Interactive Numerical Flow Visualization Using Stream Surfaces

TR95-014
1995

Jeff P.M. Hultquist

Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599-3175

Interactive Numerical Flow Visualization Using Stream Surfaces

by

Jeff P.M. Hultquist

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
1995

Approved by:

Frederick P. Brooks, Jr.

Advisor: Frederick P. Brooks, Jr.

[illegible]

Reader: James Coggins

Thomas A. Lasinski

Reader: Thomas A. Lasinski

JEFFREY PAUL MCCURDY HULTQUIST

Interactive Numerical Flow Visualization Using Stream Surfaces
(Under the direction of Frederick P. Brooks, Jr.)

Three-dimensional steady fluid flows are often numerically simulated over multiple overlapping curvilinear arrays of sample points. Such flows are often visualized using tangent curves or *streamlines* computed through the interpolated velocity field. A *stream surface* is the locus of an infinite number of streamlines rooted at all points along a short line segment or *rake*. Stream surfaces can depict the structure of a flow field more effectively than is possible with mere streamline curves, but careful placement of the rakes is needed to most effectively depict the important features of the flow.

I have built visualization software which supports the interactive calculation and display of stream surfaces in flow fields represented on composite curvilinear grids. This software exploits several novel methods to improve the speed with which a particle may be advected through a vector field. This is combined with a new algorithm which constructs adaptively sampled polygonal models of stream surfaces. These new methods make stream surfaces a viable tool for interactive numerical flow visualization. Software based on these methods has been used by scientists at the NASA Ames Research Center and the Wright-Patterson Air Force Base. Initial use of this software has demonstrated the value of this approach.

ACKNOWLEDGMENTS

I wish to thank my wife, Mary, for her constant encouragement over many years. Her faith in my eventual completion of this project is the only thing which made that completion a reality. My sons, Michael, Kyle, and Joseph, have brought me much love and joy, and they place this present work in its true perspective. My own parents have always encouraged me to find what I enjoy doing and to work hard doing it. I thank them for their love and support.

This project has benefited from extensive discussions with my colleagues at NASA, particularly Eric Raible, Kris Miceli, and David Kenwright. Tom Lasinski has been the most patient of bosses and a most helpful member of my dissertation committee. Pieter Buning graciously brought his expertise to my committee. The balance of my committee, Fred Brooks, James Coggins, and Henry Fuchs, allowed me the freedom to pursue my research on the other end of the country. I thank each of them very much.

The application software built during this project incorporates the work of Eric Raible, Kris Miceli, and David Betz. Fluid flow datasets were provided by Michael Aftosmis, Pieter Buning, Neal Chaderjian, John Ekaterinaris, Datta Gaitonde, Cetin Kiris, Goetz Klopfer, Stuart Rogers, and Miguel Visbal.

All the work described herein was conducted at the Numerical Aerodynamic Simulation Facility at the NASA Ames Research Center.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

e	energy
f, g	arbitrary fields
p	pressure
s	streamline parameter
t	time
λ	interpolation coefficient
ρ	density
ψ	stream function
$\mathcal{D}$	distance function
$\mathcal{I}$	interpolating function
$\mathcal{J}$	Jacobian matrix
$\vec{x}$	physical-space point
x, y, z	physical-space coordinates
$\vec{\xi}$	computational-space point
ξ, η, ζ	computational-space coordinates
i, j, k	grid indices
α, β, γ	offsets within grid cell
$\vec{g}$	gravity force vector
$\vec{p}, \vec{q}$	arbitrary points
$\vec{u}$	velocity vector
$\vec{u}_\infty$	freestream velocity
u, v, w	velocity components

CHAPTER I
OVERVIEW

1.1 COMPUTATIONAL FLUID DYNAMICS

The study of fluid flow is typically comprised of three phases: the construction of a grid of node points, the calculation of the flow field sample values at these points, and the post-process visualization of the resulting data. The requirements of the first two tasks motivate the use of unusual data structures, which pose difficulty for the interactive visualization of the results.

1.1.1 Grid Generation

The flow simulation effort begins with a description of the surface of an object, usually by a sequence of digitized cross-sections or a collection of surface spline patches. The object may be as simple as a sphere or as intricate as the Space Shuttle Orbiter with solid rocket boosters, external fuel tank, and interconnection hardware [Pearce *et al.* 1993]. Some volume surrounding the object is then tessellated into cells. These cells are small enough that the flow field may be expected to vary only slightly through each one. The partial differential equations which govern fluid flow are then discretized and numerically solved over the collection of points which are the vertices of these cells.

In Finite Element Analysis, the cells of an *unstructured* computational grid may have a variety of shapes and each node point may be shared by any number of cells. These general cells can be easily positioned around intricate domain boundaries, but this generality requires increased bookkeeping during the simulation

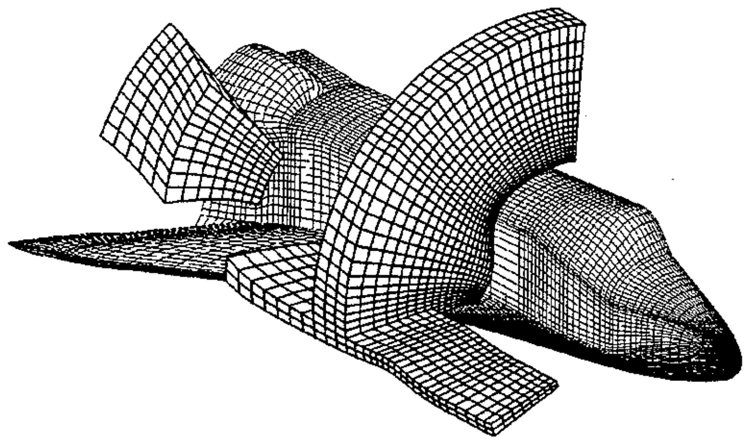


Figure 1.1: Curvilinear cells in the volume surrounding a vehicle.

and increased memory to store the grid information. This overhead is acceptable for work with structures, since a grid of a few thousand cells typically provides adequate resolution of the computed deformation and stress fields.

Unstructured meshes are not often used in Computational Fluid Dynamics (CFD). Fluid flows contain regions of detail which must be sampled at very small spatial intervals. The bookkeeping overhead of unstructured grids consumes storage space which could otherwise be used to record additional node point positions and flow field samples. Furthermore, many flow simulation codes are run on traditional vector supercomputers which cannot reach peak performance when processing over the arbitrary connectivity of an unstructured grid.

The ever greater detail of vehicle surface descriptions and the increasing availability of massively parallel computers may eventually increase the use of unstructured grids in CFD, but the majority of present-day flow calculations are performed on structured, body-conforming, *curvilinear* grids. These grids maintain a regular interconnection of hexahedral cells, but allow the placement of the node points to

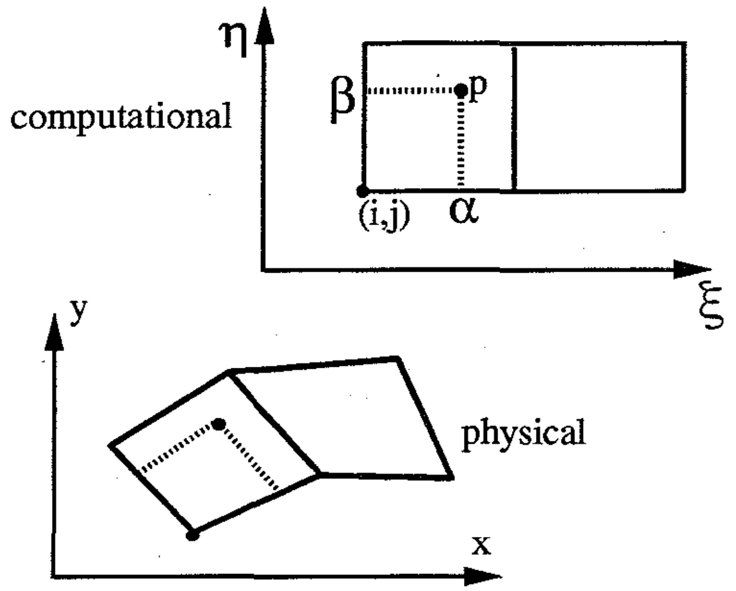


Figure 1.2: Computational and physical coordinate spaces.

follow the curved contours of the vehicle surface. As a result, the six faces of each cell are not planar, but are instead double-ruled bilinear patches. Figure 1.1 shows some nodes taken from a three-dimensional grid which has been wrapped around the surface of the Shuttle Orbiter. This data, provided by Dr. Pieter Buning, was constructed for a simulation of the Space Shuttle launch [Buning *et al.* 1988, Martin *et al.* 1990].

The position of each node point is recorded in the (x, y, z) coordinates of the Cartesian *physical* space. The interconnection of these node points is implicitly defined by their placement in three-dimensional storage arrays. This ordering also defines a *computational* coordinate space (ξ, η, ζ) in which each cell is a unit cube. Integer-valued computational coordinates map directly onto the array indices of each node point. Figure 1.2 depicts the relationship between these two spaces in two cells of a two-dimensional grid.

In simulations of the flow of a viscous fluid, the cells near the solid boundaries of the vehicle must be small enough to adequately resolve the details of the boundary

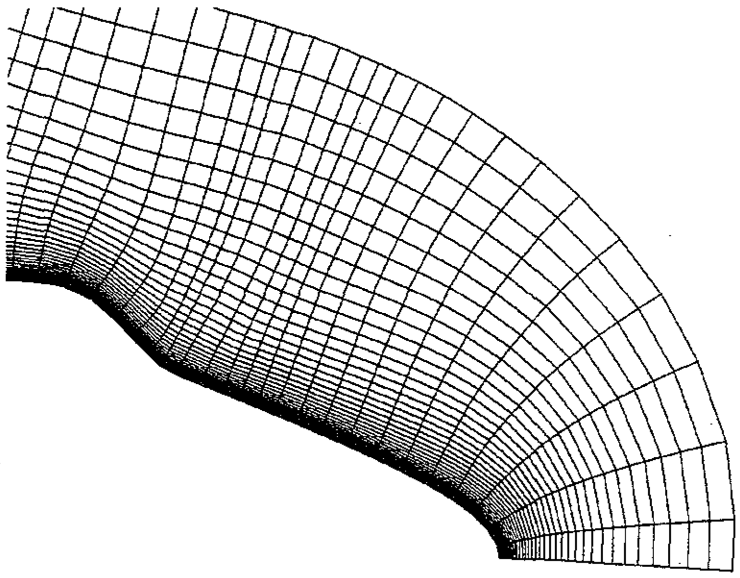


Figure 1.3: Grid cells surrounding the nose of the Shuttle Orbiter.

layer. On the other hand, for the sake of computational efficiency, the cells further from the vehicle may be rather large. This is allowable since the flow far from vehicle is relatively undisturbed. In a typical grid, the largest cells may be several thousand times larger than the smallest ones (see figure 1.3). In the grid block shown earlier, the entire Orbiter vehicle has been assigned a length of about 1.2 physical-space units. The innermost layer of cells surrounding the body has a thickness of only about 5×10^{-5}. The real Orbiter is about 120 feet long. If the innermost cells were scaled to fit the full-size vehicle, they would be less than one-tenth of an inch thick and about two feet across. In this same block, the outermost layer of cells is almost five hundred times thicker. This range of cell-size, two or even three orders of magnitude along each grid dimension, is typical for CFD grids and this is necessitated by the widely varying scale of the features present in real flows.

For numerical stability in the flow simulation, each cell should be approximately rectilinear and of generally the same size as its immediate neighbors. The spatial relationship of fuselage, wings, engines, and control surfaces can make these

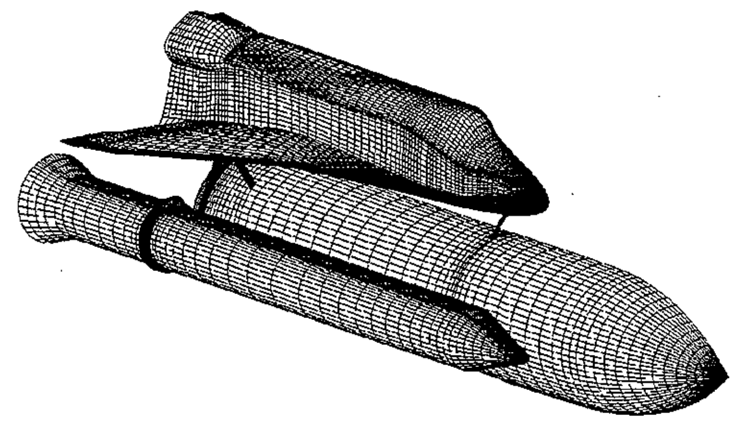


Figure 1.4: Multiple grid blocks around the Space Shuttle assembly.

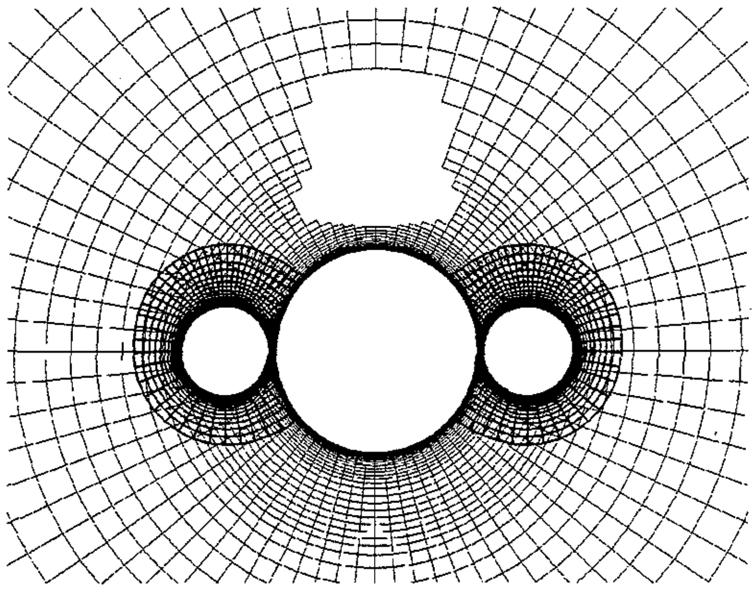


Figure 1.5: Grid slices around the Space Shuttle assembly.

restrictions very difficult to satisfy. The task of grid construction can be simplified by dividing the flow domain into sub-regions, and then sampling the flow field using multiple, partially overlapping blocks of cells. These so-called *multiple-block* or *composite grid* schemes [Benek *et al.* 1985] can allow a better positioning of the node points around vehicles for which a single block would be unsuitable. For example, figure 1.4 shows the innermost slice from the Orbiter grid block now combined with other blocks which have been constructed around the fuel tank and solid boosters. In figure 1.5, we see a frontal view of a few slices taken from this composite grid and then reflected about the mid-line of the vehicle. A single large grid has been wrapped around the main fuel tank. Holes or *voids* has been excised into this grid, and the flow in these regions is represented by other grid blocks with smaller cells. The irregularly shaped void at the upper center of the image will be filled by the body-conforming Orbiter grid block. A similar void has already been filled by a circular block surrounding the solid rocket booster. Note that this smaller block has itself been partially voided to eliminate those node points which otherwise would have penetrated the interior of the main fuel tank.

Each node point in a composite grid is marked with an integer mask to indicate which other block, if any, also occupies the same region of the flow domain. These integer tags are called *iblank* numbers, because the technique is called *blanking* and because most flow simulation codes are written in FORTRAN, in which integer variables traditionally begin with the letter "`i`."

Node points in the void regions are marked with an iblank value of 0. Usable node points in the interior of a block are marked with the value 1. The node points in the coincident regions carry a negative iblank number. The absolute value of this tag is the identifying number, counting from one, of some locally overlapping block. The meaning encoded by the iblank values must be preserved by the visualization software. The interpolation of field samples at some query point location must be preceded by a check of the iblank values at the eight corners of the cell which contains that query point. Field values may be interpolated only within cells for which all eight vertex nodes are marked with non-zero iblank numbers.

1.1.2 Flow Simulation

The equations that describe fluid flow may be written using the *material derivative* (Df/Dt), which expresses the rate of change of a field value when measured at the changing location of a massless particle that is being carried (or *advected*) by the moving fluid. This derivative is equal to the sum of the change of the field over time and the change due to the shifting position of the particle itself in the three spatial dimensions of the domain.

$$\begin{aligned}\frac{Df}{Dt} &= \frac{\partial f}{\partial t} + \left(\frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial f}{\partial z}\frac{\partial z}{\partial t}\right) \\ &= \frac{\partial f}{\partial t} + \vec{u}\cdot\nabla f\end{aligned}$$

In the above equation, the term $\partial f/\partial t$ describes the simple rate of change of the field f, while the second term is the inner product of the velocity vector $\vec{u}$ with the gradient of this field. The gradient of a field is a vector which indicates the direction of maximal increase. The product of the velocity and the gradient vector gives the rate of change introduced by the motion of the particle through that field.

The *Euler equation*, which describes the flow of an inviscid (frictionless) and incompressible fluid, specifies that the material derivative of the momentum is reduced by the pressure gradient and increased by the force of gravity.

$$\rho\frac{D\vec{u}}{Dt} = -\nabla p + \rho\vec{g}$$

Here we see that the material derivative of the momentum field is the sum of two terms. The first is the negative value of the gradient of the fluid pressure. This indicates that each moving particle is shifted in the direction of lowest pressure. The other term is product of the density ρ and the gravitational force vector $\vec{g}$. This merely represents the influence of gravity on the motion of the fluid.

Additional terms must be included if the fluid can be compressed, if it is viscous, and if the viscosity is dependent on temperature. Each new term increases the computational effort required to compute the flow field samples at each node point. In practice, the primary terms of the flow equations are accounted for directly, while the less significant and the more computationally expensive terms are approximated.

The governing equations are discretized over the node points of the grid and solved numerically with respect to a given set of boundary conditions. These constraints typically specify the undisturbed *freestream velocity* ($\vec{u}_\infty$) far from the vehicle. The constraints might also require that the velocity magnitude be zero at the vehicle surface (the *no-slip condition*). A *flow solver* program then computes the solution of these discretized equations at each node point in every grid block. On a composite grid, the evolving flow-field values must be averaged across neighboring blocks to reach a consensus in the regions of overlap.

For compressible flows, the values typically computed at each node are the local fluid density (ρ), the momentum vector ($\rho\vec{u}$), and the local energy (e). Ancillary fields can be derived as functions of these fundamental quantities. For example, the *dynamic pressure* of the fluid at each node is equal to ($\frac{1}{2}\rho|\vec{u}|^2$). Differential quantities, such as the *vorticity* ($\nabla \times \vec{u}$) or the *density gradient* ($\nabla\rho$), can be approximated using finite difference methods. These last calculations must include spatial metrics to account for the non-regular placement of the node points. These terms are included by a straightforward application of the chain rule:

$$\frac{\partial f}{\partial \vec{x}} = \frac{\partial f}{\partial \vec{\xi}}\frac{\partial \vec{\xi}}{\partial \vec{x}} = \begin{bmatrix} \frac{\partial f}{\partial \xi} & \frac{\partial f}{\partial \eta} & \frac{\partial f}{\partial \zeta} \end{bmatrix} \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{bmatrix}^{-1}$$

which combines the computational-space derivatives of the field $(\partial f/\partial \vec{\xi})$ with a spatial metric term $(\partial \vec{\xi}/\partial \vec{x})$, the inverse of the *Jacobian matrix* which describes the local shape of the grid. The field derivatives are approximated by finite differences taken across samples recorded at a few neighboring node points. The spatial derivatives may be approximated using differences over the physical-space coordinates of these same node points.

This numerical approximation of the spatial term introduces error into the calculation of the flow field [Bernard 1988, Mastin 1982]. Grids must be constructed such that this approximation error is minimized; each line of node points should define a smooth curve and these curves should cross almost orthogonally at each node point. Flaws in the grids can induce error or even non-convergence in the flow field calculation process. We shall later see that numerical error in the calculation of these spatial derivatives can also pose a problem in interactive visualization.

Once the solution has been computed to sufficient accuracy at each node point, the field data must be processed to allow evaluation of the results. The solution of a large problem can consume several tens of hours on a large supercomputer, so economy would suggest that maximal information be extracted from each new dataset. The most faithful representation of numerical data is a listing of the sample values of interest, each printed to its full significance. Although correct, this form of presentation is useless for all but the smallest of results. The total lift on a wing or the bending moment on a flap could be presented in this manner, but larger collections of data must be depicted graphically. Such qualitative depictions might be studied by a scientist studying flow behavior, a programmer debugging a solver, or by an engineer improving a vehicle design.

1.1.3 Numerical Flow Visualization

The term *flow visualization* has traditionally referred to the methods used to reveal the structure of empirical flows [vanDyke 1982]. This term has been adopted for the graphical depiction of computed flow fields, an activity more accurately termed *numerical flow visualization.* Techniques for numerical flow visualization have advanced with the increased availability of computational power. Ten years ago, most flow simulations were done in two dimensions, so simple plots were adequate. The simulation of increasingly convoluted three-dimensional flows has introduced the need for new visualization tools.

Visualization packages depict the raw data by constructing geometric *models* or *idioms* [Haber 1988]. Each of these constructs is the result of a functional mapping of some sub-domain of the data volume. The points of this sub-domain may be specified explicitly, such as for a two-dimensional slice taken from a block of node points. The sub-domain might also be selected implicitly, as when creating an iso-valued surface on which all of the selected points have the same interpolated value in some scalar field. The spatial coordinates of the model usually depend only on the spatial dimensions of the data, but the mapping of dimensions need not follow this pattern. A surface plot of a function of two variables $z = f(x, y)$ maps a scalar function value onto the third spatial dimension of the model.

Each point on a graphical model has a number of secondary attributes or *retinal variables* [Bertin 1983] including color, texture, and transparency. These variables may be assigned values which are functions of the coincident data fields, or they may be used to represent depth information in the final two-dimensional image. (Of course, the characteristics of the intended display medium determine which

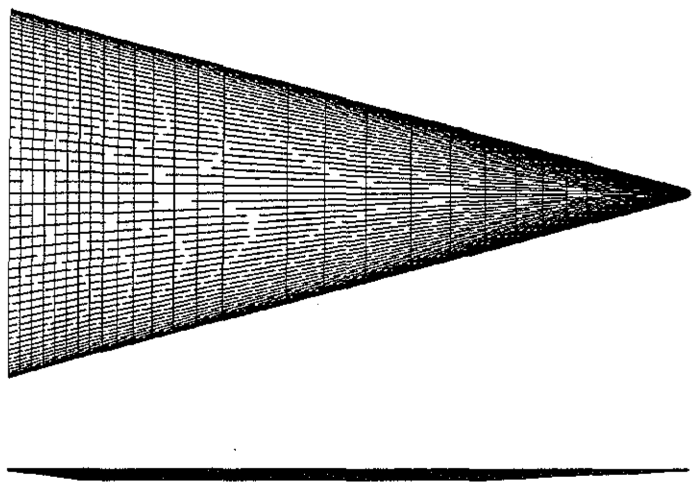


Figure 1.6: Top and side views of the beveled delta wing.

variables are available. Since many professional journals publish only monochrome line-art, color raster stills and video are currently of limited use in the archival publication of scientific results.)

1.2 VECTOR FIELD VISUALIZATION

The usefulness of any visualization method depends partly on how well it satisfies the *Visualization Principle* of Hanson and Heng [1991], which states:

> A useful data depiction must allow the viewer to reconstruct a consistent and relevant [mental] model of the original data.

In this section, we will examine the traditional models used to depict steady three-dimensional vector fields. The poverty of many of these methods with respect to the Visualization Principle motivated my further development of the stream surface method. The figures in this section depict the flow over a beveled triangular plate, flying at Mach 0.3 at an angle of attack of 40 degrees and a Reynolds number of 10^6. This flow was computed by Ekaterinaris and Schiff [1990]. The grid contains a single

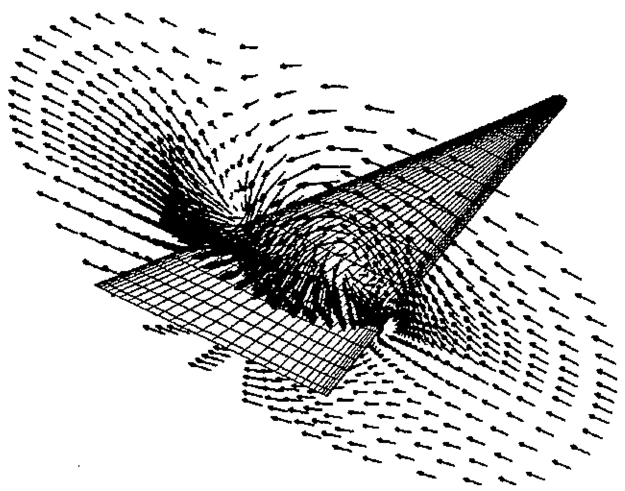


Figure 1.7: Arrow glyphs in a velocity field.

block of $(56 \times 54 \times 70) = 211680$ node points. (This is a small dataset; newer grids may contain a few million node points distributed over tens of blocks.) Primary and secondary vortices are formed over the leading edges of this delta shape. Near the back, the main vortex *bursts* into a tangle of intertwined flow. The solution was computed over only half of the vehicle; these images have been mirrored about the midline symmetry plane.

1.2.1 Arrows

Vector fields are often depicted with collections of discrete tokens or *glyphs* [Ellson and Cox 1988, de Leeuw and van Wijk 1993]. Figure 1.7 shows a collection of arrows placed on a slice through the three-dimensional domain of the flow. Each symbol depicts the magnitude and the direction of a vector quantity sampled at some point in the flow field. Notice how these arrows form a circular pattern surrounding each primary vortex above the vehicle.

This visualization method leaves the majority of the data unrepresented, but filling the whole volume with arrows would be utterly confusing and useless. These simple arrows are also ambiguous for three-dimensional fields; the length and direction of each segment cannot be determined from its two-dimensional image. This problem has led some researchers to place a small arrowhead on each segment [Weston 1987b]. This helps to convey the orientation of each individual arrow, but it can still be difficult to mentally fuse a collection of discrete symbols into an understanding of a continuous field.

1.2.2 Tangent Curves

Several useful methods of vector field visualization are based on the calculation of tangent curves. Each curve is the solution of the *initial value problem* posed by a vector field $\vec{u}(\vec{x})$ and an initial point $\vec{x}_0$. This curve can be approximated by computing a sequence of points $(\vec{x}_0, \vec{x}_1, \ldots \vec{x}_n)$ such that

$$\vec{x}_{i+1} = \vec{x}_i + \int_{t_i}^{t_{i+1}} \vec{u}(\vec{x}(t))dt$$

for a closely spaced sequence of values (t_i). Each adjacent pair of points is then linked with a line segment to produce a piecewise-linear model of the ideal curve. When the vector field is the velocity of a fluid flow, each curve defines the path traveled by a massless advecting particle. Such paths are called *particle traces* or *streamlines.*

Most flow visualization packages allow the user to specify a set of initial points from which a family of streamlines is computed. This set of starting positions is often called a *rake*, a name adopted from the tubing used to introduce multiple streams of smoke into a wind tunnel. Figure 1.8 shows a set of streamlines tracing out the same flow that was depicted in the previous figure. Note that the two vortices are more evident, but that the structure of each vortex is still quite difficult to discern.

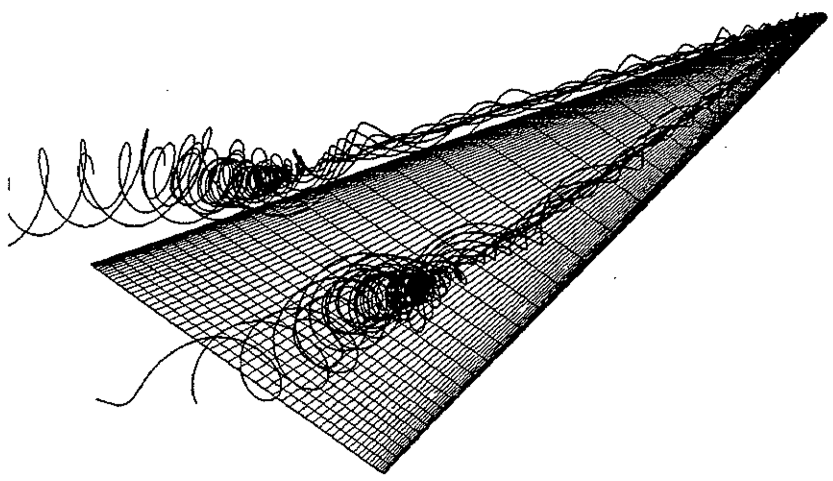


Figure 1.8: Streamlines in the flow above a delta wing.

Streamlines are the most commonly used model for investigating the structure of flow fields. As scientists have begun to study more convoluted flows, this approach has become increasingly inadequate. Some researchers [Ying *et al.* 1987] have expressed these reservations:

> It should be pointed out that even with the relatively good graphics hardware and software at our disposal, the representation of details of the flow structures, and in particular the form of multi-coiled surfaces such as depicted here, remains open to judgment, and is subject to imprecision.

Researchers ought to be working to understand their data; yet they often must work instead to understand their pictures!

Careful use of depth cues can aid in the visual interpretation of streamlines. Intensity cuing, stereopsis, and motion are perhaps the most commonly applied methods. Another aid is the representation of each path, not as a curved line, but as a space-filling and shaded cylindrical tube. These "3D space tubes" [Dickinson 1988] provide shading and occlusion to help the viewer interpret the shapes of the three-dimensional curves depicted in a two-dimensional image. Note that these bent cylinders have been called *streamtubes* by some authors, but this is a misnomer. A true streamtube is the locus of the infinite number of streamlines originating from a continuous and closed loop of rake points. (A streamtube is simply a special case of the *stream surface*!) These true tubes typically change their cross-sectional size and shape as they extend through the flow.

Schroeder *et al.* [1991] devised a model called the *stream polygon.* The deformation of an infinitesimal fluid element is tracked along the length of a streamline. This deformation is depicted graphically by the differing shapes of a sequence of polygons placed at regular intervals along the computed curve. This method conveys an impression of the fluid rotation and divergence along this path, but it shares the difficulty of the other glyph-based methods which force the viewer to perform a "mental interpolation" across a collection of discrete representations within a continuous field.

A related model is the *cloud tracing* technique of Ma and Smith [1993]. This is a depiction of flow within a combustor, with a single streamline forming the centerline of a tube of monotonically increasing radius. This distance was used to indicate the amount of mixing which had occured within the flow. Comparable to this is the turbulence depictions produced by Hin [1993,1994], in which an advecting particle is randomly displaced by an amount proportional to the flow diffusivity. Both of these methods augment a simple streamline model with some method of representing a scalar term obtained from the simulation data.

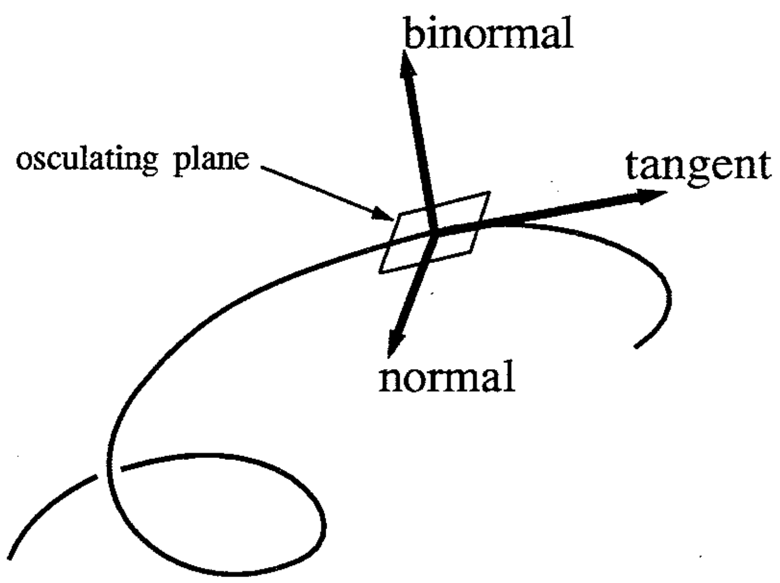


Figure 1.9: The Frenet frame on a three-dimensional curve.

1.2.3 Ribbons

A useful method for conveying the shape of a three-dimensional curve is to construct a narrow *ribbon* which is everywhere aligned with the *Frenet frame* or *moving trihedron* of differential geometry [Kerlick 1990]. This coordinate frame consists of three orthogonal vectors, shown in figure 1.9. These vectors are: the *tangent* to the curve, the *normal* in the *osculating plane*[1] which contains an infinitesimal neighborhood of the curve, and the *binormal* orthogonal to that plane. This coordinate frame is not defined where the curve is straight, and it flips 180 degrees about the normal vector at points of inflection. Bloomenthal [1990] has devised similar coordinate frames which avoid these problems.

Instead of orienting the ribbon with respect to purely differential measures of the tangent curve, Darmofal and Haimes [1992] constructed ribbons which twist about the normal vector at a rate proportional to the local value of the flow *helicity*

[1]osculate – 1. *trans.* to kiss, ... 4. *Math. trans.* to have contact of a higher order with, esp. the highest contact possible for two loci ... *Oxford English Dictionary*

$(\vec{u} \cdot (\nabla \times \vec{u}))$. This produces "corkscrews" wherever the fluid is rotating strongly about an axis defined by the local velocity.

Shirley and Neeman [1989] constructed ribbons by advecting three particles, one to trace the central spine and two others to form the edges. Belie [1985,1987] depicted flow field rotation by computing pairs of streamlines from nearby starting positions. The gap between these curves was bridged with a sequence of polygons to construct a ribbon. In divergent flow this gap can grow quite large, so he constrained the advecting particles to a constant physical-space separation distance. This restriction ensured that the apparent width of the created ribbon in the image would unambiguously encode its orientation relative to the viewing direction; however, the artificial constraint on particle motion casts some doubt on the fidelity of this method.

Ribbons provide information regarding the rotational component of the flow motion near a curved path. These models also provide visual cues to help observers interpret a two-dimensional image. Nevertheless, it can still be difficult to derive an adequate understanding of flow field structure from the image of a collection of ribbons.

1.2.4 Stream Surfaces

Flow ribbons are *ruled surfaces*; they can twist and bend lengthwise, but are flat across their narrow dimension and bordered by only two curves. A *stream surface* is the locus of all the streamlines swept out by particles rooted at every point along an initial segment rake. Stream surfaces can curve in both dimensions, both downstream and across their width. Figure 1.10 suggests that the stream surfaces computed from a few carefully positioned rakes can depict a flow field more effectively than is possible with mere curves. Surfaces can stretch in diverging flow and may fold under the influence of flow curvature. Shading, obscuration, and

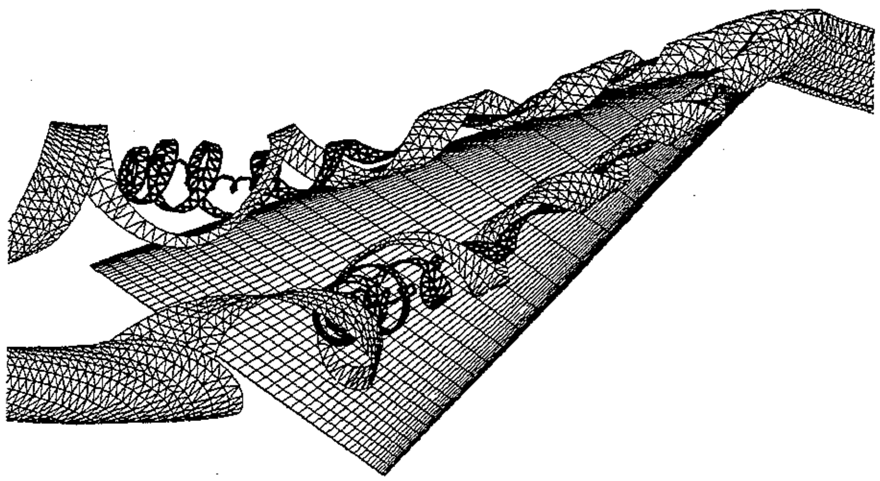


Figure 1.10: Stream surfaces above a delta wing.

texturing provide visual cues which help the viewer to interpret a two-dimensional image. Surfaces can be rendered with variable transparency to mimic the appearance of empirical smoke injections. Alternatively, surfaces can be clearly depicted in line-art.

1.2.5 Rake Placement

Each stream surface is uniquely defined by the position of its originating rake. Rakes must be positioned such that the resulting surfaces clearly depict the interesting aspects of the flow field, but most of the possible rake placements yield models which do not effectively illustrate the structure of the flow.

Some researchers have tried to automate the task of effectively placing the rakes for streamlines and stream surfaces [Globus *et al.* 1991, Helman *et al.* 1989,1991]. The stationary points in the flow, at which the velocity magnitude is zero, are found and classified by the eigenvalues of the local partial derivatives of the velocity field.

These saddle points and spirals are then interconnected by streamlines. This assemblage of points and curves diagrams the structure of the flow field. Helman's software identified the topologically significant curves on the solid boundaries of a vehicle. The program then constructed polygonal models of the *topological separatrices*, that is, the stream surfaces which emanate from these lines and extend into the flow domain.

The automated analysis of flow field data can serve as a useful tool in the visualization of flow field structure. It is an area which deserves much greater attention in the coming years. Nevertheless, interactive placement of the rakes, coupled with the rapid calculation of the resulting surfaces, is essential for the effective exploration of non-trivial flow fields. Buning [1988] reminds us that "graphical analysis is first and foremost an interactive process." Visualization tools must allow the scientist to bring his own expertise to bear on the analysis task; they should not limit a user's access to the data.

1.3 CONSTRUCTION OF SURFACES

Several researchers have written software which can construct models of stream surfaces. This software offers varying levels of speed, accuracy, and level of detail. None have been fully suitable for the interactive exploration of intricate flow fields defined over composite curvilinear grids.

