

# Visualization of Three Dimensional CFD Results

by

**David Laurence Modiano**

S.B. Massachusetts Institute of Technology (1987)

SUBMITTED IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

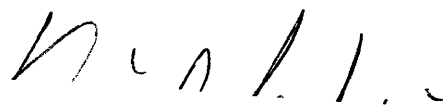
**Aeronautics and Astronautics**

at the

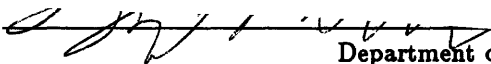
**Massachusetts Institute of Technology**

May 1989

©1989, Massachusetts Institute of Technology



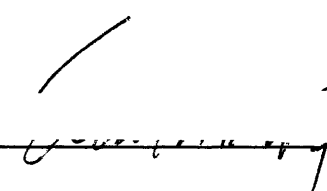
Signature of Author



Department of Aeronautics and Astronautics

May 12, 1989

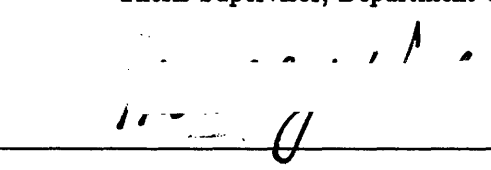
Certified by



Professor Earl M. Murman

Thesis Supervisor, Department of Aeronautics and Astronautics

Accepted by



Professor Harold Y. Wachman

Chairman, Department Graduate Committee

MASSACHUSETTS INSTITUTE  
OF TECHNOLOGY

JUN 07 1989

WITHDRAWN  
M.I.T.  
LIBRARIES

# **Visualization of Three Dimensional CFD Results**

by

**David Laurence Modiano**

Submitted to the Department of Aeronautics and Astronautics

on May 12, 1989

in partial fulfillment of the requirements for the degree of

**Master of Science in Aeronautics and Astronautics**

We have developed an interactive graphics system for the display of three dimensional CFD solutions on unstructured hexahedral grids. This system is implemented on a high-performance graphics supercomputer. Visualization methods employed are shaded color surface plots, integration of particle trajectories, interpolation of volumetric data onto a plane, interpolation of planar data onto a line segment, and extraction of numerical quantities from a plane. We have used this graphics system to examine the inviscid flow about the NTF delta wing, as solved by Becker, and found that it allows us to locate flow features quickly. We were unable to find a satisfactory method to visualize the three dimensional mesh structure. With the exception of particle path integration, the algorithms we have implemented can be used to visualize any volumetric data.

**Thesis Supervisor: Earll M. Murman,**

**Professor of Aeronautics and Astronautics**

## Acknowledgements

“If I have seen farther than others, it is because I have stood on the shoulders of giants.” Or something like that. This thesis is no *Principia*, but it’s the thought that counts. So I’d like to thank all the people upon whose shoulders I have stood.

First and foremost I would like to thank my thesis advisor, Earll Murman, who got me interested in CFD in the first place, for his enthusiastic support of a somewhat offbeat thesis topic. Also, my most profuse thanks goes to Mike Giles and Bob Haines, who both made important technical contributions to the work herein.

I’ve found that I have learned at least as much from my fellow students at the CFD Lab as from the formal education process. In particular, I’d like to thank Rich Shapiro, who taught me a lot of what I know about how these damn machines work, and my officemates, Hans Ekander and Yannis Kallinderis.

Bob Bruen and Teva Regule deserve massive public adulation for keeping most of the hardware alive, especially this year when just about everything tried to die at least once.

MIT can be a very demanding place, and its easy to lose yourself without an anchor to “reality.” I’d like to thank my friends for anchoring me, in particular Jan and Bernie, and Chris, Jeff, Eric, Eon, Jen, Mike, Laura, Rob, and, of course, all the Daves. You people are at least half the reason why I stay around here.

Last, but certainly not least, I’d like to thank my parents for accepting my adulthood with grace, and treating me with respect. This is more important than most people realize.

“What if he’s got a bunch?”

This work was supported by NASA Grants NAG-1-507 and NAG-1-855, monitored by Duane Melson, NASA Langley Research Center.

# Contents

<b>Abstract</b>	<b>2</b>
<b>Acknowledgements</b>	<b>3</b>
<b>Nomenclature</b>	<b>12</b>
<b>1 Introduction</b>	<b>14</b>
1.1 Background . . . . .	14
1.2 Overview . . . . .	16
<b>2 Graphics Concepts</b>	<b>17</b>
2.1 Drawing Primitives . . . . .	17
2.1.1 Line Drawing . . . . .	17
2.1.2 Shading Methods . . . . .	18
2.1.3 Area Filling . . . . .	19
2.1.4 Rendering Arbitrary Groups of Polygons . . . . .	20
2.2 Double Buffering and z-Buffering . . . . .	23
2.2.1 Double Buffering . . . . .	23

2.2.2	The <i>z</i> -Buffer Algorithm . . . . .	23
2.3	Colormaps . . . . .	25
2.3.1	Types of Colormaps . . . . .	26
2.4	Transformation Matrices . . . . .	29
2.4.1	Translation . . . . .	30
2.4.2	Dilation . . . . .	31
2.4.3	Rotation . . . . .	31
2.4.4	Homogeneous Coordinates . . . . .	32
2.4.5	Three Dimensional Transformations . . . . .	34
2.4.6	Unit Vector Representation . . . . .	36
2.4.7	Perspective Projection . . . . .	37
<b>3</b>	<b>The Graphics Supercomputer</b>	<b>41</b>
3.1	Stellar Hardware . . . . .	41
3.2	Stellar Software . . . . .	43
<b>4</b>	<b>Data Structures</b>	<b>44</b>
4.1	Cell Geometry . . . . .	44
4.2	Fundamental Quantities . . . . .	46
4.2.1	Mesh Geometry . . . . .	46
4.2.2	Flow Quantities . . . . .	46

4.3	Additional Quantities . . . . .	47
4.4	Memory Usage . . . . .	48
<b>5</b>	<b>Visualization Methods</b>	<b>50</b>
5.1	Surface Plot . . . . .	50
5.2	Particle Path Integration . . . . .	51
5.2.1	Trajectory Equations . . . . .	51
5.2.2	Numerical Integration Scheme . . . . .	52
5.2.3	Trilinear Interpolation . . . . .	53
5.2.4	Geometry Metrics . . . . .	54
5.2.5	Cell-to-Cell Interfaces . . . . .	55
5.3	Planar Interpolation . . . . .	58
5.3.1	Basic Interpolation Algorithm . . . . .	58
5.3.2	Enhanced Interpolation Algorithm . . . . .	60
5.3.3	Display Formats . . . . .	63
5.3.4	Thresholding . . . . .	64
5.4	Electronic Probes . . . . .	67
5.4.1	Linear Profile . . . . .	67
5.4.2	Point Probe . . . . .	68
<b>6</b>	<b>Results</b>	<b>69</b>

6.1	NTF Delta Wing Calculation . . . . .	70
6.2	Practical Guidelines and Experiences . . . . .	73
<b>7</b>	<b>Summary and Conclusions</b>	<b>83</b>
7.1	Summary . . . . .	83
7.2	Conclusions . . . . .	84
7.3	Recommendations for Further Work . . . . .	85
	<b>Bibliography</b>	<b>87</b>
<b>A</b>	<b>Disk File Format</b>	<b>90</b>
<b>B</b>	<b>Screen Layout</b>	<b>92</b>

## List of Figures

2.1	Types of lines. . . . .	18
2.2	Area filling. . . . .	19
2.3	A five-sided polygon divided using the centroid method. . . . .	20
2.4	Waddling over a six-sided polygon. . . . .	21
2.5	Two polygons joined by two zero-area triangles. . . . .	22
2.6	Hidden surface removal by $z$ -buffering. . . . .	24
2.7	Pixel value to RGB mapping with the colormap on a color display. . . .	25
2.8	Heirarchy of Colormap types. . . . .	27
2.9	Pixel value to RGB mapping for direct color and true color colormaps. .	28
2.10	A projection with two principal vanishing points. . . . .	38
2.11	Perspective projection. . . . .	38
2.12	Similar triangles in perspective projection. . . . .	38
3.1	Stellar GS-1000 hardware architecture. . . . .	42
4.1	Cell geometry. . . . .	45
5.1	Cell geometry. . . . .	53



5.2	Cell-to-cell interface of integration path. . . . .	55
5.3	Moving an interpolation plane. . . . .	62
5.4	Thresholded data. . . . .	63
5.5	Thresholding of polygons. . . . .	64
5.6	Subdivision and thresholding of a five-sided polygon. . . . .	65
5.7	Triangle thresholding situations. . . . .	66
6.1	NTF delta wing geometry. . . . .	70
6.2	Pressure on the upper side of the wing, and on a plane at the 80% axial station, and on a line normal to the wing surface through the center of the vortex. . . . .	74
6.3	Pressure on the upper side of the wing, and on a plane at the 80% axial station, and at the center of the vortex. . . . .	74
6.4	Total pressure on the upper side of the wing, and on a plane at the 90% axial station, and on a line normal to the wing surface through the center of the vortex. . . . .	75
6.5	Total pressure on the upper side of the wing, and on a plane at the 90% axial station, and at the center of the vortex. . . . .	75
6.6	Eight particle paths, viewed from the side. . . . .	76
6.7	Eight particle paths, viewed from the top. . . . .	76
6.8	Eight particle paths, viewed from the front. . . . .	77
6.9	Total pressure interpolated onto multiple planes, with threshold values at $p_0 = 0.5$ and $p_0 = 1.0$ . Oblique view. . . . .	77

6.10	Total pressure interpolated onto multiple planes, with threshold values at $p_0 = 0.5$ and $p_0 = 1.0$ . View from above, looking aft. . . . .	78
6.11	Total pressure variation in a plane at the 120% axial station, and on a line through the two vortices. . . . .	78
6.12	Total pressure variation in a plane at the 120% axial station, and at the center of the main vortex. . . . .	79
6.13	Total pressure variation in a plane at the 120% axial station, and at the center of the counter-rotating vortex. . . . .	79
6.14	Total pressure variation in a plane at the 190% axial station, and on a line through the vortex. . . . .	80
6.15	Total pressure variation in a plane at the 190% axial station, and at the center of the vortex. . . . .	80
6.16	Mach number in the symmetry plane. . . . .	81
6.17	Mach number in the symmetry plane. Trailing edge detail. . . . .	81
B.1	Arrangement of windows in screen display. . . . .	93

## List of Tables

2.1	Comparison of Colormap types . . . . .	26
4.1	Face node definitions. . . . .	45
5.1	Face node definitions. . . . .	56
5.2	Face local coordinates in the old cell. . . . .	56
5.3	Relation between face local coordinates in the old and new cells. . . . .	57
5.4	Face local coordinates in the new dell. . . . .	57

# Nomenclature

$a, b, c$	compoments of unit vector normal to interpolation plane
$c$	speed of sound
$d$	distance behind projection plane
$d$	location of interpolation plane
$E$	total internal energy per unit mass
$J$	Jacobian
$J$	Jacobian matrix
$M$	Mach number
$M$	generalized transformation matrix
$M_p$	perspective matrix
$N_C$	number of cells in the mesh
$N_V$	number of nodes in the mesh
$n_{lt}$	number of points below lower threshold contour
$n_{gt}$	number of points above upper threshold contour
$p$	static pressure
$p_0$	total pressure
$p_{0ref}$	reference total pressure
$P$	a point in space
$P'$	a transformed point in space
$q$	flow speed
$q$	a quantity
$r_{11}, r_{12}, \dots, r_{33}$	some elemnts of the matrix $R$
$R$	rotation matrix
$R_x, R_y, R_z$	rotation matrices for $x, y, z$ axes
$s$	a scalar quantity
$s_{min}$	lower threshold value of $s$
$s_{max}$	upper threshold value of $s$

$s_x, s_y, s_z$	dilation in the $x, y, z$ directions
<b>S</b>	dilation matrix
$t_x, t_y, t_z$	translation in the $x, y, z$ directions
$T_1, \dots, T_8$	interpolation functions
<b>T</b>	translation matrix
$u, v, w$	flow velocities in the $x, y, z$ directions
$U, V, W$	contravariant velocities in the $\xi, \eta, \zeta$ directions
$\bar{U}$	characteristic contravariant velocity in a cell
$U_1, \dots, U_5$	components of <b>U</b>
<b>U</b>	state vector of conserved quantities
$x, y, z$	spatial coordinates
$x', y', z'$	transformed spatial coordinates
$x_\xi, x_\eta, \dots, z_\zeta$	geometry metrics
<b>X, Y, Z, W</b>	homogeneous coordinates
$z'_{\min}$	minimum value of $z'$ in a cell
$z'_{\max}$	maximum value of $z'$ in a cell
$\alpha$	perspective factor
$\gamma$	ratio of specific heats
$\theta, \phi, \psi$	angles of rotation about $x, y, z$ axes
$\rho$	mass density
$\xi, \eta, \zeta$	computational coordinates
$\xi_n, \eta_n, \zeta_n$	computational coordinates at time level $n$
$\xi^*, \eta^*, \zeta^*$	predicted computational coordinates
$\xi_x, \xi_y, \dots, \zeta_x$	geometry metrics
$\tilde{\xi}, \tilde{\eta}$	face local computational coordinates
$\Delta t$	temporal integration step

# Chapter 1

## Introduction

Over the last twenty years, computational fluid dynamics has gone from a method available only to the most prominent government projects and corporations to an important and widely used tool for design, research and analysis. With advances in computer technology that make more realistic three dimensional calculations feasible comes the problem of understanding and visualizing the results of such simulations. The goal of this work is to develop a visualization package that enables interactive investigation of three dimensional data that is available on an unstructured mesh.

