization*, in *SIGGRAPH '95 Course Notes CD-ROM*, S. Keith, Editor. 1995, ACM SIGGRAPH: New York.

[20] Crawfis, R. and N. Max. *Direct Volume Visualization of Three-Dimensional Vector Fields*. in *Proceedings of the 1992 Workshop on Volume Visualization*. 1992. New York: ACM SIGGRAPH.

[21] Crawfis, R. and N. Max. *Texture Splats for 3D Vector and Scalar Field Visualization*. in *Visualization '93*. 1993. San Jose, CA: IEEE Computer Society Press.

[22] Crawfis, R., N. Max, and B. Becker, *Vector Field Visualization*. Computer Graphics and Applications, 1994. **14**(5): p. 50-56.

[23] Crawfis, R., *et al. Volume Rendering of 3D Scalar and Vector Fields at LLNL*. in *Supercomputing '93*. 1993. Portland, Oregon: IEEE Computer Society Press.

[24] Delmarcelle, T. and L. Hesselink. *The Topology of Symmetric, Second-Order Tensor Fields*. in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[25] Drebin, R.A., L. Carpenter, and P. Hanrahan, *Volume Rendering*. Computer Graphics, August 1988. **22**(4): p. 64-75.

[26] Forssell, L.K. *Visualizing Flow Over Curvilinear Grid Surfaces Using Line Integral Convolution*. in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[27] Franke, R., *A Critical Comparison of Some Methods for Interpolation of Scattered Data*, . 1979, Naval Postgraduate School Monterey California.

[28] Garrity, M.P., *Raytracing Irregular Volume Data*. Computer Graphics (San Diego Workshop on Volume Visualization), Nov. 1990. **24**(5): p. 35-40, 110.

[29] Glassner, A.S., *et al.*, *An Introduction to Ray Tracing*, ed. A.S. Glassner. 1989, London: Academic Press.

[30] Globus, A., C. Levit, and T. Lasinski. *A Tool for Visualizing the Topology of Three-Dimensional Vector Fields*. in *Visualization '91*. October 1991. San Diego, CA: IEEE Computer Society Press.

[31] Hall, R., *Illumination and Color in Computer Generated Imagery*. Monographs in Visual Computing, ed. D.F. Rogers. 1989, New York: Springer-Verlag. 282.

[32] Hanson, A.J. and R.A. Cross. *Interactive Visualization Methods for Four Dimensions*. in *Visualization '93*. 1993. Boston, Massachusetts: IEEE Computer Society Press.

[33] Hanson, A.J. and H. Ma. *Visualizing Flow with Quaternion Frames*. in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[34] Hardy, R.L., *Multiquadric equations of topography and other irregular surfaces*. Journal of Geophysical Research, 1971(76): p. 1905-1915.

[35] Haruyama, S. and B.A. Barsky, *Using stochastic modeling for texture generation*. IEEE Comput. Graphics and Appl. (USA), March 1984. **4**: p. 7-19.

[36] Hellman, J.L. and L. Hesselink, *Visualizing Vector Field Topology in Fluid Flows*. IEEE Computer Graphics & Applications, May 1991. **11**(2?): p. 36-46.

[37] Hin, A.J.S. and F.H. Post. *Visualization of Turbulent Flow with Particles*. in *Visualization '93*. 1993. Washington, D.C.: IEEE Computer Society Press.

[38] Hoffman, J.D., *Numerical Methods for Engineers and Scientists*. 1992, New York: McGraw-Hill Inc. 825.

[39] Hultquist, J.P.M. *Constructing Stream Surfaces in Steady 3D Vector Fields*. in *Visualization '92*. October 1992. Los Alamitos, CA: IEEE Computer Society Press.

[40] Joy, K.I., *et al.*, *Tutorial: Computer Graphics: Image Synthesis*, . 1988, Computer Society Press: Washington, DC.

[41] Kajiya, J.T. and B.P.V. Herzen, *Rendering Fur with Three Dimensional Textures.* Computer Graphics, 1989. **23**(3): p. 271-280.