1.3.1 Surface Particles

A very general approach to stream surface construction was devised by Stolk and van Wijk [1991,1992]. They placed a large number of particles in the flow domain. Each of these *surface particles* carried an associated normal vector and thereby represented an infinitesimal portion of an oriented surface. The changing direction of each normal was tracked during the advection of the points through the velocity field. After the particles had been moved over some specified time interval, the points were shaded and rendered. The lighting conveys a sense of the overall motion of the collection. With an adequate number of particles, this conveys the appearance of a surface distorted by the flow.

Unfortunately, this approach can be quite time-consuming. One image of a simple flow field contains 34,847 particles and required 32 minutes to perform the advection calculations on a Sun 3/50 workstation. The surface-particle method depicts the positions reached by a set of particles after all of them have been advected for some interval of time. It represents a *time-surface* of particles placed into the flow and advected *en masse*. A stream surface model, on the other hand, is comprised of all those points visited by a line of advecting particles *during* some time interval. Far less calculation is required, since each integration step of every particle contributes a vertex to the model.

A more serious limitation of the surface-particle method is its failure to adapt the sampling density of the model in response to the varying conditions of the flow field. Some regions of the flow will diverge, and this will yield an underpopulated region of the represented surface. This attenuation is a good mimic of some empirical methods of flow visualization, but it can be a poor method of depicting flow structure.

1.3.2 Stream Function Methods

The particle-based construction methods require the advection of many particles to form a "cloud" with a density sufficient to convey the visual impression of a coherent structure. An alternative method for constructing stream surfaces uses scalar fields which describe the behavior of the flow. The particle method is a *Langrangian* approach, which considers the flow field as one follows each moving point. The alternative *Eulerian* approach considers the flow field from a fixed spatial reference frame.

Paradoxically, it was Lagrange (1736–1813) who is generally credited[2] with showing that the velocity field of a flow in two dimensions can be described by a single scalar *stream function* (ψ), such that the separate components of the vector field can be obtained from the partial derivatives of the scalar field:

$$\vec{u} = [\frac{\partial\psi}{\partial y}, \frac{-\partial\psi}{\partial x}]$$

The stream function field has the interesting property that iso-valued contours in this scalar field are coincident with streamlines in the velocity field. (It is helpful to notice that the velocity vector is simply a ninety degree rotation of the gradient of ψ.) The numerical difference of the stream function values of any two streamlines indicates the total *mass flux* or material transport of the fluid between those two curves.

The stream function concept was generalized to three dimensions by Yih [1957] to describe a steady three-dimensional compressible flow using two coincident scalar fields. Streamlines in a three-dimensional vector field can be described as the intersection of iso-valued surfaces in these two scalar fields.

Kenwright [1992,1993] developed software which computes these *dual stream functions* in a numerical flow field. The software then constructs streamlines by locating the intersections of iso-valued surfaces computed in these two fields. This approach supports the very rapid and accurate creation of streamline models. At each new cell, the two stream functions are computed by evaluating the total mass flux through each cell face. He then found the stream function values (f, g) of the point at which the particle has entered this cell. The exit point, having those same coordinates, is then identified and the process is repeated to advance the particle

[2]Robertson [1965] gives precedence to D'Alembert (1717–1783).

across subsequent cells. The greatest limitation of this method is that features within each cell are represented by a single line segment or polygonal facet. Resolution of smaller structures would require a subdivision of the grid cells.

Some numerical problems currently prevent the pre-processed calculation of the two scalar fields across the entire flow domain. If this difficulty can be overcome, then the two fields could be pre-computed at high accuracy, and then interactively explored with a variation of an iso-surface construction technique. Arbitrary stream surfaces could be constructed as iso-surfaces in a single scalar field which is the algebraic combination of the dual stream-function fields. That is, if we can derive a function which produces a constant scalar value at all points on a user-specified rake, then the iso-surface constructed at that value in the new field will contain the stream surface emanating from that rake. Of course, some provision must be made for trimming the surface to eliminate those regions of the iso-surface which do not lie downstream of the rake and between the streamlines which define the left and right edges of the surface.

Van Wijk [1993,1994] implemented such a scheme, using a variety of methods to calculate the single scalar function in which to compute the iso-/stream-surfaces. Each new scalar field defines a family of nested iso-surfaces, and thus a family of adjacent stream surfaces. Arbitrary placement of the rakes must be accompanied by the calculation of a new scalar field. Some questions remain regarding the accuracy of the methods used to calculate this scalar field; different calculation methods have been shown to produce different stream surfaces from the same rake position in the same original flow data.

1.3.3 Streamline-based Methods

Alternative, and less controversial, methods of surface construction are based on the numerical integration of streamlines. A family of these curves is used to form the framework over which a polygonal model of a stream surface is constructed. Volpe [1989] constructed stream surfaces using a family of neighboring streamlines. Each adjacent pair of streamlines was joined by a sequence of polygons to form a ribbon. Ribbons wider than some threshold were not displayed, thus ripping the surface in diverging regions of the flow.

Belk *et al.* [1993] and Helman *et al.* [1989,1991] also used a ribbon-tiling approach to construct surfaces. Any ribbon which grew too wide was split down its entire length by the insertion of a new streamline rooted at the rake, midway between its older neighbors. The new streamline formed the common edge of two contiguous ribbons which replaced the previous wider ribbon.

Eder [1991] developed a distributed system for computing stream surfaces. After the user positioned the rake for a new surface, the system would compute between fifty and two hundred streamlines using a vectorized integration code running on a remote vector computer. The points on this family of curves were then copied into an array in the local memory of a workstation, a two-pass filtering eliminated points which were packed too closely, and those that remained were irregularly tiled with triangles to create the model.

These methods all use streamlines which are rooted at the rake and extend along the entire length of the stream surface. Since flow fields can diverge greatly, some adjacent streamlines will separate quite widely. Volpe computed a fixed number of streamlines and suppressed the display of regions which were inadequately sampled. Helman and Belk computed a small number of streamlines to form a coarse representation of the surface, then computed additional streamlines to improve the resolution of the poorly resolved regions. Eder created an abundance of streamlines and then reduced the detail of the overly sampled regions.

All of these methods compute each streamline along the entire length of the stream surface, and thereby expend computational effort across parts of the model which are already adequately sampled. The methods of Volpe and Eder can fail to adequately resolve the downstream regions of a surface in divergent flow. Helman and Belk improved the model resolution at the expense of an overabundance of points in the upstream regions.

Krueger implemented a proof-of-concept visualization system using a novel user interface called *VideoDesk* [Krueger 1991,1992]. In this system, a video camera was used to merge a real-time image of the user's hands onto a plane embedded in the three-dimensional space of the computed models. A user could then interactively and intuitively position a rake by dragging its endpoints across this plane with the image of his fingertips. The stream surface model was constructed using *adaptive refinement* [Bergman *et al.* 1986]. This interaction technique improves an initially coarse representation using an iterative sequence of calculation and redisplay. This gradual improvement of the model continues as long as the input conditions remain unchanged. The initial stream surface was constructed with a polygonal tiling of adjacent pairs of streamlines, as in the manner used by Volpe. In later phases of refinement, any ribbon which exceeded a specified width was truncated. The deleted downstream portion was then replaced by two narrower ribbons separated by a common new streamline. This new streamline was computed from a seed point placed at the middle of the truncated end of the wide ribbon.

An extension of this method was devised by Max, Becker, and Crawfis [1993], who began with a polygonal rake. A family of streamlines was then constructed from points placed at regular intervals across the two dimensions of this user-specified origin. An interconnection of these curves was then used to construct a set of tetrahedra. Overly large tetrahedra were split by the insertion of new advecting particles. This collection of cells was then rendered with partial translucency to mimic the appearance of a column of smoke.

Krueger's hierarchical splitting of ribbons produces a more equitable distribution of points in the model than does the full-length splitting method used by Helman and Belk. Neither approach allows for the merging of adjacent ribbons which may later grow too narrow in a convergent region of the flow field. Once a wide ribbon is split, the new streamline extends to the far downstream end of the surface. Thus, converging regions of flow become overpopulated by sample points. This oversampling wastes computational resources and can create excessively many small polygons in the model.

1.4 ORIGINAL CONTRIBUTIONS

Previous methods of calculating streamlines have been too slow for interactive use or else they do not provide acceptable accuracy or detail. Previous methods of constructing stream surfaces have not adequately allocated the calculation effort to produce a well-distributed set of sample points over the surface. The adaptive refinement methods are slowed by the need to fully repaint the image after each improvement of the model. I have resolved some of these problems with a set of algorithms which can allow the interactive exploration of flow field data using streamlines and stream surfaces.

1.4.1 Thesis Statement

I have devised new algorithms for the rapid and robust advection of particles through vector fields defined over composite curvilinear grids. I have devised a new algorithm which interleaves the advection of a set of particles to produce adaptively sampled stream surface models. Taken together, *these methods make stream surfaces a viable tool for interactive numerical flow visualization.*

More specifically, I have devised several algorithms and implemented these in the context of an interactive flow visualization application. This software can construct streamlines and stream surfaces in vector fields defined over multiple-block curvilinear grids. The dataset is presumed to contain no more than about two million node points, such that the node position data and the vector field samples can reside entirely in the local memory of a large workstation. I further assume that the workstation is capable of performing between one million and five million floating-point operations per second, and that it is supplied with graphics hardware capable of rendering a few thousand shaded polygons per second.

The performance tests in this thesis were all recorded on a Silicon Graphics 320-VGX workstation. This machine has dual processors running at 33 megahertz, and it is configured with 64 megabytes of memory. This hardware was about four years old at the time of these tests. Comparable performance could have been obtained from a midrange configuration of the SGI Indigo family. At the time of this writing (1995), such equipment costs in the neighborhood of $25,000 US.

Using these resources, my software is able to calculate and display a newly positioned streamline in about one second. The display of partial results, combined with the ability to abort and restart streamline calculations, allows the interactive placement of streamline seed points at a useful interactive rate of five to ten iterations per second. (The limiting factor here is the necessary redisplay of the static portions of the scene.) Stream surfaces are more time-consuming than individual streamlines to compute and display, but the construction algorithm described here minimizes this extra burden and allows the construction of complete stream surfaces at the "near interactive" rate of five to ten seconds to complete a typical surface in a typical flow field.

1.4.2 The "Flora" Application

I have implemented my advection and construction algorithms as part of a visualization package called "Flora." This application and its underlying implementation are described in chapter 2. (The "Flora Users' Guide" is reprinted in Appendix B.)

Flora was built within a programming system called "SuperGlue" [Hultquist and Raible 1992]. This environment was built by the author in cooperation with Eric Raible. It was written primarily in an object-oriented dialect of Scheme, using a heavily modified version of an interpreter originally written by David Betz [1988]. Scheme serves as the command language for a central executive process which receives input from the user and calculates geometric models of the data. Compiled primitive functions are used in this calculation. The resulting models are placed in a display list, which is accessed by a separate process that maintains the images of these models on the workstation screen.

The interpreted programming environment of SuperGlue was used to evolve the user interface of Flora during extensive trial use. The interactive framework also provided an ideal scaffolding for algorithm development, implementation testing, and performance measurement.

The source code for Flora and SuperGlue may be obtained via the World-Wide Web from the server maintained at NAS, the Numerical Aerodynamic Simulation Facility at the NASA Ames Research Center at Moffett Field, California. The URL for the NAS homepage is `http://www.nas.nasa.gov`. From this page, follow the hyperlink for "Software distribution" and then find "Flora" in the listing of available packages. All code obtained from this server is to be used only by the direct recipient; redistribution of this software to third parties is presently disallowed by NASA regulations.

1.4.3 Mixed-Space Integration

Streamlines are usually calculated by numerical integration through interpolated physical-space vector field samples, but the interpolation function is usually defined in the computational-space coordinates of each query point. Every time an interpolated field value is needed, the interpolating function must be inverted to find the computational-space coordinates which correspond to the specified physical-space coordinates of the query point. These grid-local coordinates are then used as the weights in the interpolation of the vector field. This conversion of query point coordinates greatly limits the speed of streamline construction.

An alternative method converts the vector field sample at each node from the original physical-space coordinates to the equivalent computational-space representation [Eliasson *et al.* 1989, Shirayama 1989]. The streamlines are then computed through the cubical cells of computational space and the resulting curves are mapped into physical-space coordinates for display. Since the interpolation and the integration are expressed in the same regularly sampled space, the calculation of the traces can proceed much more rapidly.

The great speed of the computational space method is attractive, but irregularities in the physical placement of the grid node points can introduce error into the conversion of the vector field samples. In chapter 3, I describe a novel *mixed-space* approach which can efficiently compute curves while still preserving accuracy in the presence of grid flaws. The rapid computational-space method is used where the node points are well positioned. The slower and more robust physical-space method is used where flaws in the grid would otherwise reduce the accuracy of the computed streamlines. The mixed-space method computed a set of test streamlines in 83% of the CPU time required by the more commonly used physical-space method, while maintaining comparable accuracy. In smoother grids, the mixed space method delivers the full speed of the computational-space method.

1.4.4 Node Tagging

Multiple-block CFD datasets include a single integer value for each node point. This mask, called the *iblank number*, indicates which node points carry valid sample data. They also indicate which other block of samples, if any, overlaps the current block in the neighborhood of a given node. This simple tag does not provide sufficient information for rapid calculation of streamlines. I have replaced the iblank values with a more descriptive form of grid annotation. Each node is still associated with a full word, but this *node tag* is divided into a several single-bit flags and a short index into a small secondary array.

One of the flags indicates if the grid lines in each computational-space dimension pass smoothly through a given node point. This indicates that the transformation from the physical to the computational coordinate space can be computed with acceptable accuracy. The logical conjunction of this bit for the eight vertices of each cell is recorded in a second flag in the tag word of the lowest-indexed vertex of that cell. This provides a rapid means of determining the local quality of the grid and the consequent reliability of the computational-space vectors which are to be interpolated within each cell.

1.4.5 Exploiting Spatial Coherence

In chapter 4, I present several methods which can further enhance the efficiency of numerical integration. The cumulative effect of these enhancements allows the interactive placement of accurate streamlines.

cell caching : Each field value interpolated at a given query point depends upon the flow field samples recorded at the eight vertices of the enclosing cell. These samples can be copied from the large arrays of raw field data into a small local buffer. This copying of the flow data isolates the interpolation code from the varying formats of the data files, it improves the speed of interpolating subsequent samples within a given cell, and it provides a convenient mechanism for on-the-fly evaluation of new vector field quantities. I apply the *working set* concept of virtual memory systems to the problem of cell caching, and demonstrate the relative performance gains to be had from increasingly larger cell caches. In a simple example, cell caching reduced the time to compute a set of streamlines by 48%.

computational-space extrapolation : When calculating a streamline in physical space, the computational-space coordinates of each query point must be found. This is usually done with a Newton-Raphson iteration which is begun from the computational-space location of a previous query point. By extrapolating along the previously computed curve, we can produce a better computational-space position from which to begin the iterative point-finding method. This improvement trims a further 11% from the time required to compute streamlines.

donor points : The separate blocks of a composite grid allow more effective placement of node points around a vehicle. In the overlap regions, the position of each node point can be described in the computational space defined by the neighboring block. These computational-space *donor points* are used by the flow solver to interpolate sample values between adjacent blocks. The points can also provide a rapid means of continuing the calculation of streamlines from one block and into the next. I describe how each donor point can be calculated from the original node position and iblank field data. I also demonstrate that computational extrapolation eliminates 8% from the CPU time consumed by this task.

implicit connection : Some blocks are bent or folded such that node points along one face of the block are coincident with other nodes on the exterior of this same block. A torus is a simple example from the world of two dimensional geometry. These *self-abutting* blocks are typically not annotated with iblank values to indicate this geometric connectivity. Because of this, many visualization packages fail to compute streamlines across these internal boundaries in the flow domain. I have devised a simple method which allows the efficient resumption of streamlines across these false boundaries.

With the algorithm improvements described above, streamlines can be computed through a single block in less than half of the time required by an unimproved physical-space method. Transitions between blocks are also handled more quickly, yielding an significant improvement over any method which might use iblank numbers alone.

1.4.6 Constructing Stream Surfaces

Images of streamlines are often difficult to interpret. Stream surfaces can more clearly depict the structure of a flow field, but the rakes must be positioned interactively for these models to be truly useful. Previous work in constructing stream surface models has been limited in efficiency, accuracy, or resolution. This has limited the adoption of stream surfaces as a tool for numerical flow visualization.

In chapter 5, I present a novel method for the construction of stream surfaces. The rake is discretized into a set of closely spaced particles. This sequence of points is then adaptively advected such that the collection is held roughly orthogonal to the local flow direction. The sampling density along this cross-flow curve is adjusted as the points are moved downstream. Triangles are added to the downstream edge of the growing surface, and these new polygons are rendering into an otherwise static background image.

This *adaptive orthogonal advancing front* technique uses a single pass of the particles through the flow field to produce a good distribution of points over the surface. Equitable sampling of the surface leads to increased performance, since fewer particles, fewer integration steps, and fewer polygons are used in the construction of the model.

1.4.7 Better Flow Visualization

Stream surfaces can be more effective than streamlines for the visualization of fluid flows. The most convincing proof of this assertion is the use of my software by NASA and Air Force scientists for the exploration of their data and the presentation of results to their peers. The sixth and final chapter shows some images produced using this software.

1.5 SUMMARY

The simulation of fluid flow begins with the construction of a grid of node points. A set of boundary conditions is imposed on some of these points and the remainder are assigned values which form a solution to the partial differential equations which govern fluid flow. The computed flow must then be visualized to judge the accuracy of the simulation or the effectiveness of a proposed vehicle design. Three-dimensional steady vector fields are often depicted using collections of streamlines, but stream surfaces offer a much more effective means of visualizing intricate flow fields.

The most effective use of stream surfaces is obtained by enabling scientists to interactively drag the originating rakes through the flow domain. I have developed rapid and accurate methods for advecting particles through steady three-dimensional vector fields. I have teamed this numerical software with an efficient and robust method for the construction of stream surface models. I have implemented these algorithms within a software package which can support the interactive exploration of CFD datasets. This software has been used by scientists at the NASA Ames Research Center and the Wright-Patterson Air Force Base. This experience has demonstrated the value of these improvements.

CHAPTER II
IMPLEMENTATION

I implemented various stream surface construction methods in several prototype applications. I reported some of this work in a previous paper [Hultquist 1990] and images created using this early software were published in a research paper by Ekaterinaris and Schiff [1990]. Each of these prototypes collapsed under its own weight before the complete set of desired features had been implemented. In cooperation with Eric Raible, I then developed an object-oriented and interpreted programming environment called "Superglue." Using this platform I was able to complete my research on stream surface construction and implement a deliverable application. This chapter describes the programming environment and the "Flora" flow visualization tool.

2.1 SUPERGLUE

This section outlines some of the significant features of the programming environment.[3]

2.1.1 Scheme

Superglue contains over 1.2 megabytes of source code, divided between the languages C, FORTRAN, and Scheme. Scheme is a small dialect of LISP. As such, it offers interpreted execution, garbage collection, dynamic type-checking, first-class

[3]A description of SuperGlue has been published in the paper "Superglue: a Programming Environment for Scientific Visualization" [Hultquist and Raible 1992].

treatment of functions, and a powerful macro facility. These features are not typically supported by the more traditional choices of FORTRAN, C, or even C++.

Scheme is concise; a given algorithm implemented in Scheme often requires much less text than its equivalent in many other languages. For example, the function that computes the magnitude of a vector would typically be implemented in C with these lines:

```
float v_magnitude (int n, float *vec)
{
    int i;
    float sum = 0.0;
    for (i=0; i<n; i++) {
        sum += (vec[i] * vec[i]);
    }
    return(fsqrt(sum));
}
```

In Scheme, this same calculation can be implemented by:

```
(define (v-magnitude vec)
    (sqrt (apply + (map * vec vec))))
```

The Scheme version can accept lists of integers or floating-point numbers, and the length of the list does not need to be provided explicitly. The brevity of Scheme, which here reduces nine lines to two, is invaluable when used to reduce nine hundred lines to two hundred.

2.1.2 The Interpreter

Another advantage of Scheme is that it is usually interpreted, rather than compiled; this allows for rapid development of code. Complex functions can be built piecemeal and tested repeatedly, without the delays of compiling, linking and restarting the application. Programmers are able to test their code more easily and to compare alternative solutions to a given problem.

Superglue is based on the "Xscheme" interpreter written by Betz [1989]. This software compiles Scheme expressions into the bytecode language of a simple stack-based virtual machine implemented in C. This interpreter originally supported little more than the features described in the *de facto* language standard [R4RS 1990]. We have extended the system to support the additional requirements of visualization and large-scale programming. We built a debugger, rewrote the garbage collector, and added support for incremental loading of compiled foreign functions. We have also constructed an object-oriented class hierarchy which implements the major components needed for interactive visualization.

Much of this extension effort could have been avoided through the use of a commercial system such as ParcPlace Smalltalk or Allegro Common LISP. We choose to use a freeware system instead of a commercial platform since many of our scientist-clients are unwilling or unable to spend money for licensing. It is our further hope that widespread distribution of the full source code will result in more rapid growth of the system. Superglue is available on the World-Wide Web, from the site maintained at the Numerical Aerodynamic Simulation Facility at NASA-Ames (`http://www.nas.nasa.gov`).

2.1.3 The Garbage Collector

Scheme systems automatically reclaim storage which is no longer in use; that is, the *garbage collector* reclaims the storage occupied by data structures which are no longer accessible from the namespace of the interpreter.[4] This liberates programmers from the responsibility of explicitly deallocating the storage which is no longer used by their programs. This avoids the gradual accumulation of consumed storage which is no longer in use, yet not released for possible reuse (a *memory leak*). It also avoids

[4]For a survey of garbage collection technology, see Wilson [1992].

the even worse problem of inadvertent release of storage which is still being accessed from elsewhere in the program (a *dangling pointer*).

We found that excessive wall-clock time was being consumed by the mark-sweep algorithm used in the original implementation of Xscheme. This collector would repeatedly traverse the data structures of our large and mostly unchanging class hierarchy. To improve the responsiveness of the system, we replaced the original collector with a *stop-copy* implementation. Under this approach, when the system has exhausted the available free memory, the execution of the program is temporarily suspended. All memory allocated by the program is then scanned and those data structures which are in use are copied to a second region of free storage space. Unused structures are ignored and the space which they occupied is then marked as available for the next collection phase. The class hierarchy was placed in a non-collected block of memory, called the *root buffer*. Only the much smaller collection of short-lived data structures is repeatedly copied between two *working buffers* (or *semi-spaces*) of active storage.

Simple reclamation of unused memory is acceptable for most data, but some items require explicit action (or *finalization*) to properly deallocate the system resources which these values represent. For example, some Scheme data items represent windows on the workstation screen. When one of these is collected, the corresponding window also should be destroyed. We extended the garbage collector to maintain a private list of items for which explicit *destructor functions* have been defined. When the garbage collector completes its scan of active storage, collected objects on this list are transferred to a namespace-visible list of *reclaimed objects* which are ready for explicit destruction. This approach requires only a single scan of the list of registered items after each invocation of the garbage collector. The actual execution of the destructor functions is deferred until the system is otherwise idle.

2.1.4 The Foreign Function Interface

Most of the manipulation of flow field data requires the speed which is available from compiled low-level languages such as C and FORTRAN. We have extended Xscheme to support the *incremental loading* of *foreign functions* written in these languages. Briefly, new routines are compiled into the object code of the workstation. These binary data are linked against the symbol table of the application, and these routines are then loaded into memory. Finally, entry points to these new functions are entered into the global symbol table (the *obarray*) of the Scheme interpreter. The new functions may then be called from the interpreter in the same manner as any statically linked routine.

Our interpreter maintains its own call stack. This simplifies the implementation of the system and enhances its portability. Foreign functions must remove their arguments from this stack, and push their single result onto the stack upon completion. Removing an argument from the stack involves ensuring that the stack is non-empty and that the uppermost item is of the desired type. The value of each argument must then be copied from the Scheme data item into a local variable in the format required by the implementation language of the foreign function. Once the body of the function has been executed, the result is converted into its equivalent Scheme representation and this item is returned to the interpreter. All of these data-handling actions have been encapsulated in a number of macros and utility functions.

2.2 THE OBJECT SYSTEM

Superglue gains much of its usefulness from a large hierarchy of class definitions. This structure provides organized access to a collection of pre-defined data types and operations on items of those types. The Superglue class mechanism most closely resembles that used in the language Smalltalk [Goldberg and Robson 1983].

(This includes single inheritance and dynamic method-lookup based on the message and the class of the receiver.) This section reviews some of the concepts of object-oriented programming and then describes the pre-defined classes of Superglue.

2.2.1 Object-Oriented Programming

Good software (and bad software) can be written in any language, but object-oriented languages encourage good practice by encouraging (or enforcing!) three design principles [Booch, 1991]:

abstraction : Users of assembly language must concern themselves with the words and bytes of computer memory. In FORTRAN, numbers and arrays form the basis of discourse. An object-oriented language encourages programmers to work with conceptual entities more closely related to those of the target application. The data types defined in Superglue include the computational grids used in CFD, the windows and events found in a graphical user interface, and the remote connections used in distributed computing.

encapsulation : The ease with which a system can be extended is inversely related to the number of interdependencies built into that system by the original designers. Objects can interact only through well-defined interfaces; implementation details within each object are protected from external access. This helps reduce the overall complexity and thereby facilitates further development of the system.

inheritance : A major problem of code development is its agonizingly slow pace. Object-oriented languages provide mechanisms which encourage the re-use of previously written software. The features of the old software can be *inherited* into a new package, which can then augment those features to satisfy new requirements.

The three principles of abstraction, encapsulation, and inheritance can increase programmer productivity and can help one to produce software systems which are easier to maintain. The hierarchy itself serves as an organizing framework which guides the future growth of the software investment.

2.2.2 Classes and Instances

An object (or *instance*) is a self-contained collection of data which is a member of some *class*. The class of an object defines the structure and the behavior of that object. For example, the class `<stack>` [5] is defined by this expression:

```
(defclass <stack>
  (instance top))
```

The single *instance variable* (`top`) forms the internal state of each instance of this class. It holds a pointer to the first link (or *cons cell*) in a chain of previously inserted items.

In traditional imperative programming, we say that a function is *called.* In object-oriented programming, instances *send messages* to other instances. The receiving instance then finds and executes the executable code (the *method*) which corresponds to that message and which implements the requested behavior. This change of terminology shifts the conceptual focus from a single abstract flow-of-control to the cooperative actions of a set of distinct and autonomous entities.

A Superglue method is written very much like any other function definition in Scheme. The syntax changes slightly to include the name of the class with which the new method is to be associated.

```
(defmethod <stack> (PUSH item)
  (set!  top (cons item top)))
```

[5]Class names are enclosed within angle brackets as a syntactic convention.

This method uses the built-in allocation function `cons` to create a new link in the chain, and it replaces the previous value of `top` with a pointer to the newly created cell.

2.2.3 Subclassing and Inheritance

Classes are nested hierarchically, such that a child class or *subclass* of another *inherits* and then augments the format and the behavior defined by its ancestors in the class hierarchy. All classes have a single superclass, except the class `<object>` which is the root of the class inheritance tree. A class may have any number of subclasses.

An instance of `<counting-stack>` has same internal state as an ordinary stack, plus one more instance variable which records the number of items which that instance contains.

```
(defclass <counting-stack>
  (super <stack>)
  (instance size))

(defmethod <counting-stack> (PUSH item)
  (set!  size (+ size 1))
  (send-super 'push item))
```

This new `push` method increments the count then re-sends the message to this same instance, but now in the guise of a generic stack. Only this difference in behavior need be specified in the implementation of the subclass. Shared format and behavior is provided by the mechanism of class inheritance.

But what happens when a message is sent to an instance which has no corresponding method? This triggers an error-handling routine, which sends the receiving instance a new message: `does-not-understand`. This message is handled by all objects, in a method defined by the uppermost class `<object>` from which all instances inherit. In the language Smalltalk-80, this method invokes the debugger. In Superglue, the method first requests from the receiving instance a *delegate*; that is, some other instance which is able to handle the original message in place of the original recipient. An instance representing a streamline calculates curves through vector fields. Messages sent to this instance and intended to specify a color are delegated to the instance of `<visual>` which handles the graphical presentation of this curve. The debugger is invoked only when no delegate is provided by the receiving instance.

Some languages, notably C++ and Common LISP, allow multiple inheritance, by which a single class can combine the characteristics of more than one superclass. A streamline class could then be a subclass of the class of computed models and also of the class of drawable entities. Other languages, such as Self [Ungar and Smith 1991], have a weak notion of class and handle all similarity of behavior using delegation. In Superglue, the combination of single inheritance and delegation provides most of the benefit of multiple inheritance with very small conceptual and implementation overhead.

2.2.4 The Class Hierarchy

Superglue currently offers the programmer a hierarchy of about 200 classes and over 2200 methods. The class hierarchy is built atop the fundamental classes `<object>` and `<class>`, which define the behavior of all of the instances and classes in the system. The class system of Superglue is *reflexive*. This means that the class hierarchy is implemented using objects. All classes are objects, thus `<class>` is a subclass of `<object>`. All classes are themselves instances of the class called `<class>`; this class is an instance of itself. Actually, each class is the sole instance of its *meta-class*, which is a subclass of `<class>`. (All this can be safely ignored most of the time!)

All instances in the system inherit the methods defined in the class `<object>`. Any instance can be assigned a destructor function. Any instance can report its class and can be assigned a name. Each instance can produce a list of the messages to which it will respond.

A class maintains the list of messages which its instances will accept, and the methods which implement the corresponding behaviors. Each class also maintains a list of all of its instances which have been assigned names. Every class can return its superclass or the entire chain of superclasses, ending with the most generic class `<object>`.

The remainder of the class hierarchy is divided into subtrees which collectively implement the generic features needed for interactive visualization. We intend to grow this hierarchy over the next few years to provide an increasingly larger suite of reusable code.

The `<structure>` subtree implements the typical data structures learned by all beginning programmers. The hierarchical implementation underscores the relationships among these abstract types. A `<collection>` organizes a number of objects, either as an unordered `<bag>` or as an ordered `<sequence>`. A `<bag>` which contains no duplicates is a `<set>`, and a `<set>` which associates a secondary value with each unique item is a `<dictionary>`. The ordered collections are implemented by the linked `<list>` and by the indexed `<vector>` subclasses. Variants of `<list>` include `<stack>` and `<queue>`. This subtree also contains support for mathematical data, including `<matrix>` and its subclass, the 4×4 `<xform>` used in computer graphics.

The `<system>` subtree provides object-oriented wrappers around many of the UNIX system resources. For example, we support `<file>` and `<directory>` objects, and instances of `<time>` and the light-weight process `<thread>`. This wrapping of the operating system resources greatly simplifies their use. We have also "objectified" the run-time environment of Superglue itself. The Scheme debugger is implemented using objects; each error condition is represented by an instance of one of the subclasses of `<error>`. Superglue may be optionally configured such that characters, numbers, text strings, and vectors all may be treated as instances of their respective classes.

The classes in the `<viewer>` subtree provide an object-oriented packaging of an asynchronous process which updates images on the workstation screen. Models are represented by instances of the class `<visual>`. A visual is contained within one or more instances of `<visual-group>`, which are themselves contained within one or more instances of the class `<visual-sequence>`. Each sequence can display an animation of models within one or more windows.

2.3 FLORA

Flora is a flow visualization package developed within Superglue. It supports the interactive manipulation of the originating rakes for streamlines and stream surfaces, and responds to this input by the repeated calculation and display of streamline and stream surface models.

2.3.1 Architecture

Flora is implemented with two processes sharing a single address space (figure 2.1). The main process executes the Scheme interpreter and is responsible for the user interface and the construction of models. These models are placed in a *display list*, a collection of models which are to be depicted in one or more windows. The main process is usually blocked on a `select` system call, which returns only when input is available from one or more sources. When an input item is found, the

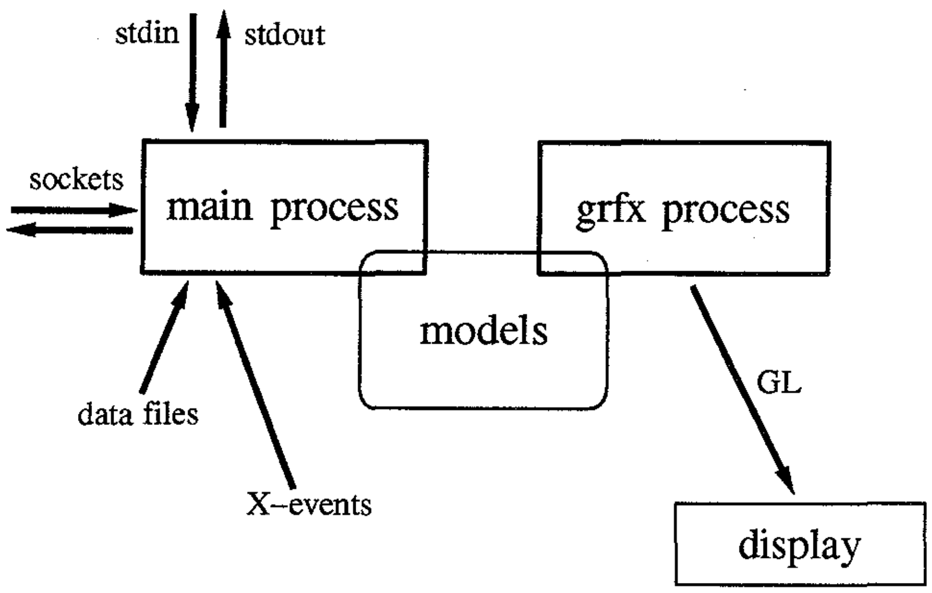


Figure 2.1: The architecture of Flora.

main process reads that input, processes it, and possibly modifies the display list. A graphics process then updates the screen images in response those changes.

When a new model must be constructed, this work is performed by the main process with a majority of the work taking place within primitives written in either C or FORTRAN. Models are placed in the display list when they are first allocated. As the model is constructed, new points, line segments, and polygons are placed into the display list by the main process, and then rendered into each scene by the graphics process. A lock is used to prevent significant modification of the display list while it is being traversed by the graphics process. When a rake is moved, the lock must be set by the main thread before the existing model may be reset to zero length. The affected scenes are then fully repainted by the graphics process.

2.3.2 Data Management

The data files produced by CFD simulations may consume tens or even hundreds of megabytes. If we were to load these numbers into memory in the usual fashion (using the UNIX library functions `malloc` and `read`), we would rapidly exhaust the available system swap space on a typically configured workstation. This fatal situation can be avoided by *memory-mapping* the contents of these large files directly into the program's virtual address space. In some sense, these large disk files become a read-only part of the system swap space.

Pointers to these data fields are encapsulated within instances of the `<field>` class. All fields defined over the same domain are held in an instance of the class `<bundle>` [Butler and Pendley 1989]. A multiple block grid is represented by a group of bundles. This structured collection of fields is the central component of a database of fields which have been read from the grid and solution files. These initial bundles are then augmented by new ancillary fields which are computed from the original data values. Alternatively, smaller bundles may be *extracted* from the bundles of the original flow data [Globus 1992]. For example, lower-dimensional sub-bundles define the geometry of the models which will be displayed to the user.

This arrangement gives us the best of both worlds: a Scheme instance with its supported methods, combined with an efficient binary data representation to be processed by foreign functions. This combines the high-level bookkeeping needed for an intuitive user interface with the speed required for the computationally intensive creation of new fields and models.

A vehicle surface is described by several two-dimensional slices, taken from blocks in the original grid. These subsets of the grid are then displayed to represent the vehicle surfaces. Each of these *gridplanes* is specified by a block number and a range of indices in each grid dimension. One index is held constant, typically the lowest-available index in the third dimension of the grid. Care must be taken here, since some researchers label their dimensions (`i,j,k`) and others use (`j,k,l`). Another source of confusion is the indexing from 1 used in FORTRAN flow solvers as opposed to the indexing from 0 used in the internals of most visualization software. Finally, one must remember that arrays are stored in column-major order in FORTRAN but the indexing is transposed to row-major order in C and C++.

A gridplane is a true subset of the original samples, but the new bundle need not have integer values for the computational-space coordinates of its points. For example, a streamline is a one-dimensional bundle embedded in the three-dimensional bundle-group of the flow domain. These points have been placed at a sequence of real-valued positions in the computational spaces defined by the three-dimensional grid blocks. In both cases (typified by gridplanes and streamlines), the sub-domain consists of a set of points with coordinates in the computational space defined by each block of the enclosing *domain bundle.* This extraction can be carried to multiple generations, as in the case of contour lines computed on a scalar iso-surface. Each new bundle is a subset extracted from an earlier bundle of possibly greater dimensionality.

2.3.3 The User Interface

Within NASA, most numerical flow visualization is done using PLOT3D [Buning and Steger 1985]. PLOT3D reads two data files: an *X-file* which lists the physical-space coordinates of each node point and a *Q-file* which records density, momentum, and energy sample values at each node. PLOT3D can then construct and display several types of models from any one of dozens of derived flow field quantities.

Flora can read data and solution files in the PLOT3D format. Flora also accepts a subset of the PLOT3D command language, including the commands for reading data files, extracting gridplanes, placing the rakes for streamlines, and viewing the resulting models. Once the scientist has finished specifying the input data and the visualization parameters, Flora creates a window in which the constructed models are displayed.

Flora adds two new features which are not provided in PLOT3D. Flora can compute and display stream surfaces, and its rakes may be repositioned interactively. While a rake is moved, new streamlines or stream surfaces are repeatedly computed and redisplayed. Rapid response to interactive repositioning of the rakes allows the user to thoroughly explore the data and to ascertain the structure of the flow.