### 1.1 Background

The numerical solution of the equations of fluid dynamics was at first greatly limited by the computational capabilities available. Panel methods, which take advantage of Green's theorem to represent the flow past a body in terms of source and doublet (or vortex) distributions on the body surface [30,9], allowed the flow over realistic geometries to be computed for the first time. Panel methods cannot, however, properly model rotational flows, such as arise through the effects of viscosity and compressibility.

In the late 1960s and early 1970s the first Euler solvers began to appear [19,21]. Limitations in computational speed and memory storage restricted the effort to the solution of the transonic full potential equations. Nevertheless, the success of the panel and potential methods led to claims that "computers should [soon] begin to supplant wind tunnels in the aerodynamic design and testing process." [6] By the end of the seventies, computer capabilities had increased enough to allow the solution of the full two dimensional Euler equations. The methods initially devised by Jameson [14] and

Ni [22] form the basis for most such finite volume schemes in use today. In the past few years, advances in computer technology have allowed the accurate solution of the three dimensional Euler equations [26], and the Navier-Stokes equations [?], as well.

The early potential and Euler schemes made use of structured grids for data storage. That is, the flow quantities were stored at a logically rectangular array of points in the flow domain. Such grids are difficult to generate about complex geometries. An alternative is to dispense with the implied ordering of the data points and to explicitly keep track of the connectivity of the points. Such unstructured methods were originally developed to solve structural mechanics problems, and were first applied to fluid mechanics shortly after the emergence of full Euler solvers [4]. Many modern algorithms use unstructured meshes to represent flows past complex geometries [17], especially for three dimensional flows [13]. In addition, an unstructured representation allows adaptation by mesh-point embedding [12,28] to increase the resolution in areas of "interesting" flow features, such as shocks, shear layers, *etc.*

The ability to numerically solve three dimensional fluid dynamics problems brings about the problem of how to display three dimensional solutions on output devices that are mostly two dimensional. Experts in computer graphics have developed sophisticated techniques to realistically render solid object models, but such capabilities are of little interest with regard to the visualization of volumetric data. Work at NASA Ames Research Center led to the development of PLOT3D [15], an interactive visualization package for structured grid solutions. PLOT3D uses the capabilities of the Silicon Graphics IRIS workstation to rotate and translate the image in real time. However, since the IRIS has only moderate floating point capabilities, many visualization strategies cannot be employed. Recent work attempts to use the computational power of the CRAY in conjunction with the graphics capabilities of the IRIS. Weston has developed a visualization package that uses many innovative techniques [32], but is incapable of real-time interaction. In the past two years, a new class of computer has emerged: the graphics supercomputer. This is a machine that combines the computational performance of minisupercomputers with the graphics power of the most advanced imaging systems. The field was initially currently composed of the Stellar GS-1000 and the Ar-

dent TITAN, with the Silicon Graphics Power Series and the Apollo DN 10000 being developed more recently. Visualization is one of the hot topics of the year, with many separate efforts producing results at the same time [20,7,29,31,27].

## 1.2 Overview

First, this thesis introduces a number of concepts from the field of computer graphics which are crucial to understand the details of our visualization package, which might not be familiar to the reader. These concepts are the basic structures that are rendered by the graphics hardware, the techniques of double buffering and  $z$ -buffering, the use of colormaps, and the representation of geometrical transformations as matrices. In chapter 3 we describe the capabilities and characteristics of graphics supercomputers in general and the Stellar GS-1000 in particular. We describe the data structures used by our methods in chapter 4.

Chapter 5 describes in detail the four visualization methods that we have implemented. Two of these methods, surface plots and particle path integration, have seen extensive previous usage, but the calculation of particle paths is much more complex when dealing with unstructured geometry data. The other two methods, planar interpolation and the electronic probes, have seen little, if any, previous use. Finally, in chapter 6, we describe the application of our package to investigate the results of a blunt delta wing Euler computation by Becker [3].



## Chapter 2

# Graphics Concepts

### 2.1 Drawing Primitives

The basic operations of rendering are the drawing of lines and the filling of areas. Simple systems are capable only of rendering line segments one at a time and of filling a single polygon with a uniform color. With such limited capabilities, complex processes, such as Gouraud shading, in which the color of a polygon is linearly interpolated between the color values at its corners, place great demands on the rendering software. With advances in hardware, particularly the use of floating-point vector units, larger and more complex data structures can be rendered as primitive objects.

#### 2.1.1 Line Drawing

A simple large data structure commonly used to draw lines is the polyline, illustrated in figure 2.1, which is a sequence of points, of which each two adjacent members define a line segment. A polyline representing  $n$  line segments requires  $n + 1$  points. The inherent advantage of using large data structures as rendering primitives is that the coordinates of the points that define the object can be processed very quickly using vector floating-point hardware. Such processing primarily consists of the geometrical transformations discussed at great length in section 2.4. Other data structures, similar to polylines, can be used when one wishes to draw a sequence of line segments that do not share adjacent endpoints. The polyhedral line is a set of points combined with a list of indices to these points that define a polyline or group of polylines. The disjoint polyline is essentially a collection of individual line segments, or a “dashed” polyline. A polyhedral line representing  $n$  line segments can contain virtually any number of points,

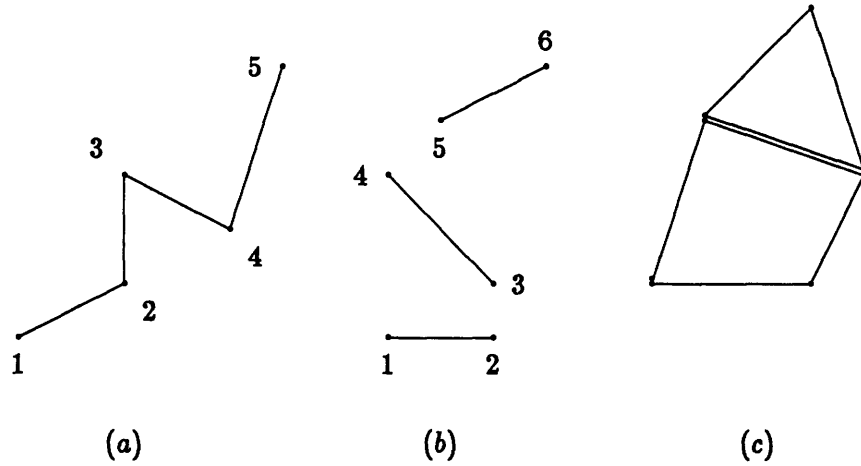


Figure 2.1: Types of lines. a) Polyline, b) Disjoint Polyline, and c) Polyhedral Line. The polyhedral line is composed of two separate polylines, one with five vertices, and one with four vertices.

whereas an  $n$  line segment disjoint polyline will contain  $2n$  points.

### 2.1.2 Shading Methods

Gouraud shading, as mentioned previously, is the linear interpolation of color values at the nodes to determine the color at an individual point in the interior of a polygon. Linear interpolation of a quantity  $q$  over a surface can be expressed as

$$q = c_0 + c_x x + c_y y. \quad (2.1)$$

The values of the coefficients  $c_0, c_x, c_y$  are determined by solving the system of equations

$$\begin{aligned} q_1 &= c_0 + c_x x_1 + c_y y_1 \\ q_2 &= c_0 + c_x x_2 + c_y y_2 \\ &\vdots \\ q_n &= c_0 + c_x x_n + c_y y_n, \end{aligned} \quad (2.2)$$

which are obtained by substituting into 2.1 the coordinates and the values of  $q$  at the corners of a polygon. It can only be performed directly on triangles, since a greater

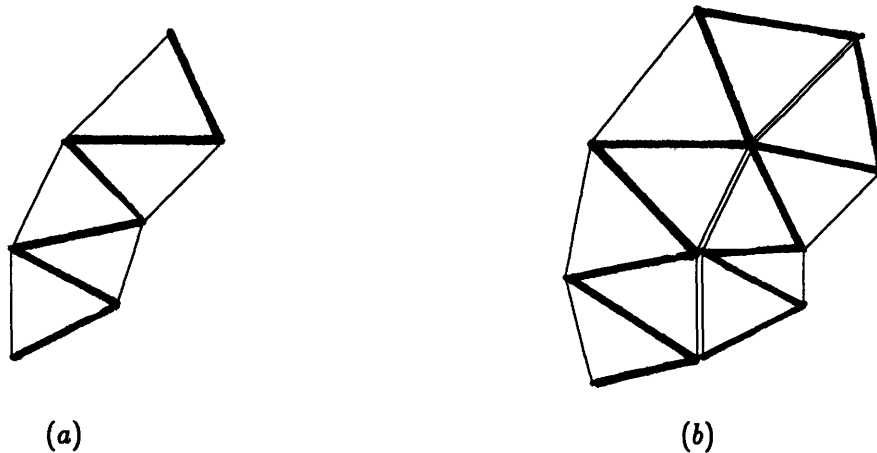


Figure 2.2: Area filling. *a*) Polytriangle surface, *b*) Polyhedral surface. The bold lines are between consecutive vertices. The nonbold lines show the triangles that are formed. The polyhedral surface is composed of two polytriangle surfaces, one with eight vertices and six triangles, the other with seven vertices and five triangles.

number of points would overdetermine equations 2.2.

Note that  $x$  and  $y$  are merely coordinates within the surface, and could be curvilinear. If we add a third coordinate  $z$ , we will have a method for interpolating  $q$  throughout a volume. In such a case, four data points, defining a tetrahedron, would be sufficient and necessary to determine the coefficients.

The hardware or software that performs Goraud shading can be used to interpolate other quantities as well, such as the surface normal vectors needed to perform lighting calculations, of the location of the polygon surface, which is used directly in the rendering process.

### 2.1.3 Area Filling

The basic large data structure used for area filling is called the polytriangle surface [1], which is a sequence of points not unlike that associated with a polyline, of

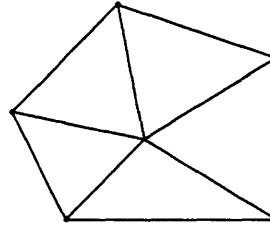


Figure 2.3: A five-sided polygon divided using the centroid method.

which every three adjacent points defines a triangle, as shown in figure 2.2. A polytriangle surface that defines  $n$  triangles will contain  $n + 2$  points. A structure that can be used to render a group of non-contiguous triangles is the polyhedral surface, similar to the polyhedral line. The polyhedral surface is a collection of points associated with a list of indices to these points that define one or more polytriangle surfaces.

#### 2.1.4 Rendering Arbitrary Groups of Polygons

The polytriangle and polyhedral surfaces described in section 2.1.3 only represent triangles. In order to draw an arbitrary set of polygons, we must devise some method to represent those polygons as a collection of triangles.

There are two methods that we use to divide a single polygon into a number of triangles. The first method divides an  $n$ -sided polygon into  $n$  triangles, each triangle having as its vertices two adjacent vertices of the original polygon and the polygon's centroid, as shown in figure 2.3. This method is used in the thresholding process, which section 5.3.4 describes. The triangles so produced cannot be arranged into a polytriangle surface unit without further manipulation.

The second method produces a polytriangle surface containing  $n$  nodes and  $n - 2$  triangles. The nodes of the polytriangle surface are, in order, the first vertex of the polygon, the last vertex of the polygon, the second vertex of the polygon, the second-to-last vertex of the polygon, and so on until all the polygonal vertices have been used.