[42] Kajiya, J.T. and B.P.V. Herzen, *Ray Tracing Volume Densities.* Computer Graphics, July 1984. **18**(3): p. 165-174.

[43] Kaufman, A., ed. *Volume Visualization*. . 1991, IEEE Computer Society Press: Los Alamitos, CA. 479.

[44] Keller, P.R. and M.M. Keller, *Visual Cues*. 1993, Los Alamitos, CA: IEEE Computer Society Press. 229.

[45] Kirk, D. and D. Voorhies, *The Rendering Architecture of the DN10000VS.* Computer Graphics, 1990. **24**(4): p. 299-307.

[46] Lacroute, P. and M. Levoy. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation.* in *SIGGRAPH 94*. 1994. Orlando, Florida: ACM SIGGRAPH.

[47] Laur, D. and P. Hanrahan, *Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering.* Computer Graphics, July 1991. **25**(4): p. 285-–288.

[48] Levoy, M. *Design for a Real-Time High-Quality Volume Rendering Workstation.* in *Chapel Hill Workshop on Volume Visualization*. 1989: Dept. of Computer Science, Univ. of North Carolina.

[49] Levoy, M., *Rendering Mixtures of Geometric and Volumetric Data*

Technical Report 88-052, . Dec. 1988, Computer Science Dept., Univ. of North Carolina at Chapel Hill.

[50] Levoy, M., *Volume Rendering by Adaptive Refinement.* The Visual Computer, Feb. 1990. **6**(1): p. 2-7.

[51] Levoy, M., *Efficient Ray Tracing of Volume Data.* ACM Transactions on Graphics, July 1990. **9**(3): p. 245-261.

[52] Levoy, M., *A Hybrid Ray Tracer for Rendering Polygon and Volume Data.* IEEE Computer Graphics and Applications, March 1990. **10**(2): p. 33-40.

[53] Levoy, M., *Display of Surfaces from Volume Data.* IEEE Computer Graphics and Applications, May 1988. **8**(5): p. 29-37.

[54] Levoy, M. *Display of Surfaces from Volume Data*. in *Ph.D. Dissertation.* May 1989: Department of Computer Science, The University of North Carolina at Chapel Hill.

[55] Levoy, M. and R. Whitaker, *Gaze-Directed Volume Rendering.* Computer Graphics (Proc. 1990 Symposium on Interactive 3D Graphics), March 1990. **24**(2): p. 217-223.

[56] Lewis, J.P., *Texture synthesis for digital painting.* Comput. Graphics (USA), July 1984. **18**: p. 245-252.

[57] Lewis, R.R. *Three-Dimensional Texturing Using Lattices*. in *Eurographics '90*. 1990: Elsevier Science Publishers B.V.

[58] Ma, K.-L. and P.J. Smith. *Cloud Tracing in Convection-Diffusion Systems*. in *Visualization '93*. 1993. Washington, D.C.: IEEE Computer Society Press.

[59] Ma, K.-L. and Z.C. Zheng. *3D Visualization of Unsteady 2D Airplane Wake Vortices*. in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[60] Max, N., *An Optimal Filter for Image Reconstruction*, in *Graphics Gem II*, J. Arvo, Editor, Academic Press: New York. p. 101-104.

[61] Max, N. *Efficient Light Propagation for Multiple Anisotropic Volume Scattering.* in *The Fifth Eurographics Workshop on Rendering*. 1994. Darmstadt, Germany.

[62] Max, N., B. Becker, and R. Crawfis. *Flow Volumes for Interactive Vector Field Visualization*. in *Visualization '93*. 1993. Los Alamitos, CA: IEEE Computer Society Press.

[63] Max, N. and R. Crawfis. *Advances in Scientific Visualization*. in *IS&T/SPIE Symposium on Electronic Imaging: Science and Technology*. 1995. San Jose: SPIE.