2.3.4 Repositioning a Rake

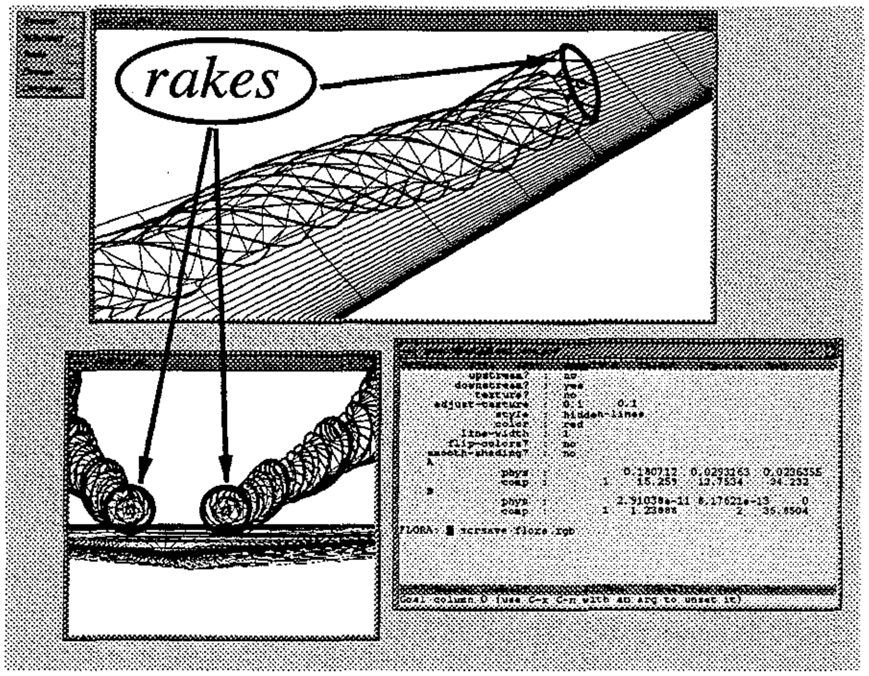
In Flora, a stream surface rake may be a circle, a cross, or a line segment. The segment is usually straight in physical space, but if the two endpoints lie within the same block, then a geodesic in computational coordinates may instead be selected. Each shape serves as the origin for a stream surface, which may be extended downstream in the flow field or (with negative integration timesteps) in the upstream direction. A rake may also be used to spawn a family of streamlines, with individual seed points placed at regular intervals along the rake.

The user may reposition either endpoint of a curve or line rake, or may translate the entire rake. While a rake is in motion, the stream surface or the family of streamlines is repeatedly computed and displayed.

Flora displays the models in an arbitrary number of windows (figure 2.2). Each window has its own viewing direction and magnification. Users may move a rake in one scene at high magnification while viewing the resulting model from another angle in a second window. This approach allows the precise placement of several rakes in a few minutes, a vast improvement over what has been previously possible with indirect control and less rapid system response.

2.4 INTERACTIVITY

Experience with early versions of the software confirmed the critical importance of interactivity in allowing effective visualization of the flow data. Three



```
     upstream? :  no
   downstream? :  yes
      texture? :  no
adjust-texture :  0.1      0.1
         style :  hidden-lines
         color :  red
    line-width :  1
  flip-colors? :  no
smooth-shading? : no
A
          phys :           0.180712  0.0293163  0.0236355
          comp :        1    15.259    12.7534     34.232
B
          phys :        2.91038e-11 8.17621e-13        0
          comp :        1   1.23888          2    35.8504

FLORA> scrsave flora.rgb

Goal column 0 (use C-x C-n with an arg to unset it)
```

Figure 2.2: The user interface of Flora.

components of the application required attention to ensure that rapid response to user input was maintained. These were: intuitive techniques for the positioning of the rake, algorithms for the rapid construction of accurate models, and a method for the efficient depiction of partially computed results.

2.4.1 Placing the Rake

The user needs to be provided with a natural and intuitive method for adjusting the location of the rakes within the flow domain. This should allow the user to concentrate on the data, with the operation of the software introducing minimal distraction.

Since few workstations are equipped with three-dimensional input devices, some mapping must be established between the two-dimensional input from the mouse and the three dimensions of the flow field domain. Flora uses a method devised by Nielson and Olsen [1986], in which each two-dimensional shift of the

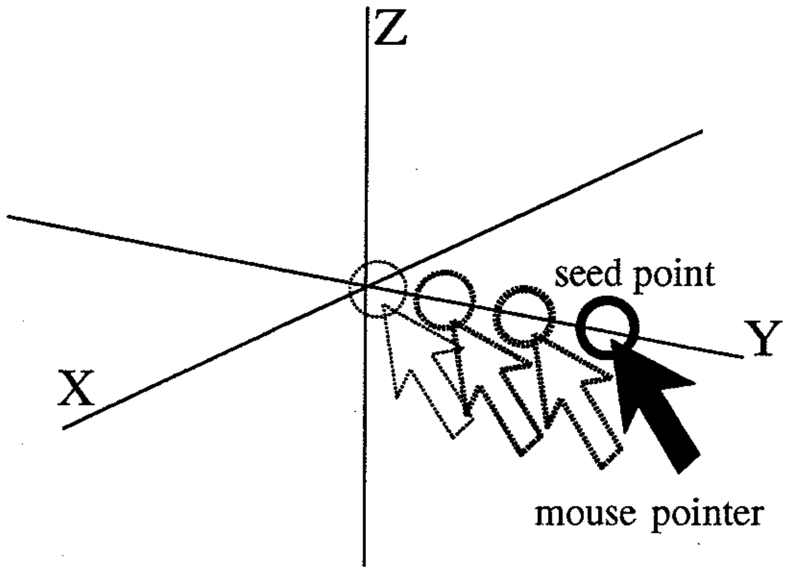


Figure 2.3: Moving a seed point in three dimensions.

mouse is mapped to a one-dimensional movement along one of the three orthogonal physical-space axes (figure 2.3). This method seems to work fairly well, although additional research on three-dimensional interaction techniques is certainly needed.

2.4.2 Constructing the Model

The stream surface must be constructed as rapidly as possibly in response to each new placement of the rake. This topic is the primary concern of this dissertation.

2.4.3 Displaying the Model

The application must quickly display a sequence of interim computed models. When the user moves the rake, any surface computed from the earlier rake position must be destroyed and remaining objects must be redisplayed. As the currently selected model is computed or when the user changes the viewing parameters, the new scene must be continuously redrawn. Once the new model has been completely constructed, then the scientist will often wish to rotate the scene to obtain a better understanding of the shape of the constructed items. This interactive rendering of highly detailed models typically requires relatively powerful graphics display hardware.

Some early versions of the Silicon Graphics graphics library allowed the drawing of wireframe surfaces into the "overlay" planes of the display, while still performing depth-buffer tests of these new graphical items against an existing static background image. This allowed the repeated drawing of new wireframe stream surface models into very complex scenes. This was never a vendor-approved behavior, and it is no longer available. Now a judicious reduction of the image quality is mandatory for maintaining an adequate response rate for surfaces repeatedly constructed and drawn into cluttered scenes.

Successive refinement of a coarse model requires the successive repainting of the entire scene. As the scene becomes more cluttered, the response rate begins to decline. The surface construction method developed for Flora generates the model in a single pass. Construction begins at the rake, and incrementally creates the model at a fully acceptable level of detail. This new model is drawn incrementally into an otherwise static background scene.

2.5 SUMMARY

Superglue encourages the reuse of source code and it supports the rapid implementation of new functions. The two main features of this system are a Scheme interpreter and a class hierarchy. Interpreted development allows new code to be written and tested from within a running target application. The class hierarchy encourages code reuse and provides an organizing structure for the incorporation of new code.

Flora is an interactive application which is intended (eventually!) to be upwardly compatible with the widely used PLOT3D package. It currently supports the interactive positioning of rakes, accompanied by the repeated calculation and display of streamline and stream surface models. The Scheme command layer responds to typed commands and mouse events by calculating new visualization models. A secondary process maintains images of these models in multiple scenes.

CHAPTER III
MIXED-SPACE INTEGRATION

A streamline is a tangent curve embedded in a velocity field. These curves are usually computed using a numerical integration method through a field which has been interpolated over a set of samples. The speed and accuracy of numerical integration methods are heavily dependent on the careful use of the two coordinate spaces defined over each grid block. A novel annotation of the grid can support the adaptive use of both spaces for maximal performance with acceptable accuracy.

3.1 COORDINATES AND INTERPOLATION

An *interpolating function* is a mapping from the real-valued coordinates in the grid-local *computational* space to the underlying *physical* coordinate system. The same function is often used to interpolate the field sample values in the interior of each cell.

3.1.1 Coordinate Spaces

In a PLOT3D grid file, the position of each node point is recorded in a Cartesian *physical space* with coordinates (x, y, z). Each block in a composite curvilinear grid defines a new coordinate space over its portion of the flow domain. The coordinates (ξ, η, ζ) of a point in this *computational space* may be divided into their integer and fractional parts:

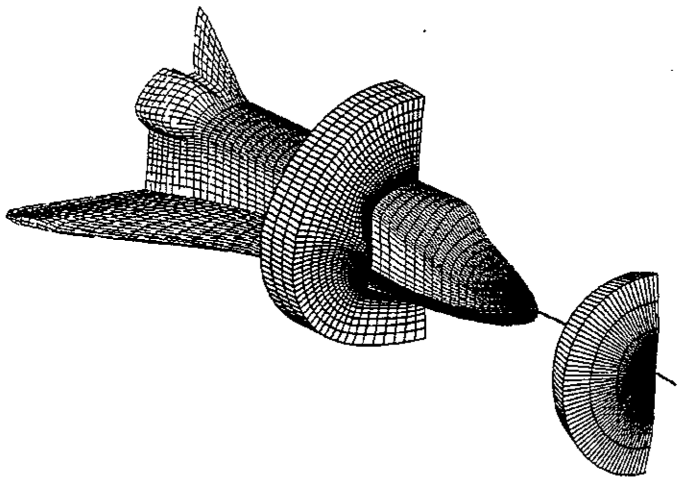


Figure 3.1: A grid with a line of singularity.

$$\begin{bmatrix} \xi \\ \eta \\ \zeta \end{bmatrix} = \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

which specify the index (i, j, k) of the cell containing the point and the fractional displacements $(\alpha, \beta, \gamma) \in [0...1]$ within that cell. Each node point has integer values for its computational-space coordinates; these integer triples form the address of each node point in the arrays which store the data values for that block. Most node points are shared by eight cells, points on a cell edge are usually shared by four cells, and most faces are held in common by two cells. The computational-space coordinates for nodes, edges, and faces therefore may have multiple valid decompositions into index and offset pairs.

The vertices of each cell are spatially ordered, such that no cell faces are interpenetrating and no cell contains negative volume. On the other hand, grids will often contain *singularities* at which an entire cell edge maps onto a single point in physical space or at which a cell face collapses into a physical-space line segment.

Such a situation often occurs when a grid must be wrapped about the nose of an aircraft. Figure 3.1 shows a number of cells and faces extracted from a single-block grid which has been fitted around the Shuttle orbiter [Rizk and Ben-Schmuel 1985]. An entire face of the grid, a two dimensional array of hundreds of cell faces, has collapsed into a line of singularity which extends forward along the central axis of the vehicle. At such locations the spatial interpolating function is not one-to-one (or *injective*), and so this mapping cannot be inverted at these points. This does not pose any difficulty for finite-differencing flow solvers, since an inversion of the interpolating function is not required by these codes. Visualization codes, however, often do invert this spatial mapping and must be able to robustly handle singularities in the grid.

3.1.2 Trilinear Interpolation

Most flow visualization software uses a cell-local trilinear interpolating function. This produces a piecewise-linear reconstruction of the ideal flow field. This interpolating function has the nice property that the extreme values must lie on the node points; higher-order interpolating schemes are more expensive to compute and also may introduce local extrema in the cell interiors.

Under any cell-local mapping, an interpolated field value depends only upon the samples recorded at the eight vertices of the cell which encloses the query point. In the commonly used trilinear interpolation formula, the relative contribution from each vertex sample is a product of the fractional displacements of the query point along each computational-space dimension:

$$\begin{aligned}
\alpha' &= (1-\alpha) \\
\beta' &= (1-\beta) \\
\gamma' &= (1-\gamma) \\
f_{ijk}(\alpha,\beta,\gamma) &= [\alpha'\beta'\gamma']f_{000} + [\alpha\,\beta'\gamma']f_{100} + \\
&\quad [\alpha'\beta\,\gamma']f_{010} + [\alpha\,\beta\,\gamma']f_{110} + \\
&\quad [\alpha'\beta'\gamma\,]f_{001} + [\alpha\,\beta'\gamma\,]f_{101} + \\
&\quad [\alpha'\beta\,\gamma\,]f_{011} + [\alpha\,\beta\,\gamma\,]f_{111}
\end{aligned}$$

3.1.3 Interpolation Error

Strictly speaking, the flow solver produces no information about the flow field quantities in the interior of each cell. One may typically assume that the flow field measures vary smoothly across each cell, since the grids are constructed with very small cells placed in regions where the flow is expected to exhibit high gradients in its various measures.

If some cell lies in a *freestream* region, far from the vehicle, then a trilinear interpolating function perfectly reconstructs the constant field quantities throughout that cell. Similarly, flow which exhibits a uniform *rigid-body rotation* can also be accurately recovered using a trilinear interpolating function. Real flows, however, often contain non-linear variation in the various independent and derived fields. These variations cannot be recovered by the trilinear interpolating function; error is thereby introduced in the samples interpolated within each cell.

An analytical model of fluid flow defines an *exponential velocity profile* in the boundary layer near the surface of the vehicle in a viscous fluid. The fluid velocity has zero magnitude at the vehicle surface, with the speed increasing exponentially with distance above the surface. A discrete sampling of this non-linear variation, followed by an interpolation of these samples, introduces error into the visualization. In this particular case, however, it is common practice for the size of the grid cells to

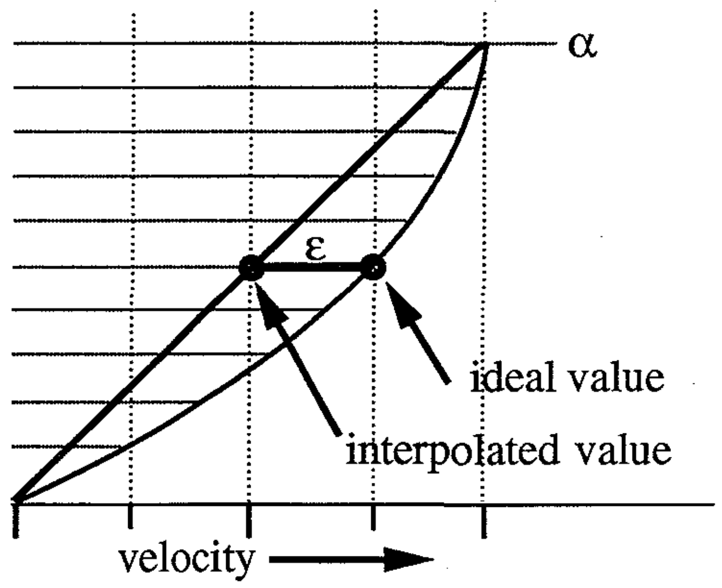


Figure 3.2: Interpolation error in a quadratically-varying flow.

vary exponentially in the boundary layer region. This adaptation of the grid to the expected variation of the flow is used to improve the accuracy and the efficiency of the flow simulation code. It also reduces the interpolation error in the post-process visualization.

A related model of fluid motion is the *Poiseuille flow* within a circular pipe. In this case, the flow velocity varies as the inverse of the square of the radial distance from the center. An interpolation of this idealized flow along a single dimension is diagrammed in figure 3.2. In this situation, the velocity magnitude is fixed at zero at the wall, and it increases quadratically to a value v somewhere within the flow. Imagine a linear interpolating function using an interpolant of α across the cell in the vertical direction. This function incurs an error of ($\epsilon = v(\alpha - \alpha^2)$), with a maximum error of $v/4$ at the center of the cell. Doubling the size of a cell will square the velocity magnitude at the upper node point and will also square the error bound. The expected interpolation error in this case is thus related to the square of the cell size.

Higher order interpolation functions could be used for visualization, but this is rarely necessary. Small cells are created during grid construction to properly represent large gradients in flow field. The direct correlation between cell size and the interpolation error ensures that values interpolated within each cell are bounded within acceptable limits.

3.2 POINT FINDING

Consider a point specified by its computational-space coordinates in a given block. The physical-space coordinates for this point can be determined by interpolating the physical-space locations of the vertices of the enclosing cell. Converting in the opposite direction (from a known physical-space position to a possibly non-unique computational-space equivalent) is more difficult. This is called the *point-location* or *point-finding* problem [Preparata and Shamos 1985].

The distance between each node point and a specified position in physical space $\vec{x}$ defines a scalar "distance" field $\mathcal{D}_{\vec{x}}$ at each node point. The point-location problem is then equivalent to identifying a computational-space position which yields a distance of zero when used in the interpolation of this field: $\mathcal{D}_{\vec{x}}(\vec{\xi}) = \vec{0}$.

Since the grid is bent in physical space, the field may have local non-zero minima. The true zero-point is usually found using two phases: a *searching* method that finds a candidate point in the neighborhood of the proper minimum and a subsequent *polishing* or *stencil-walk* method that refines this approximate value into the final real-valued computational-space coordinates [Buning 1989]. Recall that the flow domain is usually subdivided into partially overlapping regions. The coarse search must consider points from each each grid block, and the refinement method must be able to cross inter-block boundaries in its search for the query location.

3.2.1 Coarse Search

The coarse searching method could simply be a "brute-force" examination of every node in the grid to find the one which yields the smallest distance measure. But since every line of node points in a typical grid curves only slightly over a distance of a few cells, it is usually sufficient to test every n-th point in each dimension. The candidate which is selected by this more sparse search will never be farther than roughly $n\sqrt{3}/2$ cells away from the actual closest node point. On typical grids, a value of n equal to ten is sufficient and means that only one node point in every thousand need be tested. The test itself may be approximate; a simple sum-of-magnitudes (ℓ_1) norm may select a node point which is not truly the closest, but which will be an adequate starting position for the stencil-walk.

A pre-computed access structure may be built to provide a more rapid means of coarse searching. The grid-construction software "DCF" uses a coarse rectilinear *background grid* for this purpose [Meakin 1991]. Octrees and k-D trees are data structures which organize multi-dimensional data [Samet 1984, Bentley 1975]. In each of these, the space of the data is recursively divided into sub-volumes and each interior node of the tree may be annotated with the minimum and maximum data values contained within that subtree. Globus *et al.* [1991] used an octree to implement a coarse point search. Williams [1992] used a somewhat more complicated structure to implement coarse searching in unstructured grids.

An approximate initial point may be improved using a *hill-climbing* algorithm [Mastin 1988]. This method tentatively advances a test point one cell width in either direction along one of the three computational-space dimensions. This new location is selected as the new starting position if it yields a smaller error measure than does the current location. The iteration continues until none of the six immediate neighbors of the current node offers reduced distance measure. The final node is then likely to be quite close to the specified query point.

Flora uses a subsampled or *decimated* block of node points, in which the position and offset of one node point of every thousand is copied into a contiguous array. This very small array is then searched for the node nearest to the specified query point. The offset of this point is then used to locate the same node in the original array of grid data. A hill-climbing iteration is then performed to improve this estimate and to identify the node which lies closest to the query point. Only the seed point of each streamline must be found by a coarse search. Each subsequent point along the curve lies close to its predecessor; therefore, only the local stencil-walk routine is required to identify the computational-space coordinates of the integration query points.

3.2.2 Stencil-Walking

The second phase of the point-finding task attempts to identify the real-valued computational-space coordinates which interpolate onto the given physical-space coordinates of the query point. The computational-space result may be described by the current block number, the index (i, j, k) of the enclosing cell, and the offsets (α, β, γ) within that cell. The point-finding problem may then be posed as a multidimensional root-finding problem on a function $\mathcal{D}$ which measures the difference between the specified physical-space location $(\vec{x}^\star)$ and the interpolated result at a some offset (α, β, γ) in a given cell:

$$\mathcal{D}(\alpha, \beta, \gamma) = \mathcal{I}_{ijk}(\alpha, \beta, \gamma) - \vec{x}^\star = \vec{0}$$

The Newton-Raphson method is commonly used to solve such problems. This method begins at an approximate computational-space point $\vec{\alpha}_n = [\alpha, \beta, \gamma]_n$, and repeatedly shifts to a new point $\vec{\alpha}_{n+1}$ which, one hopes, lies more closely to the unknown correct location $\alpha^\star$. This method repeatedly evaluates the recurrence relation:

$$\vec{\alpha}_{n+1} = \vec{\alpha}_n - \frac{\mathcal{D}(\vec{\alpha}_n)}{\mathcal{D}'(\vec{\alpha}_n)}$$

which contains the partial derivatives of the interpolation function along each of the grid dimensions. After a few iterations, the distance between successive points (α_i) should be quite small. If the distance is increasing, then the method has failed. The iteration must be restarted from a new initial point, or else a different point-finding method may be tried.

Once the iteration has converged, the offsets are examined. If any offset lies outside the range [0...1], then the identified location is outside the current cell and outside the domain of the cell-local distance function $\mathcal{D}_{ijk}$. The cell index is shifted by plus or minus one along one or more grid dimensions and the iteration is restarted using a new set of coefficients calculated from the physical-space coordinates of the vertices of the new cell. If the iteration converges to a computational-space position which is beyond the boundaries of the current block, then search must be resumed in the neighboring block. The method is successful only when it converges with each offset in the range of [0...1]. These fractional offsets may then be used to interpolate the flow field sample values at this location. This composite point-finding method, which executes a Newton-Raphson method in one or more cells, is often called *stencil-walking* [Buning 1988].

The Newton-Raphson method can be described geometrically, as depicted in figure 3.3. In this view of the problem, an approximate computational-space point $\vec{\alpha}_n$ is mapped onto its physical-space coordinates $\vec{x}_n$. The error vector $(\Delta\vec{x}_n)$ is the difference between the image point $\vec{x}_n$ and the sought-after position $\vec{x}^*$. This error vector is then mapped back into computational coordinates to produce a computational-space error vector $(\Delta\vec{\alpha}_n)$. This new vector is subtracted from the current computational-space position to produce a new point $\vec{\alpha}_{n+1}$, which should

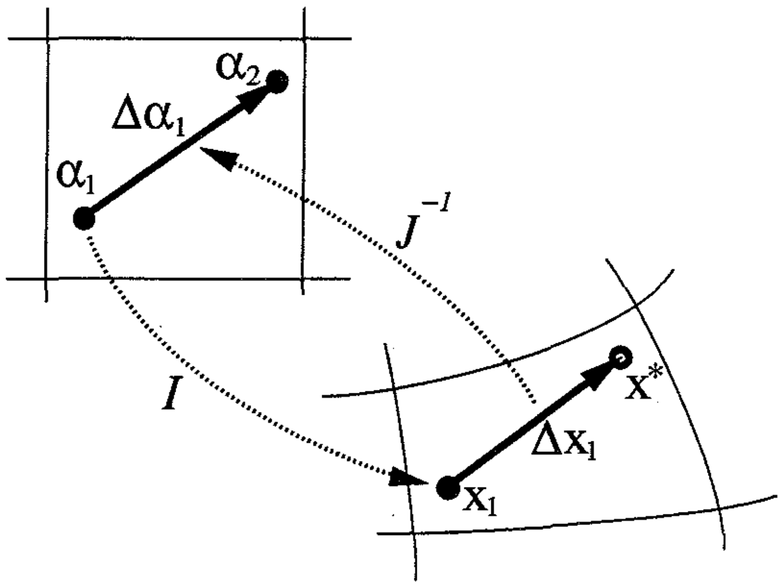


Figure 3.3: Point-finding with the Newton-Raphson method.

map more closely to the correct physical-space location. The earlier equation can be augmented as such:

$$\begin{aligned}
\vec{\alpha}_{n+1} &= \vec{\alpha}_n - \frac{\mathcal{D}(\vec{\alpha}_n)}{\mathcal{D}'(\vec{\alpha}_n)} \\
&= \vec{\alpha}_n - \Delta\vec{\alpha}_n \\
&= \vec{\alpha}_n - (\Delta\vec{x}_n)\mathcal{J}^{-1} \\
&= \vec{\alpha}_n - (\vec{x}_n - x^*)\mathcal{J}^{-1} \\
&= \vec{\alpha}_n - (\mathcal{I}(\vec{\alpha}_n) - x^*)\mathcal{J}^{-1}
\end{aligned}$$

Since the computational space is curved with respect to the underlying physical space, the subtraction of an instantaneous error vector ($\Delta\vec{\alpha}$) is only a partial correction of the interim computational-space location. Subsequent improvements must be made in an increasingly smaller neighborhood about the unknown $\vec{\alpha}^*$.

3.2.3 Implementing the Newton-Raphson Method

The Newton-Raphson method requires the evaluation of $\vec{\mathcal{D}}'$ at the point $\vec{\alpha}$. This can be computed by straightforward expansion of the nine partial derivatives $\partial\mathcal{D}(x_i)/\partial\alpha_j$ of the trilinear interpolating function, computed across each physical space component (x_i) and in each grid dimension (α_j). This produces the *Jacobian matrix*:

$$\mathcal{J} = \frac{\partial\vec{x}}{\partial\vec{\xi}} = \frac{\partial\vec{x}}{\partial\vec{\alpha}} = \begin{bmatrix} \frac{\partial x}{\partial\alpha} & \frac{\partial x}{\partial\beta} & \frac{\partial x}{\partial\gamma} \\ \frac{\partial y}{\partial\alpha} & \frac{\partial y}{\partial\beta} & \frac{\partial y}{\partial\gamma} \\ \frac{\partial z}{\partial\alpha} & \frac{\partial z}{\partial\beta} & \frac{\partial z}{\partial\gamma} \end{bmatrix}$$

which describes the local relationship of the basis vectors of the physical and the computational coordinate systems

An approximation of the Jacobian matrix can be computed using finite-differences of the physical-space coordinates of the vertices of the current cell.

```
DOTIMES(i,3) {
  d_a   = (p100[i] - p000[i]);
  d_b   = (p010[i] - p000[i]);
  d_g   = (p001[i] - p000[i]);
  d_ab  = (p110[i] - p010[i] - p100[i] + p000[i]);
  d_ag  = (p101[i] - p001[i] - p100[i] + p000[i]);
  d_bg  = (p011[i] - p001[i] - p010[i] + p000[i]);
  d_abg = (p111[i] - p011[i] - p101[i] + p001[i] -
            p110[i] + p010[i] + p100[i] - p000[i]);

  jac[i][0] = d_a + d_ab*b + d_ag*g + d_abg*(b*g);
  jac[i][1] = d_b + d_ab*a + d_bg*g + d_abg*(a*g);
  jac[i][2] = d_g + d_ag*a + d_bg*b + d_abg*(a*b);
}
```

This factoring of the partial derivatives is due to Buning, who used it in the implementation of PLOT3D. The variables p000 through p111 contain the physical-space coördinates of the eight corners of the enclosing cell. The variables (a,b,g) hold the fractional parts of the computational-space coordinates of the query point. The subscript i is iterated over the three dimensions of computational space. Note that the first seven equations yield constant values throughout each cell and that only the last three lines of the loop body need be computed anew for subsequent query points in the same cell.

The resulting matrix (`jac`) is then inverted. If this matrix cannot be inverted, this indicates that the current position lies on or very near a collapsed cell edge or face. When this occurs, the point $\vec{\alpha}_n$ can be shifted a short distance in an attempt to move away from the grid singularity. The Jacobian matrix at this new location is computed, and the iteration is resumed. The method will sometimes converge without encountering further difficulty. Stubborn cases can be solved by resorting to another method, such as a fractional variant of the hill-climbing method described earlier.

3.3 PHYSICAL-SPACE INTEGRATION

PLOT3D computes streamlines using a second-order Runge-Kutta integration method through a physical-space vector field. The field values within each cell are defined by the trilinear interpolating function, which requires the computational-space coordinates of each query point. Since each new query point is originally specified in physical-space coordinates, this position must be converted into its computational-space representation prior to the interpolation of the vector field values.

3.3.1 Implementing the Integration Method

The point-finding method is used to determine the computational-space coordinates for each query point. This procedure, let us call it `find_point`, contains internal state which allows it to begin the root-finding iteration from the most recently identified, and presumably nearby, computational-space location. A second procedure, `interpolate`, returns the interpolated value of some field at a specified computational-space location.

These two procedures, `find_point` and `interpolate`, may be combined to implement a physical-space numerical integration routine. Each iteration advances the particle through physical space, using `find_point` to identify the computional-space coordinates of each new query point and `interpolate` to find the local velocity vector at these locations.

```
phys = seed_xyz;
comp = find_point(GRID, phys);
time = 0;
LOOP {
  output(phys, comp, time);
  pvec = interpolate(PVEC, comp);
  phys = phys + (h * pvec);
  comp = find_point(GRID, phys);
  time = time + h;
}
```

As the integration proceeds, the physical-space coordinates, the computational-space coordinates, and the accumulated stepsize of each new point are written to the result buffers. The iteration is stopped when the advecting particle falls outside the flow domain, when the user moves the seed point, or when the output buffer has been filled. This code works transparently over multiple-block grids, since the transition from one block to the next is handled within the procedure `find_point`.

This example uses the simple *forward Euler* algorithm, which accumulates error at a rate proportional to the stepsize parameter `h`. This simple routine is generally inadequate for computing streamlines in most flow fields [Murman and Powell 1988]. The accuracy of the computed result can be improved by replacing the body of the loop with these lines:

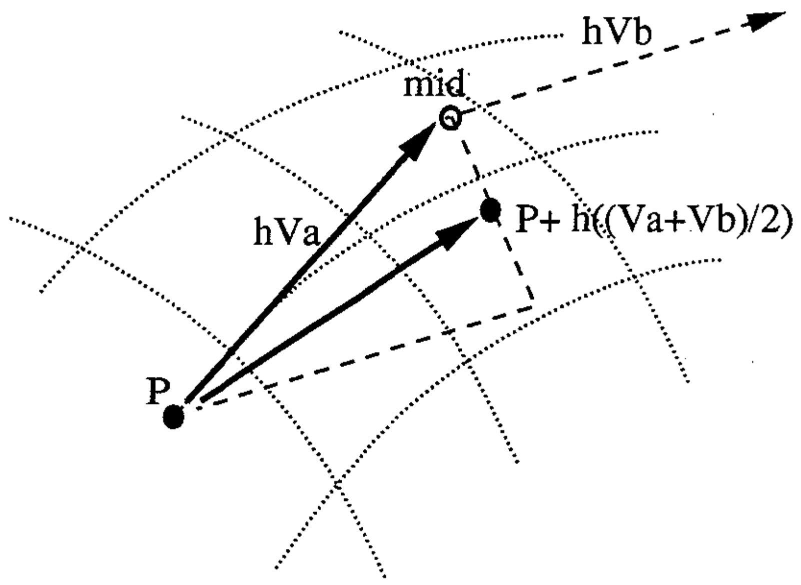


Figure 3.4: Heun's method.

```
output(phys, comp, time);
va   = interpolate(PVEC, comp);
pmid = phys + (h * va);
cmid = find_point(GRID, pmid);
vb   = interpolate(PVEC, cmid);
phys = phys + (h * ((va+vb)/2);
comp = find_point(GRID, phys);
time = time + h;
```

This is *Heun's method*, one of the second-order Runge-Kutta schemes (figure 3.4). In this method, a simple Euler step is used to sample the field at a test location `mid`. The two velocity values (`va` and `vb`) are averaged, and this combined value is used to advance the particle to its new location. This algorithm has an error rate equal to the square of the stepsize parameter (`h`). This usually produces acceptable results when the particle is advanced some fraction of the cell width for each iteration. This second-order algorithm requires two calls of `find_point` per iteration, but this additional work is partially offset by the reduced error rate which allows the use of larger stepsizes. Both PLOT3D and Flora use a version of this routine, with additional modifications to adjust the stepsize parameter during the iteration.

3.3.2 Adjusting the Stepsize

A large improvement in the speed of numerical integration may be obtained through the use of *adaptive stepsizing*, which extends the length of the integration stepsize wherever the flow field samples are fairly uniform. The stepsize is shortened where the flow makes a large change in its speed or direction. This adjustment of the stepsize is typically based on a comparison of the results of two different integration methods started from the same initial point. The more accurate method produces the new point which is copied to the output buffer. The simpler method is used as a test of the error-sensitivity in this region of the flow. When the results of the two methods are very close, the stepsize is increased slightly. When the results differ by more than some small amount, then the stepsize is reduced to improve the accuracy of the next few integration steps. (Other methods of adjusting the stepsize are based on the explicit evaluation of the local curvature of the vector field, as in [Dickinson and Bartels 1988] or [Darmofal and Haimes 1992].)

In regions of rapid velocity change, the stepsize must be so small that the resulting points are separated by tiny fractions of a cell width. But in a large percentage of the volume of a typical flow field, the stepsizes can be much larger. Note, however, that the the flow field is defined by sample values at each node point. This sets a limit on the maximum acceptable stepsize. In order to detect and to adequately resolve small flow features, the stepsize must be constrained such that each iteration advances the particle at most one cell width. This avoids errors that would be caused by taking a single large step completely through small flow structures that may be represented by sample values at only one or two neighboring node points. This limit on the stepsize also guarantees an adequate sampling rate for any subsequent texturing of the model, which might be used to represent a coincident scalar field.

The program STREAM3D [Eliasson *et al.* 1989] computes the vector magnitudes at the eight cell corners. It uses the average of these as the nominal fluid velocity in the cell interior. The stepsize is then adjusted to ensure about three steps per cell. This adjustment generally ensures that any curvature of the field within a cell will be adequately represented, at the cost of being overly conservative in regions of simple flow behavior. On the other hand, the size of each grid cell is a fairly good indication of the complexity of the flow in that region, since the grids are created with great care to ensure exactly this property. The averaging of the local samples is adequate for a majority of the flow domain, but is inaccurate within any cell which has greatly differing velocity values at its eight vertices.

PLOT3D adjusts the stepsize to ensure that adjacent points on the streamline are separated by roughly one-fifth of the width of the current cell in any grid direction. This is implemented by converting the interpolated velocity sample into its computational-space equivalent, finding the component with the maximum absolute value, and scaling the stepsize by the inverse of this length. This conversion into computational space requires the inverse of the local Jacobian matrix. Since this adjustment is based on the local Jacobian matrix, the stepsize may be not quite correct in highly stretched regions of the grid. A stepsize which is correctly sized for one cell might be much too large for the neighboring cell into which the particle next moves.

Flora uses an alternative method to maintain a specified computational-space distance between computed points on the curve. Each integration step is performed using the previously determined stepsize. The computational-space position of each new point is then subtracted from that of the preceding point on the growing curve. The magnitude of each component of this incremental displacement is then evaluated to adjust the stepsize for the next integration step. This *retroactive adjustment* produces acceptable results in most cases. If the new point is overly distant from its predecessor, then it is discarded and the integration step is repeated with a smaller stepsize. This occasional recalculation of a step can be less costly than the calculation of computational-space velocities for the sole purpose of adjusting the stepsize. This method also ensures that the stepsize will always be correctly adjusted, even in cases of rapid local change of cell size or flow speed.

3.4 COMPUTATIONAL-SPACE INTEGRATION

Numerical integration in physical space is slowed by the need to find the computational-space coordinates of each query point. As we have seen, finding these coordinates involves the creation and the inversion of a sequence of Jacobian matrices within the iterative body of a Newton-Raphson method. A promising alternative is to convert vector samples into their computational-space representations. The particle is then advected through the cubical cells of computational space, and no point-finding of the query point locations is needed. This approach can be quite rapid, but the conversion of the vector samples must use finite-differences to approximate the local Jacobian matrix. This error can yield inaccurate streamlines through some regions of the grid.

3.4.1 Implementing the Integration Method

The Newton-Raphson method uses the inverse of the Jacobian matrix to convert a physical-space error vector into its approximate computational-space equivalent. The inverse of the Jacobian matrix can also be used to convert vector field samples into computational coordinates.

Consider a particle with an arbitrary position specified in computational-space coordinates. These coordinates may be split into the (i, j, k) cell index and the

(α, β, γ) offsets, which are then used to interpolate the physical-space velocity field and also to approximate the local Jacobian matrix. A multiplication of the vector through the inverse of this matrix yields the computational-space representation of the interpolated vector sample. This new value may be used to advance the particle through the current block:

```
comp = find_point(seed_xyz);
time = 0;
LOOP {
  phys = interpolate(GRID, comp);
  output(phys, comp, time);
  pvec = interpolate(PVEC, comp);
  va   = pvec * J_inverse(comp);
  cmid = comp + (h * va);
  pvec = interpolate(PVEC, cmid);
  vb   = pvec * J_inverse(cmid);
  comp = comp + (h * ((va+vb)/2));
  time = time + h;
}
```

In this approach, the computationally expensive function `find_point` is called only to identify the computational-space coordinates of the initial seed point. The loop now contains three calls of the function `interpolate`, twice to query the physical-space vector field and once to identify the physical-space coordinates of the newly computed point for eventual output to the display device. Each iteration also contains the creation and inversion of a Jacobian matrix at each query point location, in order to convert the interpolated vector sample into computational-space coordinates.

This new method fails, of course, where the Jacobian matrix cannot be inverted. The mapping into computational-space coordinates is non-unique at grid singularities; there is no computational-space equivalent to the interpolated physical-space vector. Even where the matrix *can* be inverted, there is concern over the accuracy of the vector transformation. Tamura and Fujii [1990] warn that, in the common case of highly stretched grids, a constant physical-space vector field can yield a wide variation in the magnitudes and directions of the resulting computational-space vector samples. This problem reaches its extreme case near singularities, at which a zero-length cell edge might suggest an infinite magnitude for the computational-space vector quantity! This wide range of vector magnitudes is likely to cause accuracy problems in the simpler numerical integration algorithms. Before considering these issues further, two variations of the computational-space method should be examined.