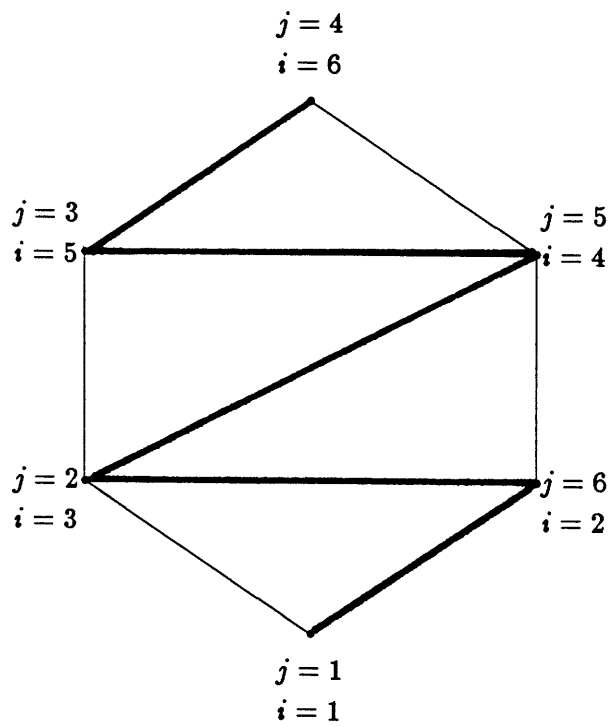


Figure 2.4: Waddling over a six-sided polygon.



Figure 2.5: Two polygons joined by two zero-area triangles.

Figure 2.4 illustrates this method, which is called polygon waddling. In this figure, the corners of the polygon are ordered by the index  $j$ , and the points comprising the polytriangle surface are ordered by the index  $i$ . Unlike the polygon-centroid method, the collection of triangles that is produced is dependent upon the polygon vertex numbering. However, it has the advantage of producing a polytriangle surface unit composed of fewer triangles, and does not require any additional floating-point calculations. We render the interpolation plane, the details of which are in section 5.3, using the polygon waddling method. Both methods are only guaranteed to produce sensible results when operating on convex polygons, but the geometry expected of a good CFD mesh will be associated only with convex polygons.

Once a polytriangle surface has been generated for each individual polygon, they must be assembled into one large data structure for more efficient rendering. One possibility is to create a polyhedral surface as described in section 2.1.3, with each individual polytriangle surface representing one polygon. A more efficient method is to assemble the triangles produced by waddling into one large polytriangle surface. This is done by the simple method of inserting two zero-area triangles between each two numerically adjacent (which are not necessarily physically adjacent) pair of polygons, as illustrated in figure 2.5. This is possible since zero-area triangles are ignored by the rendering hardware. The zero-area triangles are created by duplicating the first and last vertices of the small polytriangle surface.

## 2.2 Double Buffering and $z$ -Buffering

The techniques of double buffering and  $z$ -buffering both serve to increase the quality of a displayed image as well as decreasing the amount of computation needed to render that image, at the expense of vastly increased memory storage. Double buffering increases the quality of a series of animated images and allows an image to be rendered more quickly, while  $z$ -buffering is a method for the removal of hidden lines and surfaces. With the decreasing cost of large memory systems, they are becoming commonly used alternatives to the sometimes cumbersome algorithms previously used.

### 2.2.1 Double Buffering

Double buffering is a simple concept that is used to increase the visual quality of an animated series of images, and to increase the speed of the rendering process. A problem arises because of the finite time needed to render an image. If the display area is erased between two successive images, a flickering effect will result as the new image is drawn. This can be circumvented for a restricted class of situations (such as the animated motion of a large group of points) by erasing each small component of the image immediately before that component is redrawn. However, it is possible to render an image not into display memory, but into an arbitrary offscreen area of memory, which is copied directly to display memory when the rendering process is complete. This offscreen memory is often called the *refresh buffer*. The refresh buffer can be accessed more quickly than the display memory during the rendering process. A  $z$ -buffer can also be associated with such an offscreen image, so that parts of an image can be rendered into separate offscreen buffers, and then combined later.

### 2.2.2 The $z$ -Buffer Algorithm

One of the most difficult problems in achieving visual realism in computer graphics is that of hidden line, or hidden surface, removal. Essentially, the problem is how to

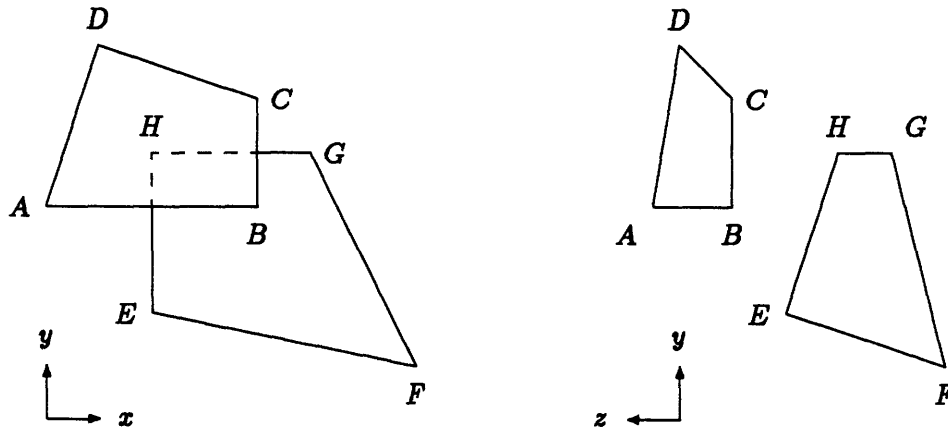


Figure 2.6: Hidden surface removal by  $z$ -buffering. Polygon  $ABCD$  partially obscures polygon  $EFGH$ .

determine which components of an object are visible, and which are obscured by other components in front of them. One of the simpler algorithms is the depth-sort method, in which the polygons that make up the object are sorted according to the distance of their centroids from the observer, and then rendered in a back-to-front order. This simple algorithm works well so long as polygons do not intersect, in which case it will render incorrectly, and computational costs increase rapidly with the number of polygons comprising the object. When used in conjunction with extent checking, in which two polygons are only compared if they overlap in  $x$  and  $y$ , the depth-sort algorithm is useful when rendering objects composed of few polygons.

Much work has also been done with the scan-line methods, in which an image is rendered sequentially in horizontal lines, like a television image is drawn. Scan-line methods are considerably more complicated than either the depth sort or the  $z$ -buffer algorithms, but can generate realistic images, and in competitive times for objects of medium complexity [8].

The basis of the  $z$ -buffer technique is to store in memory the  $z$  value of *every* pixel in the image. The  $z$ -buffer is initialized with the lowest  $z$  value that can possibly appear. When a polygon, or other primitive, is rendered, it is only drawn at those pixels whose



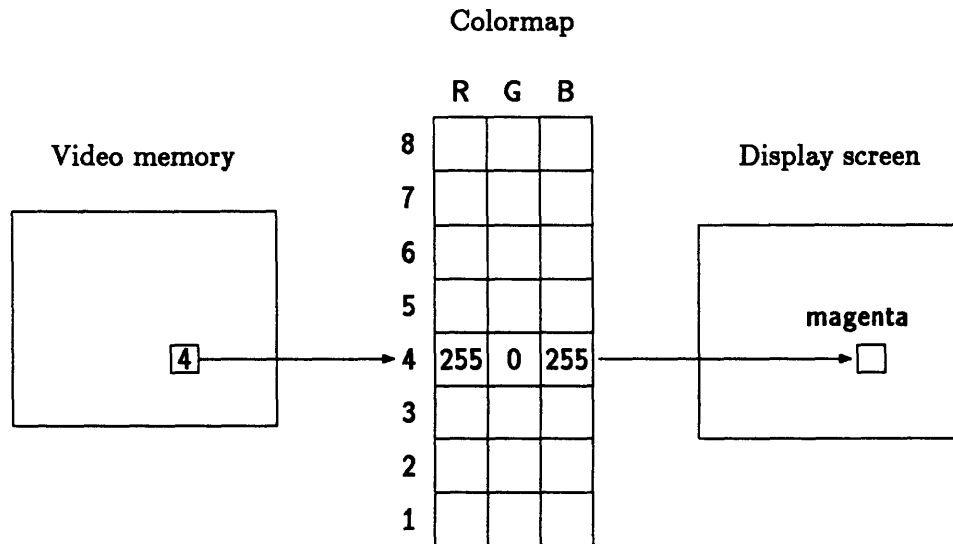


Figure 2.7: Pixel value to RGB mapping with the colormap on a color display.

$z$  values exceed those already stored in the  $z$ -buffer (see figure 2.6). Thus, a comparison must be made for every pixel in the image. Also, a  $z$  value must be calculated at pixels in the interior of polygons. Hardware or software designed to perform Gouraud interpolation of triangles (see section 2.1.2) to determine pixel value (color) can also be used to calculate integer  $z$  values quickly. The computational resources needed are dependent primarily on the size of the image, so  $z$ -buffering is most efficient when rendering large objects.

### 2.3 Colormaps

Most color display hardware available today is based on the RGB color model, in which the color of a pixel on the screen is determined by the intensities of red, green and blue phosphors that make up the pixel. When drawing an object, rarely are the RGB values directly specified, but a *pixel value*, which is an index into a lookup table called the colormap, which contains the RGB values that appear on the screen (see figure 2.7), is used instead.

Each group of three intensities in a colormap is called a *colorcell*. The RGB inten-

Colormap type	read/write	read-only
Gray shade	gray scale	static gray
Single index for RGB	pseudocolor	static color
Decomposed index for RGB	direct color	true color

Table 2.1: Comparison of Colormap types

sities tend to be represented as integers, as opposed to floating-point numbers. The number of bits available for each colorcell entry determines the number of colors that are possible on a certain graphics system. Many systems have eight bits available for each primary, allowing  $(2^8)^3$ , or about 16 million, different colors. In such a case, the RGB values would range from 0 to 255.

The number of colorcells available on a certain display, which is equal to the number of different colors that can be displayed *at the same time*, is determined by the number of bits available for the *pixel value*. This number is often referred to as the *depth* of the colormap, or the number of *bit planes* that the colormap contains. For example, an eight-plane system would allow  $2^8$ , or 256, different colors at the same time, whereas a 24-plane system would allow  $2^{24}$ , or about 16 million, distinct colors simultaneously (assuming the display has that many distinct pixels, which is unlikely).

### 2.3.1 Types of Colormaps

There are six types of colormaps, which depend upon two factors: first, whether the colormap is read/write or read-only, and second, the method by which the pixel value is translated into an actual color [23]. Table 2.1 summarizes the six types. There is a distinct hierarchy among these six types, which figure 2.8 shows, whereby any colormap can imitate a colormap that is below itself in the hierarchy. We will not concern ourselves here with the two gray shade colormaps.

The two most popular types of colormap are pseudocolor and true color. Most graphics systems do not support direct color colormaps, and they often provide little

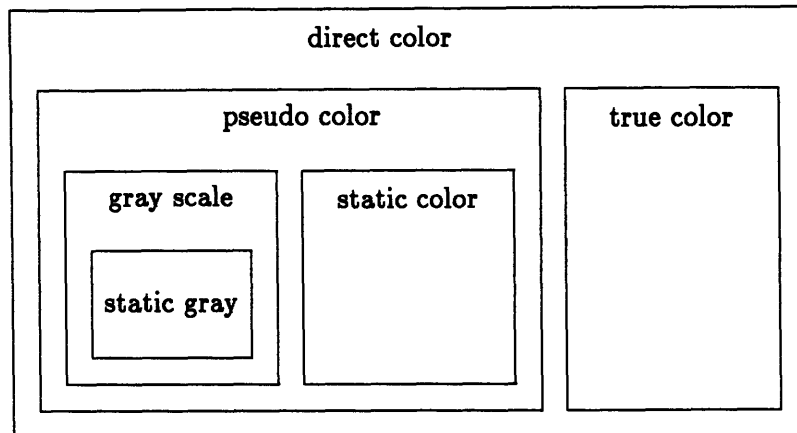


Figure 2.8: Hierarchy of Colormap types.

additional benefit over true color. Static color colormaps are read-only, and cannot be modified once they are created. They have no advantages over pseudocolor, and are rarely used.

The colormap shown in figure 2.7 is a pseudocolor colormap. This means that the pixel value is used directly as the index into the colortable. A pixel with a value of four will have the red intensity of the fourth entry, the green intensity of the fourth entry, and the blue intensity of the fourth entry. This means that with  $n$  colormap entries, one can represent  $n$  colors. However, the colormap entries can be changed at any time, and all pixels on the screen will change colors to reflect the new colormap entries.