[64] Max, N., R. Crawfis, and C. Grant. *Visualizing 3D Velocity Fields Near Contour Surfaces*. in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[65] Max, N., R. Crawfis, and D. Williams, *Visualization for Climate Modeling.* Computer Graphics and Applications, 1993. **13**(4): p. 34-40.

[66] Max, N., P. Hanrahan, and R. Crawfis, *Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions.* Computer Graphics, 1990. **24**(5): p. 27-33.

[67] Max, N.L., *Sorting for Polyhedron Compositing*, in *Focus on Scientific Visualization*, H. Hagen, H. Muller, and G.M. Nielson, Editors. 1992, Springer-Verlag: Berlin. p. 259-268.

[68] Meinzer, H.-P., *et al.*, *The Heidelberg Ray Tracing Model.* IEEE Computer Graphics and Applications, Nov. 1991. **11**(6): p. 34-43.

[69] Mitchell, D.P., *Generating Antialiased Images at Low Sampling Densities.* Computer Graphics, 1987. **21**(4): p. 65-72.

[70] Montani, C. and R. Scopigno, *Rendering Volumetric Data using the STICKS Representation Scheme.* Computer Graphics (San Diego Workshop on Volume Visualization), Nov. 1990. **24**(5): p. 87-93, 112.

[71] Nishimura, H., *et al. Object modelling by distribution function and a method of image generation*. in *Electronics Communication Conference*. 1985.

[72] Pagendarm, H.-G. and B. Walter. *Feature Detection from Vector Quantities in a Numerically Simulated Hypersonic Flow Field in Combination with Experimental Flow Visualization.* in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[73] Perlin, K., *An Image Synthesizer.* Comput. Graphics (USA) (em Proc. SIGGRAPH '85; held in San Francisco, CA; 22-26 July 1985, July 1985. **19**(3): p. 287-296.

[74] Porter, T. and T. Duff, *Compositing Digital Images.* Computer Graphics, 1984. **18**(3): p. 253-259.

[75] Preparata, F.P. and M.I. Shamos, *Computational Geometry*. Texts and Monographs in Computer Science, ed. D. Gries. 1985, New York: Springer-Verlag. 390.

[76] Press, W.H., *et al.*, *Numerical Recipes in C*. 1988, Cambride: Cambridge University Press. 735.

[77] Reeves, W.T., *Particle Systems - A Technique for Modeling a Class of Fuzzy Objects.* ACM Transactions on Graphics, April 1983. **2**(2): p. 91-108.

[78] Reeves, W.T. and R. Blau, *Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems.* Computer Graphics, July 1985. **19**(3): p. 313-322.

[79] Sabella, P., *A Rendering Algorithm for Visualizing 3D Scalar Fields.* Computer Graphics, August 1988. **22**(4): p. 51-58.

[80] Schroeder, W.J., C.R. Volpe, and W.E. Lorensen. *The Stream Polygon: A Technique for 3D Vector Field Visualization.* in *Visualization '91*. October 1991. San Diego, CA: IEEE Computer Society Press.

[81] Shepard, D. *A Two Dimensional Interpolation Function for Irregularly Spaced Data.* in *23rd National Conference of the ACM*. 1968: ACM.

[82] Shirley, P. and A. Tuchman, *A Polygonal Approximation to Direct Scalar Volume Rendering.* Computer Graphics, November 1990. **24**(5): p. 63-70.

[83] Sims, K., *Particle Animation and Rendering Using Data Parallel Computation.* Computer Graphics, 1990. **24**(4): p. 405-413.

[84] Sobierajski, L.M. and A.E. Kaufman. *Volumetric Ray Tracing*. in *1994 Symposium on Volume Visualization*. 1994. Washington, D.C.: ACM Press.

[85] Stalling, D. and H.-C. Hege. *Fast and Resolution Independent Line Integral Convolution.* in *SIGGRAPH 95*. 1995. Los Angeles, California: ACM SIGGRAPH.