3.4.2 Isoparametric Mapping

The method sketched above converts each interpolated vector sample into computational space by constructing the inverse of the Jacobian matrix at each query point. An alternative approach converts the user-supplied samples at the node points. These converted samples are then interpolated at the query locations. The calculation of the streamlines is again conducted entirely in the computational space. This approach differs in a significant way from the computational-space method of the previous section. The previous method requires the approximation of the Jacobian matrix at arbitrary points in the flow domain. We have seen that these matrices are computed using finite differences within the current cell, thereby constructing the partial derivatives of the interpolating function.

The alternative method of *isoparametric mapping* requires Jacobian matrices only at the node points [Hughes 1987]. These matrices are calculated using finite-differences taken across neighboring node points along each grid dimension. Once the Jacobian matrix at a node point has been approximated in this way, the matrix is inverted and the vector sample is multiplied through the new matrix to derive the computational-space representation of the vector sample at the node. Once again, the matrix cannot be inverted at grid singularities, so some alternative calculation method must be applied at these nodes.

The program STREAM3D [Eliasson *et al.* 1989] converts the flow field vectors on the corners of each newly encountered cell as the streamline is extended through the flow domain. Since the interpolating function is cell-local and linear, they suggest that a cell-local construction of the Jacobian matrix is attractive. These computed matrices would then be in exact accordance with the cell-local interpolating function. Indeed, the Jacobian matrix consists simply of the partial derivatives of the interpolating function itself, computed at the eight corners of the cell.

This calculation of the Jacobian matrix is implemented with two-point finite-differences taken across each cell edge. Using the same conventions as used in earlier code fragments, the Jacobian matrix for the lowest-indexed vertex (p000) is computed from the differences in each physical-space component along the three cell edges which join at that node point.

```
DOTIMES(i,3) {
  jac[i][0] = (p100[i] - p000[i]);
  jac[i][1] = (p010[i] - p000[i]);
  jac[i][2] = (p001[i] - p000[i]);
}
```

The other seven matrices are computed in a similar manner. Each matrix is inverted, and the user-supplied sample vectors are multiplied through the inverted matrices to yield the computational-space vector samples for the eight vertices of the current cell.

Unfortunately, the cell-local differencing shown above creates up to eight distinct velocity samples at each node point. The storage cost of this is prohibitive, and so an averaged value might well be used instead. A centered-difference can be used to compute these averaged computational-space velocity samples each node, thus reducing storage costs to a single new vector sample per node point. In three-point centered-differencing, a partial derivative at a node point is approximated as the average of the differences taken across the two neighboring cell edges. This is equivalent to the one-half the total difference taken across two successive cell edges:

$$\partial\vec{x}/\partial\xi = (\vec{x}_{(i+1,j,k)} - \vec{x}_{(i-1,j,k)})/2$$
$$\partial\vec{x}/\partial\eta = (\vec{x}_{(i,j+1,k)} - \vec{x}_{(i,j-1,k)})/2$$
$$\partial\vec{x}/\partial\zeta = (\vec{x}_{(i,j,k+1)} - \vec{x}_{(i,j,k-1)})/2$$

These centered-differences can, of course, be applied only at the nodes in the interior of each grid block. Non-centered finite-differences must be used at nodes which lie on the domain boundaries. One drawback of this method is that the resultant computational-space vectors are formed using spatial metrics which have been averaged across multiple cells.

On the other hand, it has the appealing property that each vector field sample is mapped to one and only one computational space representation; therefore, these samples may be pre-computed and stored for later use in interactive visualization [Volpe 1989]. Once this full pre-process conversion is complete, the streamlines can be computed using only interpolation of the computational-space vector field samples and a mapping of the resulting points into physical space. No conversion of the vector field values is needed during the integration of the streamlines, and no matrices need be constructed or inverted during this calculation.

3.4.3 Performance

The physical-space method requires the point-finding Newton-Raphson method to be applied at each query point. This typically incurs a computational effort of about two matrix inversions per query point, and hence four inversions per step of the second-order Runge-Kutta method. This yields about sixteen matrix inversions in each cell encountered by the particle, assuming that it passes through each cell in about four integration steps.

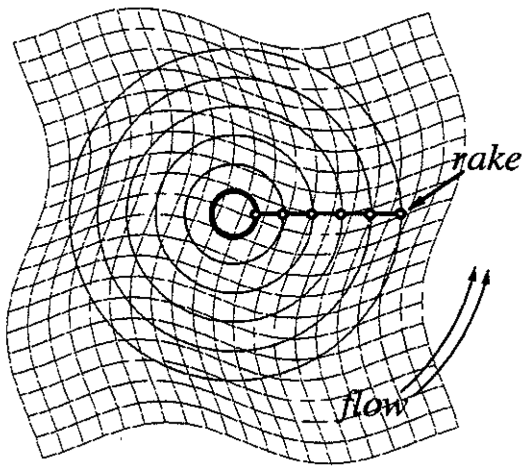


Figure 3.5: Smooth grid with physical-space streamlines.

Local conversion of the field samples requires one inversion of the Jacobian matrix at each query point. This happens twice per iteration of the second-order Runge-Kutta method, and typically eight times per cell.

On-the-fly conversion of the node samples requires eight matrices to be inverted at each newly encountered cell. These computational-space samples may be saved and interpolated for all subsequent query points within that cell.

Pre-process conversion of the field data allows the calculation of streamlines using only the interpolation of the vector field at each query point. No conversion of vector samples is required during the calculation of streamlines.

A simple grid was constructed for testing the relative speed of these calculation methods. This is essentially a two-dimensional grid, as it varies only along the i and j dimensions. Each k-plane is an identical copy of those above and below, with a constant value of z and the w velocity component held at zero. This yields a stack of identical two-dimensional flow fields varying only in x and y, in this case a rigid circular flow. (The program used to generate this test grid and flow data may be

METHOD	phys	local	two-pt	central
CELLS LOADED	1453	1433	1435	1439
NR INVOCATIONS	10000	0	0	0
INVERSE(J)	18453	10000	11480	0
CPU SECONDS	7.3	4.5	4.5	2.3

Table 3.1: Relative performance of four differencing methods.

found in Appendix A.) One layer of this grid is shown in figure 3.5, which has been overlaid with a family of streamlines. The rake is a horizontonal line segment which has been discretized to form six seed points. The flow defines six circular streamline curves, with the particles advected counter-clockwise about a fixed center.

Streamlines were calculated from this same rake using the physical space method and all three variants of the computational space method. All the tests used a second-order Runge-Kutta algorithm with the retroactive stepsizing scheme used to produce points separated by about one-fifth of the smallest dimension of the current grid cell. I counted the number of interpolations and matrix inversions calculated by each method. I also measured the elapsed CPU time, measured on a single processor of a Silicon Graphics VGX-320 workstation. A total of five thousand integration steps were performed in each trial, and the trials were repeated several times.

The results are listed in table 3.1. Each method evaluated the vector field at 10000 query locations. Due to differences in accuracy, the curves intersected slightly different numbers of cells. The physical-space method required 18453 iterations of the Newton method for the point finding. The three computational-space methods do not use the Newton-Raphson point finding method, but instead convert the field samples by inverting the Jacobian matrices. This requires 10000 inversions for the local computational-space method, eight per cell (or 11480) for the two-point differencing across the cell edges, and none at all if the conversion was already performed by a pre-processing step. The elapsed CPU time confirms that the computational-space methods are indeed more rapid, particularly so the method using pre-converted velocity samples.

3.4.4 Accuracy

Shirayama [1991] tested the effect of grid smoothness on the accuracy of the computational-space integration methods. He constructed a two-dimensional grid of square cells, and then perturbed each node point position by some random amount in the two physical-space dimensions. The maximal displacement of any node point was less than one-half the cell size, so that no cell edges were allowed to cross. An analytically defined physical-space vector field was then sampled on these perturbed node points, the samples converted using a central-differencing pre-process, and streamlines were then calculated through the computational-space field. The resulting curves maintained good qualitative agreement with the ideal curves defined by the original analytic vector field.

There is unfortunately some doubt that a random perturbation of the node points can provide a suitable test of the error-sensitivity of the computational-space methods. Grids used in practice are generally quite smooth over much of the flow domain, but can exhibit large first-order discontinuities along gridlines at a sequence of neighboring node points. These correlated flaws can inject error into the calculation of computational-space vector samples and can produce inaccurate streamlines. For example, Sadarjoen *et al.* [1994] demonstrated troubling error rates using a number of computational-space methods on a collection of test grids and flow fields.

To further investigate this effect, I took the simple test grid from the performance test and introduced a large first-order discontinuity across one gridline (figure 3.6). The same rotational flow field was then sampled at these new node points. These new samples were used in the calculation of streamlines, once again using the physical space and the three computational space methods.

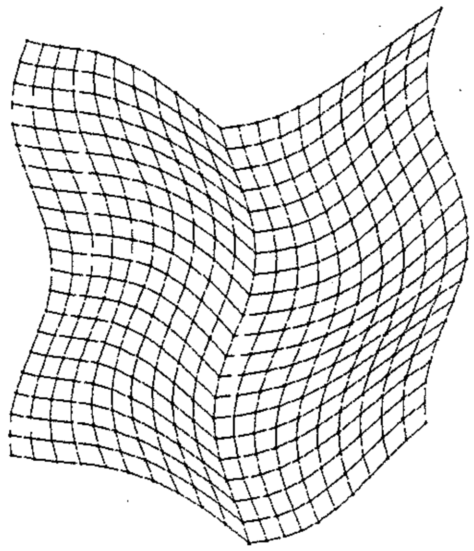


Figure 3.6: A grid with a strong discontinuity.

The streamlines computed in physical space adequately reconstruct the circular pattern of the flow in both the smooth and the creased grids (figure 3.7). This is not surprising, since this vector field is strictly linear and is perfectly reconstructed in each case by the trilinear interpolating function. The only significant error introduced is that due to the truncation of the numerical integration formula, as evidenced by a slight outward spiraling of the innermost streamline. A smaller stepsize improves this result, as would a higher-order integration method.

The streamlines computed with local Jacobian matrices are similar to those calculated using two-point differencing across the edges of the current cell (figures 3.8 and 3.9). Both methods construct Jacobian matrices as the set of partial derivatives of the cell-local interpolating function. Both families of streamlines exhibit some increased spiraling, which is exacerbated by the crease.

Figure 3.10 shows the streamlines computed through pre-converted samples which have been derived by two-point central-differencing. Here we see that accuracy is quite poor when the particles are advected for several revolutions. In both

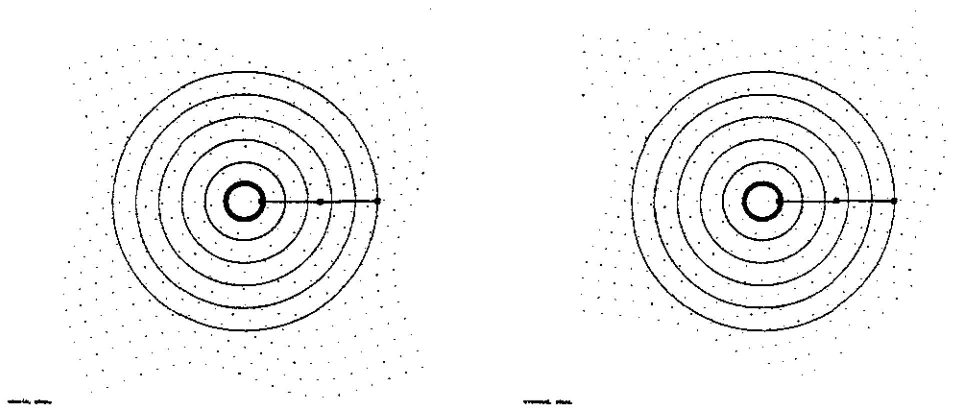


Figure 3.7: Streamlines computed in physical space, in a smooth grid (left) and a creased grid (right).

the smooth and the creased tests, many of the streamlines exhibit significant inward spiraling. Only the outermost curve maintains approximate closure to form a circular path. The presence of the grid crease has introduced a strong local perturbation of the resulting curves; each curve exhibits a "bump" at each crossing of the grid crease. But this diversion contributes only minor additional reduction of the global accuracy, as the full streamlines in the creased grid are qualitatively similar to those computed using this same method in the smooth grid.

In a magnified view of some curves computed with locally constructed Jacobian matrices (figure 3.11), we see that the streamlines exhibit very strong discontinuities near the crease. The cause is demonstrated in figure 3.12, which depicts a uniform horizontal flow sampled at the vertices of two cells. In physical-space integration, the two query points each yield the same uniform flow sample, and the averaged vector value carries the particle to the position P^*. But when the particle is advected in computational space, the vector samples are now different because the conversion of each vector is based on Jacobian matrices computed in different cells. The particle is advected to a new location in computational space, then mapped into physical coordinates. This mapped location P' is not coincident with P^*, due to the error introduced by the combination of incompatible spatial transformations.

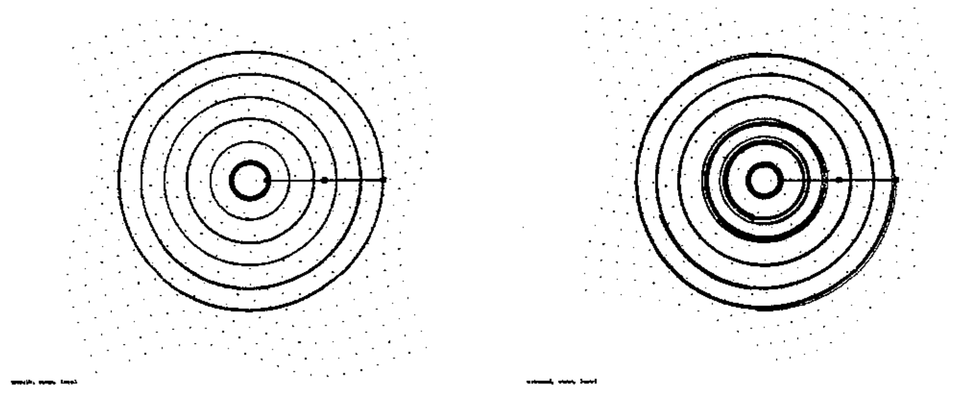


Figure 3.8: Streamlines computed with local Jacobians, in a smooth grid (left) and a creased grid (right).

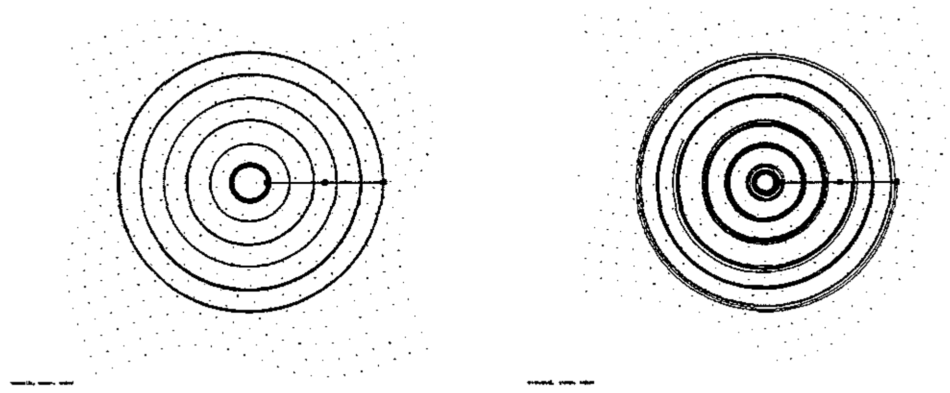


Figure 3.9: Streamlines computed with edge-differencing, in a smooth grid (left) and a creased grid (right).

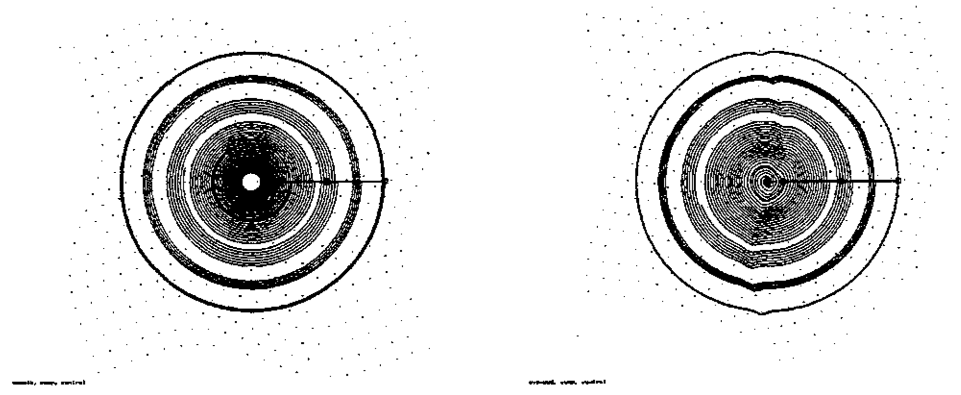


Figure 3.10: Streamlines computed with central-differences, in a smooth grid (left) and a creased grid (right).

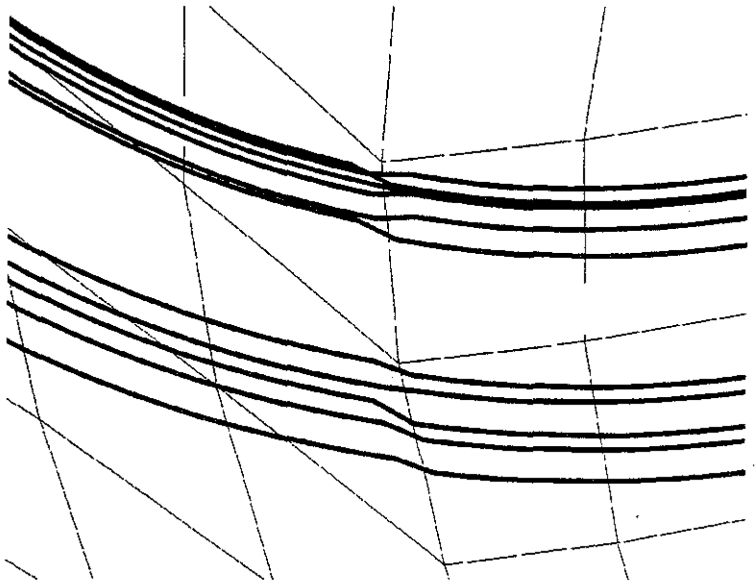


Figure 3.11: Detail of streamlines computed with local Jacobians.

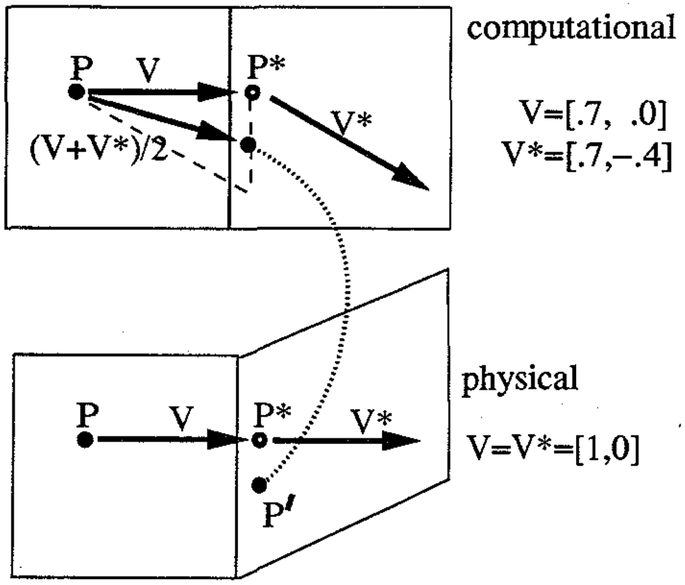


Figure 3.12: Heun's method with cell-local differencing.

3.4.5 Constrained Integration

Computational-space calculation suffers from inaccuracy near grid flaws and it fails entirely at singularities. This approach is, however, much faster than the physical-space method. The computational-space algorithms offer one additional advantage which can further motivate an attempt to salvage this approach: the computational-space methods are more easily adapted to compute streamlines near the vehicle surface.

An iblank value of 2 is used to mark any node which lies on a impermeable boundary. These node points carry valid field samples, but if all four corners of a cell face are so marked, then no fluid may pass through that face. PLOT3D handles this situation using a special constraint called *wall bouncing.* By explicitly restricting the curves away from these body surfaces, the software avoids one particularly annoying artifact of numerical error: streamlines that leave the flow domain by passing through the skin of the vehicle.

Wall bouncing is implemented by constraining the position of the advecting particle slightly away from any nearby wall. (This is usually in the positive ζ or k direction.) This constraint can also be enforced by clamping the appropriate component of the velocity to positive values for integration with a positive timestep, and negative values when advecting particles upstream. When the integration is performed in physical coordinates, the particle position and the interpolated vectors must be converted into computational-space coordinates. These values are then adjusted if necessary, and the new values converted back into physical coordinates. Integration by the computational-space method avoids this conversion and its inverse.

It is often useful to depict the tangential movement of the fluid just off the surface of the vehicle. These are called *oil flow plots*, since they are similar to the empirical visualization method in which a scale model is coated with a layer of oil or paint prior to its session in a wind tunnel. The numerical equivalent is constructed by restricting the calculation of streamlines to the gridplane which lies one cell-height above the no-slip boundary surface. The implementation of this restriction is comparable to that of the wall-bouncing constraint described above. In wall bouncing, the particle is forced away from the boundaries. In the oil flow visualization, the particle must remain on or just above these surfaces. This is done by clamping the off-body component of the particle's computational-space position to a fixed value. The velocity component along this same grid dimension is forced to zero.

Once again, this restriction of particle motion is implemented most easily under the computational-space integration methods. As in wall bouncing, the tracing of oil flow lines through a physical-space vector field requires the repeated conversion of the particle location and velocity vectors into computational space, followed by the possible modification of these values and a return to physical-space coordinates.

3.4.6 Block Transitions

Computational-space integration is fast, albeit error-prone, and it easily adapts to the special constraints of wall bouncing and oil flow. Unfortunately, the integration methods cannot easily carry a particle across the inter-block boundaries of a composite grid.

In physical-space integration, the transition across block boundaries is handled within the `find_point` procedure. But when a particle is advected through computational space, the valid domain is limited to the current block. Reaching a block boundary suspends the integration until the new location of the moving particle has been determined in the computational coordinate space defined by the new block. It may be that the last known position of the particle lies within a region in which two neighboring blocks overlap. In this case, the physical-space position of the particle is found by interpolation, and this same location is then identified in the computational space of the new block.

If the grids merely abut, then the computational-space integration method will fail when it attempts to extend the curve beyond the boundaries of the current block. This situation can be handled by reverting to the physical-space integration method until the boundary has been crossed. Once the boundary has been crossed, the computational-space method may be resumed. This adaptive use of both coordinate spaces is more fully considered in the next section.

3.5 MIXED-SPACE INTEGRATION

We have seen that the fastest computational-space integration method can compute a streamline in about one-third of the time required by the same integration method cast in physical coordinates. But the faster methods are sensitive to error near grid flaws and are unable to cross block boundaries. A novel hybrid approach, presented here, uses precomputed measures of local grid-quality to adaptively select the most suitable integration method for each newly encountered cell. This preserves accuracy, yet still provides most of the speed of the computational-space approach.

3.5.1 Node Tagging

The *iblank* field in a composite grid allocates a full thirty-two bit word to every node point. Blanked node points, which do not carry valid field samples, are marked with an iblank value of 0. Usable node points in the interior of a block are marked with the value 1. Node points in regions of block overlap are marked with negative values, and the value 2 marks nodes on the impermeable boundaries formed by the vehicle surface and the flow field symmetry planes.

In the Shuttle dataset shown earlier, nine grid blocks contain a total of 941,159 node points. Of these, 92% have an iblank value of one, about 5% percent of the

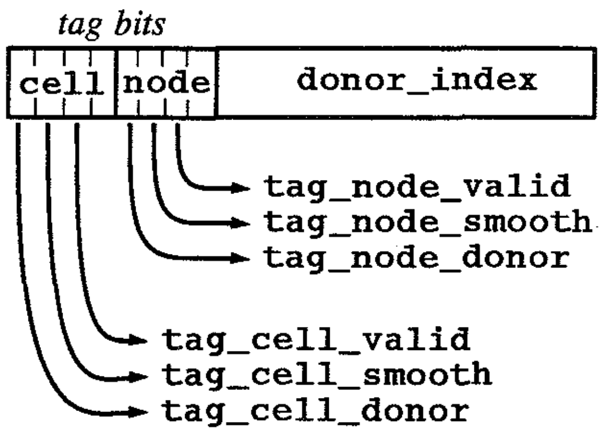


Figure 3.13: A tag word.

node points have been assigned a zero iblank value, and the remaining nodes carry negative iblank numbers. No points have been marked as walls in this dataset. The iblank annotation of this grid is rather wasteful, consuming four megabytes for a moderately sized grid of about one million nodes and encoding fewer than four bits of information in a 32-bit word. A more efficient encoding can simplify the querying of the field data.

In Flora, each node point is assigned a *tag word*, which is subdivided into several single bit flags and a small integer field (figure 3.13). Information for each node is stored in its tag, and comparable information for each cell is stored in the tag of its lowest-indexed corner. (The nodes on the highest-indexed face in each dimension of each grid block have no associated cell and the cell bits within these tag words are ignored.)

The tag word contains a Boolean flag called `TAG_NODE_VALID`, which is set to TRUE for nodes at which the original iblank number was non-zero. This indicates whether a node point carries valid sample data. A comparable flag is assigned to each cell and set to the logical product (and) of the node flags for its eight vertices. A test of this single `TAG_CELL_VALID` flag is therefore sufficient for determining the validity of interpolated values within a given cell.

Other bits can be assigned to encode additional information about the grid. We have seen that a vector field sample may be accurately converted into computational space only at those node points through which the grid lines pass smoothly. Such nodes can carry the value TRUE in the flag TAG_NODE_SMOOTH. As before, the corresponding cell flag is assigned the product of the bits for its eight vertices. Only a cell with a TRUE value for its TAG_CELL_SMOOTH flag will have reliable computational-space vectors at all eight vertices. A cell marked FALSE is adjacent to some grid flaw, and does not have reliable field data from which to interpolate computational-space vector samples in its interior. When an advecting particle encounters such a cell, the integration method can revert to the slower but more robust physical-space method to carry the curve through this problematic region. The software thereby exploits the faster computational-space method in most of the volume of the data, while limiting the introduction of error in non-smooth regions of the grid.

3.5.2 Calculation of the Tag Values

We wish to mark each node point with a value which indicates the local smoothness of the grid. A simple estimation of grid smoothness is obtained by comparing the two cell edges which share a given node along a given gridline. If these two vectors are of comparable magnitude and direction, the the grid may be assumed to be smooth along that grid dimension. A test of the other two edge pairs at the current node is required to then determine if the grid is indeed smooth in all three grid dimensions. If the three pairs of edges are found to join cleanly, then the grid is deemed acceptable at this node and the TAG_NODE_SMOOTH bit is set to TRUE.

A cell is valid when all eight corner nodes are valid. A cell is deemed smooth when all eight nodes are smooth. When advecting a particle in computational coordinates, sample values will only be interpolated within cells which are valid and smooth.

Arbitrarily complicated tests of grid quality could be conducted during the grid pre-processing with the result encoded in the node and cell tags. The user might wish to enforce the use of the physical-space method through regions of particular interest. The tags in these regions could be manually specified, overriding the computed values. Such flexibility would be more costly to implement and execute under a mechanism which selects the appropriate integration method using programmatic quality measures at each newly encountered cell.

3.5.3 Implementing the Integration Method

To implement mixed-space integration, the software must record the particle location at the start of each new iteration. It can then attempt a single step of one of the computational-space methods described earlier. If this attempt fails, due to FALSE cell tags or because the particle has encountered a block boundary, then the particle is restored to its previous location and the more robust physical-space method is tried:

```
LOOP {
  prev_comp = comp;
  ok = step_in_comp_space();
  if (!ok) {
    reset_position(prev_comp);
    ok = step_in_phys_space();
    if (!ok) QUIT();
  }
}
```

The routines `step_in_comp_space` and `step_in_phys_space` are simply those integration methods shown earlier. A minor modification to the code shown here can improve the performance slightly. It is likely that when the computational-space method fails, it will continue to fail on the next few integration steps. It will succeed only after the physical-space method has carried the particle past the grid flaw or into the new block. It is a good idea, therefore, to revert entirely to the physical-space method for the next five or six steps before once again trying the computational-space method.

3.5.4 Evaluation

Physical-space integration is a robust method which uses the vector samples read from the grid and solution files. Computational-space integration offers a rapid means to compute streamlines through smooth grids. Mixed-space integration allows the adaptive use of the two methods to ensure fidelity to the data while enabling maximal performance.

To verify this assertion, I have taken the creased grid from the earlier test case and assigned a tag word to each node. Nodes on the crease were marked non-smooth, as were the cells adjacent to these nodes. The same test of circular streamline calculation was then retried, now comparing the mixed-space method to the three strictly computational-space approaches.

As shown in the the figure below, the curves computed by local-conversion (figure 3.14) and by edge-differencing (figure 3.15) are significantly improved by the mixed-space methods. The improved curves on the right-hand sides of these figures are now comparable to the results of the physical space method. The pre-processed central-differencing method is improved somewhat by the mixed-space approach, but even this method still gives rather poor results. Both sets of curves shown in figure 3.16 are unacceptable.

The performance improvement gained from the hybrid method is summarized in table 3.2. The physical space method computes 5000 integration steps in about 7.5 seconds. This time was measured using one processor of a Silicon Graphics 320-VGX, as noted earlier. The computational-space methods compute the same number of output points in 4.5, 4.7, and 2.3 seconds. But, as we have seen, the purely computational-space methods have problems with accuracy. The local and

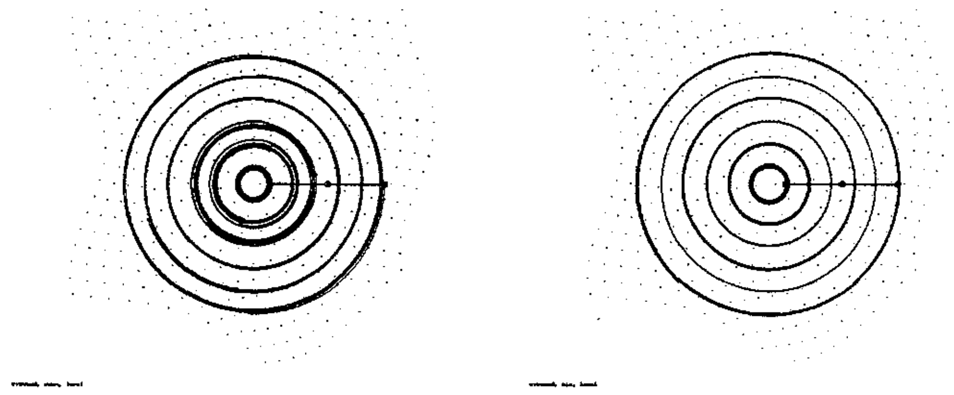


Figure 3.14: *Streamlines computed with local Jacobians in a creased grid, by computational (left) and mixed space (right) methods.*

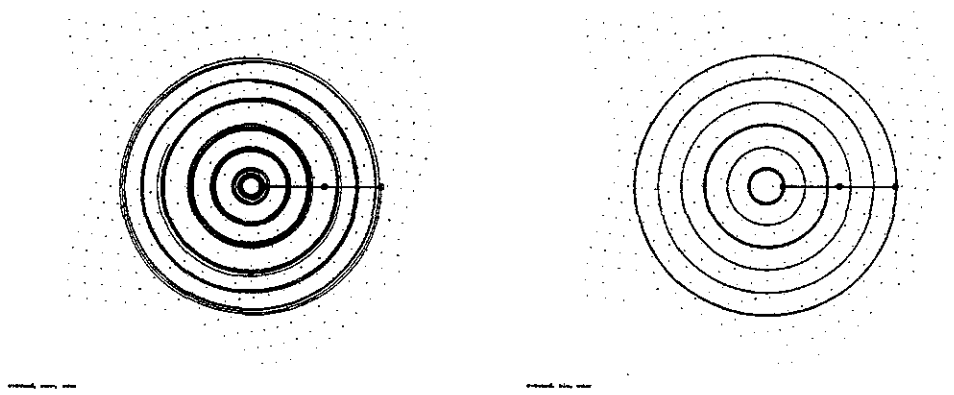


Figure 3.15: *Streamlines computed with edge-differencing in a creased grid, by computational (left) and mixed space (right) methods.*

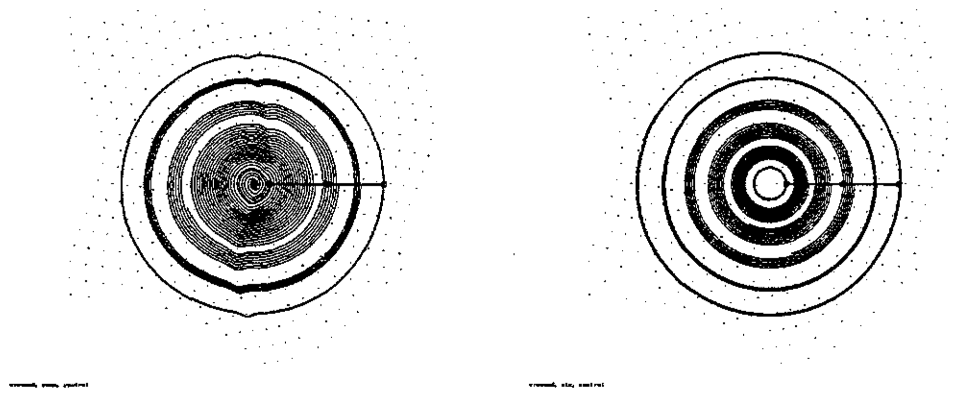


Figure 3.16: *Streamlines computed with central-differencing in a creased grid, by computational (left) and mixed space (right) methods.*

	phys steps	comp steps	CPU seconds
PHYSICAL	5000	0	7.5
COMP LOCAL	0	5000	4.5
COMP TWO-PT	0	5000	4.7
COMP CENTRAL	0	5000	2.3
MIXED LOCAL	1374	3626	6.1
MIXED TWO-PT	1387	3613	6.2
MIXED CENTRAL	1555	3445	4.9

Table 3.2: Performance of mixed-spaced streamline computation.

two-point hybrid methods can compute quite good curves in just over 6 seconds, about 83% of the time required by the physical space method.

This is, admittedly, a rather unsatisfying result and hardly merits the additional implementation difficulty. On the other hand, almost one-fourth of the integration steps in these mixed-space tests were performed in physical space. Some CFD grids contain a lower percentage of non-smooth cells, and would thus allow a correspondingly higher percentage of the particle advection to be computed in computational space. The single-block Shuttle dataset shown earlier [Rizk and Ben-Schmuel 1985] contains 226,800 points. About nine percent of these nodes are located on grid creases of greater than twenty degrees, and would therefore be deemed unsuitable for the computational-space method. The remaining subvolume of the grid can still support this more rapid method.

Even in very smooth grids, the hybrid method has some additional drawbacks. The first problem is that vector samples from both coordinate spaces must be available at some of the node points. The computational-space vector samples are needed at the eight vertices of each smooth cell. Cells with even one non-smooth corner require physical-space vector samples at all eight of its vertex nodes. The nodes which are shared by cells of differing quality must have vector field samples defined for both coordinate spaces. Hence, no single array with intermingled physical-space and computational-space velocity samples can be used. Two separate arrays are needed, although large regions of each will usually be swapped out to disk.

The calculation of the cell tags can be rather time-consuming, depending upon the size of the grid and how carefully one wishes to measure grid smoothness. Unless these tags are computed once in a pre-process phase and saved in a disk file for future use, the additional time required to compute the grid quality metrics is likely to be bothersome to the user.

An additional concern is the rather arbitrary threshold value which must be chosen to determine which regions of the grid are "smooth enough" to support accurate calculation of computational-space values. Perhaps a more acceptable solution would be to compute and store a multi-valued grid quality metric at each node, and then to adjust the limit of acceptability upwards or downwards depending on the current goal of the user. Rapid exploration of the flow field could be enhanced by using the computational-space method almost exclusively, with the caveat that any interesting streamlines should be verified by recomputing these curves entirely in physical space.

This has not been implemented in Flora, since the available computing hardware is fast enough to provide adequate response on most datasets without resorting to this adaptive approach. Larger datasets could be processed on smaller workstations if this method were exploited.

3.6 SUMMARY

The integration of streamlines through physical-space coordinates requires the use of a point-finding method at each new query point. A conversion of the physical-space vector field samples into the computational coordinate space allows the calculation of streamlines to be conducted much more rapidly. Unfortunately, these computational-space methods are unreliable near grid flaws and are untenable across block boundaries.

A set of precomputed *node tags* can allow the adaptive use of the two coordinate spaces. The faster computational-space algorithms are used wherever possible and the more robust physical-space method is applied where grid flaws or block boundaries make such caution advisable. In a heavily creased test grid, the mixed-space method computed streamlines nearly identical to those of the strictly physical-space method, and it did so using 83% of the CPU time. In more typical grids, better performance should be safely attainable.

CHAPTER IV
EXPLOITING SPATIAL COHERENCE

Mixed-space tracing is one approach toward improving the efficiency of streamline calculation. Improvement can also be obtained by exploiting *spatial coherence*; that is, the geometric proximity of the successive query points generated by a numerical integration method.[6]

4.1 CELL CACHING

Most flow visualization software uses a cell-local trilinear interpolating function to define the field values within each hexahedral cell. Under such a mapping, an interpolated field value depends only upon the samples recorded at the vertices of the cell which contains the query point. Rather than indexing directly into a large arrays of grid position and solution data, Flora copies the node position and vector field samples for the eight corners of the current cell into two small buffers of (8×3) words each. This copying isolates the interpolation software from the format of the data files, it reduces the time required to interpolate field values at successive query points in the same cell, and it supports on-the-fly calculation of derived flow field quantities.