When a true color colormap is in use, the pixel value is decomposed into three separate indices which individually point to the separate red, green and blue colormap entries. Figure 2.9 shows this schematically. The decomposition is effected as follows. Consider a 24-plane true color colormap. When represented in hexadecimal notation, the pixel values will have six digits. The index to the red intensity is the first two digits of that representation, the green index is the middle two digits, and the blue index is the last two digits. Typically the value of the intensity entries will be the same as that of the index to that entry itself. For example, a pixel with a value of 9 662 683, or 9370db in hexadecimal, will have the red intensity of the 147th index, the green

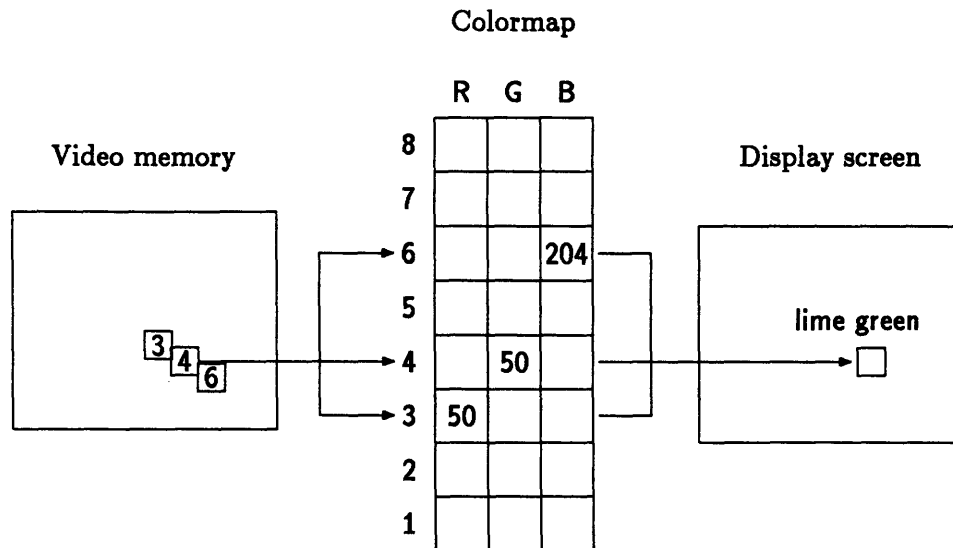


Figure 2.9: Pixel value to RGB mapping for direct color and true color colormaps.

intensity of the 112th index, and a blue intensity of the 219th index. With an intensity range of 0 to 255, which is common, the pixel will be colored aquamarine. With  $n$  colormap entries,  $n^3$  colors can be represented. True color balances the ease of selecting a colormap entry when the RGB intensities are known against the inability to change the colors of pixels that are already drawn.

The choice between pseudocolor and true color depends upon the color requirements of an application. If one requires a wide range of colors that cannot be predicted in advance, as is the case when doing solid modeling, true color is best. Some graphics systems perform lighting calculations in hardware, taking advantage of the setup of the standard true color colormap. The main use of color in our application, however, is to represent the values of a scalar field. We want to select our colors from a predetermined, orderly sequence of colors, each of which represents a different value of the scalar. In such a case, pseudocolor is a better choice. We can linearly convert the scalar value directly to the appropriate pixel value.

If we wished to use true color for the same application, we would need another level of indirection, in the form of an array listing the pixel values that correspond to each scalar value. Also, the use of pseudocolor guarantees that all pixel values will represent a scalar value, whereas many of the pixel values of a true color colormap would not.

This can cause problems when Gouraud shading of the pixel values (see section 2.1.2) is performed.

## 2.4 Transformation Matrices

Each point in a space can be mapped onto another point in the same space by a transformation, which need not preserve any convenient geometric properties, such as length, angular measure, or the definition of a straight line. The transformations that are of interest here are those that map points from one right-handed orthogonal coordinate system to another such system. In particular, we would like to transform points in the *world*, or *object*, or *modeling* coordinate system, in which the geometry of the object we would like to display is given, into the *display*, or *viewing*, or *screen* coordinate system, in which the object is drawn on the display screen. Any such transformation will be linear, and can be represented (in  $n$  dimensional space) as an  $n + 1$  by  $n + 1$  element matrix. An arbitrary transformation can be expressed as a sequence of elementary transformations of three types, namely translation, dilation, and rotation.

For the sake of convenience and simplicity, we will consider two dimensional transformations in detail, and then make a sweeping generalization to include higher dimensions.

### Caution!

Note that many references in computer graphics literature use row vectors to represent points and represent transformations by right multiplying by the transformation matrix, whereas we use column vectors to represent points and represent transformations by left multiplying by the transformation matrix. We differ from the literature because the use of column vectors is standard in mathematics and other branches of computer science. In the notation used in the literature, all matrices will appear transposed with respect to the notation we use here [8,24].

### 2.4.1 Translation

If point  $P$  has coordinates  $(x, y)$  in the original coordinate frame, then it is translated to a new point  $P'$  with coordinates  $(x', y')$  by moving it a distance  $t_x$  parallel to the  $x$ -axis and an amount  $t_y$  parallel to the  $y$ -axis. We can write

$$\begin{aligned}x' &= x + t_x \\y' &= y + t_y\end{aligned}\tag{2.3}$$

which can be expressed in vector form as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

or, if we define the column vectors

$$\mathbf{P} = \begin{bmatrix} x \\ y \end{bmatrix}, \mathbf{P}' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

this transformation can be expressed most concisely as

$$\mathbf{P}' = \mathbf{P} + \mathbf{T}.$$

An object, which is made up of a collection of points, can be translated by applying equations 2.3 to each point that composes the object. This process is greatly simplified by noting that translating a line segment can be accomplished by translating only the endpoints and defining the new line segment to have those endpoints. This property of linearity holds for dilational and rotational transformations, as well.

### 2.4.2 Dilation

Dilation, or scaling, of a point is dependent on the location of the origin of coordinates. A point can be scaled by a factor  $s_x$  in the  $x$  direction and by a factor  $s_y$  in the  $y$  direction by the relations

$$\begin{aligned}x' &= s_x x \\y' &= s_y y.\end{aligned}\tag{2.4}$$

Now, defining the scaling matrix

$$\mathbf{S} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix},$$

we can write the scaling transformation as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix},$$

or,

$$\mathbf{P}' = \mathbf{S}\mathbf{P}.$$

As with translation, an object can be scaled by applying equations 2.4 to each point of which the object is composed.

If the two scaling factors  $s_x$  and  $s_y$  are equal, the scaling is *uniform*, and the proportions of the object are retained, and any angles are unchanged. If  $s_x$  and  $s_y$  are unequal, the scaling is said to be *differential*. We mainly use uniform scaling, but the possibility of differential scaling is not be excluded.

### 2.4.3 Rotation

As with dilation, rotation can only be performed with respect to the origin of coordinates. Rotation is the most complex of the elementary transformations, and is the

reason for additional complexity in higher dimensions.

A point is rotated about the coordinate origin through an angle  $\theta$ , considered to be positive in the counterclockwise direction, by

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta.\end{aligned}\tag{2.5}$$

This can be written in matrix form as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix},$$

or, with the definition of the rotation matrix

$$\mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix},$$

as

$$\mathbf{P}' = \mathbf{R}\mathbf{P}.$$

Once again, an object is rotated by applying equations 2.5 to each point in the object.

#### 2.4.4 Homogeneous Coordinates

A more complex transformation can be represented as a composition of elementary transformations, for example the general transformation

$$\begin{aligned}x' &= s_x x \cos \theta - s_y y \sin \theta + t_x \\y' &= s_x x \sin \theta + s_y y \cos \theta + t_y\end{aligned}\tag{2.6}$$

can be written as

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$



or

$$\mathbf{P}' = \mathbf{RSP} + \mathbf{T},$$

and represents a dilation, followed by a rotation, followed by a translation. Note that while rotation and dilation are represented as matrix multiplication, translation is represented as vector addition. We would like to treat the three transformations in a consistent or homogeneous way, so that all transformations can be composed by matrix multiplications of elementary transformations. To do this, we introduce the homogeneous coordinate.

In the homogeneous coordinate system, the point  $(x, y)$  in two dimensional space is represented by the point  $(X, Y, W)$  in three dimensional space, such that  $X = Wx$  and  $Y = Wy$ , and likewise  $x = X/W$  and  $y = Y/W$ . In practice, the homogeneous coordinate  $W$  will almost always be equal to unity, so that  $X = x$  and  $Y = y$ . We must also redefine the column vectors

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \mathbf{P}' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}.$$

This may seem like a strange thing to do, but now our translation relations

$$x' = x + t_x$$

$$y' = y + t_y$$

can be expressed as a matrix multiplication,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

or, with the expected definition

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix},$$

as

$$\mathbf{P}' = \mathbf{TP}.$$

It is important to note that the homogeneous coordinate  $W$  has no intrinsic physical meaning, but is a mathematical convenience to allow arbitrary transformations to be expressed as matrix multiplications.

It is necessary to make small changes to the scaling and rotation matrices to take account of the homogeneous coordinate, namely

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{R} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

With these redefinitions, any transformation can be expressed as a matrix multiplication

$$\mathbf{P}' = \mathbf{MP},$$

where  $\mathbf{M}$  is the product of the appropriate sequence of elementary transformation matrices. When transforming from the object coordinates to the viewing coordinates, the matrix  $\mathbf{M}$  is often called the *object*, or *viewing matrix*.

#### 2.4.5 Three Dimensional Transformations

Translation and dilation are trivially extended to three dimensions. The three dimensional translation

$$x' = x + t_x$$

$$y' = y + t_y$$

$$z' = z + t_z$$

(2.7)

can be expressed as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix},$$

or, as before,

$$\mathbf{P}' = \mathbf{TP}.$$

Likewise, scaling can be written as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix},$$

or

$$\mathbf{P}' = \mathbf{SP}.$$

Rotation, however, is more complicated. In three dimensions, rotation can be defined in many different ways, such as a rotation about an arbitrary axis, or as a sequence of rotations about the coordinate axes, or as the set of unit vectors that define the new coordinate frame. We will primarily express three dimensional rotation as a sequence of rotations about the coordinate axes, although the unit vector representation will also be used.

The full three dimensional rotation matrix,

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

is the product of the individual axial rotation matrices  $\mathbf{R}_x$ ,  $\mathbf{R}_y$  and  $\mathbf{R}_z$  that represent to the sequence of pure axial rotations of that were performed to yield the entire trans-

formation. The meaning of the entries  $r_{ij}$  will be discussed in section 2.4.6. The axial rotation matrices are described here.

Rotation about the  $z$  axis is equivalent to the two dimensional rotation discussed in section 2.4.3. The  $z$ -rotation matrix is

$$\mathbf{R}_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

in which the angle  $\theta$  is positive counterclockwise when viewed in the negative  $z$  direction, that is, from the  $x$ -axis to the  $y$ -axis. Similarly, the  $y$ -rotation matrix is

$$\mathbf{R}_y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi & 0 \\ 0 & \sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

in which the angle  $\phi$  is positive in the direction from the  $z$ -axis to the  $x$ -axis, and the  $x$ -rotation matrix is

$$\mathbf{R}_x = \begin{bmatrix} \cos \psi & 0 & \sin \psi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \psi & 0 & \cos \psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

in which the angle  $\psi$  is positive in the direction from the  $y$ -axis to the  $z$ -axis.

## 2.4.6 Unit Vector Representation

The full three dimensional rotation matrix,

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

is orthogonal, that is, any two of its columns (or rows), when treated as vectors, are perpendicular. In addition, all rows and columns have unit magnitude. The inverse and the transpose of an orthogonal matrix are the same, so that  $\mathbf{R}^T\mathbf{R} = \mathbf{R}\mathbf{R}^T = \mathbf{I}$ . The inverse of a transformation matrix clearly is the matrix representation of the inverse transformation, that is, if

$$\mathbf{P}' = \mathbf{R}\mathbf{P},$$

then

$$\mathbf{P} = \mathbf{R}^{-1}\mathbf{P}' = \mathbf{R}^T\mathbf{P}'$$

which allows us to multiply each unit vector of the primed coordinate system by  $\mathbf{R}^T$  to find its value in the original, unprimed frame. That is,

$$\mathbf{i}_{x'} = \mathbf{R}^T \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{i}_{y'} = \mathbf{R}^T \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \mathbf{i}_{z'} = \mathbf{R}^T \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

or

$$\mathbf{i}_{x'} = \begin{bmatrix} r_{11} \\ r_{12} \\ r_{13} \end{bmatrix}, \quad \mathbf{i}_{y'} = \begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \end{bmatrix}, \quad \mathbf{i}_{z'} = \begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \end{bmatrix}.$$

Thus, when one knows the unit vectors of the primed frame with respect to the unprimed frame, it is simple to construct the rotation matrix  $\mathbf{R}$  by setting the rows of  $\mathbf{R}$  equal to the components of the unit vectors. This method will be used in section 5.3 to display a perpendicular view of an arbitrary plane in space.