[86] Stander, B.T. and J.C. Hart. *A Lipschitz Method for Accelerated Volume Rendering.* in *1994 Symposium on Volume Visualization*. 1994. Washington, D.C.: ACM Press.

[87] Stein, C., B. Becker, and N. Max. *Sorting and Hardware Assisted Rendering for Volume Visualization*. in *1994 Symposium on Volume Visualization*. 1994. Washington, D.C.: ACM Press.

[88] Stoer, J. and Bulirsch, *Introduction to Numerical Analysis.* 1980, New York: Spring-Verlag. 609.

[89] Totsuka, T. and M. Levoy. *Frequency Domain Volume Rendering.* in *SIGGRAPH '93*. 1993. Anaheim, California: ACM SIGGRAPH.

[90] Trousset, Y. and F. Schmitt. *Active-Ray Tracing for 3D Medical Imaging.* in *Eurographics '87.* Aug. 1987. Amsterdam, North-Holland: Elsevier Science Publishers.

[91] Twiddy, R., J. Cavallo, and S.M. Shiri. *Restorer: A Visualization Technique for Handling Missing Data.* in *Visualization '94*. 1994. Washington, D.C.: IEEE Computer Society Press.

[92] Upson, C. and M. Keeler, *V-BUFFER: Visible Volume Rendering.* Computer Graphics, August 1988. **22**(4): p. 59-64.

[93] Watanabe, Y. and Y. Suenaga, *Drawing human hair using the Wisp model.* The Visual Computer, 1991. **7**: p. 97-103.

[94] Webber, E., *Ray Tracing Voxel Data via Biquadratic Local Surface Interpolation.* The Visual Computer, Feb. 1990. **6**(1): p. 8-15.

[95] Westover, L., *Footprint Evaluation for Volume Rendering.* Computer Graphics, August 1990. **24**(4): p. 367-376.

[96] Westover, L. *Interactive Volume Rendering.* in *Proceedings of the Chapel Hill Workshop on Volume Visualization*. May 1989. Chapel Hill, NC.

[97] Whitted, T., *An Improved Illumination Model for Shaded Display.* CACM, June 1980. **23**(6): p. 343-349.

[98] Wijk, J.J.v., *Spot Noise-Texture Synthesis for Data Visualization.* Computer Graphics, July 1991. **25**(4): p. 309-318.

[99] Wijk, J.J.v., *Flow Visualization with Surface Particles.* IEEE Computer Graphics and Applications, July 1993. **13**(4): p. 18-24.

[100] Wijk, J.J.v. *Rendering Surface-Particles.* in *Visualization '92*. October 1992. Los Alamitos, CA: IEEE Computer Society Press.

[101] Wijk, J.J.v. *Implicit Stream Surfaces*. in *Visualization '93*. October 1993. Los Alamitos, CA: IEEE Computer Society Press.

[102] Wilhelms, J. and A.V. Gelder, *A Coherent Projection Approach for Direct Volume Rendering.* Computer Graphics, July 1991. **25**(4): p. 275-284.

[103] Williams, P., *Visibility Ordering Meshed Polyhedra.* ACM Transactions on Graphics, April 1992. **11**(2): p. 103-126.

[104] Williams, P. and N. Max. *A Volume Density Optical Model*. in *Proceedings of the 1992 Workshop on Volume Visualization*. October 1992. New York: ACM.

[105] Yagel, R., D. Cohen, and A. Kaufmann, *Discrete Ray Tracing.* IEEE Computer Graphics and Applications, Sept. 1992. **12**(5): p. 19-28.

[106] Yagel, R. and A. Kaufmann, *Template-Based Volume Viewing.* Computer Graphics Forum (Eurographics '92), Sept. 1992. **11**(3): p. 153-167.

[107] Yagel, R. and Z. Shi. *Accelerating Volume Animation by Space-Leaping.* in *Visualization '93*. 1993. Boston, Massachusetts: IEEE Computer Society Press.

[108] Yang, W.-J., *Computer-Assisted Flow Visualization*. 1994, Boca Raton, FL: CRC Press. 342.