4.1.1 Implementing the Caching

Access to the user-supplied flow field samples is mediated in Flora by an instance of a `particle` data structure, which specifies its current location in physical

[6]Most of the material in this chapter has been published in the paper "Improving the Performance of Particle Tracing in Curvilinear Grids" [Hultquist 1994].

coordinates and also in the computational coordinate space defined by the current grid block. The particle structure contains pointers to the large arrays of grid and solution sample data. It also caches a copy of the position and vector field samples for the eight corner nodes of the cell which currently contains the particle.

As a particle is advected through the flow domain, the computational-space coordinates of each new query point are split into their integer and fractional parts. The integers form the index of the node at the lowest-indexed corner of the containing cell. If this cell index is unchanged from that of the previous query point, then the interpolation may proceed with the previously cached samples. This avoids the repeated fetching of these values from scattered locations throughout the large grid and solution data arrays.

When the particle moves into a new cell, then the node position data and field samples for the new cell must be loaded into the particle structure. The three index values (`i,j,k`) are multiplied by the indexing offsets for each block dimension, and the products are summed to form the total address offset for the lowest-indexed node. The additional offsets for the other seven corners of the cell depend only upon the dimensions of the current block. The position and vector field samples are loaded into contiguous memory words in the particle structure, starting from the lowest-indexed vertex and continuing in column-major (FORTRAN) order to the sample data for the diagonally opposite cell vertex:

```
di = 1;
dj = idim;
dk = idim * jdim;
offset = ((i*di) + (j*dj) + (k*dk));
for each component in (x,y,z) {
  src = field_data + offset;
  dst = particle_cache;
  dst[0] = src[ 0 ];
  dst[1] = src[ di ];
  dst[2] = src[ dj ];
  dst[3] = src[ di + dj ];
  dst[4] = src[ dk ];
  dst[5] = src[ di + dk ];
  dst[6] = src[ dj + dk ];
  dst[7] = src[ di + dj + dk ];
}
```

By this collective loading of the samples for all eight corners of the current cell, we can avoid much of the indexing arithmetic which would be required for accessing individual elements from from the grid and solution arrays. This *caching* of the vertex positions and field samples also isolates the interpolation code from the format of the field data, which sometimes interleaves the components of vector samples and sometimes stores the components for each physical-space dimension in disjoint arrays.

This local software caching of the sample values supports the efficient deferred or *lazy evaluation* of field data, such as for computing velocity from the density and momentum samples. Lazy evaluation of flow field data has also been implemented by Kerlick [1990] and by Globus [1992]. If only a few streamlines are to be calculated, this deferred calculation is much more efficient than code which first computes the value of the new field at every node point in the entire grid before computing any streamlines.

If we then save (or *tabulate*) these computed values for use in subsequent queries in this same cell, then calculation of new values is needed only when the particle encounters a new cell. This reuse of computed values can significantly improve the efficiency of advecting particles through some derived fields.

4.1.2 Working Set Size

In any caching scheme, the notion of the *working set* may be applied [Denning 1967]. In a virtual memory computer system, the working set is the collection of memory segments or pages which have been accessed by a process over some specified interval of time. The number of different segments or pages in the working set is called the *working set size*. This is a measure of the *locality* of the process; that is, how many different areas of memory have been accessed by the process over that time interval. A process with a smaller working set will tend to run more quickly, since a higher percentage of its text and data can reside in the faster hardware of the CPU cache.

We may consider the locality of an advecting particle. Clearly, the particle resides at any given moment in a single cell; however, the query points generated by an integration algorithm may fall within neighboring cells. Thus, the speed of particle advection might be further improved by caching the data for more than one cell in each particle structure. A *cache miss* occurs when a newly queried cell is not already loaded in the cell-cache. The data for one of the previously encountered cells must then be overwritten by the data for the new cell. Typically, the cell which is removed from the cache is that which was "least recently used" and presumably (but not always!) the least likely to be used again.

To investigate the effect of cell-cache size on the frequency of cell-cache misses, I computed several streamlines through the smooth test grid. The integration method was second-order Runge-Kutta in physical space, using stepsizes scaled to produce points separated by at most one-fifth of a cell-width. In this test, the momentum was divided by density to find the fluid velocity at the vertices of each

CACHE SIZE	query points	load pos	load vel	cpu seconds
0	10008	36328	10014	13.9
1	10008	1385	1387	7.3
2	10008	1342	1344	7.3
3	10008	1342	1344	7.4
4	10008	1342	1344	7.3

Table 4.1: Effect of cache size.

new cell. The test was repeated several times, with a cache size ranging from zero to four cells. As for all of the results reported in this thesis, the CPU times were measured on a Silicon Graphics 320-VGX, using one 33 megahertz processor for the streamline calculation.

The caching results are shown in table 4.1. Note the great difference in speed between having zero and one cell in the cache, a savings of 48% of the CPU time. For this second-order Runge-Kutta scheme, the locality is quite high and little improvement is gained by having space for more than one cell in the cache. Other methods have somewhat worse locality, as for a higher-order scheme which might generate several query locations within each integration step or a truly adaptive scheme which effectively executes two integration methods in parallel. A cache of two cells or more would allow each method to advance independently, without unfortunate interaction in the particle data cache.

Additional savings could be obtained by recognizing that adjacent cells share four vertices. The already cached values for these shared nodes can be copied directly into four of the slots in the newly vacated cell-cache entry; these samples need not be recomputed from the original flow data. This has not yet been implemented in Flora, but would be worthwhile for vector fields which are more time-consuming to compute.

Summary: The flow field values at each query point are typically defined by an interpolation of the values stored at the vertices of the enclosing cell. By caching *these samples, we can reduce the cost of accessing this data for subsequent query points in this same cell. The software cell-cache can also support the efficient* lazy calculation *of derived field values. By* reusing *these computed values, we can avoid their recalculation for subsequent queries in this same cell.*

4.2 INTERPOLATION

If the position and vector field samples for the current cell are loaded into small contiguous buffers, then a rapid and general implementation of the trilinear interpolating function becomes possible.

4.2.1 Trilinear Interpolation

In the cell-local trilinear interpolating function, the relative weight applied to each vertex sample is the product of three numbers; each one either a computational-space offset or that same offset subtracted from one:

$$\begin{aligned}
\alpha' &= (1-\alpha) \\
\beta' &= (1-\beta) \\
\gamma' &= (1-\gamma) \\
\mathcal{I}_{ijk}(\alpha,\beta,\gamma) &= [\alpha'\beta'\gamma']f_{000} + [\alpha\,\beta'\gamma']f_{100} + \\
&\quad [\alpha'\beta\,\gamma']f_{010} + [\alpha\,\beta\,\gamma']f_{110} + \\
&\quad [\alpha'\beta'\gamma\,]f_{001} + [\alpha\,\beta'\gamma\,]f_{101} + \\
&\quad [\alpha'\beta\,\gamma\,]f_{011} + [\alpha\,\beta\,\gamma\,]f_{111}
\end{aligned}$$

The interpolation weight which is applied to each vertex sample is the computational-space volume bounded by the opposite vertex and the position of the query point. For example, if the offsets (α,β,γ) are all equal to one third, their product of $1/27$

is applied as the coefficient for the sample value (f_{111}) at the uppermost-indexed vertex of the current cell.

4.2.2 Operation Counts

The interpolating function as written above requires 10 additions and 24 multiplications for each component of the field being sampled. The interpolating of a vector in three dimensions would therefore require $3 \times (10+24)$ or 102 floating-point operations. Factoring of common sub-expressions and the pulling of some invariant expressions out of the loop body can bring these totals down to 3 additions and 12 multiplications in a pre-iterative setup phase, with 7 additions and 8 multiplications required for each component of the sampled field. This rearrangement brings the total down to 60 operations in the case of three-dimensional vectors interpolated from the eight vertices of a three-dimensional hexahedral cell.

A different factoring uses a sequence of seven linear interpolations or *lerps* [Hill 1994]. Each lerp produces the weighted average of two values, using an interpolation coefficient (λ) ranging between zero and one.

$$\text{LERP}(\lambda, \vec{p}, \vec{q}) = ((1-\lambda)\vec{p} + \lambda\vec{q}) = (\vec{p} + \lambda(\vec{q} - \vec{p}))$$

A trilinear interpolation may then be computed by using the third computational-space offset γ to combine the eight original vertex samples into four interpolated values. These intermediate results are combined pairwise using β as the interpolation weight, and the final value is computed using α as the weight in a single final step. This method is implementated by seven evaluations of the LERP function, producing a sequence of intermediate values which culminate in the final interpolated result:

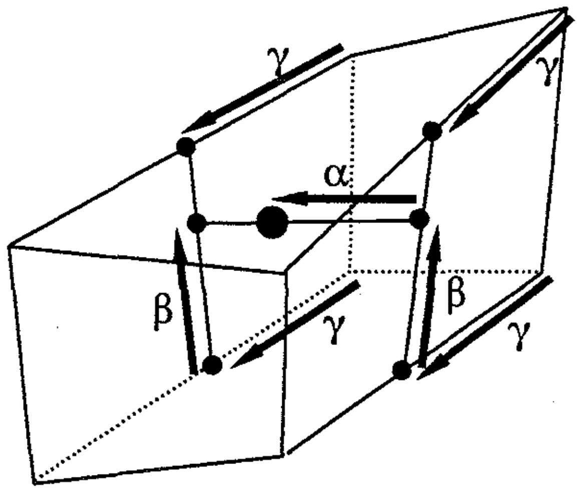


Figure 4.1: Trilinear interpolation by seven linear interpolations.

$$
\begin{aligned}
f_{00\gamma} &= \text{LERP}(\gamma, f_{000}, f_{001}) \\
f_{10\gamma} &= \text{LERP}(\gamma, f_{100}, f_{101}) \\
f_{10\gamma} &= \text{LERP}(\gamma, f_{010}, f_{011}) \\
f_{11\gamma} &= \text{LERP}(\gamma, f_{110}, f_{111}) \\
f_{0\beta\gamma} &= \text{LERP}(\beta, f_{000}, f_{\alpha 10}) \\
f_{1\beta\gamma} &= \text{LERP}(\beta, f_{101}, f_{\alpha 11}) \\
f_{\alpha\beta\gamma} &= \text{LERP}(\alpha, f_{0\beta\gamma}, f_{1\beta\gamma})
\end{aligned}
$$

This method is depicted graphically in the figure 4.1. The first four steps produce the values interpolated along four edges of the cell. The next two steps compute values on opposite faces. The final evaluation yields the interpolated value at the specified query point in the interior of the cell.

This "seven-lerp" implementation requires 14 additions and 7 multiplications for each component of the sampled field, with no provision for pulling common operations outside the loop. The operation counts for each method are summarized

	OPERATIONS		$(n=3)$
METHOD	add	mul	total
SIMPLE VOLUME	$10n$	$24n$	102
FACTORED VOLUME	$3+7n$	$12+8n$	60
SEVEN-LERP	$14n$	$7n$	63

Table 4.2: Implementations of trilinear interpolation.

in table 4.2. The seven-lerp method and the factored version of the volume method have roughly a comparable performance of about 60 operations for vector samples in three dimensions. Each runs in about two-thirds of the time required by the straightforward and non-optimized implementation of the volume method.

4.2.3 A General Implementation

The caching of the vertex samples allows a very general implementation of the seven-lerp method for evaluating the interpolating function. This simple code can interpolate field samples with any number of components within cells having any number of dimensions. For field samples in a three-dimensional cell, the calculation of the interpolated value at a query point consists of three iterations of a loop which repeatedly combines the upper and lower halves of an small and shrinking array of cached samples:

```
num = cache_size;
src = cache_data;
dst = tmp_buffer;
for each offset (...$\gamma$, $\beta$, $\alpha$)
  del =  next fractional offset
  num = num/2;
  lo  = src;
  hi  = src + num;
  DOTIMES(i,num) {
    dst[i] = LERP(del, lo[i], hi[i]);
  }
  src = tmp_buffer;
}
```

The first pass combines the four cached samples on half of the cell with their counterparts on the opposite cell face. These interpolations use the offset γ as the coefficient to produce four new values which are written into a temporary buffer. These intermediate values are then combined pairwise using β as the interpolation weight, again merging the two halves of the array by pairwise interpolation. These results can be written back into the same temporary buffer, since the previous values are no longer needed. These two new values are finally combined into a single result using α as the interpolation coefficient.

These two nested loops, one over each cell dimension and the other over the sample components, should be unrolled for the common and important case of three-dimensional vector samples interpolated within a three-dimensional cell.

Summary: The query point lies within some cell. The position and vector field data at the corners of this cell are copied into a small contiguous buffer. This buffer allows a efficient and general implementation of the interpolating function.

4.3 COMPUTATIONAL-SPACE EXTRAPOLATION

When a particle is advected by physical-space integration, the computational-space coordinates for each new query point must be found. Finding these coordinates typically involves a Newton-Raphson method which iteratively computes a sequence of computational-space positions which lie increasingly close to the specified query location. The iteration is often started from the computational-space position identified by the previous invocation of the point-finding method. A closer starting point can be found by extrapolating, in computational coordinates, some short distance beyond the end of the growing streamline. This improved initial estimate can reduce the number of iterations required for the convergence of the Newton-Raphson method, thus increasing the speed of the physical-space integration of streamlines.

4.3.1 Implementation

In Huen's integration method, two query points are generated by each integration step. Each is placed some distance beyond the end of the partially computed streamline, the first using the local velocity value and the second using an averaged quantity. The point-finding method for each of these query points is typically begun from the computational-space coordinates of the immediately preceding query location.

Let $\vec{p}$ be the computational-space position of the most recently computed point on the streamline. Let $\vec{q}$ be the coordinates of its immediate predecessor on this partially computed curve. Then $(\vec{p} - \vec{q})$ is a first-order approximation of the local value of the computational-space vector field. The point $(\vec{p} + (\vec{p} - \vec{q}))$ is a good approximation of the computational locations of the query points to be used in the next integration step. We might therefore expect that the point-finding method should converge more quickly if the initial point of the iteration $(\vec{\alpha}_0)$ be placed at this extrapolated computational-space location.

If the two most recent points ($\vec{p}$ and $\vec{q}$) on the streamline lie in different grid blocks, then this extrapolation approach cannot be used. The numerical difference between the computational-space coordinates of points in two separate blocks has no useful meaning. Similarly, the extrapolation of the curve can yield an estimate beyond the bounds of the current block. Once again, the extrapolation fails in this case. Such failures should occur in only a small percentage of the integration steps; most blocks are several tens of cells in each dimension and each integration step typically advances the particle by only fraction of the current cell size.

	Extrapolation?		
	none	1st	2nd
Query Points	10008	10008	10008
Cells Loaded	1342	1352	1343
NR invocations	11393	10133	10154
NR iterations	18161	14096	14125
Iters per Invoc	1.59	1.39	1.39
CPU seconds	7.2	6.4	6.5

Table 4.3: Effect of computational-space extrapolation.

4.3.2 Performance Measurement

Several streamlines were calculated with a physical-space second-order Runge-Kutta method, both with and without the computational extrapolation enhancement. In each case, I counted the number of query points generated by the integrator, the number of cells loaded into a particle cell-cache, the number of times the Newton-Raphson method was executed, the total number of iterations of this method, and the elapsed CPU time. These measures are listed in table 4.3.

Extrapolation prior to the invocation of the point-finding method reduces the CPU time by about 11%. This improvement can be attributed to a reduction in the total number of iterations of the Newton-Raphson method, each of which includes the construction and the inversion of a Jacobian matrix. By extrapolating to find a better initial point, we reduce the average cost of each invocation of the Newton method by thirteen percent, from 1.59 down to 1.39 iterations per invocation.

Some improvement is also obtained by reducing the number of times the Newton-Raphson method converges to an offset which lies outside the current cell. When this does happen, the cell index must be adjusted and the method must be restarted in the neighboring cell. By extrapolating in computational-space, the proper cell is used more frequently in the initial attempt and the number of recalculations is reduced. This is shown in the reduction in the number of invocations of the Newton-Raphson method, from 11393 to 10133, the latter being only 125 invocations above the lower bound of once per query point.

Note, however, that the computational-space extrapolation method also resulted in the spurious loading of ten additional cells. These cells were loaded because the extrapolation pushed the initial point into the next cell, even though the new query point had not yet crossed through this face. This extraneous loading is reduced to a single cell by second-order extrapolation, but no net improvement in speed is obtained from this higher-order method.

Summary: By extrapolating the partially constructed streamline, we can more accurately estimate the computational-space location of each new query point. This reduces the number of invocations of the Newton-Raphson method and it reduces the number of iterations executed within that method.

4.4 FURTHER USE OF CELL TAGS

As mentioned earlier, the *iblank* field in a composite grid allocates a full thirty-two bit word to every node point. Voided grid points, which do not carry valid field samples, are marked with an iblank value of 0. Usable node points in the interior of a block are marked with the value 1. Node points in regions of block overlap are marked with negative values, and the value 2 marks nodes on impermeable boundaries.

In Flora, each node point is assigned a *tag word*, which is subdivided into several single-bit flags and a small integer field. Information for each node is stored in its tag, and comparable information for each cell is stored in the node tag of its lowest-indexed corner. These tags may be used to record information about the grid in a form which is more convenient to use than are the iblank numbers.

4.4.1 Valid Cells

As noted before, each tag word contains a single-bit flag called TAG_NODE_VALID. This is set to TRUE for nodes at which the original iblank number was non-zero. The state of this bit indicates whether a node point carries valid sample data. A comparable flag is assigned to each cell and set to the logical-product (and) of the node flags for its eight corner nodes. A fetch of a single tag word and a test of the TAG_CELL_VALID bit-flag within that tag is therefore sufficient for determining the validity of interpolated values within each newly encountered cell.

4.4.2 Wall Cells

An iblank value of 2 marks any node which lies on an impermeable boundary. Vehicle walls are marked in this manner, as are symmetry planes within the flow domain. These node points do carry valid field samples, but if all four corners of a cell face are so marked, then no fluid may pass through that face. PLOT3D handles this situation using a special constraint called *wall bouncing.* By explicitly restricting the curves away from these surfaces, the software avoids one particularly annoying artifact of numerical error: streamlines that leave the flow domain by passing through the skin of the vehicle.

Wall-bouncing is not yet implemented in Flora, but the cell tags would allow the convenient marking of these impermeable cell faces. The integration software could then test certain bit-flags in the cell tag and displace the advancing particle as appropriate in any marked cell.

4.4.3 Overlap Cells

A composite CFD grid contains several partially overlapping or abutting blocks of node points. The node points in the overlap regions carry a negative iblank number. The absolute value of this tag is the number, counting from one, of a locally overlapping block.

The node flag TAG_NODE_DONOR is true at those nodes in the overlap region between blocks; these are nodes which carry a negative iblank value. A cell flag TAG_CELL_DONOR is set to the logical sum (or) of the donor-flag for the corner nodes of each cell. When a query point falls outside the current block or into an invalid cell, then the donor flag of the previous cell is tested. A TRUE value indicates that one or more vertices of the cell lie in an overlap region. The neighboring block can then be searched for the position of the new query point.

Summary: By allocating a tag word to each node, and subdividing these words into subfields, we can encode more information about the grid than is provided by the iblank number. This allows a more convenient and slightly more efficient implementation of certain field querying operations.

4.5 DONOR POINTS

Within regions of block overlap, the position of each node can be expressed in the computational coordinate space defined by the other block. During the solving of the flow field, every such node point *receives* the interpolated field quantities sampled at the corresponding computational-space *donor point.* This exchange of data allows for the eventual calculation of consensus values for the flow field quantities in these regions.

Visualization software must be able to continue the calculation of streamlines through the boundaries which separate adjacent blocks. A negative iblank number indicates which other block overlaps or abuts the current block in some region of interest, but a coarse search of the new block is required before the point-finding method may be applied. Augmenting the grid with donor-point locations eliminates the need for the coarse search and improves the speed of streamline block transition.

4.5.1 Storing the Donor-Receiver Equivalences

In a typical grid, only about five percent of the nodes are located in overlap regions. Because so few node points are receivers, the donor-receiver information can be recorded in a separate small array of records which accompanies the node position data of each block. When a grid is loaded into Flora, the software counts the nodes which carry a negative iblank number. Storage is then allocated to hold the records for that many donor-receiver pairs. This secondary array is indexed by a small integer which is stored in the tag word of each node. Nodes which are not within an overlap region carry a zero index; a non-zero index identifies the donor record for that receiver node. The number of bits presently allocated in Flora to store the index in the tag word is currently twenty. This allows the indexing of over one million receiver points per block while still leaving twelve bits for the storage of node and cell bit-flags.

4.5.2 Computing the Donor Points

The computational-space donor point coordinates are sometimes recorded in a secondary data file (usually called "`fort.2`") which is produced by some grid generation codes. The "Virtual Wind Tunnel" uses these files to speed the transition of particles across block boundaries [Bryson 1992].

In those cases for which these files are not available, the donor-receiver equivalences can be recomputed directly from the node position data and the iblank numbers. In the simplest approach to this problem, the first donor-point for each block is found by a coarse search, followed by the usual stencil-walk point-finding method. The donor points for the receiver nodes along a single line of grid points are then computed by repeated invocations of the point-finding method, each time commencing from the computational-space coordinates of the previously identified donor point.

	EXTRAPOLATION?		
	none	1st	2nd
RECEIVERS	30862	30862	30862
MISSING DONORS	27	29	27
NR INVOCATIONS	71986	55446	55862
NR ITERATIONS	127522	99405	99151
ITERS PER INVOC	1.77	1.79	1.77
CPU SECONDS	70.4	64.7	67.9

Table 4.4: Effect of extrapolation in finding donor points.

We can improve the speed of this calculation by using computational-space extrapolation. The first two receiver node points on a gridline are processed as before. Donor points for subsequent nodes on this same receiving gridline can then be identified by first advancing by the computational-space distance between two previously identified donor points. The Newton-Raphson method is started from this improved initial position, thus limiting the number of invocations of and iterations within the method. As in streamline calculation, this enhancement can be used only if the two previous computational-space points reside in the same donor block, and only if the extrapolated position also lies within that block. Furthermore, this extrapolation is meaningful only when the two previous receiver nodes and the current receiver node are successors along the same gridline.

Donor points were calculated for the nine-block Shuttle grid [Buning *et al.* 1988, Martin *et al.* 1990] shown in Chapter 1. The methods described here was used, both with and without computational-space extrapolation along each gridline of successive receiver points. The values listed in table 4.4 are the number of receivers, the number of receivers for which donors could not be readily found, execution counts within the Newton-Raphson method, and the total expended CPU time.

The use of computational-space space extrapolation reduced the total computation time by only 8%, from 70.4 CPU seconds down to 64.7 seconds. Second-order extrapolation reduced the total number of iterations, but caused a net increase in

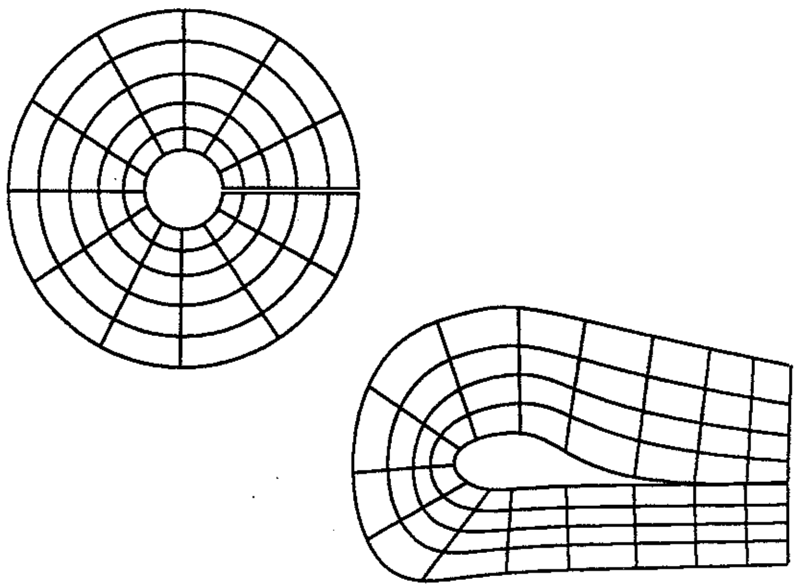


Figure 4.2: O-type and C-type grid blocks.

the elapsed time. Overall, these results are disappointing. This can be attributed to the relatively short sequences of contiguous receiver nodes, and the additional overhead of block-searching required at the start of each new sequence.

Summary: Donor-point coordinates can be used as a means to rapidly continue the calculation of streamlines into neighboring blocks. Computational-space extrapolation can be used to slightly improve the speed at which these donor-point coordinates can be computed from the node position and iblank data.

4.6 SELF-ABUTTING BLOCKS

Single blocks are often wrapped around a cylindrical body such that node points on opposite block faces are coincident. Blocks are also folded around low-speed wings, bringing one block face to abut against itself. These two varieties of self-abutting block, the O-type and C-type topologies [Thompson 1982], exhibit a branch cut in the flow domain which is comparable to the inter-block boundaries discussed earlier.

Negative iblank numbers are not often specified along the seams of these self-abutting blocks, since this juncture is handled using other mechanisms in most flow solver software. Unfortunately, the visualization software has traditionally had access only to the iblank field, and not the auxiliary files which describe the presence and the form of self-abutting blocks. Many visualization packages (including PLOT3D [Buning 1985], FAST [Bancroft 1990], and the test code used by Sadarjoen *et al.* [1994]) thereby fail to continue streamlines across these unmarked branch cuts.

Some flow researchers edit the iblank field to explicitly indicate the connectivity of self-abutting blocks; this is done after the simulation and prior to the visualization of the computed fields. It would be preferable if the visualization software itself were to handle these special cases without the need for manual post-simulation editing of the iblank data. A novel heuristic test can be used to cross these boundaries efficiently.

4.6.1 Implicit Connection

Whenever a streamline exits a block, Flora checks the tag word of the most recent valid cell to determine whether there are donor points assigned to any of its corner nodes. If no donor points are found, then either the particle has exited the computational domain or it has crossed the unmarked boundary of a self-abutting block. If we assume that the node points are coincident across this boundary, then there are three possible points near which the streamline may have re-entered this same block. One of these is found in blocks of the O-type and the two other cases occur in C-type blocks. If any one of these three re-entry positions prove to be the starting position for a successful stencil-walk, then the calculation can be resumed at this new location.

A streamline crossing the seam of an O-type block will exit one block face and re-enter the same block through the opposite face. For a specific example, let us assume that the streamline exits the block through the lower block face in the first grid dimension, near the node with index (i_{min}, j, k). In this case, the point of re-entry will be near the node (i_{max}, j, k) in that same block.

A C-type block may be folded along one of two grid dimensions to bring one block face into a self-abutting state. A streamline which exits the lower block face near the node (i_{min}, j, k) will re-enter the block at either $(i_{min}, j_{max} - j, k)$ or $(i_{min}, j, k_{max} - k)$.

Summary: The branch cuts in C-type and O-type blocks are rarely marked by negative iblank numbers. Visualization packages often fail to continue streamlines which cross these unmarked boundaries. One simple method of streamline resumption requires only the testing of three node points whenever an advecting particle exits a block from a cell which carries no donor records. This resumption technique requires no user intervention. It incurs no startup cost for pre-processing the grid. It requires no storage of additional donor records. Most importantly, it properly continues streamlines through self-abutting blocks which match node-to-node across the branch cut.

4.7 SUMMARY

The successive query points of a numerical integration method occur close together and in a predictable pattern. This behavior can be exploited by caching the sample values for the vertices of one or more cells, lazy calculation of derived fields, and re-use of the computed values. Extrapolating along partially computed streamlines can reduce the calculation required in the point-finding routine.

Precomputing the donor point coordinates avoids the loss of coherence which would otherwise occur at the block boundaries. Extrapolation along sequences of donor points improves the efficiency of donor point calculation. Checking for particle re-entry in self-abutting blocks allows the resumption of streamlines across these previously troublesome boundaries.

All of these methods increase the speed of streamline calculation, and thereby improve the interactive response to the repositioning of rakes within the flow domain. In a test case in a single block, a family of streamlines curves was computed in less than half the time consumed by an unimproved method.

CHAPTER V
CONSTRUCTION OF STREAM SURFACE MODELS

Stream surfaces can be more effective than streamlines for depicting the structure of complicated flow fields. Earlier algorithms for the construction of stream surface models are not very efficient, nor do they produce models which adequately represent the highly convoluted surfaces which are produced in many flows. This chapter presents a new algorithm which efficiently constructs accurate models of stream surfaces.[7]

5.1 OVERVIEW

Just as a streamline is the locus of a single point advected over time, a *stream surface* is the two-dimensional locus of an advected curve. The initial curve is a one-dimensional continuous analog of the seed point used in the placement of a single streamline. Alternatively, a stream surface is the locus of all the streamlines with a seed point on the initial curve.

An ideal stream surface may be approximated by a polygonal model. Figures 5.1 and 5.2 suggest that a few carefully placed rakes can create surfaces which more clearly convey the shape of a complicated flow field. In each of these images the flow travels from the right to the upper left of the images, spiraling with increasing diameter to form a conical helix. Just above the trailing edge of the vehicle body, the innermost coils of the flow turn inward and forward to travel back toward the

[7]An early version of this material may be found in the paper "Constructing Stream Surfaces in Steady 3D Vector Fields" [Hultquist 1992].

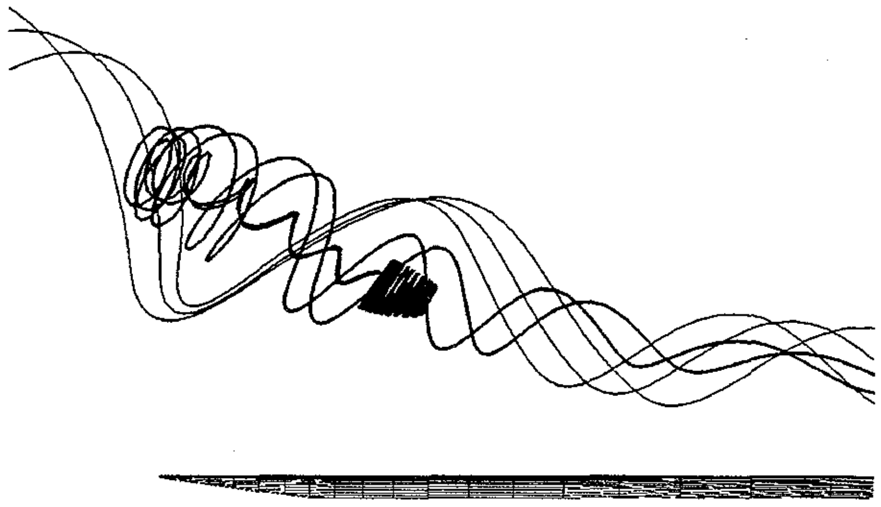


Figure 5.1: Streamlines in the flow over a delta wing.

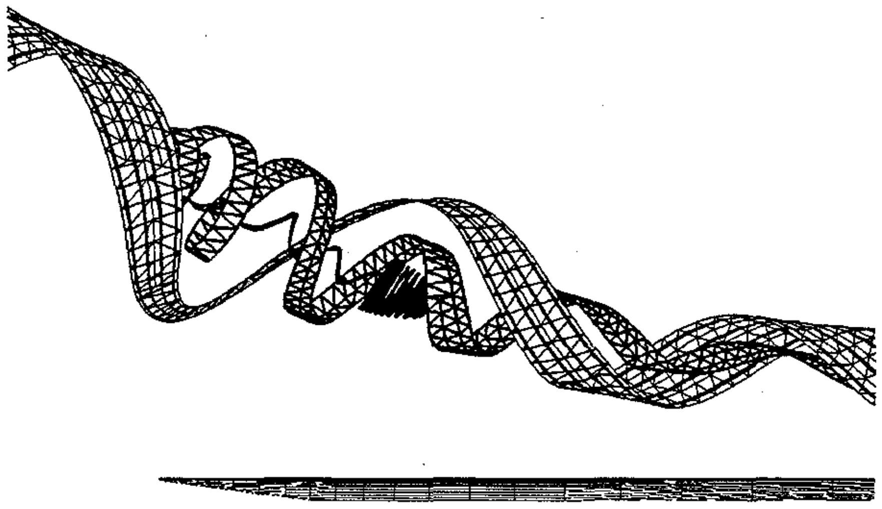


Figure 5.2: Stream surfaces in the flow over a delta wing.

nose of the vehicle. This pocket of spiraling air forms a *recirculation bubble* which greatly reduces the lift on the wing.

The surface provides visual cues which help one to interpret two-dimensional images of this three-dimensional shape. Surfaces can be overlaid with texture to represent additional measures of the flow, may be rendered with variable transparency to mimic the appearance of empirical smoke injections, and can be effectively depicted in monochrome. Surfaces can be formed from circular rakes to create *stream tubes*.

The construction of a polygonal stream surface model typically begins with a discretization of the rake to form a sequence of particles positioned at small intervals along this initial line segment. These particles are the initial seed points from which a family of streamlines is computed through the flow domain. The points along these curves form the underlying skeleton upon which a set of polygons is then defined. Surface normals are computed over the mesh and the result is displayed by the rendering hardware of a graphics workstation. Surface construction algorithms differ in how they adapt polygon size to the local deformations of the surface and also in the manner in which they allocate the processing effort used to advance each particle.

5.1.1 Parameterization

A stream surface is a two-dimensional parametric surface embedded in the three-dimensional domain of a flow velocity field. An ideal stream surface is the locus of an infinite number of streamlines, each uniquely identified by the fractional offset $s \in [0...1]$ of its seed point along a continuous originating rake. The surface supports a second family of curves, which are the positions of the advected rake at an infinite sequence of downstream displacements. These cross-flow curves are called *timelines*, and these may be labeled by a parameter $t \in [0...\infty]$.

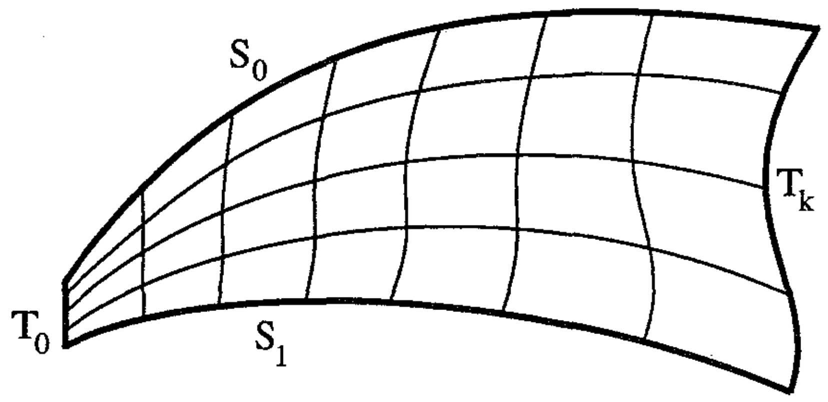


Figure 5.3: Parametric space over a stream surface.

These two families of curves define a two dimensional coordinate space (s, t) over the surface, as illustrated in figure 5.3. The surface is bounded on the upstream edge by the rake itself, which defines the initial timeline (t_0). The surface is bordered by the streamlines (s_0) and (s_1). These are joined by an infinite sequence of timelines (t_k).

The simplest construction method creates a quadrilateral mesh of points at regular intervals in the (s, t) coordinate space. Each streamline (s_i) may be computed by numerical integration using a fixed stepsize to produce a sequence of points (s_i, t_0) through (s_i, t_n). An adaptive stepsizing method could be used, if followed by a resampling of the curve at some fixed interval. Volpe [1989] and Belk *et al.* [1994] implemented such a scheme, computing a family of streamlines and joining each adjacent pair of curves with a sequence of quadrilaterals to form a set of contiguous ribbons. This approach is adequate for well-behaved flows, but the surfaces embedded within many flow fields twist and fold greatly. This deformation complicates the construction of accurate models.

The distortions to which stream surfaces are subjected may be categorized into those caused by shear in the flow field, by divergence in the local plane of the surface, by local curvature, and by convergence. Shearing tends to stretch the timelines locally; divergence in the local tangent plane of the surface causes adjacent streamlines to separate. Surfaces often fold lengthwise, introducing high curvature along timelines. Finally, in many engineered flow fields, streamlines within the flow are pushed together, causing a shrinkage of the distance between neighboring particles.

5.1.2 Earlier Work

Krueger [1991,1992] implemented a proof-of-concept system for the visualization of flow fields. One of the visualization models supported in this package was a simple stream surface, constructed with a polygonal tiling of adjacent pairs of streamlines. Ribbons which exceeded a specified width were truncated, with the downstream portion of that ribbon substituted by two narrower ribbons.

Helman [1989,1990] implemented a system which identified the topologically significant curves on the solid boundaries of a vehicle. The software then constructed the topological separatrices, that is, the stream surfaces which emanate from these lines and extend into the surrounding flow. Helman used a ribbon tiling approach to construct these surfaces, and refined this representation by splitting the widest ribbons down their entire length.

These adaptive ribbon-splitting methods result in scattered access patterns to the memory pages and cache lines of the sampled flow data, since individual complete streamlines are computed through the flow domain. Both methods also compute an overabundance of points in regions of convergent flow. When used in an interactive context, both methods also require the repeated display of the entire scene, as refined stream surface models are successively substituted for their coarser predecessors.

Eder [1991] developed a distributed system for computing stream surfaces. After the user specified the rake for a new surface, the system computed between fifty and two hundred streamlines using a vectorized code running on a remote

Siemens-Nixdorf VP-200 mini-supercomputer. These curves were copied into the memory of a workstation, then a two-pass filtering removed those points which did not significantly contribute to the accuracy of the surface model.