#### 2.4.7 Perspective Projection

Among the geometric transformations that can be modeled as matrices are perspective viewing projections. A perspective projection is characterized by the number of its principal vanishing points, which are the points at which lines parallel to the coordinate axes intersect, as figure 2.10 shows. We use one vanishing point, situated at the center of the viewing window, *i. e.*, at  $x = y = 0, z = \infty$ . Consider a projection with a

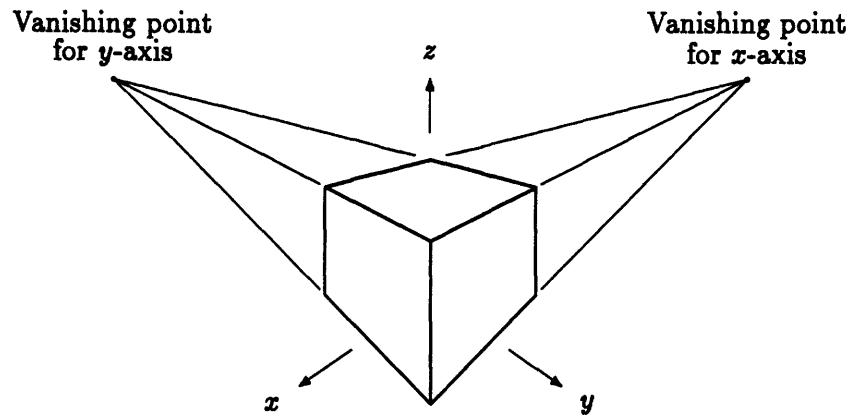


Figure 2.10: A projection with two principal vanishing points.

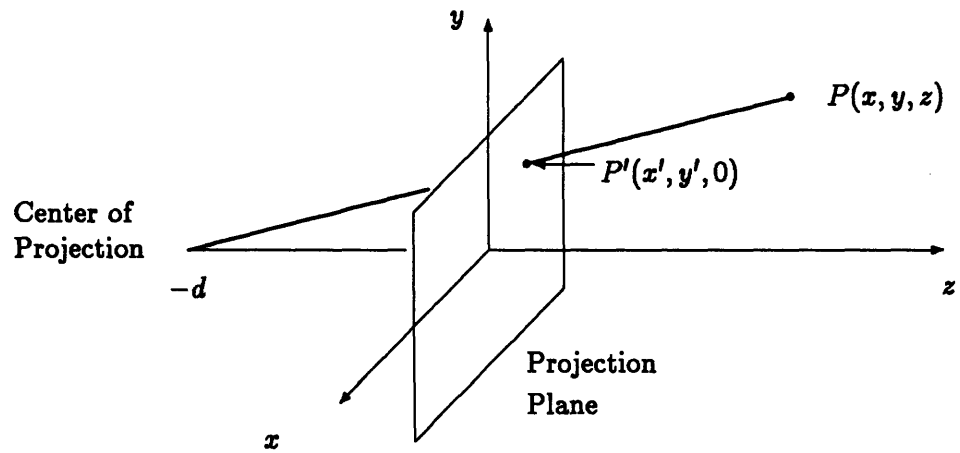


Figure 2.11: Perspective projection.

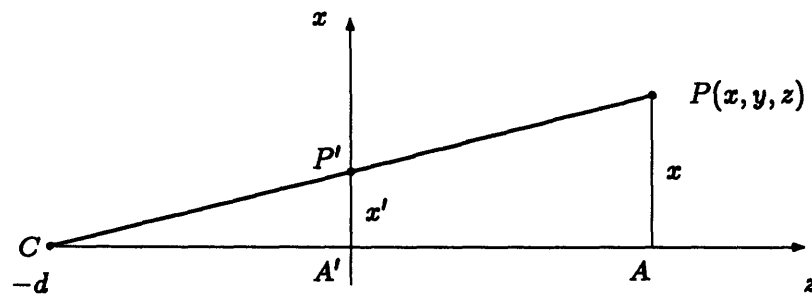


Figure 2.12: Similar triangles in perspective projection.  $\Delta P'CA' \sim \Delta PCA$ .

projection plane  $z = 0$ , and the location of the observer, or the *center of projection*, along the  $z$  axis a distance  $d$  behind the plane, as shown in figure 2.11. By use of similar triangles (see figure 2.12), we find the relations

$$\frac{x'}{d} = \frac{x}{z+d}$$

$$\frac{y'}{d} = \frac{y}{z+d}$$

so that the coordinates of the projection of point  $P$  onto the plane of projection are

$$x' = \frac{x}{z/d + 1}$$

$$y' = \frac{y}{z/d + 1}. \quad (2.8)$$

This can be represented by the perspective matrix

$$\mathbf{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}.$$

To see this, multiply the coordinate vector of the point  $\mathbf{P}$  by the matrix  $\mathbf{M}_p$  to find the homogeneous point, as follows:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d + 1 \end{bmatrix}.$$

Note that this is one of the rare scenarios in which the homogeneous coordinate  $W$  is not unity. Now, homogenize to find the coordinate vector of the projected point,  $\mathbf{P}'$ :

$$\mathbf{P}' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \frac{1}{W} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \frac{1}{z/d + 1} \begin{bmatrix} x \\ y \\ z \\ z/d + 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d + 1} \\ \frac{y}{z/d + 1} \\ \frac{z}{z/d + 1} \\ 1 \end{bmatrix}.$$

Note the value of  $z'$  that results. The actual value is quite irrelevant, but it is important that  $z'$  be monotonically increasing with  $z$ , so that hidden surface removal will be accurate.

It is convenient and useful to define the *perspective factor* as

$$\alpha = \frac{1}{d}$$

so that the projection equations are

$$\begin{aligned}x' &= \frac{x}{\alpha z + 1} \\y' &= \frac{y}{\alpha z + 1},\end{aligned}\tag{2.9}$$

and the perspective matrix is

$$M_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \alpha & 1 \end{bmatrix}.$$

The advantage of this representation is that when the perspective factor vanishes, the projection equations (2.9) reduce to

$$\begin{aligned}x' &= x \\y' &= y\end{aligned}$$

and the effects of perspective are eliminated. Such a projection is called *orthographic*. This effect is achieved with the original formulation by setting the distance to the projection plane equal to infinity, which is, in practice, unwieldy at best.

Since perspective projection is highly dependent upon the orientation of the observer with respect to the object, perspective projection should only be performed after all other geometric transformations.



## Chapter 3

# The Graphics Supercomputer

The graphics supercomputer is a system that combines the computational prowess of the minisupercomputer with the graphics capabilities of advanced imaging systems. The class is an outgrowth of the rapidly increasing capabilities of single-user graphics workstations with the decreasing cost of high-performance processors originally developed for supercomputers. The recent emergence of products in the field is a result of progress in very large scale integration (VLSI) techniques, that allow a computer manufacturer to create application-specific integrated circuits (ASICs). The two systems initially available, the Stellar GS-1000 and the Ardent TITAN, are nearly identical in stated graphics and computational performance.

### 3.1 Stellar Hardware

The architecture of the GS-1000 differs considerably from that of previous systems. It consists of several major components each of which is connected to a central Data Path (DP), as shown in figure 3.1. The central processing unit is composed of a Multi-Stream Processor (MSP) and a Vector Floating-point Processor (VFP). The MSP is capable of executing four instruction streams simultaneously, with three-stage pipeline capability in each. It thus simulates a four processor concurrent architecture, and can be reconfigured to run as a two-processor or one-processor system as well. It has a fifty nanosecond clock speed, and can achieve between twenty and twenty-five million instructions per second (MIPS).

The VFP is directly connected to the Rendering Processor (RP), which oversees the actual graphics process. The VFP performs all floating-point geometric transfor-

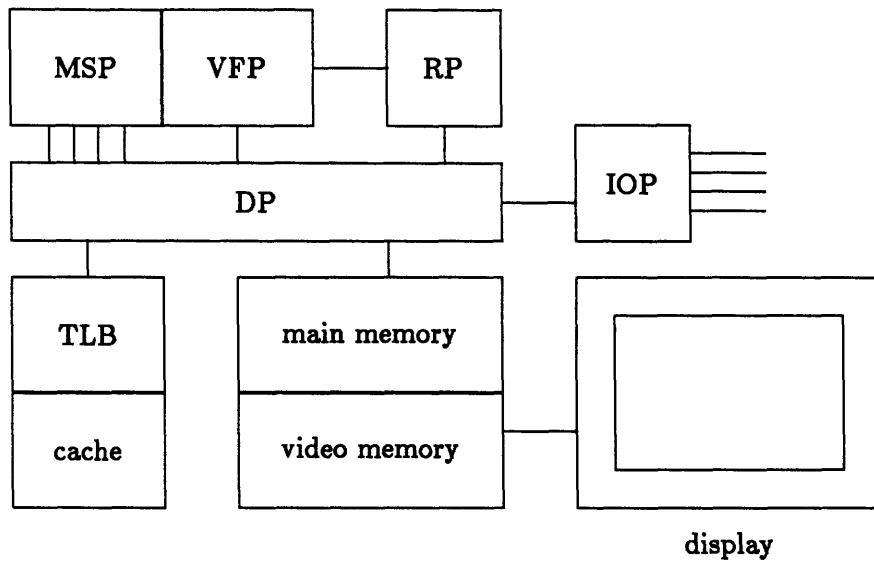


Figure 3.1: Stellar GS-1000 hardware architecture.

mations (see section 2.4). The core of the RP is a “footprint processor,” a four by four array of individual processors called “toes.” During the rendering process, the footprint processor “walks” across the image, performing  $z$ -buffer and pixel value generation for a four by four array of pixels. The RP performs at about twenty MIPS. The graphics hardware is capable of performing geometric transformations on 800,000 points per second, and can render 600,000 short vectors (10 pixels or less in length) and 150,000 small  $z$ -buffered Goraud-shaded polygons (100 pixels or fewer in area) per second.

The GS-1000 can be configured to have from 32 to 128 megabytes (MB) of main memory, and has a 1 MB cache memory (TLB is the Translation Look-aside Buffer, which maps virtual address space in the cache to physical address space). The video memory is connected to the main memory by a high-speed link. The display screen contains 1024 by 1280 pixels, and has a refresh rate of 74 Hz. Traditional input and output functions are handled by the Input/Output Processor (IOP).

## 3.2 Stellar Software

There are three low-level graphics packages available on the GS-1000. At the highest level is the Stellar extensions to the Programmers' Hierarchical Interactive Graphics System (PHIGS+), which is an object-oriented graphics system intended to be used for solid modeling [5]. PHIGS+ subroutines can be called by programs written in both FORTRAN and C, and call additional low-level routines that perform the actual rendering. These low-level routines include the X Window System library [23], an industry standard. Xlib operates exclusively upon integer arguments, and is intended to be called by programs written in C. It allows graphics functions to be performed on the screens of remote hosts via Ethernet. The high-performance rendering hardware is accessed via the X Floating-point Device Interface library [1], or XFDI. XFDI is Stellar-specific and is not portable. It is intended to be callable by both FORTRAN and C, but the FORTRAN interface has not been released.

We decided to use Xlib and XFDI in favor of PHIGS+. The primary factor affecting our decision was that PHIGS+ allows a program to render in only one window at the same time. We desired to have two main graphics windows in use. We use Xlib for the creation and management of windows, the maintenance of colormaps (see section 2.3), the drawing of text and simple graphics, and interaction with the mouse cursor and keyboard [23]. We use XFDI for high-performance graphics needs, including the actual rendering, the maintenance of transformation matrices (see section 2.4), and for hidden-line and hidden-surface removal via  $z$ -buffering (see section 2.2.2).

## Chapter 4

# Data Structures

The representation of unstructured three dimensional data is an open problem. The most general data structure is a three-tiered representation in which the mesh is composed of a group of cells, which are in turn defined by an arbitrary number of polygonal faces, which are composed of a number of edges, which are at last defined by the two nodes that are their endpoints. This is an incredible amount of data. Consider a large structured rectangular mesh as an example. If the mesh contains  $N_C$  cells, it will contain about  $N_C$  nodes, and about  $3N_C$  faces and edges. Each cell has six faces and each face has four edges, yielding a total of  $24N_C$  words of data storage to define the mesh topology. We choose to forego the middle two levels of indirection and restrict ourselves to cells of a fixed topology, defined by the mesh nodes that are their vertices.

### 4.1 Cell Geometry

Our meshes are composed entirely of hexahedral cells. The basic strategies described herein are completely extendible to meshes composed of tetrahedral cells, which represents a future development. Also, cells in the shapes of tetrahedra, pyramids, and triangular prisms can be represented as degenerate hexahedra, by assigning several nodes the same location.

Each cell is completely defined by the eight nodes that are its vertices, as shown in figure 4.1. Within each cell a set of computational coordinates  $(\xi, \eta, \zeta)$  is defined, which take values of  $\pm 1$  at the corners. Thus, the cell edges have a length of two, and the cells have a volume of eight.