This filtering reduces the cost of displaying the completed model, but excess calculation is performed to produce points which are later removed from the model. This is acceptable in Eder's approach, since the latency of the network connection would unduly penalize the adaptive placement of advecting particles. In a strictly local implementation, excess calculations may be more easily avoided. Furthermore, when these streamlines diverge, the approach taken by Eder can leave sparsely sampled regions in the finished polygonal sheet. The regions would need to be "filled in" by the calculation of additional streamlines, and by the insertion of those points into the previously computed model.

To support the interactive placement of stream surfaces, we should like to have methods for rapidly advecting particles through the flow domain. The two previous chapters have described such improvements. We now require a method to manage the advection of a set of particles to generate an adaptively sampled representation of a surface. Furthermore, it would be helpful if the interim results could be efficiently presented to the user, to allow early judgment of the quality of the rake placement.

5.2 THE ADVANCING FRONT METHOD

I have devised an improved algorithm which advects a row (or *front*) of particles in a tightly-clustered group. This *advancing front* algorithm interleaves the advection of the particles with the creation of polygons across the trailing downstream edge of a growing surface. During the advance of the front, the relative positions of the particles are examined to determine if they adequately represent this cross-flow curve which spans the surface. Additional particles may be added to the front,

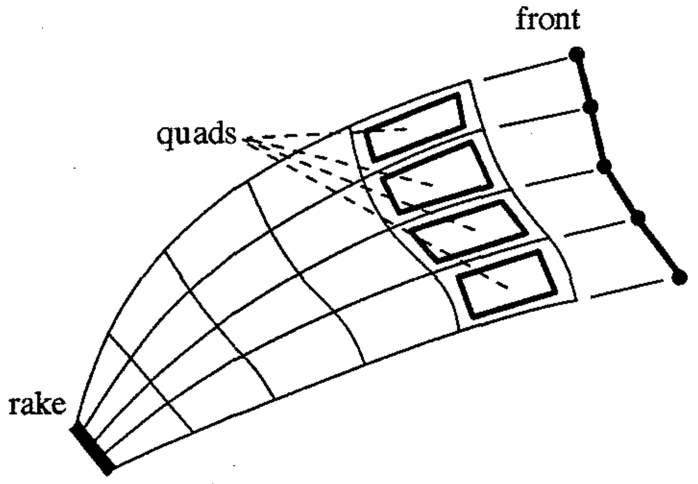


Figure 5.4: Appending quadrilaterals to a growing surface.

or particles may be removed, to maintain an adequate sampling density over the growing model. This approach more efficiently accesses the sampled field data and provides better control over the sampling density across the width of the growing model. Each new triangle can be painted into an otherwise static background image, thereby reducing the performance required from the display hardware.

5.2.1 Implementation

In the simplest variant of the advancing front methods, all of the particles in the front are repeatedly advanced some short distance through the flow domain. Each particle may be advanced by some fixed distance in physical or computational space, or by some fixed integration stepsize, or even by an amount chosen for each particle by some adaptive-stepsizing integration method. However this distance is selected, every particle is advanced by some amount during each iteration of the algorithm. After every particle has been advanced, a row of quadrilaterals is appended to the downstream edge of the partially constructed model. Figure 5.4 shows a surface computed over some number of iterations, with the new positions of the advecting particles just beyond the end of the polygonal sheet.

If the sampling density is deemed too high along any portion of the front, then some of the particles may be deleted from this advancing cross-flow curve. When a portion of the front requires more samples (as is more often the case), then additional particles must be inserted into this sequence. Of course, the removal or the insertion of particles requires some local patching of the otherwise regular polygonal tiling of the model.

This *lockstep advancement* allows good control over the cross-flow sampling density of the model. Interim depiction of the surface during its construction is simple, since the newly created polygons can be immediately rendered into the depth and color buffers of an otherwise static background image. Even the pattern of data access is improved, since the particles advance in company and (barring recirculation of the flow field) any region of the flow domain will be visited at most once during the construction of each new surface. Finally, particles are advected only when the resultant points will be used in the model; excess calculation and oversampling are avoided.

Unfortunately, advancement by a fixed integration stepsize will favor those particles in more rapid regions of the flow field, causing these particles to advance more rapidly and unduly lengthening the front. Similarly, an adaptive stepsizing method will advance farther those particles which travel through regions of relatively constant velocity. Siclari [1989] and Volpe [1989] resampled the streamline curves to generate a sequence of points separated by a constant physical-space distance. This properly handles variations in the velocity magnitude, but raises problems in helical flow. The particles on the "outside lane" of each turn will fall further behind those which must travel a shorter distance.

The result in any of these cases is an advancing front of increasingly greater length, as some of the particles outdistance their companions. More particles are then needed to represent this curve, even though the surface might otherwise be

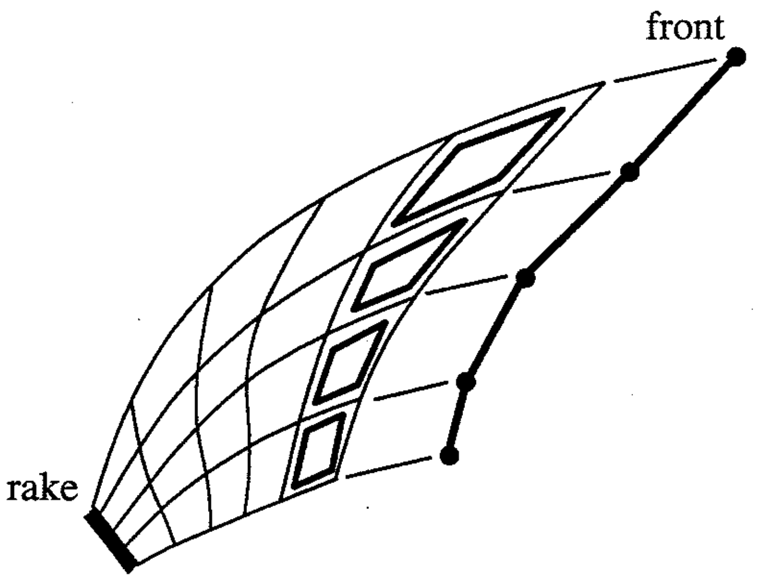


Figure 5.5: Shearing of a front in lockstep advance.

rather simple and the front relatively straight. This situation, illustrated in figure 5.5, produces an overabundance of points in a model, and increasingly skewed and stretched quadrilaterals.

5.3 HANDLING SHEAR

When the particles are advanced in a lockstep fashion, shearing of the front can greatly increase the number of particles needed to adequately represent the front. An *orthogonal advance* scheme avoids this problem by advancing the laggard points more rapidly. This tends to keep the front locally orthogonal to the flow velocity, thus reducing the length of the front and reducing the number of particles which must be advected through the flow domain. The resulting surface model is composed of well-shaped polygonal facets with sizes adjusted to the local two-dimensional curvature of the surface.

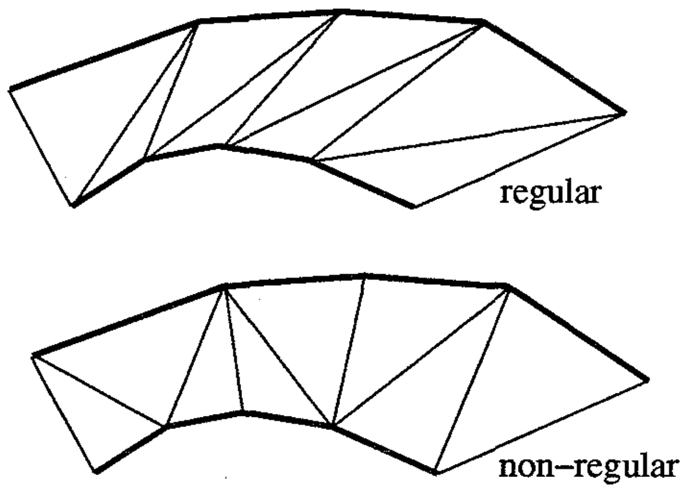


Figure 5.6: Regular and non-regular tiling of a ribbon.

5.3.1 Ribbon Tiling

The gap between any two streamlines can be bridged by a sequence of triangles to form a ribbon. Each new triangle shares an edge with its predecessor and obtains its third point from one of the two bordering streamlines. Two curves with $(n + 1)$ points in each contain $2n$ edges, each of which serves as the base for a new triangle. This ribbon can be formed in $(2n\ \textit{choose}\ n)$ or $((2n)!/(n!n!))$ different ways. Any one of several algorithms may be used to select a suitable tiling from among these numerous possibilities.

The simplest tiling method alternates sides, thereby consuming points from the two streamline curves at an equal rate. Krueger [1991] used this method in his "VideoDesk" application. This tiling method is satisfactory when the flow field is well behaved; however, in many flows this simple approach can produce long skinny triangles and a poor representation of the true surface. Irregular tiling of adjacent streamlines can be used to avoid the problems encountered with the regular tiling methods. The construction and the display of such ribbons is slightly more difficult, but the shape of each triangle is closer to equilateral (figure 5.6). Nicely shaped triangles are particularly important when the surface is drawn in wireframe or with outlined polygons. They also improve the lighting and texturing of the surface, since color and texture values are usually interpolated linearly across each surface facet.

The surface which is globally optimal in some scalar measure can be obtained by the method of Fuchs, Kedem and Uselton [1977]. This algorithm uses a dynamic programming approach to find a tiling for two closed loops of points, such that some scalar quality measure is minimized. Since the initial points of two adjacent streamlines $((s_i, t_0)$ and $(s_{i+1}, t_0))$ must be connected to one another, a simpler version of the general algorithm may be used. Helman [1989,1990] used this variation to construct ribbons of minimal total surface area.

5.3.2 Incremental Tiling

The globally minimal algorithm requires that each streamline be computed to its full length before any triangles can be created. In contrast, an *incremental tiling* method creates new triangles beginning at the upstream end of the ribbon. It creates triangles in succession, down the length of a growing ribbon, using only local information. The particles can be advected just one step ahead of the end of the growing ribbon, and the triangles can be drawn into the static background image as that ribbon is extended through the flow domain. The user may then evaluate this interim model, and perhaps abort the ongoing calculation by shifting the rake to a new location.

Figure 5.7 illustrates the implementation used in Flora, in which a `ribbon` structure is used to connect the points generated from a pair of neighboring particles. These two curves of points are joined with a sequence of triangles, each appended to the end of the growing ribbon. The surface itself is composed of a family of contiguous growing ribbons, which are extended together through the flow field.

Flora represents each advecting particle with a `particle` structure, which maintains the information required to advance a particle through a vector field.

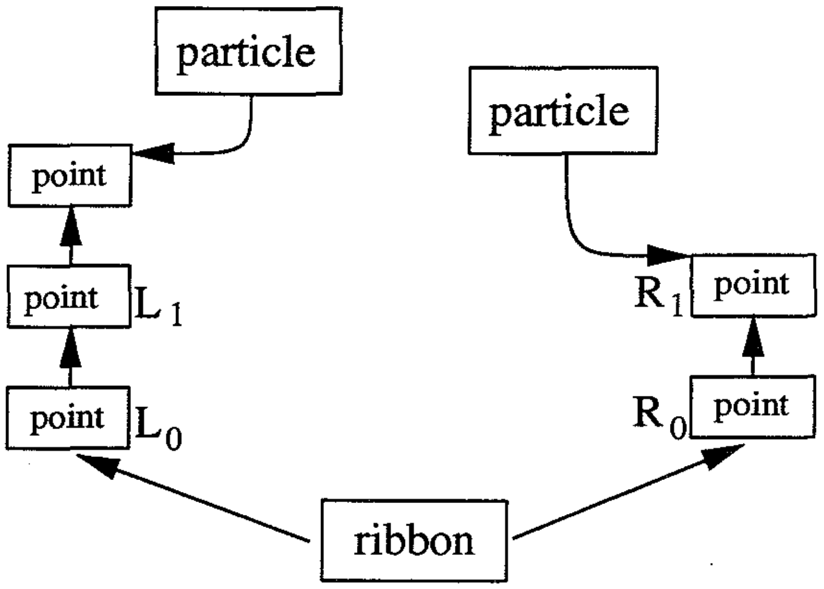


Figure 5.7: A ribbon structure connecting two particles.

Flora maintains a physical-space parameter `facet_size`, which is the user-preferred length of the triangle edges. Each particle is advanced with a stepsize chosen within the particle integration code. The resulting sequence of successive particle locations is then filtered, such that the points used in the surface construction are separated by at least this desired distance. Points from this filtered sequence are represented by a linked list of instances of a `point` structure. The position of each point is recorded in both physical space and computational space. These coordinates are copied into output arrays that define the computed model. Each point structure also stores the integer index at which that point's data was written to the output buffers. A triangle is represented by the integer indices for the data of its three vertices. Each vertex is written only once, but most indices appear in the index triple for several triangles.

At each step in the ribbon-growing iteration, the two most recently joined points (L_0, R_0) and their successors (L_1, R_1) form a quadrilateral which must be split along one of its two diagonals. This is illustrated in figure 5.8. The shorter of

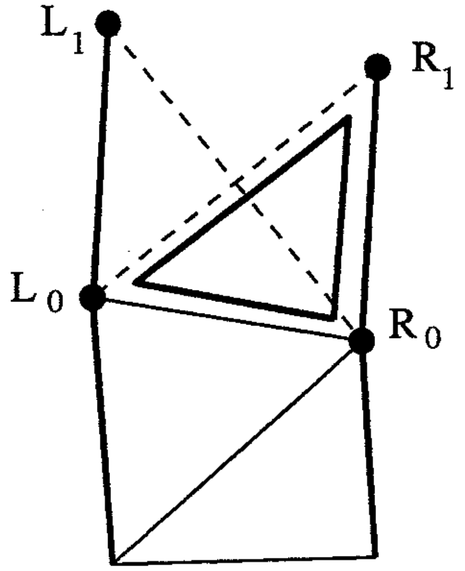


Figure 5.8: Appending a triangle to a ribbon.

the two diagonals ($|L_0 - R_1|$ or $|L_1 - R_0|$) forms an edge of the new triangle, the new end of the ribbon, and the base of the next quadrilateral. In this case, the length $|L_0 - R_1|$ is the shorter and the new triangle is constructed using a new point from the righthand curve. The data for this triangle is written to the output buffers, and the point R_0 is released. This is implemented by decrementing the reference count for this item. When that count goes to zero, the point structure is returned to a free-storage list.

The ribbon-growing is implemented in code similar to this:

```
L0,L1, R0,R1 ←  next quad
lft_diag     = length(R0,L1);
rgt_diag     = length(L0,R1);
lft_advance = (lft_diag < rgt_diag);
if (lft_advance) {
  write_triangle(L0,R0,L1);
  free_point(L0);
} else {
  write_triangle(L0,R0,R1)
  free_point(R0);
}
```

The four vertices of the current quadrilateral are obtained; perhaps after advecting one or both particles an additional step through the flow. The two diagonals are measured, and the shorter is selected. One triangle is written to the output buffers, and one point structure is possibly released. Note that the "`length`" of each diagonal may be computed as a sum-of-squares; no square root is required since only a relative comparison of the two lengths is performed.

This *local minimal width* algorithm can produce poor results when the distance between the streamlines is significantly greater than the distance between successive points on either of these curves. In practice, when the curves are "reasonably" close together, this incremental method produces quite acceptable results.

This simple tiling algorithm allows the construction of a single ribbon, proceeding immediately behind the advance of a pair of neighboring particles. This technique can be combined with a scheduling method which determines which of several ribbons should be advanced next. This adaptive advance is used to combat the effect of fluid shear, thereby shortening the overall length of the front and reducing the number of particles which are needed to generate the model.

5.3.3 Implementing the Orthogonal Advance

Each step of the surface advance begins with the creation of a single triangle at the end of the leftmost ribbon. If the new point for this triangle is selected on the righthand streamline, then the second ribbon will lag behind the first. To erase this deficit, triangles are appended to the second ribbon until the two ribbons are once again the same length along the shared streamline curve. If necessary, the third ribbon is brought abreast with the second, and so on.

By extending the ribbon-growing code from above, we now have the *orthogonal* surface-growing function shown below. In this routine, each ribbon is advanced at least until it has reached the same point as its predecessor ribbon along the shared curve. The **caught_up** flag is then set, and no further advancement along the shared curve will be allowed during this iteration. Additional triangles may be added to the ribbon as long as the new triangles consume points on the righthand streamline and as long as the leading edge of each new triangle is shorter than that of it predecessor.

```
caught_up = FALSE;
prev_diag = 0;
while (1) {
   —- select the diagonal
  L0,L1, R0,R1 ← next quad
  lft_diag  = length(L1,R0);
  rgt_diag  = length(L0,R1);
  lft_advance = (lft_diag < rgt_diag);

  if (caught_up AND
      (lft_advance OR (rgt_diag > prev_diag)))
    break;

   —- output the triangle and free the point
  if (lft_advance) {
    write_triangle(L0,R0,L1);
    free_point(L0);
    prev_diag = lft_diag;
     —- we have pulled aside predecessor
    caught_up = TRUE;
  } else {
    write_triangle(L0,R0,R1)
    free_point(R0);
    prev_diag = rgt_diag;
     —- let neighbor catch up with us
    advance_ribbon(right_neighbor(self));
  }
}
```

An example is shown in figure 5.9. The first and second triangles are constructed at the base of ribbon A. This introduces a debt for ribbon B, which lags behind A along the common streamline. The third triangle is added to ribbon B, but now this incurs a debt for ribbon C. C must be advanced to bring it alongside B, then triangle number 6 is appended to C to shorten its trailing edge. Ribbon

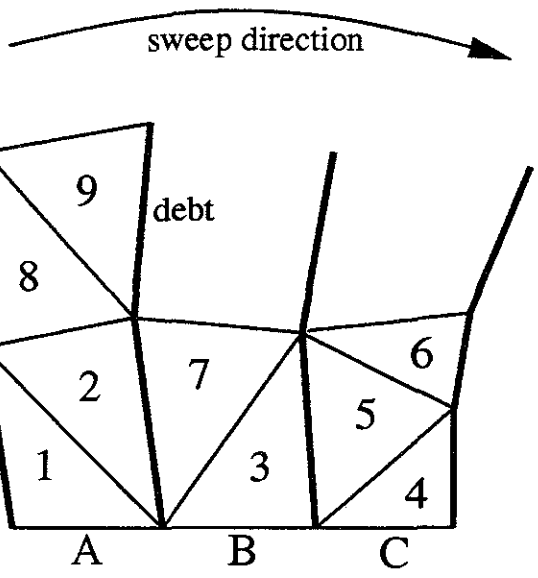


Figure 5.9: Extending the surface using orthogonal advance.

B is then advanced some more and this finally removes its deficit with respect to the first ribbon. At this stage, triangles has been appended to each ribbon, the surface consists of seven triangles, and it once again has a smooth downstream edge. The surface construction continues with the creation of two more triangles, which introduce the need for once again extending the second ribbon. The advance of the front begins with one or more triangles appended to the first ribbon, and with any mismatch along shared streamlines subsequently eliminated by the extension of the neighboring ribbons.

(The labels *left* and *right* in this example are simply conventions, since the rake is arbitrarily oriented in three-dimensional space. Indeed, the actual implementation performs the sweeps in alternating directions across the front. This *boustrophedonic*[8] implementation slightly improves the locality of data access, better handles regions of high curvature, and is fun to watch.)

[8]*boustrophedon*: (Written) alternately from right to left and from left to right, like the course of the plow in successive furrows; ..., *Oxford English Dictionary*

5.3.4 Recovering the Timelines

When advanced with the orthogonal method, the triangles which are drawn into the image are not drawn in a strictly increasing time order. Instead, particles are advanced to maintain a short front length. This can produce a misleading impression of the flow behavior.

A more faithful representation of the growth of the surface could be implemented by deferring the display of triangles. The particles would be advanced as before, with triangle creation still directly associated with an orthogonal advance of the front. However, new triangles would not be immediately displayed, but would instead be placed in a list which is kept sorted by the time parameter of the newest vertex of each triangle. Triangles would be displayed only after triangles had been created across the entire front. Triangles from the tail end of the sorted list would then be drawn in the proper temporal order. This decoupling of creation and display would consist of two tasks: one which advances particles according to spatial constraints, and the other which renders the resulting triangles in temporal order. This would yield a more visually intuitive development of the image.

Another approach to preserving the temporal information is to apply a striped texture to the polygonal surface. These *timeline stripes* depict the relative speed of the flow across the surface, even after the surface construction has been completed. Where the flow increases in speed, these stripes become wider and further apart. Figure 5.10 shows striped texturing overlaid on a set of outlined triangles. Note that the stripes are not aligned with the triangle edges. (Also note that the textures are linearly interpolated across each facet, thus a slight local distortion of the timelines may be incurred.)

The deferred display of new triangles provides a more physically accurate representation of the growth of a surface. The timeline texturing conveys temporal

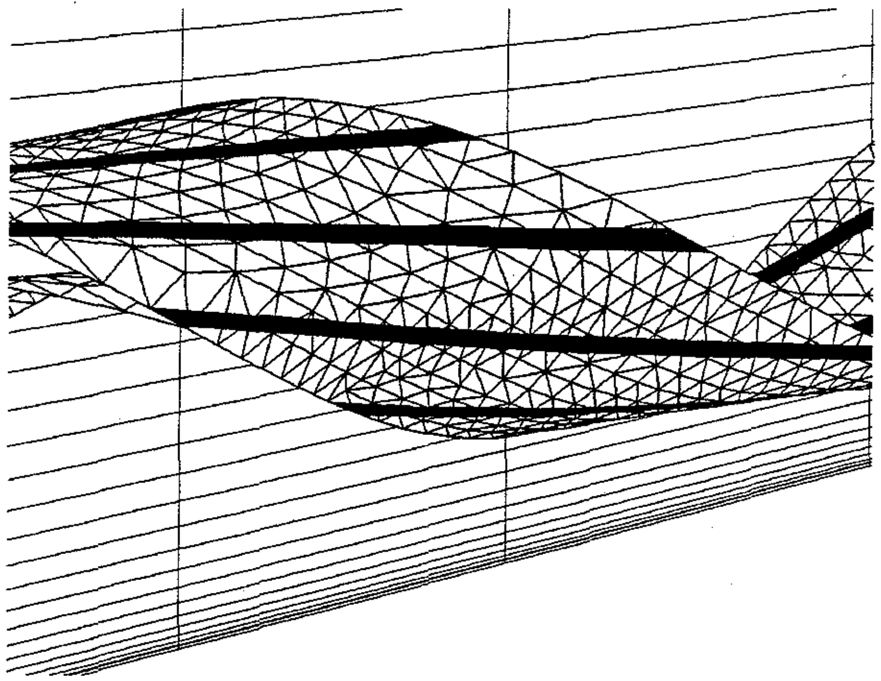


Figure 5.10: Timeline textures on a stream surface model.

information in static images. The Flora application does support texturing of timelines, but the deferred display has not been implemented. Bear in mind that, in each case, what is represented is the relative velocity of a steady flow or the *pseudo-time* frozen in a single instantaneous snapshot of an unsteady flow field.

5.4 HANDLING DIVERGENCE

Many flow fields exhibit regions of strong divergence, in which a pair of neighboring streamlines are separated by an increasingly greater distance.[9] gap would grow increasingly wider, and this would produce a coarse and possibly misleading depiction of the flow structure. We may improve the stream surface model by introducing additional particles in these regions, thereby splitting some of the ribbons lengthwise. To implement this refinement, we must determine when new particles

[9]A pair of streamlines may separate even in a flow which is strictly divergence-free in three dimensions ($\nabla \cdot \vec{u} = 0$).

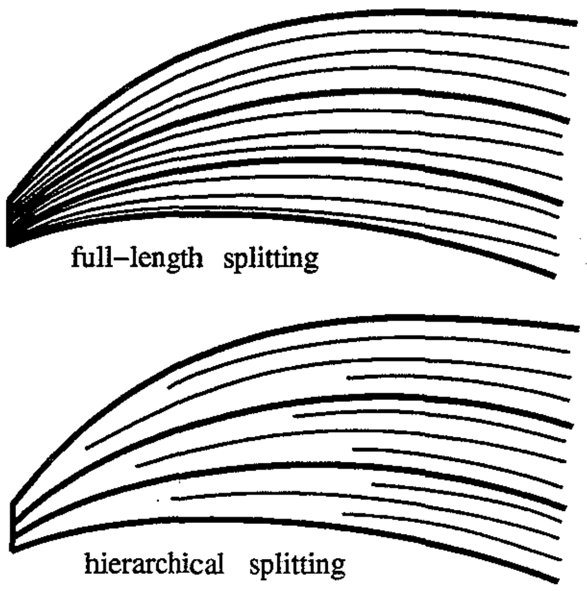


Figure 5.11: Full-length and hierarchical ribbon splitting.

are needed, where each new particle should be placed, and how the polygonal tiling of the surface must be modified to accommodate this new sequence of points.

5.4.1 Hierarchical Splitting

Helman [1989,1990] computed coarse stream surface models using a polygonal tiling of a family of streamlines. He then refined these models by splitting each overly wide ribbon along its entire length with a new streamline. Each wide ribbon was then replaced with two narrower ribbons which shared the new streamline as a common edge. Under this method, the distribution of points over the surface tends to be biased in favor of the upstream portion of the surface, which will be narrower in many flow fields. In the extreme case, the seed points of adjacent streamlines can be separated by very tiny distances, and yet the resulting curves can separate quite widely in the far downstream regions of the surface model. Full-length ribbon-splitting can therefore produce too many points in the upstream end of a stream surface model, and yet create too few points in the downstream end.

This imbalance can be avoided by the *hierarchical splitting* of ribbons, in which ribbons are truncated when their width exceeds some tolerance [Krueger 1991]. Only the unacceptably wide downstream portion of a ribbon is then replaced by a pair of narrower ribbons which more accurately represent that portion of the stream surface. Figure 5.11 demonstrates how this partial splitting of ribbons more evenly distributes sample points over a widening surface. The advancing front method enables one to easily implement hierarchical splitting of ribbons.

5.4.2 Splitting a Ribbon

Some evaluation must be made to determine when a new particle should be inserted into the front. (Here we shall consider only the effect of divergence in the local plane of the surface; adjusting the sampling density in response to changes in surface curvature will be considered later.) For now, it is sufficient to consider the distance between neighboring particles, and to insert a new particle between any pair of particles which drift overly far apart.

In Flora, a ribbon is split by the insertion of a new `particle` structure and a new `ribbon` structure into the linked-list which represents the front. The particle advection code produces new particle locations which have been filtered to generate a sequence of point structures separated by the user-specified `facet_size` distance. Each new triangle is constructed from three of the points taken from the current quadrilateral (L_1, L_0, R_0, R_1). Flora introduces a new particle when the width of the current quadrilateral $|L_0 - R_0|$ grows to more than twice the facet-size.

One new triangle, shown in figure 5.12, must be added to the surface to make the transition from one ribbon to its two narrower replacements. The split introduces a new triangle (L_0, M, R_0) to the surface, followed by two additional triangles which fill what remains of the original quadrilateral. The two new ribbons share point M, which also acts as the seed of the new streamline.

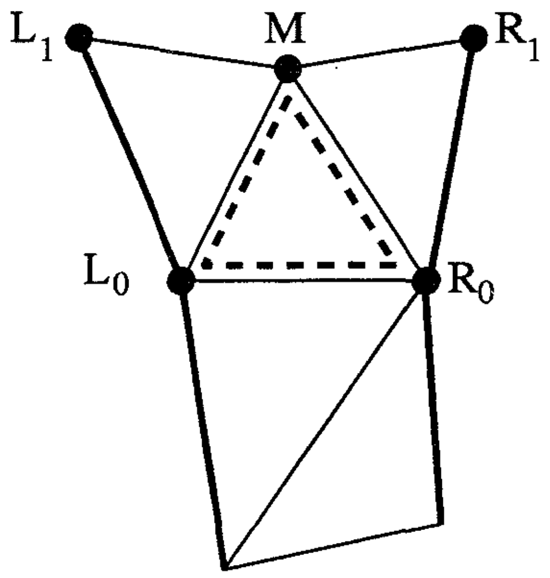


Figure 5.12: Splitting a ribbon with a new particle.

5.4.3 Placing the New Particle

When a new particle must be added, its position M is initially identified only by its location in the parametric coordinate space of the surface. This new particle must have an (s, t) location roughly between that of the points L_1 and R_1. Three-dimensional physical and computational-space positions must then be determined which correspond to this parametrically specified location.

The simplest method interpolates in physical coordinates midway between L_1 and R_1. This was the method used by Krueger [1991] in his hierarchical splitting; the starting position of the new streamline was found by linear interpolation in physical space midway across the trailing edge of the truncated ribbon. This introduces significant error into the surface, since the new particle is positioned by linear interpolation at a place in the surface which has just been deemed to be inadequately sampled.

The physical and computational coordinates of a particle at (s_j, t_k) may instead be found by computing a streamline curve from a point $(s_j, 0)$ on the initial rake. In a typical surface, the ribbon which is to be split is quite narrow along a large percentage of its length. Integration of a new curve which bisects the entire ribbon requires significant calculation, most of which is spent bisecting a very narrow gap. In the extreme case, the initial points of the two bordering streamlines differ in their computational coordinates by an amount only slightly greater than the limits of machine floating-point precision. A ribbon with an initial width of almost zero cannot be reasonably split any further, since the limitations of the floating point representation prevent the accurate calculation of new seed locations in this narrow space.

I have implemented a compromise method which produces new particle positions with good accuracy and low cost. This approach periodically saves the particle locations across the entire front. When a ribbon must be split, a new particle location is interpolated from this saved data, midway across the ribbon and some distance upstream. The new particle is then advected forward to its place on the front. The three-dimensional position of this new particle is more accurate than a locally interpolated point would be, since it was obtained by interpolation in a well-sampled region of the surface. This improved accuracy is bought at relatively low computational cost, just the few integration steps needed to advect the new particle to its position on the current location of the front.

5.4.4 Ripping the Surface

The introduction of new particles into the front serves well to adapt the sampling density of an expanding surface. But when a portion of the front is stretched very rapidly, it is sometimes preferable to sever the front and to continue the independent advancement of two separate portions.

Consider the flow illustrated in figure 5.13. Two neighboring streamlines, once very close together, reach a place in the flow at which they begin to travel in almost

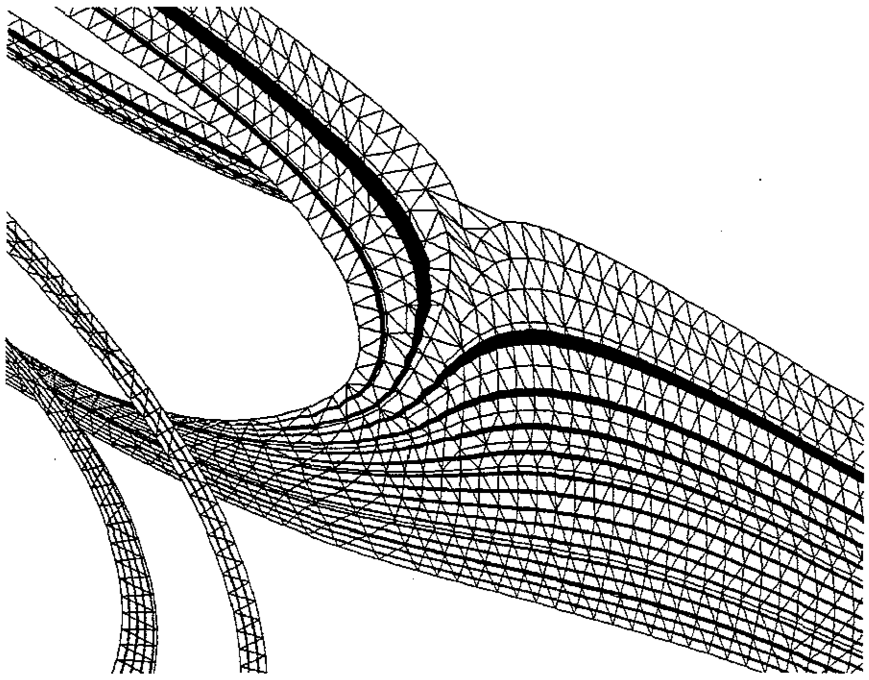


Figure 5.13: Ripping a heavily divergent surface.

opposite directions. The ribbon between these two streamlines is truncated and the two curves become the bordering edges of two separate fronts. These independent groups of particles are then advanced independently by interleaved processing of all of the active fronts.

The comparison of the direction of the neighboring particles may be implemented most easily with a dot product of the velocity vectors. If the particle directions differ by greater than ninety degrees, then this dot product will be negative. Ribbons are truncated when this occurs.

In a commonly encountered situation, neighboring streamlines pass above and below a wing, as shown in figure 5.14. If the facet-size is larger than the thickness of the wing, then the dot-product test may not detect the imposition of this obstacle in the path of the growing ribbon. Performing the same test with computational-space velocities, however, does properly handle this common case. The particles in this figure travel through an O-grid which has been wrapped around the wing. As the

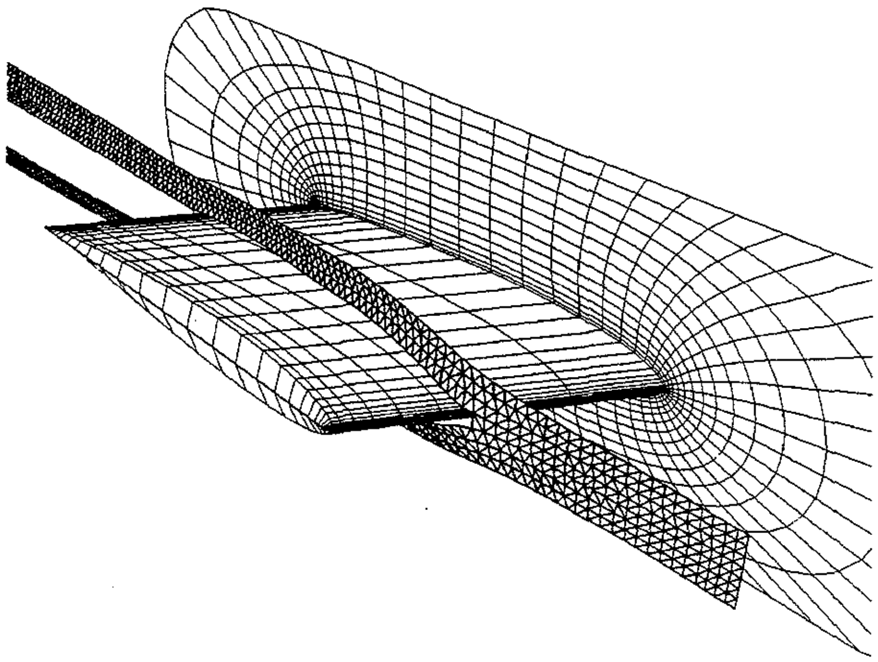


Figure 5.14: Ripping a surface on the leading-edge of a wing.

front nears the leading edge of the wing, the particles above and below the wing begin traveling in almost opposite directions in the computational coordinate space defined by the surrounding grid block. The surface is therefore torn just slightly in front of the wing.

In many grids, the cell height along the **k** or ζ dimension is quite small relative to the other two computational space directions. The dot-product of computational-space vectors may be negative even for closely aligned vectors, if those vectors are aligned with a **k**-gridplane. A somewhat more reliable, albeit non-general, test of vector angles ignores the third computational component, and considers only the projection of the computational-space vectors into the two dimensions of ξ and η. This refined test properly handles most ripping in O-grids, and does not inadvertently tear the surface in the compressed cells of the boundary layer.

5.5 HANDLING CURVATURE

Adaptive stepsizing is often used in numerical integration to adjust the sampling of curves in response to changes in the curvature of the flow. In the parametric notation of our stream surfaces, the smaller stepsizes are used to adequately resolve regions which have relatively large values of $\partial^2\vec{x}/\partial t^2$ at a point $\vec{x}$ along some streamline curve. For stream surface construction, we would also like a method for adapting the sampling in response to changes in $\partial^2\vec{x}/\partial s^2$; that is, changes in the curvature of the surface along the cross-flow dimension.

5.5.1 Measuring Surface Curvature

Local curvature of the surface in both dimensions may be easily estimated by measuring the angle between adjacent unit-length surface normals. This is convenient, since these normals must be computed anyway for the rendering of shaded polygons. If the angle between the normals on adjacent facets is greater than some threshold value, we may assume that the curvature is excessive and may increase the number of particles in order to more faithfully represent that region of the surface. Conversely, if the angle is quite small, then the surface is relatively flat and the number of particles might well be reduced.

The angle between two unit-length vectors is equal to the arc-cosine of their dot-product. We can avoid the calculation of this transcendental function by taking the cosine of both sides of the comparison, thereby comparing the dot-product with the cosine of the original threshold value. Assume that we wish to refine any pair of ribbons which join at an angle of greater than twenty degrees. These ribbons should each split when the dot-product of their most recent surface normals falls below $cos(20) \approx 0.94$.

5.5.2 Modifying the Effective Facet-Size

If a ribbon is split due to an local increase in surface curvature, then the subsequent quadrilaterals along that ribbon will be narrower than we might prefer and the constructed triangles will be long and skinny. To maintain the proper aspect ratio of the triangles, any change in the local ribbon width should be accompanied by a similar change in the separation distance of subsequent points along the neighboring streamlines. This may be done by maintaining a *local facet-size* which is smaller than the global facet-size which was specified for the entire surface. When the local facet-size is reduced, new points generated from each particle will be positioned more closely. This distance will match the new width of the ribbon, and this will produce triangles with a more acceptable aspect ratio.

One may also choose to limit the reduction of the facet size, so that very sharp creases in the surface do not cause the introduction of a great quantity of tiny triangles. Instead, a very sharp crease in the surface will first trigger a reduction of the facet size, some splitting of the ribbons, and an eventual ripping of the surface when further reduction of the facet size is disallowed.

5.6 HANDLING CONVERGENCE

The orthogonal advance method helps to minimize the number of particles needed to construct a model in a flow with heavy shearing. Insertion of new particles is used to improve model accuracy in the face of surface divergence and curvature. With these methods, we may efficiently construct well-sampled models in many complicated flow fields.