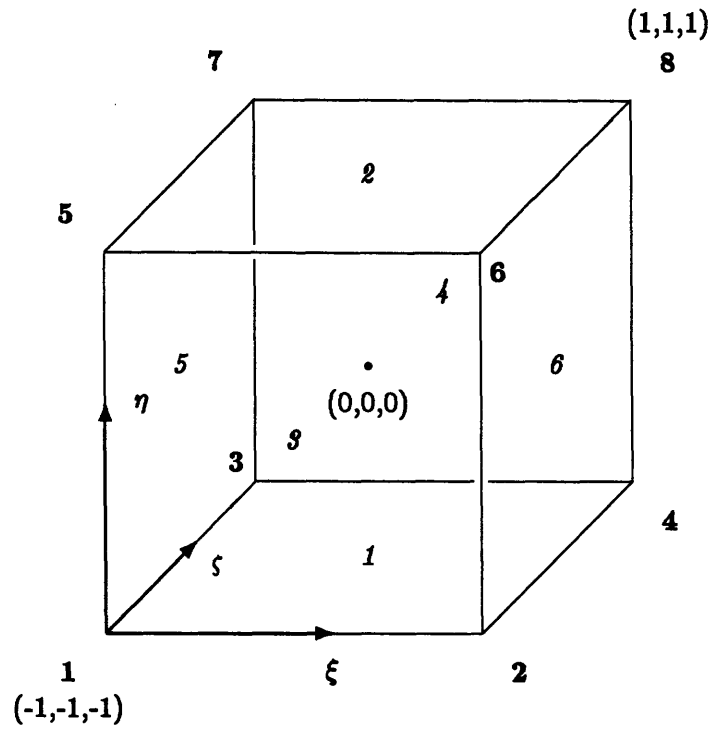


Figure 4.1: Cell geometry. Nodes are numbered in bold. Faces are numbered in *italic*.

face number	nodes on face
<b>1</b>	<b>1 2 4 3</b>
<b>2</b>	<b>5 7 8 6</b>
<b>3</b>	<b>1 5 6 2</b>
<b>4</b>	<b>3 4 8 7</b>
<b>5</b>	<b>1 3 7 5</b>
<b>6</b>	<b>2 6 8 4</b>

Table 4.1: Face node definitions.

The cell faces are not part of our mesh data structure, but knowledge of them is required to perform particle path integration (see section 5.2.5). Since our cells have a fixed geometry, information about the faces can easily be determined from the list of nodes that define the cell. Table 4.1 lists the faces of the cell shown in figure 4.1. The nodes of a face are given in a counterclockwise order when viewed from within the cell.

## 4.2 Fundamental Quantities

### 4.2.1 Mesh Geometry

The complete geometry of the mesh is stored as a list of coordinates of the nodes, and a list of the eight nodes which define each cell. The boundaries of the mesh, which are needed to produce surface plots, as explained in section 5.1, are stored as a list of quadrilaterals. The boundaries are divided into groups of polygons, as specified by the user. As an example, the far-field boundary could be group 1, the body surface group 2, the inflow boundary group 3, and the outflow boundary group 4.

### 4.2.2 Flow Quantities

Flow data is stored at the nodes as a five element state vector of conserved quantities,

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{bmatrix},$$

where  $\rho$  is the fluid density,  $u$ ,  $v$  and  $w$  are the Cartesian components of velocity, and  $E$  is the total internal energy per unit mass. Also stored at the nodes are a scalar quantity and a vector quantity. The scalar quantity is used for all plotting functions, and can be chosen from the following:

**Density:**  $\rho = U_1$

**x velocity:**  $u = \frac{U_2}{U_1}$

**y velocity:**  $v = \frac{U_3}{U_1}$

**z velocity:**  $w = \frac{U_4}{U_1}$

**Total internal energy:**  $E = \frac{U_5}{U_1}$

**Velocity:**  $q = \sqrt{u^2 + v^2 + w^2}$

**Pressure:**  $p = (\gamma - 1) \left( U_5 - \frac{1}{2} \rho q^2 \right)$

**Speed of sound:**  $c = \sqrt{\frac{\gamma p}{\rho}}$

**Mach number:**  $M = \frac{q}{c}$

**Total pressure:**  $p_0 = p \left( 1 + \frac{\gamma-1}{2} M^2 \right)^{\frac{\gamma}{\gamma-1}}$

**Total pressure loss:**  $\Delta p_0 = 1 - \frac{p_0}{p_{0\text{ref}}}$

The fluid is assumed to be an ideal gas, with a constant ratio of specific heats of  $\gamma = 1.4$ .

The vector quantity is used for particle path integration, which is detailed in section 5.2, and is presently restricted to be either velocity or momentum density.

The composition of the state vector and the choice of scalar or vector quantities are not rigid, and can readily be modified if an investigator wishes to display a different type of data. For example, three extra elements could be added to represent fluid vorticity, or an entirely different state vector could be used to display the results of a structural simulation.

### 4.3 Additional Quantities

In addition to the data described in section 4.2, we need many other physical quantities and pointer lists to perform our algorithms. These include the coordinates of the

centroids of each cell, which are used to determine the cell in which to start particle path integration. The quantity  $z'$ , which is critical in the process of planar interpolation (see section 5.3.1) is calculated and stored at each node and cell centroid. The minimum and maximum value of  $z'$  in each cell is also stored.

We also calculate the surface normal vectors at each node on the mesh boundary, to aid the future inclusion of a lighting model for the mesh surface.

Several additional pointer lists are created, based on the cell-to-node and the boundary polygon-to-node lists described in section 4.2.1. In order to create the polytriangle surface strips needed to produce surface plots (see sections 2.1.3 and 5.1) we create a node-to-boundary polygon pointer list, which allows us to determine in which polygons a node is contained.

Similarly, particle path calculations require a cell-to-cell pointer list, which is consulted when a trajectory leaves a cell. This list is created in two stages. First, we create a node-to-cell pointer list, and then use that list to compare each cell to all other cells that also contain its first and last nodes. Examination of figure 4.1 shows that every face of a cell contains either node 1 or node 8 of that cell, and thus any adjacent cell must contain one of those nodes as well. Adjacent cells are those that have four nodes in common. Table 4.1 is used to determine which face it is that the two cells share.

## 4.4 Memory Usage

The memory that we use is dominated by quantities stored at the cells and nodes, since the amount of nodes and cells will be far greater than the amounts of any other group of objects, such as boundary polygons, where data is stored. For each cell, eight words are required to indicate the nodes that are its corners. The cell-to-cell pointer list described in section 4.3 occupies six words per cell. The two pointer lists,  $i'_{\min}$  and  $i'_{\max}$ , which are described in section 5.3.2, account for two words per cell, as do  $z'_{\min}$  and  $z'_{\max}$ , the minimum and maximum values of  $z'$  in each cell. One word of storage is



required for the value of  $z'$  at the cell centroid, and the coordinates of the centroid take up three words. The total memory usage is 22 words per cell.

The quantities stored at the nodes are as follows. Three words are required for the spatial coordinates of the nodes. The value of  $z'$  at the nodes takes up one word per node, while the state vector, the scalar plotting quantity, and the vector plotting quantity (see section 4.2.2) account for five, one, and three words per nodes, respectively. The pointer list that identifies the cells which contain a node occupies seventeen words per vertex (just to be safe, since in a general unstructured grid it is not known from the shape of the cells alone how many cells will contain a given node). The total is 30 words of storage per node.

For a hexahedral mesh such as we use, the number of cells is roughly the same as the number of vertices. Thus, the total memory usage will be approximately  $52N_V$  words, where  $N_V$  is the number of nodes. With the standard 32 megabytes of memory for the GS-1000, this would limit us to meshes of 150 000 or fewer cells. Our system, equipped with 96 MB, can contain meshes of up to 450 000 nodes. It is possible to decrease memory requirements by dynamically allocating storage for some arrays, but this has not yet been implemented. In particular, if the space taken up by the node-to-cell pointer list can be re-used, we can handle solutions with up to 700 000 nodes.

A mesh entirely composed of tetrahedra would differ in that the cell-to-node pointer array would use four words per cell, as would the cell-to-cell pointer list. This would decrease the memory storage needed per cell, but such a mesh would contain about five times as many cells as nodes. A tetrahedral mesh would require about  $110N_V$  words of storage, limiting us to meshes of 70 000 or fewer nodes when 32 MB are available, and to 215 000 or fewer nodes on our system with 96 MB. The type of dynamic memory allocation discussed above would increase the capacity to 250 000 nodes.

## Chapter 5

# Visualization Methods

### 5.1 Surface Plot

Surface plotting is the simplest visualization method that we use. A scalar quantity, as chosen by the user, is displayed in color shading on a subset of the surfaces of the computational mesh, which are also defined by the user. Surface plots are the simplest three dimensional visualization method because the geometry is directly available from the mesh and that geometry is static. Neither of these is true of the other visualization methods we use. Also, surface plotting is useful for comparison with experiment and/or analysis of surface loading. For these reasons it has previously been the most widely used three dimensional visualization method.

However, surface plotting provides no information about the interior of the flow field. This is satisfactory for potential or nearly potential flows, in which the most interesting phenomena will be located at the boundaries,<sup>1</sup> but is inadequate for rotational flows.

As mentioned in section 4.2.1, the boundary is composed of a collection of quadrilaterals, which are divided up into groups of possible physical significance. These groups are defined by the user as input, who can select which groups are to be displayed, and on which groups a scalar variable will be plotted. Refer to section 4.2.2 for a list of flow variables currently available.

The polygons that make up each boundary group are further divided into a number of polytriangle surface strips (see section 2.1.3), which are constructed with the aid of

---

<sup>1</sup>This is a statement of Green's Theorem, which declares that the properties of a potential flow field can be completely defined by a distribution of point sources and doublets on the flow boundary.

a node-to-polygon pointer list which determines, for each node on the boundary, which polygons contain it. The point of view from which the surface is seen is determined interactively by the user, and affects the plotting process through the use of viewing transformation matrices, which are discussed in detail in section 2.4.

## 5.2 Particle Path Integration

One way to visualize the interior of a flow field is to integrate the velocity field to form particle pathlines. Since we are presently concerned only with steady flows, pathlines are equivalent to streamlines and streaklines. Similarly, one can also integrate the vorticity field to generate vortex lines. A suggestion by Levy *et al.* [16] is to integrate in the upstream direction to locate the origin of a vortex. Trajectory calculations are numerically intensive and cannot be done interactively on ordinary graphics workstations.

### 5.2.1 Trajectory Equations

It is possible to directly integrate the trajectory equations

$$\begin{aligned}\frac{dx}{dt} &= u \\ \frac{dy}{dt} &= v \\ \frac{dz}{dt} &= w,\end{aligned}\tag{5.1}$$

but this leaves one with the problem of how to detect when a path crosses from one cell to another. Also, previous research has found [25] that accuracy limitations require about five numerical integration steps to be performed within each cell. We therefore perform trajectory integration in computational space, in which the trajectory equations are

$$\begin{aligned}
\frac{d\xi}{dt} &= U \\
\frac{d\eta}{dt} &= V \\
\frac{d\zeta}{dt} &= W,
\end{aligned} \tag{5.2}$$

where  $\xi$ ,  $\eta$  and  $\zeta$  are the computational coordinates, as depicted in figure 5.1, and  $U$ ,  $V$  and  $W$  are the contravariant velocities

$$\begin{aligned}
U &= \xi_x u + \xi_y v + \xi_z w \\
V &= \eta_x u + \eta_y v + \eta_z w \\
W &= \zeta_x u + \zeta_y v + \zeta_z w,
\end{aligned} \tag{5.3}$$

and are the components of the velocity resolved in the directions of the computational coordinates  $\xi$ ,  $\eta$  and  $\zeta$  [2].

### 5.2.2 Numerical Integration Scheme

The predictor-corrector integration scheme that we use is

$$\begin{aligned}
\xi^* &= \xi_n + U(\xi_n, \eta_n, \zeta_n) \Delta t \\
\eta^* &= \eta_n + V(\xi_n, \eta_n, \zeta_n) \Delta t \\
\zeta^* &= \zeta_n + W(\xi_n, \eta_n, \zeta_n) \Delta t \\
\xi_{n+1} &= \xi_n + \frac{1}{2} [U(\xi^*, \eta^*, \zeta^*) + U(\xi_n, \eta_n, \zeta_n)] \Delta t \\
\eta_{n+1} &= \eta_n + \frac{1}{2} [V(\xi^*, \eta^*, \zeta^*) + V(\xi_n, \eta_n, \zeta_n)] \Delta t \\
\zeta_{n+1} &= \zeta_n + \frac{1}{2} [W(\xi^*, \eta^*, \zeta^*) + W(\xi_n, \eta_n, \zeta_n)] \Delta t
\end{aligned} \tag{5.4}$$

where the time step  $\Delta t$  is chosen so that  $\bar{U} \Delta t = \frac{1}{6}$ , where  $\bar{U}$  is a characteristic value of the contravariant velocity for the entire cell, and is approximated by magnitude of the vector average of the contravariant velocities at the vertices of the cell.