The speed of construction and display may be improved by reducing the number of particles wherever the surface converges and flattens. Note that insufficient merging of ribbons will produce needless additional polygons in the final result, and will consume excess computation time in advecting this surfeit of particles.

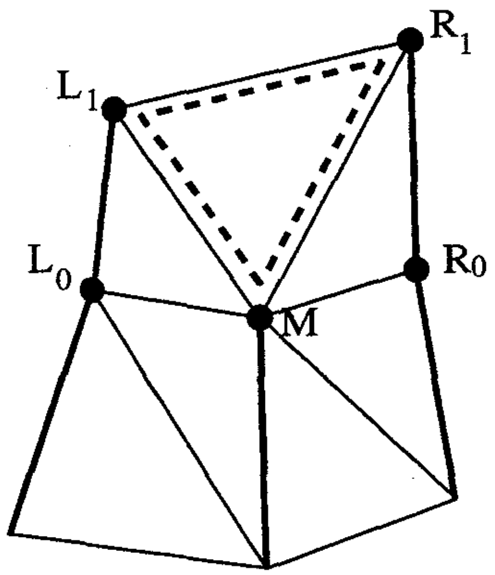


Figure 5.15: Removing a particle and merging two ribbons.

The test for when to remove a particle must consider the abutting quadrilaterals of two neighboring ribbons; merging of ribbons can therefore be applied only when the two ribbons have advanced the same distance along the shared streamline. If these two ribbons are roughly coplanar and if the height of the two quadrilaterals is greater than their combined width, then these two ribbons may be merged into a single wider replacement. The shared streamline is stopped here, and the shared particle structure is returned to a list of free storage.

Merging of ribbons requires the creation of three new triangles which make the transition between the leading edges of the two ribbons and the single leading edge of the new wider replacement. This transition is shown in figure 5.15, which also demonstrates that the merging of ribbons advances the new ribbon by one step along both its neighboring curves, and thus may require the subsequent advancement of a neighboring ribbon to maintain the restrictions of the orthogonal advance method.

5.7 DISCUSSION

The *adaptive orthogonal advancing front* method makes feasible the interactive exploration of flow field data. It offers increased performance, produces improved sampling densities over the constructed surface, and allows the incremental display of interim results.

sampling quality : The advancing front method begins at the rake and works downstream across the width of the surface. The number of particles on this front is continually adjusted to maintain a sampling density appropriate to the local differential measures of the surface.

performance : Overall performance is improved, since particles are used sparingly in the construction of the surface. We avoid computing integration steps which do not contribute to the quality of the constructed model.

interaction : The earlier ribbon-splitting methods repeatedly replace a single previously computed ribbon with two narrower ribbons. If the user interface is designed to continually update an image of the refining surface, then that image must be entirely redrawn. The advancing front method requires a full redisplay of the image only when the rake is repositioned. While a surface is being computed, newly created triangles are simply drawn into the depth and color buffers of an evolving image. Every new triangle is a contribution to the final model.

5.8 SUMMARY

Stream surface models can be constructed by a polygonal tiling of adjacent pairs of streamlines. Since flow fields tend to diverge, this method produces far too many points in the upstream portion of most stream surfaces and not enough further downstream. Hierarchical splitting of the ribbons can distribute the samples more evenly, but this approach does not allow for the reduction of the sampling rate in regions of convergence. Advection of timelines avoids many of these problems, but it falls prey to shearing of the flow.

Maintenance of an orthogonal front of particles is an efficient method of generating a set of sample points over a two-dimensional stream surface model. The advancing front of particles forms a discretization of a cross-flow line on the ideal surface. Orthogonal advancement of these particles tends to minimize the length of this discretized curve. Particles may be added to or removed from the front to maintain an acceptable sampling density in the presence of local curvature, divergence, or convergence. A front which encounters massive divergence of the flow may be split, thus ripping the surface around obstacles. The triangles in the constructed model are approximately equilateral and of a size adapted to the local curvature of the surface. This yields representations of the flow structure which are more visually intuitive than other, more commonly used, models.

CHAPTER VI
CONCLUSIONS

Stream surface models can be used to depict fluid flows more effectively than is possible when using streamlines. I have developed algorithms which rapidly construct adaptively sampled models of stream surfaces. I have built an application which allows the interactive placement of the originating rakes for such models. The usefulness of these algorithms and this implementation has been confirmed by the use of my software by scientists at the NASA Ames Research Center and at the Wright-Patterson Air Force Base.

6.1 EARLY USES OF FLORA

An early version of Flora was described in my paper "Interactive Numerical Flow Visualization Using Stream Surfaces" [Hultquist 1990]. This primitive software was used to produce images for the paper "Vortical Flows over Delta Wings and Numerical Prediction of Vortex Breakdown" by Ekaterinaris and Schiff [1990].

This simple application was difficult to use and cumbersome to extend, and so I spent two years revising the construction algorithms and devising an advanced implementation platform. I then published a refined algorithm for stream surface construction [Hultquist 1992], and (with Eric Raible) a description of the SuperGlue programming environment [Hultquist and Raible 1992].

After two additional years (in 1994), I finished a completely new version of Flora, which was installed for use by CFD scientists at the NASA Ames Research Center in California and at the Wright-Patterson Air Force Base in Ohio. The user documentation for this new version of Flora is included in Appendix B of this dissertation.

6.1.1 Video Production

The first useful work performed with the "new Flora" was in the production of a videotape. This work was intended primarily for presentation to non-specialists, to give an overview of the kinds of research being done at NASA. This "1994 CFD Technical Highlights" videotape [Hirsch and Gong 1994] includes several minutes of animation which uses stream surfaces to depict three different flow fields. The first piece demonstrated the vortex formed over the edge of a deployed flap on a wing. The second segment depicted the flow inside a small heart-assist pump which has been designed for surgical implantation. The final segment showed the flow of air surrounding and trailing in the wake of a magnetically levitated "bullet train."

Before this video could be made, I had to extend Flora to support video production. Since Flora was implemented under SuperGlue, this modification was much simpler than it might otherwise have been. The position of the models for each frame of animation was computed in the Scheme command layer, and a new class was defined to allow convenient program-level control of video-recording hardware. To be sure, the features added to this revised copy of Flora were only the minimal set required to control very simple camera motions. A small amount of extra work was then needed to adjust model colors and scene lighting for a more acceptable appearance when recorded on NTSC-encoded videotape.

Once these modifications had been made, the animations were created. I was helped in this by Ms. Vi Hirsch, a video production specialist at the Numerical Aerodynamics Simulation Facility at NASA-Ames. She and I created each video segment in consultation with the scientists who had computed the datasets. Each scientist described to us the particular aspects of the flow field which they felt were of interest. Visualization models were created and then modified under the direct supervision of the participating scientists. Once an acceptable set of models had been constructed, we would define the motions of the camera and record the scenes on an Abekas disk-video system, with final transfer of the images onto videotape.

The first dataset was of a flap extended from the wing of a generic commercial-transport airplane [Mathias *et al.* 1995] . One of the participating scientists, Dr. Stuart Rogers, wanted us to illustrate how a vortex formed in the flow around the edge of the flap. Three small rakes were placed near the top of the flap. While the scene was recorded, the stream surface models were animated to depict the relative speed of the flow. Since the advancing front method does not march the particles through the flow at the same speed, the motion of the fluid was depicted using a texture. This texture was a single rectangular patch of color, interleaved with transparent stripes. The stripes were laid across the models to form gaps parallel to the time-lines. As the animation was recorded, this texture was translated along the model in the downstream direction. This yielded an illusion of motion, with opaque bands of surface advancing with proper relative speed through the vortical flow. In a later interview [Rogers 1995], Rogers stated that "part of the analysis definitely benefited from the [Flora] software."

The second segment to be recorded was a depiction of the flow inside a helical pump (figure 6.1). This "Left Ventricular Assist Device" (LVAD) is a small cylinder about one inch in diameter and about three inches long. It is designed to be surgically implanted just upstream of the heart. Rapidly spinning helical channels drive blood into the heart, thereby improving the blood circulation of an ailing patient. The computational grid defined only one helical channel. For the visualization, this data was replicated three times about the central axis of the pump. All three instantiations were then rotated slowly while stream surfaces were extruded from the inlet and into the pump body.

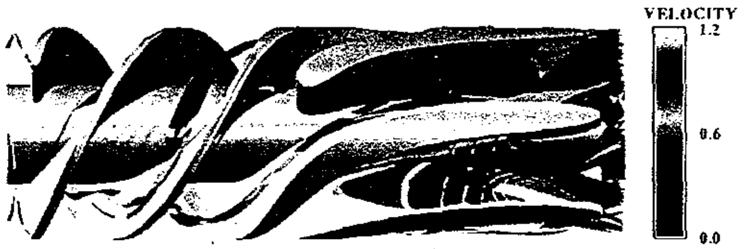


Figure 6.1: Flow inside a heart-implant pump.

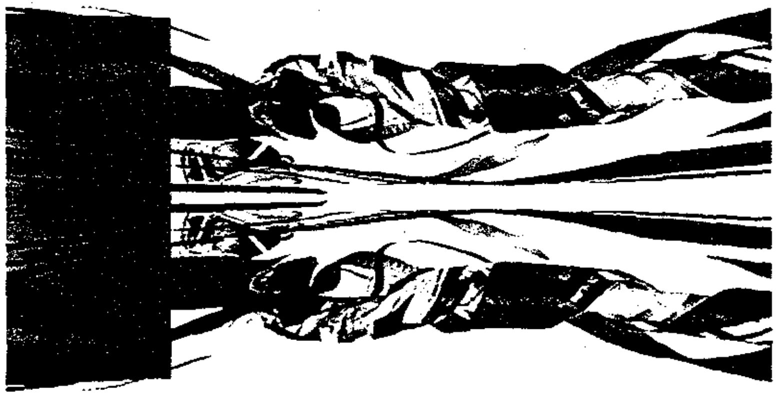


Figure 6.2: Flow behind a magnetically levitated train.

Each stream surface was displayed with the same advection of transparent timeline stripes. The principle scientist, Dr. Cetin Kiris, asked if we might also indicate fluid speed by depicting each model in a range of colors corresponding to the variations of the local velocity magnitude. This involved yet more modification of Flora, but this was completed in about two hours of additional programming. The animation segment was re-recorded onto the video system and reviewed by Doctor Kiris, who then approved the work. Still images from this animation have been sent to his colleagues at the Baylor College of Medicine. These pictures are useful to physicians who might be less comfortable with the more abstract plots used by CFD scientists. These images also aid the interpretation of photographs taken of empirical tests of the pump prototypes.

The final piece of animation depicted the motion of air caused by the passage of a train. This vehicle was designed to be magnetically levitated and propelled along a sunken semi-circular trough. The nose is smoothly tapered, but the tail is blunt. Dr. Goetz Klopfer, the principle scientist, particularly wanted to show the highly convoluted flow in the wake of the speeding vehicle. The originating rakes were placed just off the surface of the train, in an arc surrounding the nose. The stream surfaces were extruded over the length of the vehicle and then spilled over the trailing edge of the roof. There the stream surface models intertwine in a tangle of air in the wake region behind the train.

For the train dataset, yet more modifications to Flora were required. The advecting particles were found to repeatedly impact the skin of the train as they progressed down the length of the vehicle. A very primitive implementation of wall-bouncing was introduced to allow the stream surface models to flow just slightly above the skin of the train. Very careful color selection was then required to clearly depict the tangled knot of fluid in the wake region (figure 6.2). This task was further complicated by the significant loss of detail introduced by the NTSC-encoding of video images.

Several lessons were learned from the experience of creating these animations. The first surprise was that extremely precise control of rake placement was wanted. Flora does allow the user to reposition each rake using the mouse; what was lacking was some kind of *vernier mode* which would allow precise and tiny movements of each rake. Also desired was an *undo mechanism*, which would allow one to return a rake to its previous location after trial repositioning of the rake proved to be unsatisfactory.

Another lesson was that a full depiction of a very complicated flow field can easily generate hundreds of thousands of tiny polygons. This great quantity of graphical data may require several seconds to display. This prevents the *dynamic* exploration of the flow data, and thereby impedes the work of the scientist. It was necessary to modify Flora so that it would automatically reduce the image quality to sustain an acceptable frame rate even at the expense of some coarsening of the depicted polygonal models. A related concern is that much better coordination was needed between the separate computational processes which perform input-event handling, model calculation, and image redisplay.

Despite these limitations of the user interface and despite the very limited support for video production, Vi Hirsch and I were able to create three pieces of computer animation which clearly depicted the important features of three very different flow fields.

6.1.2 Topological Analysis

A copy of Flora was sent to a CFD research group at the Wright-Patterson Air Force Base. The scientists there specialize in the topological analysis of the structures created in various flow fields. For this kind of work, they needed a visualization tool which would clearly depict complicated interacting structures. Despite the great distance and minimal interaction between these scientists and myself, Flora has proven itself to be a useful tool for their needs.

Figure 6.3 was created by Dr. Miguel Visbal using Flora. It shows the flow impinging on a vertical post, with the fluid is moving from the right background to the left foreground. Three stream surfaces have been placed in the flow. The uppermost surface travels downwards to envelop a large region at the base of the flow domain. The middle surface develops a vortical structure which is compressed along the vertical dimension

Another of these scientists, Dr. Datta Gaitonde, had earlier used some primitive methods to construct stream surface models. He would first use a computer program to compute a family of streamlines from a textually-specified list of starting seed locations. The point values from these curves were saved in a file, which was processed by a second program to create a two-dimensional array of points. The final point of the shorter streamlines was replicated so that each streamline curve would contain the same number of points. Finally, this rectangular mesh was loaded into FAST [Bancroft 1990] and rendered as though it were a gridplane. Gaitonde told me that he had spent an entire week creating images using this collection of tools. Using Flora, he can now create similar models in one or two hours [Gaitonde 1995].

Although their opinion of Flora was generally enthusiastic, the Air Force scientists did request some changes to the package. One early suggestion was for the creation of circular rakes to form streamtubes. A second modification was to allow the models to be scaled by differing amounts in each of the three physical-space dimensions; this enables one to exaggerate the height of flow structures and hence to more easily investigate the boundary layer flow near the skin of the vehicle. Both of these modifications were simple to implement, and a revised version of Flora has

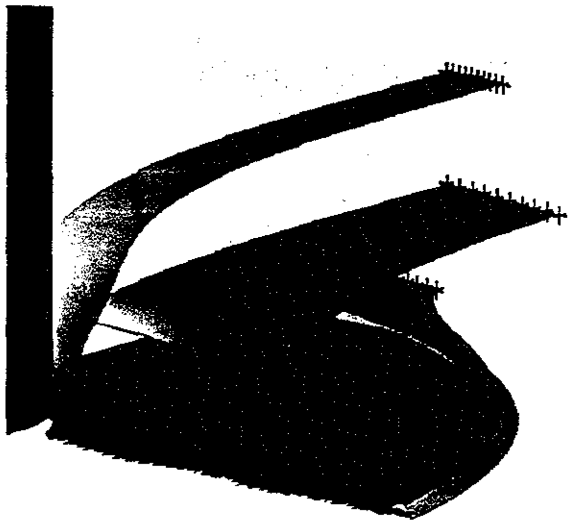


Figure 6.3: Laminar juncture flow around a vertical post.

helped these scientists to create more useful depictions of certain flows. A requested enhancement which I have not yet implemented is a facility for "slicing open" a coiled stream surface to reveal its presently hidden interior.

6.1.3 Vortex Roll-Up

I have used Flora to depict the *shear layer* or *roll-up* of the vortices formed across the beveled leading edges of a delta wing. This is illustrated in figure 6.4, which shows two stream surfaces textured with both streamline and timeline stripes. This data was computed by Dr. Neal Chaderjian of NASA-Ames. He told me that these textured models were not sufficient for illustrating the important aspects of this data. Instead, he requested some means of seeing the interior of these structures. To satisfy this requirement, I incorporated a simple contouring method into Flora and then generated a family of contour lines at constant intervals of the physical-space x-coordinate (figure 6.5). In this image, the surface shape is conveyed by the family of spiral slicing curves. Note how these curves clearly depict the formation of the shear-layer vortices.

These spirals are a commonly used technique for the visualization of such flows; however, these curves are usually only approximated. They are typically computed by projecting the three-dimensional vectors of the flow field onto a planar slice. A streamline is then computed in the two-dimensional vector field define over this restricted domain. Since the direction of the original flow data is not strictly perpendicular to the slicing planes, the projection introduces error which alters the shape of the computed spirals. (See [Buning 1985,1989] for further discussion of this problem.) The physical-space contouring shown here is a direct implementation of the conceptual visualization model. A comparable slicing of simple stream surfaces is described in Belk and Maple [1993].

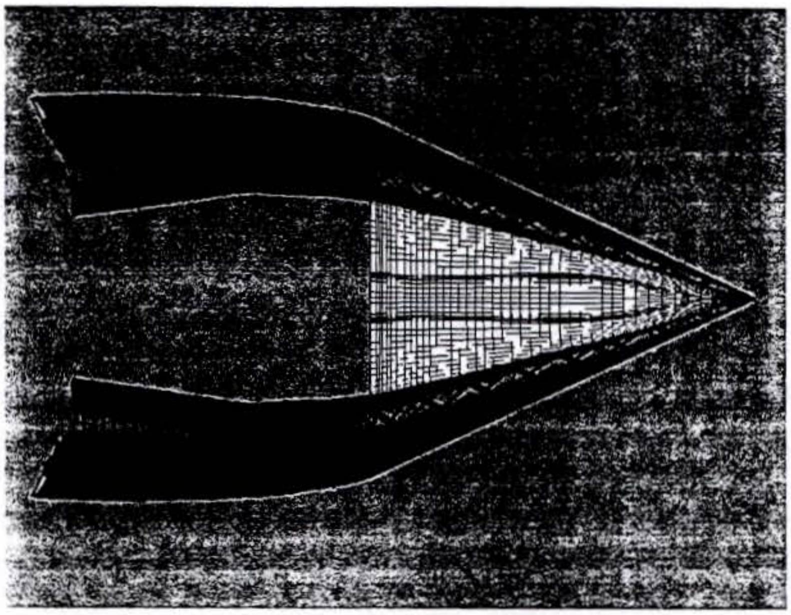


Figure 6.4: Stream surfaces above a delta wing.

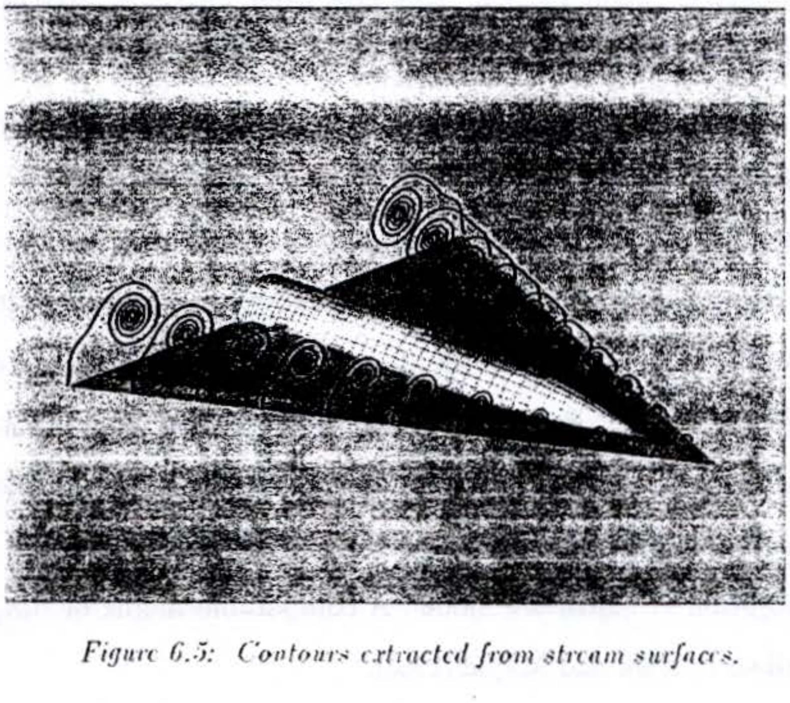


Figure 6.5: Contours extracted from stream surfaces.

The implementation of this final image was based on two visualization techniques: the first to compute stream surfaces from the three-dimensional flow data, the second to compute a set of contour curves from these surfaces. I have more fully considered this *functional composition* of visualization techniques in the report "Numerical Flow Visualization in a Functional Style" [Hultquist 1989].

6.1.4 Summary

Flora has been used for the exploration and the presentation of many different flow fields. It has been repeatedly customized to accommodate the special requests of several different researchers. Some new features are still desired, but these are mostly confined to minor changes to the user interface and to support additional visualization methods.

Indeed, it has become clear that a very important requirement of any visualization package is that it provide a large suite of complementary techniques. To be truly useful, Flora must be further extended so that scientists may use iso-surfaces, cutting planes, contours, streamlines, and stream surfaces all through an intuitive and expressive user interface. The ease of extensibility of the package is also important, so that customized visualizations may be created with minimal delay.

6.2 CONTRIBUTIONS OF THIS WORK

Software based on the algorithms described here has assisted scientists in the exploration of highly convoluted flow fields. The usefulness of this code can be attributed to the interactive response made possible by the rapid advection of particles and the robust and efficient use of these particles to create well-sampled stream surface models. Several specific contributions are enumerated below.

6.2.1 Mixed-Space Integration

Computational-space integration has been recommended by some authors [Eliasson *et al.* 1989, Belk and Maple 1993, Shirayama 1991] ; yet rejected by others [Buning 1989, Sadarjoen *et al.* 1994] . The speed obtained from pre-processed conversion of the vector field offers very rapid calculation of streamline curves. The error introduced by finite-difference calculation of spatial metrics raises doubt over the trustworthiness of these results. The selective use of both methods has been shown to provide rapid calculation of streamlines, while reducing the introduction of error in regions of computational sensitivity.

6.2.2 Test Cases

I have published (in Appendix A) the source code which generates two test cases. These new grid and solution files may be used as benchmark tests for future work by others.

6.2.3 Retroactive Stepsize Adjustment

Numerical integration through cell-based data is often constrained to generate a specified number of points within each cell. This has been implemented in Plot3d by estimating the computational-space velocity magnitude of each newly interpolated physical-space vector, and then scaling the integration timestep accordingly. I have proposed a new method which measures the actual computational-space distance between successively computed points on a streamline. The stepsize is then adjusted to maintain the desired separation distance. This retroactive adjustment is slightly more efficient, and is also more robust in regions of rapid change of cell size.

6.2.4 Node Tags

The standard iblank annotation on multiple-block grids provides only limited information. An augmentation of this data by the more descriptive *node tags* increases the speed of many visualization tasks. A single-bit flag marks each valid cell. A second flag marks cells in smooth regions of the grid. A third flag marks cells in regions of block overlap. An integer stored within each tag word indexes into an array of donor-point records for each block.

6.2.5 Exploiting Spatial Coherence

The numerical integration of streamlines generates a sequence of closely positioned query points. The spatial coherence of these samples can be exploited by caching the sample values for the eight corners of one or more recently encountered cells. This cache of node position and field sample data improves the speed of constructing Jacobian matrices and interpolating flow field data. Extrapolation in computational space can reduce the calculation time spent in the point-finding method.

Pre-calculation of the donor points can greatly reduce the performance penalty imposed by block transitions. Computational-space extrapolation can slightly reduce the time required to compute these donor-receiver equivalences.

6.2.6 Tracing through Self-Abutting Blocks

Many grids contain C-type and O-type blocks. The branch cut boundaries of self-abutting blocks are rarely marked by negative iblank values. These internal block faces are then incorrectly treated as domain boundaries by many flow visualization applications; streamlines which encounter these faces stop in the middle of the flow domain. I have introduced a simple heuristic test which constructs and then tests three "pseudo-donor" locations corresponding to the three likely re-entry points found in the two common block topologies. This method efficiently resumes the advection of particles across the boundaries of many self-abutting blocks.

6.2.7 Stream Surface Construction

The generalization of the streamline model into a full two-dimensional sheet is a natural and effective means of depicting flow fields. I have devised an efficient and robust method for constructing polygonal models of these surfaces. A set of particles is advected together through the flow domain, with this front kept locally orthogonal to the flow velocity. Particles are added to or removed from this set to maintain an acceptable sampling density. This method offers increased performance, produces improved sampling densities over the constructed model, and allows the simple incremental display of the interim results.

6.2.8 Publications

I have published several papers which describe various aspects of this work:

- "Numerical Visualization in a Functional Style,"

  technical report RNR-89-008, NASA Ames Research Center, June 1989.

- "Interactive Numerical Flow Visualization Using Stream Surfaces,"

  Computing Systems in Engineering, 1(24), pp. 349–353, 1990.

- "SuperGlue: A Programming Environment for Scientific Visualization,"

  (with Eric Raible), *Proceedings of Visualization '92*, pp. 243–250, October 1992.

- "Constructing Stream Surfaces in Steady 3d Vector Fields,"

  Proceedings of Visualization '92, pp. 171–178, October 1992.

- "Improving the Performance of Particle Tracing in Curvilinear Grids," AIAA Paper 94-0324, January 1994.

6.2.9 Better Flow Visualization

The algorithms presented in this thesis are of interest only in as much as they contribute to the productivity of scientists. A survey paper on numerical flow visualization [Weston 1987a] has this to say:

> Particle traces can be challenging to display and time-consuming to set up because of the need to keep the number of lines small in order to avoid confusion, while still capturing the essence of the features of interest in the flowfield.

The software implemented during this research has alleviated some of this difficulty. Use of this code by scientists has demonstrated that the interactive placement of stream surface rakes can be useful for exploring complicated flow fields. Operation counts and empirical tests have verified that these new algorithms offer increased performance over earlier algorithms and implementations. Further work is needed to improve the user interface and to expand the suite of visualization models supported by the Flora application.

6.3 OPEN QUESTIONS

The field of numerical flow visualization is very young, and it faces great challenges in the coming years as computational grids become larger and the flows being simulated become more intricate. The present work has not considered certain issues which will become more important for newer flow field datasets.

6.3.1 Query Optimization

Each computed model contains a set of fields defined over some domain. When this data is queried for a field which it does not yet contain, the software can either compute the values of this new field or it may extract those values from the field of the same name in the enclosing higher-dimensional domain. In the example of a streamline model, the fluid density at the points on the curve may be obtained by interpolating the corresponding field samples in the original flow field data.

Some fields cannot be extracted from fields in the enclosing dataset. The surface-normal field over a gridplane has no equivalent in the higher-dimensional domain; the arc-length of a streamline cannot be extracted from any scalar field in the grid and solution samples of the enclosing flow data.

Now consider the pressure field defined over a streamline. This field may be obtained by computing pressure over the entire flow domain, and then sampling this new field at each point on the curve. A more efficient approach would be to extract the momentum and density samples at points along the curve, and to compute the pressure only at these points on the streamline. Such *query optimization* for scientific data management is a rich area for future research.

6.3.2 Time-Varying Surfaces

In the coming years, the simulation of time-varying flow will become increasingly more common. These datasets typically consist of a sequence of between one hundred and one thousand snapshots of the flow data. A new grid is required for each timestep if the geometry also changes over time, such as for the oscillations of a flexible wing.

In unsteady flow fields, we must determine not only *where* a rake has been placed, but also *when*. As the front of the *streak surface* is marched through the flow, the previously constructed portions of the surface continue to be carried downstream.

The advection of many particles through an unsteady flow field is quite computationally demanding. The construction of unsteady *streaklines* and *streak surfaces* will require much more research, but see [Lane 1994] for notable early progress in this area.

6.3.3 Distributed Implementation

An unsteady flow simulation can produce a massive quantity of data, perhaps several gigabytes of vector field samples. This outstrips the resources of even the largest of workstations. A distributed implementation might be required, in which the data is stored and processed on a supercomputer. The workstation would receive the geometric data of each model across a high-speed network link. The Distributed Virtual Wind Tunnel [Bryson and Gerald-Yamasaki *et al.* 1992] and PV3 [Haimes 1994] are early examples of work in this important and promising area.

6.4 SUMMARY

This work has examined a number of different methods for the calculation of streamlines and stream surfaces through multiple-block curvilinear grids. Mixed-space integration and the exploitation of spatial coherence was empirically shown to increase the speed of particle advection. An *adaptive orthogonal advancing front method* constructs surfaces by the interleaved advection of a set of particles. This method distributes the computed points more evenly over a stream surface model and thereby uses fewer polygons to describe the surface at a given level of resolution.

The most important contribution of this work has been an improvement in the ability of scientists to create useful images of their data. By enabling the interactive exploration of vector fields, this approach facilitates the identification of important flow features. By producing visually intuitive models, it simplifies the presentation of those features to others.

APPENDIX A
CODE FOR THE TEST GRID AND SOLUTION

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <math.h>
#include <assert.h>

#define DOTIMES(I,N) for(I=0; I<N; I++)
#define MAX(A,B) ((A)>(B)?(A): (B))
#define ABS(A)   ((A)>0  ?(A):-(A))

int main (int argc,
          char *argv[])
{
#define NUM 21
#define NK  3
  typedef float ARRAY[NUM*NUM*NK];

  int dims[3];
  float info[4];
  ARRAY xyz[3], qqq[5];
  int xfile, qfile, crease;
  int i,j,k,n;
  float x,y,z, xx,yy,zz, del;

  assert(argc == 4);
  xfile = open(argv[1], O_WRONLY|O_CREAT, 0644);
  qfile = open(argv[2], O_WRONLY|O_CREAT, 0644);
  crease = atoi(argv[3]);
  assert(xfile && qfile);
```

```
dims[0] =  NUM;
dims[1] =  NUM;
dims[2] =  NK;

info[0] = 0.0;
info[1] = 0.0;
info[2] = 0.0;
info[3] = 0.0;

del = (NUM-1)/2.0;
n   = 0;
DOTIMES(k,NK) {
  DOTIMES(j,NUM) {
    DOTIMES(i,NUM) {
      x = (i-del)/del;
      y = (j-del)/del;
      z = (1/del) * (k-1);

      if (crease) {
        y = y + 0.5*ABS(x);
      }
      xx = x + sin(4*y)/10;
      yy = y - sin(4*x)/10;
      zz = z;

      xyz[0][n] = xx;
      xyz[1][n] = yy;
      xyz[2][n] = zz;

      qqq[0][n] =  1.0;
      qqq[1][n] =   yy -0.1;
      qqq[2][n] =  -xx;
      qqq[3][n] =  0.0;
      qqq[4][n] = 10.0;

      n++;
    }
  }
}
```

```
  write(xfile, dims, sizeof(dims));
  write(xfile, xyz,  sizeof(xyz));
  close(xfile);

  write(qfile, dims, sizeof(dims));
  write(qfile, info, sizeof(info));
  write(qfile, qqq,  sizeof(qqq));
  close(qfile);
}
```

APPENDIX B
THE FLORA USERS' GUIDE

FLORA 1.10

NAME

flora – interactive flow visualization

SYNOPSIS

flora [-nomail] [command-file ...]

DESCRIPTION

Flora is an interactive application for visualizing steady flow fields defined over multiple-block curvilinear CFD datasets. It reads data defined in the Plot3d format, and it accepts a subset of the Plot3d commands. Flora presently can construct and display gridplanes, gridblocks, streamlines, and stream surfaces. Each new graphical "model" and plot window is assigned a name and (optionally) an alias. These names and aliases may be used as new commands by which previously created objects are modified. The mouse is used to change the view and to move rakes within the flow domain.

Commands can be typed while the mouse-cursor lies within the boundary of a plot window. This is useful for typing commands into a graphics window which covers the entire screen. This text will be displayed in the window title bar. These commands will be processed with default values used for any values not supplied in this single line of input.

When Flora starts running, it first tries to locate the file ".florarc" in the current directory, and then in the user's home directory. This file may contain commands to initialize Flora to suit user preference. Flora then processes the command line arguments. These are treated as the names of command files, which will be loaded in the order in which they have been specified.

Flora normally sends email to its author upon the completion of each interactive session. This information is being gathered to assist in the completion of a dissertation. If you do not wish this mail to be sent, you may execute Flora with the command-line argument "-NOMAIL". This option will prevent the gathering and sending of these statistics.

A typical interactive session begins with the READ command, followed by the names of the grid file and the solution file. The locations of the boundary surfaces are specified using the WALLS command, and rakes are embedded in the flow field using the RAKES command. The PLOT command then places an image of these walls and rakes on the screen. The mouse is used to control the view and to relocate the rakes. When the rakes are properly positioned, such that the resulting streamlines and stream surfaces adequately depict the structure of the flow, then printable images are saved, a checkpoint file created for restoring the program state in future sessions, and the program is exited. A typical session is depicted in the demo script, which is executed with the DEMO command.

COMMAND SYNTAX

As mentioned above, the Flora command interface is similar to that of Plot3d. This includes recognition of (almost) any non-ambiguous prefix of each command, qualifier, and argument. (For safety, Flora will not accept an abbreviated form of the commands DEMO, EXIT, or QUIT.) Upper- and lower-case distinctions are ignored for most input text, excepting only filenames. Hyphens and underline characters may be used interchangeably or omitted entirely, again excepting within filenames.

An interrupt character (typically control-C) will stop any long-running calculation within Flora and return control to the top level of the command input loop.

Flora will prompt for each required input. At all times, a question mark may be entered to obtain a description of the next value expected by the command parser. In many cases, a default value will be indicated within square-brackets. A carriage-return is sufficient to select this default value. Logical (or Boolean) arguments will typically default to the opposite of the current value.

Some commands accept "qualifiers", which follow the command name and are marked with a slash character. For example, the READ command accepts the qualifier "/MGRID" to indicate that the grid contains multiple blocks. Some qualifiers must be accompanied with a value. Each such value is attached to its qualifier with an equals-sign, as in the pair "/X=mygrid.bin", which supplies the grid filename for the READ command. Flora will prompt for any needed qualifier values which have not been so specified.

Any input text surrounded by single quotes is treated as a shell environment variable, and the quoted text is replaced by the value of that variable. Flora will also accept the tilde character in filenames as a shorthand for the user's home directory. Any item may be surrounded by double-quotes; this is useful to protect filenames which contain slashes, which otherwise would be processed as a sequence of qualifiers.

Additional features adopted from Plot3D include the use of an exclamation point to preface comments, the at-sign to invoke a command file, and a dollar sign to escape to a UNIX command shell. A hyphen at the end of a line allows the continuation of that text onto multiple lines of input. A percent-sign is used by Flora to mark any command which has been typed into one of the graphics windows; a command line marked in this manner will be processed with no prompting, and a default value will be used for any parameter not specified by the user in this single line of text.

Flora has been implemented in the programming languages C and Scheme. Scheme is a dialect of LISP. Input text which begins with a left parenthesis is assumed to be an expression in the Scheme language. These expressions are evaluated by an interpreter embedded within Flora. Scheme expressions may be placed in a command or initialization file to customize the behavior of Flora. This kind of thing is not generally recommended, but it does appeal to some folks, which is why I mention it.

BASIC COMMANDS

The currently supported commands are listed here with a brief description of their purpose. Additional information is available from the online help facility; type HELP followed by the command name.

CHECKPOINT – writes a file of commands which will restore the program to its current state. This file may be edited and loaded into Flora at any time; for example, to create the same set of walls in a number of related datasets. Any filename may be supplied to the CHECKPOINT command; the default value is "flora.com". If a file of the specified name already exists, then a unique filename will be generated by inserting a integer version number just prior to the filename extension.

DEMO – loads a demonstration command file. This file loads a small grid and a solution file, creates some grid walls, and places a plot window on the screen. A rake is added to this image with a family of streamlines rooted at intervals along this line segment. Some object-modifying commands are then shown. The demonstration file is heavily commented and makes for very informative reading.

EXIT and QUIT – stops the program and returns control to the originating shell. The /SAVE qualifier will cause Flora to write checkpoint commands into the file "flora.com". If this file already exists, the checkpoint data will be written into the file "flora-N.com", where *N* has been replaced by a small integer.

GRIPE – sends email to the author of Flora. Use this command to send in bug reports, suggestions, and lavish praise. You are also invited to call the author (Jeff Hultquist) at: xxx.xxx-xxxx (office) or xxx.xxx-xxxx (pager).

HELP – prints information about the commands. Just typing "HELP" with no arguments will print some general information and a list of the available commands. Type "HELP" followed by a command name for more information about a particular command. Type "HELP *" to see all of the available on-line documentation.

MAP – prints a list of the fields and grid-blocks which have been read. This command is exactly equivalent to "SHOW READ".

MINMAX – specifies the extent of the "area of interest" in each physical-space dimension. The low and high limits in each dimension will be requested, unless one or more pairs are suppressed with the qualifiers /NOX, /NOY, and /NOZ. If the qualifiers /X, /Y, or /Z are provided, then pairs will be requested only for the specified dimensions.

Subsequent plots will be centered about the midpoint of this rectangular volume. The near and far clipping limits of the graphics hardware will be set either side of this center-point, separated by a distance equal to the diagonal of the volume. The initial magnification of the image will be such that this region fits just within the boundaries of the window. If not specified by the user, the minmax region will be set to the bounding extent of the previously constructed models.

PLOT – creates an image of the previously specified walls, rakes, and resulting streamlines and stream surfaces. The /OVERLAY qualifier allows the placement of new walls and rakes into previously created windows. The argument for this command may be the integer index, the name, or the alias of an existing plot. The special symbol "P*" refers to all existing plots, and can be used to place new walls and rakes into every window.

The /BACKGROUND qualifier enables one to specify the background color. Flora will recognize several color names, or the letters "RGB" followed by red, green, and blue values between zero and one. The /ALIAS qualifier can be used to assign a meaningful alternative name to a plot. Multiple plots may be active on the screen at any time.

RAKES – specifies the originating seed location for a single stream surface or a family of streamlines. A rake is a short continuous curve which may be either a line segment in physical coordinates, a curve which follows the computational space of a single block, a circle, or a cross. The shape of the rake may be selected with the /SHAPE qualifier. If a shape is so specified, then Flora will request either the coordinates of the two endpoints or the position of the center point and a radius.

These point locations may be specified in either computational or physical space. The /XYZ and /IJK qualifiers are used to select which coordinate space will be used. If neither qualifier is supplied, then each new point location may be specified in either coordinate space. Each new set of coordinates is preceeded by a keyword, either PHYS or COMP.

When no shape is specified with a qualifier, then the rake is assumed to be a computational-space curve. The position of this curve will be obtained in the traditional Plot3D style, with the program requesting the beginning and ending values along each physical or computational dimension. Note that only the minimal and maximal values of the index range are used; these form the coordinates of the endpoints of a single curve.

By default, particles are advected in the downstream (positive pseudo-time) direction. This can be changed with the qualifier /+-TIME or /-TIME, to construct the model in both directions or only upstream. The qualifier /+TIME may be used, but has no real effect since it merely specifies the default value. Note that, while hyphens are optional within most commands and qualifiers, the hyphens in these qualifiers are clearly not optional.

The size of the result buffers, and therefore the overall length or size of the streamlines or surface, is controlled with the qualifier /MAX-POINTS. The stepsize used in the numerical integration is normally adjusted to maintain a separation of about one-fifth of a cell-width between successively computed points. This distance may be changed with the /COMP-STEP qualifier. The sequence of points is then filtered, such that the remaining points are separated by a specified minimum distance in physical space. This value defaults to zero length, and may be changed by the /FACET-SIZE qualifier. Each new seed point will be separated from its immediate neighbors by this same physical space distance. The number of streamlines may be changed from this default value using the /NUM-SEEDS qualifier.

The /ATTRIBUTES and /NOATTRIBUTES qualifiers are used to control whether Flora requests information about how to display the created models. The /ADD qualifier allows the creation of more than one rake in the next plot. Note that the object-modifying commands provided by Flora allow the rendering attributes to be changed at any time, and also enable one to insert and remove previously constructed models from each plot. The RAKES commands accepts the /ALIAS qualifier to assign an alternate name to the created model.

READ – loads the grid and solution data files. The /XYZ and /Q qualifiers may be used to specify the filenames, but if neither qualifier is used then both names will be requested. Other qualifiers may be used to indicate the use of multiple blocks (/MGRID) and iblanking (/BLANK). Only the 3D, whole, binary format is currently supported.

SHOW – prints information about the state of the program. This commands requires one or more arguments, each of which is a command name. It will then print information relevant to those commands; such as what data has been read, or what walls, rakes, and plots have been created.

VPOINT – specifies the direction from which the models will initially be displayed. This data can be input as an azimuth and elevation. Alternatively, the viewing direction may be specified by a position in physical space. The angles will then be computed from this location, relative to the physical-space origin. This command accepts the qualifiers /XYZ and /ANGLES, with the Cartesian input style being the default. The default viewing direction creates a "three-quarter view" with a slight elevation.

WALLS – specifies subregions of the grid for display. In a multiple-block grid, Flora will always ask for the number of the grid-block from which the wall is to be extracted. This is different from Plot3D, in which the /GRID qualifier must be specified to create a wall in any block but the first. Flora will request the low and high index values for each grid dimension. The special symbols "ALL" and "LAST" may be used here to represent all the grid indices or only the uppermost value. Although a step increment is also requested for each index range, the current version of Flora always displays walls with an stepping increment of one.

Plot3d and Flora each manage a list of "active" models which will be displayed in the next plot. By default, both programs will allow only one wall to be constructed from each block. The qualifier /ADD may be used to override this behavior, to create multiple walls within a single block.

The pair of qualifiers /ATTRIBUTES and /NOATTRIBUTES specify whether the rendering attributes should be requested. If neither of these qualifiers is used, then /ATTRIBUTES is the default and Flora will ask how this item should be displayed. The choices for the "rendering style" are POINTS, LINES, SHADED-SURFACE, and HIDDEN-LINES. This is followed by a request for the color of the item, and perhaps the width of any lines or the size of the points. The color value may be any one of several color names or an explicit listing of the separate red, green, and blue color components. Type a question mark at the prompt to get a listing of the allowed names. Additional rendering attributes may be requested for compatibility with Plot3D, but these other parameters are not used in the present version of Flora.

The WALLS command also accepts an /ALIAS qualifier, which may be used to give a more meaningful name to each item, such as "WING", "TAIL", or "FLAP". This alias can be used in place of the program-assigned name when referring to this item.

USING THE MOUSE

The mouse is used within each plot window to control the viewing parameters and to reposition the rakes within the three-dimensional volume of the flow domain.

If the mouse is clicked on (or very near) a point on a model then a small marker is placed at this location. Any previously placed marker on this same model will be removed. The new point then serves as the new center of rotation and zooming for this plot. Furthermore, the name or alias of the selected item will appear in the title-bar of the current plot window. Clicking an existing marker will make it disappear, but has no other effect. Clicking on a rake will also set the center of rotation, but no marker will be created. Later versions of Flora may use markers for other purposes, such as for accessing popup menus. There is no need to remove the markers prior to saving an image to a disk file; all of the rakes and markers will be hidden while the image is copied to the image file.

As in Plot3d, the RAKES command is used to place the initial seed points from which streamlines are calculated. In Flora, the initial rake is represented as a line segment, a geodesic curve in computational space, or a circle or cross shape. Each rake serves as the root for a family of streamlines or for a single stream surface. This rake can be moved interactively through the flow domain using the mouse. Either endpoint of a segment or curve rake can be repositioned. Selecting a rake endpoint with the middle mouse button will cause only this one point to move, leaving the other endpoint fixed and stretching the new rake between the fixed point and the one changing position. The middle mouse button, when used to select the center of a circle or cross, is used to control the radius of the rake shape. Selecting the midpoint or endpoint of a rake with the left mouse button will cause the entire rake to be translated. In other words, the left button will reposition the entire rake; the middle mouse button will change the size of the rake.

Obviously, some mapping is needed between the two degrees of freedom of the mouse and the three degrees of freedom for the position of a selected point. Each small change of the mouse position is compared against the projected directions of the three orthogonal physical-space axes. The selected three-dimensional point is shifted along whichever axis lies most closely aligned with the incremental change in the mouse direction. Experiment with this for a little while, it becomes easier to use after a little practice.

While a rake is moved, new streamlines or a new surface will be repeatedly computed and displayed. One can move a rake in one scene at high magnification, while watching the resulting model from another angle in a second window. If the mouse button is released before the advection is complete, then the remainder of the lines or surface will be computed by a low-priority background task.

The rake endpoints will resist being moved outside the flow domain. This is a feature. It allows one to push one end of a rake up against a non-slip wall without penetrating this boundary.

CHANGING THE VIEW

Each model is displayed in one or more windows, and each window may have its own distinct viewing direction and magnification. The view may be changed by pressing a mouse button while the cursor does not lie near a rake, and then dragging the mouse pointer across the screen. The view-controlling function of each mouse button mimicks that of Plot3d. As in Plot3d, the rate of any viewing change is proportional to the amount by which the mouse has been moved. The rate of response for the viewing control may be adjusted by tapping the up-arrow and down-arrow keys. Holding down the control-key while dragging the mouse will decrease the rate of change, thus allowing precision adjustment of the viewing direction.

The left mouse button invokes rotation of the scene. Side-to-side motions of the mouse will invoke a rotation about the vertical axis in the physical-space of the data. Up and down movements of the mouse cursor will cause rotation about a horizontal axis in screen-space. More clearly perhaps, the two directions of mouse movement control the azimuth and the elevation of the viewing direction. Both rotational axes pass through the "center of rotation." This is originally the center of the MINMAX box, but may also be the geometric center of the computed models, and more frequently is the most recently selected point on a model.

When the middle button is pressed, the apparent size of the depicted models may be changed. Upward movement of the mouse cursor will reduce the size of the displayed models; pulling the mouse downward will increase that size. More precisely, when the scene is displayed under a orthographic projection, dragging the middle button will alter the magnification factor in the viewing transformation. When the image is displayed with perspective foreshortening, the motions of the mouse will alter the image by moving the models closer to or farther from the viewing position. This type of motion is called "trucking," and this parameter may also be changed with the TRUCK command. When the ALT key is held during the middle-button press, then the motions of the mouse will alter the angle-of-view. This is called "zooming," and it is equivalent to the use of the ZOOM command.

With a press of the righthand button, the image may be translated horizontally and vertically. Remember that the speed at which the models move across the screen is proportional to the distance between the current location of the mouse cursor and the point at which the mouse-button was originally pressed.

OBJECT-MODIFYING COMMANDS

Each new wall is given a name and (optionally) an alias. The name for each wall is the letter "W" followed by an integer. Similarly, names are also assigned to rakes ("R") and plots ("P"). Each name serves as a new command by which one can modify the state of the associated item. This is done by following the name or alias with a command sub-option and perhaps some argument values. For example, the command "W2 COLOR RED" changes the color of the second wall. Note the noun-verb structure of these commands: first the name of the object, then the action to be applied to that object.

There are three special "group names": "W*", "R*", and "P*". These refer to all of the items of each type: walls, rakes, and plots. Many of the commands can be applied to all of the items in that group. For example, "P* BACKGROUND WHITE" sets the background color of all the plots to white, as does "P* BA WH".

All of the available commands for any item are listed in response to the object's name followed by a question mark or the word "HELP". This produces a listing similar to this one:

```
help  -- print a list of available commands
show  -- print a description of this item
edit  -- modify this item interactively
alias -- assign an alias to this item
...
```

All items support the command SHOW. This prints out information about that item, such as its position and current rendering attributes. EDIT allows one to enter a sequence of commands to a single item; one command per input line, terminated with a blank line. For example:

```
w1 edit
   style lines
   color red
   linewidth 2
```

Finally, all items will react to the ALIAS command, which allows one to assign meaningful alternate names to each item. Plots, walls, and rakes each support additional commands. These are described in the sections below.

CHANGING A PLOT

Each plot supports the commands HELP, SHOW, EDIT, and ALIAS; as well the additional commands listed below. Remember that each of these commands is invoked by typing the name or alias of an existing plot, followed by a non-ambiguous prefix of the command to be applied to that item.

```
close           -- close this window
open            -- re-open this window
iconify         -- iconify this window
xywh            -- set the window position
duplicate       -- create a new window like this one
find-view       -- show my models, wherever they are
mark-view       -- remember this view
goto-view       -- use the remembered view
snap-view       -- align the view to the nearest axis
zoom            -- set the camera angle-of-view
magnify         -- change the apparent size of the items
truck           -- change the distance to the items
pan             -- shift the image across the screen
angles          -- set the view azimuth and elevation
sphere          -- set a spherical ``region of interest''
include         -- insert a wall, rake, or plot
exclude         -- remove an item from this plot
mirror          -- mirror the displayed models
scale           -- adjust size in each dimension
rotate          -- rotate/replicate models around an axis
outline-shift   -- adjust the relative depth of lines
background      -- specify a new background color
save-image      -- write the current image to a file
monochrome?         -- display only in black and white?
```

A plot may be closed with the "go-away" button in the upper-left corner of the window, or with the CLOSE command. Windows are never actually destroyed, and so a previously closed plot can be returned to the screen with the command OPEN. Windows may also be iconified, using either the small button in the upper-right corner of the window frame or by using the ICONIFY command. An open window may also be positioned on the screen by using the XYWH command to specify the position and size. The position is the pixel location of the lower-left corner of the image, relative to the lower-left corner of the screen. The DUPLICATE command will create a new plot similar to the existing plot which was the recipient of this command.

The mouse is used to control the viewing direction and magnification, but four commands are provided for other view-control tasks. FIND-VIEW will calculate a bounding rectangular volume about the walls and rakes, and will place these models nicely centered in the current plot. Any view may be saved with MARK-VIEW, and the most recently saved view can be recalled with the GOTO-VIEW command. Thus, one can mark the view in one plot, and "goto" that same view in any number of the other plots on the screen. The group-command "P* GOTO" will apply this saved view to every plot. The command SNAP-VIEW shifts the viewing angles of azimuth and elevation each to the nearest multiple of ninety degrees.

Plots will normally display their contents in an orthographic projection, but each plot can be changed to display its models in a perspective view. The command ZOOM takes an angle, between 0 and 160 degrees, for the field of view of the camera lens. An angle of zero describes the orthographic projection, small angles create a "telephoto" appearance, and large angles create a "fisheye" effect. (Plot3d uses a field-of-view of twenty degrees.) When the view is orthographic, then the MAGNIFY command can be used to change the apparent size of the displayed items. Under a perspective projection, the apparent size of the models can be reduced by changing to a wider angle or by moving the objects further away. The TRUCK commands is used to alter the distance between the camera and the models. As in Plot3d, the vertical dragging of the middle mouse button may be used change the apparent size of the displayed objects.

The view may also be modified by adjusting the horizontal and vertical displacement of the image with the PAN command, or by dragging the mouse with the right button held down. The azimuth and elevation of the view using the ANGLES command, or by a drag of the left mouse button.

The near and far clipping planes are automatically positioned around a spherical "region of interest" which is constructed around the MINMAX box. This sphere may be changed using the SPHERE command to indicate a new point and radius which encloses those models which one wishes to view. Note that the center of the region of interest is not generally the center of rotation. The latter location is always determined to be the point on a model which has been most recently selected with the mouse.

Each wall or rake may appear in one or more plots. A model may be placed into a plot with the command INCLUDE, which takes a model name or alias as its argument. An item may be removed from a plot with the command EXCLUDE. If a plot name is given as an argument to either of these commands, this is considered to represent the entire group of models within that plot. Group-commands work here as well, but consider the difference between "P* EXCL W1" and "P1 EXCL W*". The former removes the first wall from all plots, while the latter deprives the first plot of all of its walls.

Each plot typically displays each of its models only once. Three commands are available to change this behavior. The MIRROR command accepts the keywords X, Y, Z, or NONE. If any of the axes are selected, then all the models within the plot window will be reflected about the zero-plane in that dimension. The SCALE command accepts three numbers which specify the independent magnification to be applied to each physical space dimension. This is useful for expanding the image along one dimension relative to the other two, perhaps for investigating small features in the boundary layer. Finally, the ROTATE command accepts an axis, an angle, and a small integer. It then creates that many copies of the models, each rotated about the specified axis by a multiple of the specified number of degrees.

The polygons and line segments of each model are drawn into a integer-valued hardware Z-buffer. The OUTLINE-SHIFT command can be used to displace the line segments slightly forward in the computed depth, thereby improving the appearance of outlined polygons. The value of OUTLINE-SHIFT defaults to three thousand; experimenting with other values may improve the quality of some scenes. The BACKGROUND command can change the color used for the background (no surprise here!); it accepts a few pre-defined color names or the letters "RGB" followed by three component values between zero and one.

The image displayed in a window can be saved to an "RGB" or "BW" SGI-format image file using the SAVE-IMAGE command. This takes a filename as its argument, which defaults to "flora" with the appropriate extension, either "RGB" or "BW". Multiple images can be saved to the same filename; a version number will be inserted between the base part of the file name and its extension to prevent any loss of previously saved images.

Use the MONOCHROME? command to view models in a style suitable for dumping to a laser-printer. All lighting will be disabled and the visuals will be drawn with black lines and textures on a white background. Image files taken from a monochrome window can be printed with this command:

```
$ /usr/sbin/tops fname -b 1 -p 300 | lpr
```

This converts the image data to PostScript format with one bit per pixel and three hundred pixels to the inch. When printed to a 300-dpi printer (such as a LaserWriter), the hardcopy output from a full-screen image will be about four inches wide.

CHANGING A MODEL

The WALLS and the RAKES commands both create graphical objects or "models" for display. Models recognize the commands HELP, SHOW, EDIT, and ALIAS. Additional commands supported by all models are these:

```
style            -- change the rendering style
color            -- change the item color
line-width       -- change the line width
flip-colors?         -- exchange the front/back colors
smooth-shading?  -- average the surface normals?
include          -- place this model into a plot
exclude          -- remove this model from a plot
recompute        -- rebuild this model
```

These first of commands allows one to change the style in which a model is drawn. As mentioned above, the options for rendering style are POINTS, LINES, SHADED-SURFACE, and HIDDEN-LINES. Streamlines (of course) will only be drawn as points or lines. Flora draws hidden-line objects with normal coloration and shading, but adds a black outline around each polygon.

Other commands allow one to change the color of the model and the width of its lines. The color may be any one of a set of recognized names or three RGB values. The FLIP-COLORS command changes the Flora's notion of "front" and "back" for polygonal objects, thus exchanging the colors on these two sides. In the current implementation, the front color may be specified by the user, but the back color is always light gray. The surface normals used in the lighting calculation may be calculated for each polygonal facet of the model, or they may be computed as average values at each vertex point. The SMOOTH-SHADING command allows one to switch between these faceted and smooth representations for each polygonal model.

The INCLUDE and EXCLUDE commands allows one to place a model into a specified plot, or to remove a model from a plot. The RECOMPUTE command simply regenerates the model from previously specified parameters, for whatever reason this might be needed.

CHANGING A WALL

Each wall recognizes the generic model commands described in the previous section. A wall can be moved with these additional commands:

```
move     -- change grid number and all indices
grid     -- change the grid number
I-move   -- change the limits along I
J-move   -- change the limits along J
K-move   -- change the limits along K
```

These commands allow one to specify the grid number, or one of more sets of new indices. The index specification may include the keywords "ALL" or "LAST". An increment may also be specified, but this version of Flora always draws walls with an increment of one.

CHANGING A RAKE

Rake items support these additional commands:

```
move           -- change rake location
shape          -- change the rake shape
radius         -- size of circle or cross
surface?               -- create a surface (instead of lines)?
upstream?             -- advect particles upstream?
downstream?         -- advect particles downstream?
facet-size     -- max length of segments and edges
comp-step      -- integration step-length
improve        -- increase facets by 3/2
degrade        -- decrease facets by 2/3
num-seeds      -- change number of streamlines
max-points     -- change limit of saved points
duplicate      -- create a new rake, near this one
snap-rake      -- align rake to nearest gridline or axis
texture?               -- apply striped texturing?
adjust-texture -- specify spacing of stripes
```

The location of a rake is usually changed using the mouse, but it may also be repositioned using the MOVE command to specify a new center point or two new endpoints. The SHAPE command allows one to select between the SEGMENT, CURVE, CROSS, or CIRCLE rake forms. The cross and circle shapes also require the specification of a radius. This is most easily controlled with the middle button of the mouse, but may also be specified with the RADIUS command.

The SURFACE? command determines whether the model computed from this rake will be a single stream surface or a family of streamlines. The commands UPSTREAM? and DOWNSTREAM? control advection along the negative and positive directions of pseudo-time. The length of the lines and the area of the surfaces can be changed with the MAX-POINTS command, which alters the number of points which have been allocated for the result buffers of the particle advection. The integration

COMP-STEP is normally set to one-fifth of a cell, but may be reduced to improve the accuracy of the computed curves and surfaces.

Flora maintains an upper-bound on the physical-space distance along any edge of any triangle in the surface or any segment along a streamline. The default length is about one 1/100th of the diagonal of the bounding volume of the created models. A new value may be specified explicitly using the FACET-SIZE qualifier or the like-named command. This size may also be adjusted with the commands IMPROVE and DEGRADE, which multiply or divide the current facet size by two-thirds.

The FACET-SIZE command also controls the separation distance between the initial seed points of a family of streamlines. This can be overruled by specifying a integer argument for the NUM-SEEDS command, in place of its default value of the keyword AUTO. The DUPLICATE command creates a new rake, similar to but offset slightly from the original rake. SNAP-RAKE aligns the rake to one of the Cartesian axes or nearby gridlines.

A stream-surface or a set of streamlines may be overlaid with a striped texture. The TEXTURE? command enables this rendering option. The texture contains two series of orthogonal stripes which depict the streamlines and the timelines. The spacing of each set of stripes may be altered with the command ADJUST-TEXTURE. The spacing of the streamline stripes is based on the distance in physical space, while the timeline separation is specified in the total pseudo-time accumulated by the numerical integration method. Note that the same texture spacing is applied to every rake; changing the spacing for any rake will apply this change to all existing rakes and any rakes created later during that session. This global change of the texture spacing is done to avoid the inadvertant and misleading construction of rakes showing differently spaced timelines.

IMPLEMENTATION DETAILS

Flora assumes the common trilinear cell-local interpolating function within each hexahedral grid cell. Particle advection is performed with a second-order Runge-Kutta method in physical coordinates. This integration method uses a "reactive" stepsize adjustment which limits the stepsize such that successively computed points along the curve are typically separated by about one-fifth of a cell-width along any grid dimension.

Streamlines and stream-surfaces are computed in the velocity field, which is computed on an as-needed basis during the advection of the particles. This is done by dividing the momentum components (Q2,Q3,Q4) at each corner of the currently enclosing cell by the density value (Q1). The resulting velocity values are then interpolated at subsequent query locations within this same cell.

No "wall bouncing" is presently performed. That means that interpolation, truncation, and numerical errors may result in streamlines which strike a non-slip boundary and incorrectly terminate at this location.

Flora reads the iblank data from the grid file, and then computes the locations of (most of) the donor-points. These values are used to speed the transition of advecting particles across inter-block boundaries. Flora also manages to correctly traverse non-marked internal branch-cut boundaries of O-type and C-type grid blocks.

FILES

`.florarc`	`initial Flora commands`
`flora[-N].com`	`default checkpoint file`
`flora[-N].bw`	`default grayscale raster image filename`
`flora[-N].rgb`	`default color raster image filename`

SEE ALSO

```
fast (1)
plot3d (1)
rip (1)
ufat (1)
visual3 (1)
```

BUGS

Some important differences between Flora and Plot3D are these:

- Most Plot3d commands are not yet supported by Flora.
- Flora only runs on SGI workstations.
- READ only accepts the 3D, whole, binary format.
- WALLS only supports an index increment of one.
- RAKES creates a curve; not a set of points.
- The distinction between qualifiers and commands is ugly.

This package is intended to eventually be fully and upwardly compatible with Plot3d. Please contact the author if you need any particular command implemented soon; perhaps I will be able to rearrange the development schedule to accommodate your needs.

AUTHORS

David Betz wrote the underlying Scheme interpreter. This was extended into the "SuperGlue" package by Eric Raible and Jeff Hultquist. This package was used by Jeff Hultquist as a platform for the creation of Flora. Flora was heavily patterned after Plot3d, which was written by Pieter Buning.

Please send comments and bug reports via the GRIPE command. Suggestions and requests for help will be welcomed by the author, who may be contacted at:

```
Jeff Hultquist
(xxx) xxx-xxxx  -- office
(xxx) xxx-xxxx  -- pager
hultquist@xxx.xxx.xxx
```

BIBLIOGRAPHY

Gordon V. Bancroft et al. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Proceedings of Visualization '90*, pages 14–27, San Francisco, CA, October 1990.

R. Gary Belie. Flow visualization in the space shuttle's main engine. *Mechanical Engineering*, pages 27–33, September 1985.

R. Gary Belie. Some advances in digital flow visualization. In *AIAA Aerospace Sciences Meeting*, Reno, NV, January 1987. AIAA Paper 87-1179.

Davy M. Belk and Raymond C. Maple. Visualization of vortical flows with Yet Another Post Processor. In *AIAA Aerospace Sciences Meeting*, Reno, NV, January 1993. AIAA Paper 93-0222.

John A. Benek, Pieter G. Buning, and Joseph L. Steger. A 3D Chimera grid embedding technique. In *AIAA 7th Computational Fluid Dynamics Conference*, Cincinnati, OH, July 1985. AIAA Paper 85-1523-CP.

John Bentley. Multi-dimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), September 1975.

Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach. Image rendering by adaptive refinement. In *Computer Graphics (Proceedings of SigGraph)*, volume 20, pages 29–37, August 1986.

Robert S. Bernard. Grid-induced computational flow separation. In S. Sengupta et al., editors, *Numerical Grid Generation in Computational Fluid Mechanics*, pages 955–964, Swansea, Wales, 1988. Pineridge Press.

Jacques Bertin. *The Semiology of Graphics.* University of Wisconson Press, 1983. Translated by W.Berg.

David M. Betz. XSCHEME: an object-oriented Scheme, 1988. (http://www.cs.indiana.edu/scheme-repository/imp.html).

Jules Bloomenthal. Calculation of reference frames along a space curve. In Andrew Glassner, editor, *Graphics Gems.* Academic Press, Cambridge, MA, 1990.

Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991. Chapter 2.

Steve Bryson and Michael Gerald-Yamasaki. The distributed virtual windtunnel. In *Proceedings of Supercomputing '92*, pages 275–284, Minneapolis, MN, November 1992.

Pieter G. Buning. Numerical algorithms in CFD post-processing. von Karmann Institute for Fluid Dynamics, Lecture Series 1989-10, Computer Graphics and Flow Visualization in CFD, September 1989.

Pieter G. Buning. Sources of error in the graphical analysis of CFD results. *Journal of Scientific Computing*, 3(2):149–164, 1988.

Pieter G. Buning, I.T. Chiu, S. Obayashi, Yehia M. Rizk, and Joseph L. Steger. Numerical simulation of the integrated space shuttle vehicle in ascent. In *AIAA Atmospheric Flight Mechanics Meeting*, Minneapolis, MN, August 1988. AIAA Paper 88-4359.

Pieter G. Buning and Joseph L. Steger. Graphics and flow visualization in CFD. In *AIAA 7th CFD Conference*, pages 162–170, Cincinnati, OH, July 1985. AIAA Paper 85-1507-CP.

David M. Butler and M.H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3:45–51, Sep/Oct 1989.

Dave Darmofal and Robert Haimes. Visualization of 3D vector fields: Variations on a stream. In *AIAA Aerospace Sciences Meeting*, Reno, NV, January 1992. AIAA Paper 92-0074.

Willem C. de Leeuw and Jarke van Wijk. A probe for local flow visualization. In *Proceedings of Visualization '93*, pages 39–45, San Jose, CA, October 1993.

Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 26(1):43–48, January 1983. (Reprinted here from a 1967 conference proceedings.).

Robert R. Dickinson. A unified approach to the design of visualization software for the analysis of field problems. In *Three-dimensional Visualization and Display Technologies*, volume 1083, pages 173–180. SPIE, 1989.

Robert R. Dickinson and Richard H. Bartels. Fast algorithms for tracking contours, streamlines, and tensor field lines. In *Third International Conference on Computing in Civil Engineering*, Vancouver BC, August 1988.

Erich Eder. Visualisierung von Teilchenstroemungun mit Hilfe eines Vektorrechners. Master's thesis, Fachberiech Informatik, Fachhochschule Muenchen, Munich, February 1991. (Visualization of Flow Fields Using Vector Computers (*diplomarbeit*)).

John A. Ekaterinaris and Lewis B. Schiff. Vortical flows over delta wings and numerical prediction of vortex breakdown. In *AIAA Aerospace Sciences Conference*, Reno, NV, January 1990. AIAA Paper 90-0102.

Peter Eliasson, Jesper Oppelstrup, and Arthur Rizzi. STREAM3D: Computer graphics program for streamline visualization. *Advances in Engineering Software*, 11(4):162–168, 1989.

Richard Ellson and Donna Cox. Visualization of injection molding. *Simulation*, 51(5):184–188, November 1988.

Henry Fuchs, Zvi M. Kedem, and Samuel P. Uselton. Optimal surface reconstructions from planar contours. *Communications of the ACM*, 20(10):693–702, October 1977.

Datta Gaitonde. Personal communication, January 12, 1995.

Al Globus. A software model for visualization of time dependent 3D computer fluid dynamics results. Technical Report RNR-92-031, Numerical Aerodynamic Simulation Systems Division, NASA Ames, Moffett Field, CA 94035, November 1992.

Al Globus, Tom Lasinski, and Creon Levit. A tool for visualizing the topology of three-dimensional vector fields. In *Proceedings of Visualization '91*, pages 33–40, San Diego, CA, October 1991.

Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

Robert B. Haber. Visualization in engineering mechanics: Techniques, systems and issues. In *Visualization Techniques in the Physical Sciences*, Atlanta, GA, August 1988.

Robert Haimes. PV3: A distributed system for large-scale unsteady CFD visualization. In *AIAA Aerospace Sciences Meeting*, Reno, NV, January 1994. AIAA Paper 92-0321.

Andrew J. Hanson and Pheng A. Heng. Visualizing the fourth dimension using geometry and light. In *Proceedings of Visualization '91*, pages 321–328, San Diego, CA, October 1991.

James L. Helman and Lambertus Hesselink. Representation and display of vector field topology in fluid flow data sets. *IEEE Computer*, pages 27–36, August 1989.

James L. Helman and Lambertus Hesselink. Surface representations of two- and three-dimensional fluid flow topology. In *Proceedings of Visualization '90*, pages 6–13, San Diego, CA, October 1991.

Steve Hill. Tri-linear interpolation. In Paul Heckbert, editor, *Graphics Gems IV*. Academic Press, Cambridge, MA, 1994.

Andrea J.S. Hin. *Visualization of Turbulent Flow*. PhD thesis, Department of Informatics, Technische Universiteit Delft, Netherlands, September 1994.

Andrea J.S. Hin and Frits H. Post. Visualization of turbulent flow with particles. In *Proceedings of Visualization '93*, pages 46–52, San Jose, CA, October 1993.

Vi Hirsch and Chris Gong. CFD technical highlights videotape, 1994. Numerical Aerodynamic Simulation Systems Division, NASA Ames Research Center, Moffett Field, CA 94035.

Thomas J. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall, 1987. Chapter 3.

Jeff P.M. Hultquist. Numerical flow visualization in a functional style. Technical Report RNR-89-008, Numerical Aerodynamic Simulation Systems Division, NASA Ames, Moffett Field, CA 94035, June 1989.

Jeff P.M. Hultquist. Interactive numerical flow visualization using stream surfaces. *Computing Systems in Engineering*, 1(2-4):349–353, 1990.

Jeff P.M. Hultquist. Constructing stream surfaces in steady 3d vector fields. In *Proceedings of Visualization '92*, pages 171–178, Boston, MA, October 1992.

Jeff P.M. Hultquist. Improving the performance of particle tracing in curvilinear grids. In *AIAA Aerospace Sciences Meeting*, Reno, NV, January 1994. AIAA Paper 94-0324.

Jeff P.M. Hultquist and Eric Raible. Superglue: A programming environment for scientific visualization. In *Proceedings of Visualization '92*, pages 243–250, Boston, MA, October 1992.

David Kenwright. *Dual Stream Function Methods for Generating 3-Dimensional Stream Lines*. PhD thesis, Department of Mechanical Engineering, University of Auckland, New Zealand, August 1993.

David Kenwright and Gordon Mallinson. A streamline tracking algorithm using dual stream functions. In *Proceedings of Visualization '92*, pages 62–68, Boston, MA, October 1992.

G. David Kerlick. Moving iconic objects in scientific visualization. In *Proceedings of Visualization '91*, pages 124–129, San Francisco, CA, October 1990.

Myron W. Krueger. Personal communication, February 26, 1992.

Myron W. Krueger. *Artificial Reality.* Addison-Wesley, Reading, MA, second edition, 1991. pages 175-176.

David Lane. UFAT – a particle tracer for time-dependent flow fields. In *Proceedings of Visualization '94*, pages 257–264, Washington D.C., October 1994.

Kwan-Liu Ma and Philip J. Smith. Cloud tracing in convection-diffusion systems. In *Proceedings of Visualization '93*, pages 253–260, San Jose, CA, October 1993.

Fred W. Martin, Jr. and Jeffrey P. Slotnick. Flow computations for the space shuttle in ascent mode using thin-layer navier-stokes equations. In P.A. Henne, editor, *Applied Computational Aerodynamics (Progress in Astronautics and Aeronautics)*, volume 125, pages 863–886. AIAA, Washington, D.C., 1990.

C. Wayne Mastin. Error induced by coordinate systems. In J.F. Thompson, editor, *Numerical Grid Generation*, pages 31–40. Elsevier, 1982.

C. Wayne Mastin. Fast interpolation schemes for moving grids. In S. Sengupta et al., editors, *Numerical Grid Generation in Computational Fluid Mechanics*, pages 63–73, Swansea, Wales, 1988. Pineridge Press.

Donovan L. Mathias, Karlin R. Roth, James C. Ross, Stuart E. Rogers, and Russell M. Cummings. Navier-stokes analysis of the flow about a flap edge. In *AIAA Aerospace Sciences Meeting*, Reno, NV, January 1995. AIAA Paper 95-0185.

Nelson Max, Barry Becker, and Roger Crawfis. Flow volumes for interactive vector field visualization. In *Proceedings of Visualization '93*, pages 19–24, San Jose, CA, October 1993.

Robert L. Meakin. A new method for establishing intergrid communication among systems of overset grids. In *AIAA 10th Computational Fluid Dynamics Conference*, Honolulu, HI, June 1991. AIAA Paper 91-1586-CP.

Earl M. Murman and Kenneth G. Powell. Trajectory integration in vortical flows. *AIAA Journal*, 27(7):982–984, August 1988.

Gregory M. Nielson and Dan R. Olsen Jr. Direct manipulation techniques for 3d objects using 2d locator devices. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pages 175–182, Chapel Hill, NC, October 1986. ACM, New York.

Daniel G. Pearce, Scott Stanley, Fred Martin, Ray Gomez, Gerald Le Beau, Pieter Buning, William Chan, Ing-Tsau Chiu, Armin Wulf, and Vedat Akdag. Development of a large-scale chimera grid system for the space shuttle launch vehicle. In *AIAA Aerospace Sciences Meeting*, Reno, NV, January 1993. AIAA Paper 93-0533.

Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

Jonathon Rees and William Clinger. Revised (4th) report on the algorithmic language Scheme, 1990. (http://www.cs.indiana.edu/scheme-repository/doc.standards.html).

Yehia M. Rizk and Shmuel Ben-Shmuel. Computational of the viscous flow around the shuttle orbiter at low supersonic speeds. In *AIAA Aerospace Sciences Meeting*, Reno, NV, January 1985. AIAA Paper 85-0168.

J.M. Robertson. *Hydrodynamics Theory and Applications.* Prentice-Hall, 1965. page 73.

Stuart Rogers. Personal communication, February 13, 1995.

Ari Sadarjoen, Theo van Walsum, Andrea J.S. Hin, and Frits H. Post. Particle tracing algorithms for 3D curvilinear grids. In *Eurographics Workshop on Visualization in Scientific Computing*, Rostok, Germany, May 1994.

Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):365–369, June 1984.

W.J. Schroeder, C.R. Volpe, and W.E. Lorenson. The stream polygon: A technique for 3d vector field visualization. In *Proceedings of Visualization '91*, pages 126–131, San Diego, CA, October 1991.

S. Shirayama. Visualization of vector fields in flow analysis I. In *29th Aerospace Sciences Meeting*, Reno, NV, January 1991. AIAA Paper 91-0801.

Peter Shirley and Henry Neeman. Volume visualization at the center for supercomputing research and development. Technical report, University of Illinois at Urbana-Champaign, Urbana, IL, January 1989. CSRD Report Number 849.

M. Siclari, 1991. Personal communication regarding the picture on the cover of *Science*, 245(28), July 1989.

J. Stolk and Jarke J. van Wijk. Surface-particles for 3d flow visualization. In *Proceedings of the Second Eurographics Workshop on Visualization in Scientific Computing*, pages 22–24, Delft, The Netherlands, April 1991.

Yoshiaki Tamura and Kozo Fujii. Visualization for computational fluid dynamics and the comparison with experiments. In *8th Applied Aerodynamics Conference*, Portland, OR, August 1990. AIAA Paper 90-3031-CP.

Joseph F. Thompson. General curvilinear coordinate systems. In J.F. Thompson, editor, *Numerical Grid Generation*, pages 1–30. Elsevier, 1982.

David Ungar and Randall B. Smith. Self: The power of simplicity. *LISP and Symbolic Computation*, 4(3):187–205, July 1991.

Milton van Dyke. *An Album of Fluid Motion.* The Parabolic Press, Stanford, CA, 1982.

Jarke J. van Wijk. Rendering surface particles. In *Proceedings of Visualization '92*, pages 54–61, Boston, MA, October 1992.

Jarke J. van Wijk. Implicit stream surfaces. In *Proceedings of Visualization '93*, pages 245–252, San Jose, CA, October 1993.

Jarke J. van Wijk, Andrea J.S. Hin, Willem C. de Leeuw, and Frits Post. Three ways to show 3D fluid flow. *IEEE Computer Graphics and Applications*, 14(5):33–39, September 1994.

G. Volpe. Streamlines and streamribbons in aerodynamics. In *27th Aerospace Sciences Meeting*, Reno, NV, January 1989. AIAA Paper 89-0140.

Robert P. Weston. Applications of color graphics to complex aerodynamics analysis. In *AIAA 25th Aerospace Sciences Meeting*, Reno, NV, January 1987. AIAA Paper 87-0273.

Robert P. Weston. Color graphics techniques for shaded surface displays of aerodynamic flowfield parameters. In *AIAA 8th CFD Conference*, Honolulu, HI, June 1987. AIAA Paper 87-1182-CP.

Peter L. Williams. Visibility ordering meshed polyhedra. *Transactions on Graphics*, 11(2):103–126, April 1992.

Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the 1992 International Workshop on Memory Management (Springer Lecture Notes #637)*, 1992. (ftp://cs.utexas.edu/pub/garbage/gcsurvey.ps).

Chia-Shun Yih. Stream functions in three-dimensional flows. In *Selected Papers*, pages 893–898. World Scientific, Teaneck, NJ, 1991. (First published in *La Houlle Blanche*, 1957).

Susan X. Ying, Lewis B. Schiff, and Joseph L. Steger. A numerical study of three-dimensional separated flow past a hemisphere cylinder. In *AIAA 19th Fluid Dynamics, Plasma Dynamics and Lasers Conference*, Honolulu, HI, June 1987. AIAA Paper 87-1207.