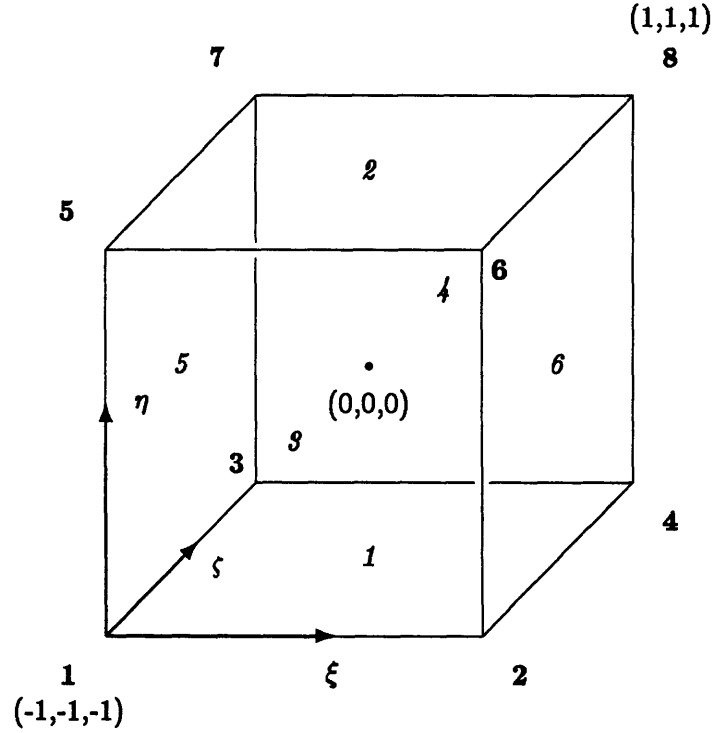


Figure 5.1: Cell geometry. Nodes are numbered in bold. Faces are numbered in *italic*.

### 5.2.3 Trilinear Interpolation

The cell geometry was first discussed in section 4.1. Figure 5.1 depicts a cell in computational space, with the node and face numbering shown. The computational coordinates take a value of  $\pm 1$  at the nodes. This geometrical representation is based on finite element formulations [28]. The geometry of the cell is determined by trilinear interpolation between the nodes. In addition, it is necessary to interpolate the contravariant velocities within the cell in the same manner. With the node numbering shown in figure 5.1, the trilinear interpolation functions

$$T_1 = \frac{1}{8}(1 - \xi)(1 - \eta)(1 - \zeta)$$

$$T_2 = \frac{1}{8}(1 + \xi)(1 - \eta)(1 - \zeta)$$

$$T_3 = \frac{1}{8}(1 - \xi)(1 + \eta)(1 - \zeta)$$

$$T_4 = \frac{1}{8}(1 + \xi)(1 + \eta)(1 - \zeta)$$

$$\begin{aligned}
T_5 &= \frac{1}{8}(1 - \xi)(1 - \eta)(1 + \zeta) \\
T_6 &= \frac{1}{8}(1 + \xi)(1 - \eta)(1 + \zeta) \\
T_7 &= \frac{1}{8}(1 - \xi)(1 + \eta)(1 + \zeta) \\
T_8 &= \frac{1}{8}(1 + \xi)(1 + \eta)(1 + \zeta)
\end{aligned} \tag{5.5}$$

will yield any quantity  $q$  at a point within the cell as

$$q = q_1T_1 + q_2T_2 + q_3T_3 + q_4T_4 + q_5T_5 + q_6T_6 + q_7T_7 + q_8T_8, \tag{5.6}$$

where  $q_1, \dots, q_8$  are the nodal values of  $q$ . The quantities we interpolate according to this procedure are the physical coordinates  $x$ ,  $y$  and  $z$ , and the contravariant velocities  $U$ ,  $V$  and  $W$ .

Derivatives with respect to the computational coordinates are considered to be constant throughout each cell, and are determined by the formulas

$$\begin{aligned}
q_\xi &= \frac{1}{8}(q_2 + q_4 + q_6 + q_8 - q_1 - q_3 - q_5 - q_7) \\
q_\eta &= \frac{1}{8}(q_3 + q_4 + q_7 + q_8 - q_1 - q_2 - q_5 - q_6) \\
q_\zeta &= \frac{1}{8}(q_5 + q_6 + q_7 + q_8 - q_1 - q_2 - q_3 - q_4),
\end{aligned} \tag{5.7}$$

where  $q$  is any scalar quantity.

#### 5.2.4 Geometry Metrics

The geometry metrics  $x_\xi, x_\eta, \dots, z_\zeta$  are evaluated according to equations 5.7. The metrics  $\xi_x, \xi_y, \dots, \zeta_z$ , which appear in the expressions for the contravariant velocities (equations 5.3), are determined by inverting the transformation matrix

$$\mathbf{J} = \begin{bmatrix} x_\xi & y_\xi & z_\xi \\ x_\eta & y_\eta & z_\eta \\ x_\zeta & y_\zeta & z_\zeta \end{bmatrix}$$

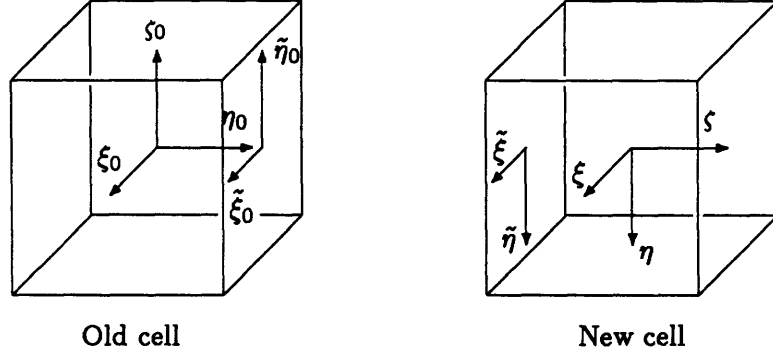


Figure 5.2: Cell-to-cell interface of integration path.

to yield the set of expressions

$$\begin{aligned}
 \xi_x &= \frac{y_\eta z_\zeta - z_\eta y_\zeta}{J} & \eta_x &= \frac{z_\xi y_\zeta - y_\xi z_\zeta}{J} & \zeta_x &= \frac{y_\xi z_\eta - z_\xi y_\eta}{J} \\
 \xi_y &= \frac{z_\eta x_\zeta - x_\eta z_\zeta}{J} & \eta_y &= \frac{x_\xi z_\zeta - z_\xi x_\zeta}{J} & \zeta_y &= \frac{z_\xi x_\eta - x_\xi z_\eta}{J} \\
 \xi_z &= \frac{x_\eta y_\zeta - y_\eta x_\zeta}{J} & \eta_z &= \frac{y_\xi x_\zeta - x_\xi y_\zeta}{J} & \zeta_z &= \frac{x_\xi y_\eta - y_\xi x_\eta}{J}
 \end{aligned}$$

where

$$J = |\mathbf{J}| = x_\xi y_\eta z_\zeta + y_\xi z_\eta x_\zeta + z_\xi x_\eta y_\zeta - z_\xi y_\eta x_\zeta - y_\xi x_\eta z_\zeta - x_\xi z_\eta y_\zeta$$

is the Jacobian of the transformation.

### 5.2.5 Cell-to-Cell Interfaces

With this representation, the trajectory can be seen to leave a cell when the absolute value of one of the computational coordinates exceeds unity. When this occurs, the trajectory is clipped so that the offending coordinate has an absolute value of exactly unity. If more than one coordinate exceeds these bounds, it may be necessary to repeatedly clip until all coordinates fall within the cell.

The face through which the path exits is determined by table 5.1. The next cell through which the trajectory passes is determined from a cell-to-cell pointer list, in-

face number	nodes on face	definition
1	1 2 4 3	$\zeta = -1$
2	5 7 8 6	$\zeta = 1$
3	1 5 6 2	$\eta = -1$
4	3 4 8 7	$\eta = 1$
5	1 3 7 5	$\xi = -1$
6	2 6 8 4	$\xi = 1$

Table 5.1: Face node definitions.

face index	$\tilde{\xi}_0$	$\tilde{\eta}_0$
1	$\xi_0$	$\eta_0$
2	$\eta_0$	$\xi_0$
3	$\zeta_0$	$\xi_0$
4	$\xi_0$	$\zeta_0$
5	$\eta_0$	$\zeta_0$
6	$\zeta_0$	$\eta_0$

Table 5.2: Face local coordinates in the old cell.

troduced in section 4.3, which lists the six neighbors of each cell. The neighbor cells are indexed according to which face of the old cell they share. We also need to know through which face of the new cell the trajectory enters. This is determined by examining the pointer list to see which face of the new cell is shared with the old cell. The values of the new computational coordinates are determined by a three-step process. In the following discussion, the zero subscript indicates quantities in the old cell. Two adjacent cells are shown in figure 5.2, to which we refer in the following discussion.

First, we define a two dimensional *face local* coordinate frame  $(\tilde{\xi}_0, \tilde{\eta}_0)$ , in the old cell according to table 5.2, based upon the index of the face through which the trajectory leaves the old cell. In figure 5.2, the trajectory exits the old cell through face 4.

Now it is necessary to find the face local coordinates in the new cell. The face of the new cell can have four possible orientations with respect to the face of the old cell, which will determine how the face local coordinates in the new cell are related to the face local coordinates in the old cell. This arises because even though the two faces have



orientation	$\tilde{\xi}$	$\tilde{\eta}$
1	$\tilde{\eta}_0$	$\tilde{\xi}_0$
2	$-\tilde{\xi}_0$	$\tilde{\eta}_0$
3	$-\tilde{\eta}_0$	$-\tilde{\xi}_0$
4	$\tilde{\xi}_0$	$-\tilde{\eta}_0$

Table 5.3: Relation between face local coordinates in the old and new cells.

face index	$\xi$	$\eta$	$\zeta$
1	$\tilde{\xi}$	$\tilde{\eta}$	-1
2	$\tilde{\eta}$	$\tilde{\xi}$	1
3	$\tilde{\eta}$	-1	$\tilde{\xi}$
4	$\tilde{\xi}$	1	$\tilde{\eta}$
5	-1	$\tilde{\xi}$	$\tilde{\eta}$
6	1	$\tilde{\eta}$	$\tilde{\xi}$

Table 5.4: Face local coordinates in the new cell.

the same four nodes as their corners, the nodes might be listed in a different order. The orientation is determined as follows. We look up the number of the first node in the old face. We then query all the nodes in the new face, and the orientation is set to be the index, with respect to the new face, of the previously determined node. In figure 5.2, the faces of the old and new cells have an orientation of 4.

Finally, the computational coordinates in the new cell depend upon through which face of the new cell the path enters, according to table 5.4. In figure 5.2, the trajectory enters the new cell through face 1.

Integration now proceeds in the new cell. The trajectory is complete when it exits the mesh, which is indicated in the cell-to-cell pointer list by an index of zero.

## 5.3 Planar Interpolation

A very useful technique to visualize the interior of the flow field is to display the projection of the three dimensional data onto a two dimensional surface within the flow domain. This provides more information about the interior flow field than any other method that we use. This method is easy to implement if the three dimensional data is stored on a structured grid, by holding one of the grid indices constant to define an interior surface. In that case, however, the choice of surfaces is limited by the grid structure, and the resulting surfaces often are curvilinear. Unless the geometry of the curvilinear surface is very clearly evident, the observer will mentally “flatten” the surface, and misinterpret the data. The method we use, which would be equally applicable to structured grid data as to unstructured mesh data, involves the linear interpolation of the three dimensional data onto an arbitrarily define planar surface in the flow field.

### 5.3.1 Basic Interpolation Algorithm

A plane in three dimensional space is defined by the equation

$$ax + by + cz = d.$$

If the coefficients are normalized such that

$$a^2 + b^2 + c^2 = 1$$

then the vector

$$\mathbf{i}_{x'} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

is a unit vector in the direction perpendicular to the plane, and the quantity  $d$  is the perpendicular distance from the plane to the origin of coordinates, and is positive in the direction of  $\mathbf{i}_{x'}$ . We can now define a right-handed coordinate system  $(x', y', z')$  with  $\mathbf{i}_{x'}$

given above, and the other two basic directions defined by

$$\begin{aligned} \mathbf{i}_{x'} &= \begin{cases} \frac{\mathbf{i}_x \times \mathbf{i}_{x'}}{|\mathbf{i}_x \times \mathbf{i}_{x'}|} & \text{if } \mathbf{i}_x \neq \mathbf{i}_{x'} \\ \mathbf{i}_x & \text{if } \mathbf{i}_x = \mathbf{i}_{x'} \end{cases} \\ \mathbf{i}_{y'} &= \begin{cases} \mathbf{i}_{x'} \times \mathbf{i}_x & \text{if } \mathbf{i}_x \neq \mathbf{i}_{x'} \\ \mathbf{i}_y & \text{if } \mathbf{i}_x = \mathbf{i}_{x'} \end{cases} \end{aligned} \quad (5.8)$$

yielding the orthogonal triad

$$\mathbf{i}_{x'} = \begin{bmatrix} -\frac{b}{\sqrt{a^2+b^2}} \\ \frac{a}{\sqrt{a^2+b^2}} \\ 0 \end{bmatrix}, \mathbf{i}_{y'} = \begin{bmatrix} -\frac{ac}{\sqrt{a^2+b^2}} \\ -\frac{bc}{\sqrt{a^2+b^2}} \\ \sqrt{a^2+b^2} \end{bmatrix}, \mathbf{i}_{z'} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}. \quad (5.9)$$

These definitions are completely arbitrary and were developed solely to insure that the primed coordinate frame is orthogonal and right-handed. The expressions for the unit vectors are used to form an orthogonal transformation matrix, as detailed in section 2.4.6, which is then composed with a viewing transformation (specified by the user) to display a perpendicular view of the interpolation plane.

The quantity  $z'$  is calculated and stored at every node point in the mesh, according to the relation

$$z' = ax + by + cz.$$

The equation of the plane is most simply expressed as  $z' = d$ , and every cell of the mesh is examined to determine how it intersects the plane. The intersection of a cell with the plane will be a polygon. The location of the corners of this polygon are determined as detailed in the following paragraphs.

Each edge of the cell in question is examined as to whether its endpoints lie on different sides of the plane, so that if

$$(z'_1 - d)(z'_2 - d) < 0$$

the edge intersects the plane, and a point is defined by interpolating the coordinates  $x, y$  and  $z$ , and the scalar  $s$  to the plane by the formulas

$$\begin{aligned}
x_p &= x_1 + \frac{d - z'_1}{z'_2 - z'_1}(x_2 - x_1) \\
&= \frac{(z'_2 - d)x_1 + (d - z'_1)x_2}{z'_2 - z'_1} \\
y_p &= y_1 + \frac{d - z'_1}{z'_2 - z'_1}(y_2 - y_1) \\
&= \frac{(z'_2 - d)y_1 + (d - z'_1)y_2}{z'_2 - z'_1} \\
z_p &= z_1 + \frac{d - z'_1}{z'_2 - z'_1}(z_2 - z_1) \\
&= \frac{(z'_2 - d)z_1 + (d - z'_1)z_2}{z'_2 - z'_1} \\
s_p &= s_1 + \frac{d - z'_1}{z'_2 - z'_1}(s_2 - s_1) \\
&= \frac{(z'_2 - d)s_1 + (d - z'_1)s_2}{z'_2 - z'_1}. \tag{5.10}
\end{aligned}$$

This point is appended to a list of points that will be the corners of the intersection polygon. In the case that the edge lies entirely within the plane, no interpolation is performed and both endpoints of the edge are appended to the list. The edges of the polygon are formed by generating the convex hull of this collection of points. When doing so, the values of  $x'$  and  $y'$  must be calculated for each corner, so that the convex hull will be correct. This method will yield the intersection of any convex cell with the plane. A good CFD mesh will contain only convex cells.

### 5.3.2 Enhanced Interpolation Algorithm

The basic algorithm described above is extremely inefficient. It is unnecessary to examine every cell of the mesh, since most of them will not intersect the plane at all. When the values of  $z'$  are calculated at the mesh nodes, two cell-based arrays are constructed: the minimum and maximum values of  $z'$  of all the nodes of a cell, which are called  $z'_{\min}$  and  $z'_{\max}$ , respectively. A cell is only capable of intersecting the plane if  $z'_{\min} < d < z'_{\max}$ , providing a quick test for avoiding a lot of calculation and comparison.

In some circumstances, it is even unnecessary to examine  $z'_{\min}$  and  $z'_{\max}$  for the majority of the cells. When a large number of planes are to be drawn using the same set of nodal  $z'$ , that is, when they are parallel, it is possible to sort the  $z'_{\min}$  and  $z'_{\max}$  lists and quickly isolate a subset of cells that might intersect a plane with a particular value of  $d$ . The sorting process requires  $\mathcal{O}(N_C \log N_C)$  operations, where  $N_C$  is the number of cells, but it is only performed once for a given set of parallel planes.

As mentioned previously, only cells for which  $z'_{\min} < d < z'_{\max}$  can intersect the plane. A binary search is performed upon the two lists to isolate separately the cells that satisfy each of these criteria. These indices are termed  $i'_{\min}$  and  $i'_{\max}$ . Now, the determination of whether a cell can intersect the plane is reduced to a comparison of the indices of the cell in the sorted  $z'_{\min}$  and  $z'_{\max}$  arrays with the indices returned by the binary searches. A cell with indices of  $i'_{C_{\min}}$  and  $i'_{C_{\max}}$  in the  $z'_{\min}$  and  $z'_{\max}$  lists, respectively, can only intersect the plane if  $i'_{\min} < i'_{C_{\min}}$  and  $i'_{C_{\max}} < i'_{\max}$ .

It is even unnecessary to compare the indices of all cells in the mesh. When we calculate the intersection of one plane, the list of cells that actually intersect it is saved. When many parallel planes are drawn, this list of cells can be used to quickly restrict the range of cells that need to be examined. This list forms the beginning of a list of cells that are capable of intersecting the current plane. If the current plane has a greater value of  $d$  than the previous plane, all cells that have a value of  $z'_{\min}$  between  $d_{\text{old}}$  and  $d$  are appended to the list of cells to be examined. Likewise, if  $d < d_{\text{old}}$ , then the cells for which  $d < z'_{\max} < d_{\text{old}}$  are appended to the list. These sub-lists of cells can also be quickly determined using the sorted lists mentioned in the previous paragraph.

Figure 5.3 displays this process graphically. On the left, a plane has moved to a lower value of  $d$ . The plane will “pass through” the thirteenth through seventeenth cells of the  $z'_{\max}$  array, and these cells will be checked for intersection. On the right, the analogous process is illustrated for a plane that has moved to a higher value of  $d$ . In this case, the second through sixth cells of the  $z'_{\min}$  array will be checked for intersection. After we perform the new intersection calculations, all cells of the list that do not intersect the plane are stricken from the list. With these algorithms, we can greatly narrow the list

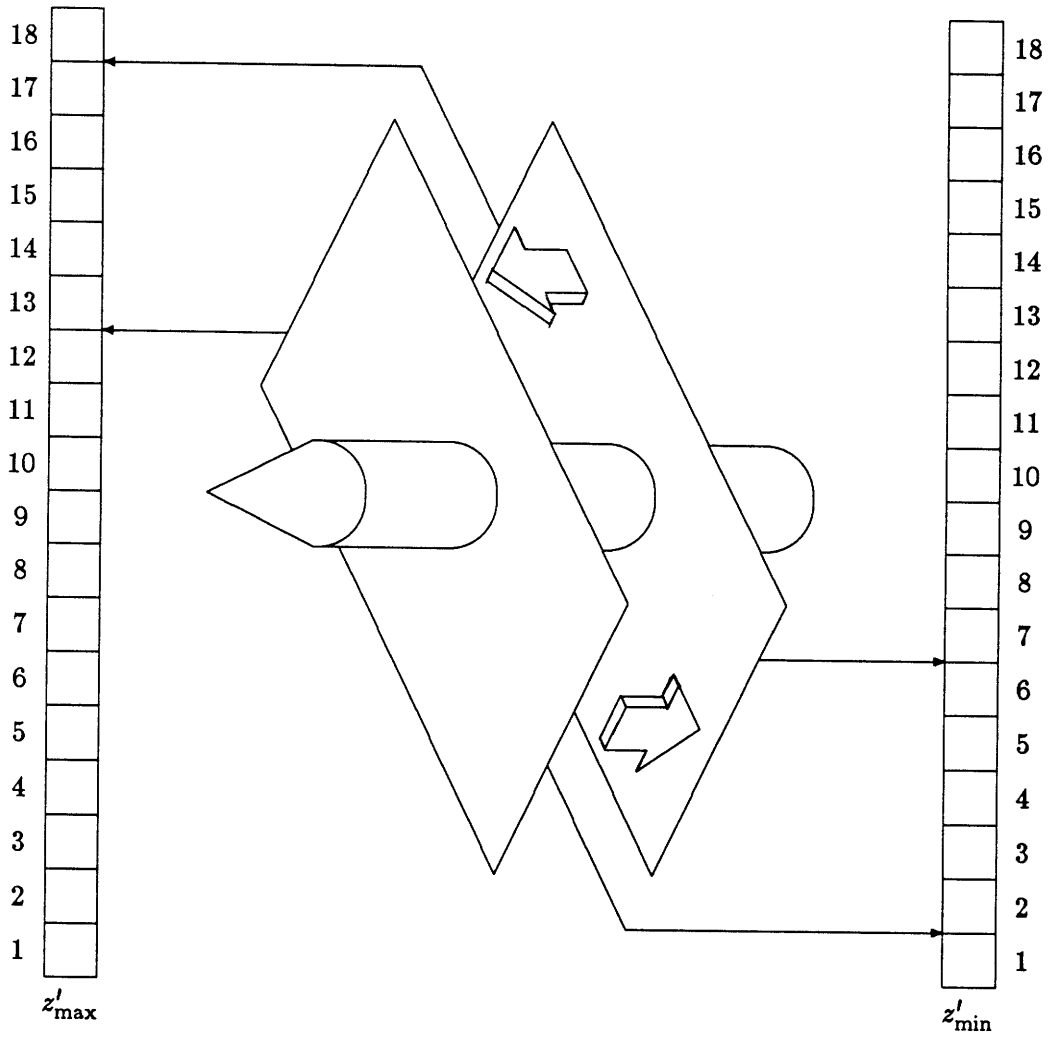


Figure 5.3: Moving an interpolation plane.

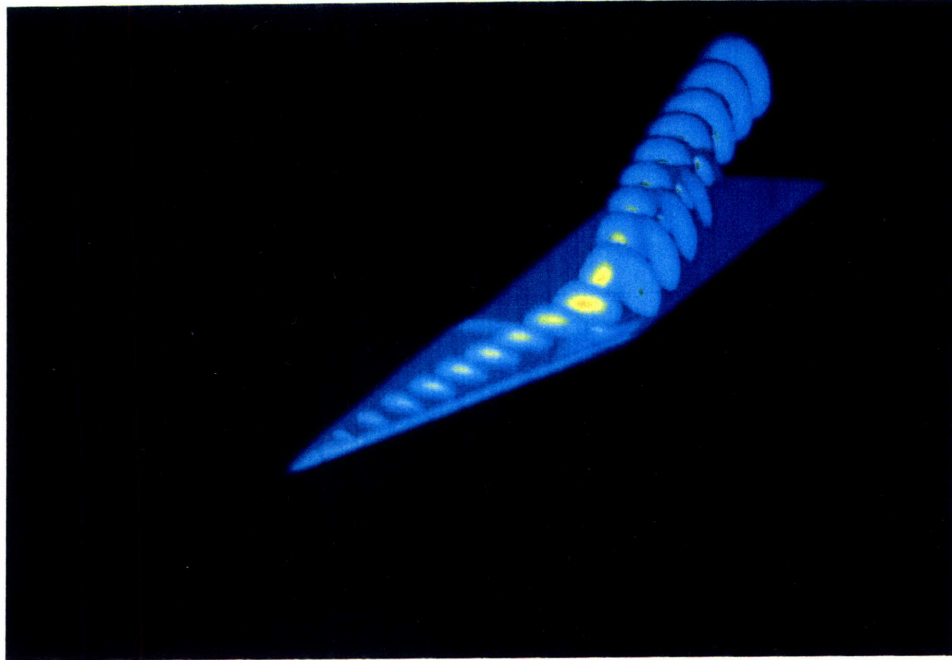


Figure 5.4: Thresholded data.

of cells that need to be examined for intersection, and we have a quick test to perform on this reduced list to determine intersection with the plane.

### 5.3.3 Display Formats

We use two display formats that both take advantage of the strategies discussed in section 5.3.2 that yield enhanced performance when multiple parallel planes are displayed. Both formats are intended to help the user visualize the third dimension. In the first format, a single plane moves along the  $z'$  axis, resembling a laser sheet. This format is particularly apropos with flow geometries in which an analytic analogy can be made between time and one of the spatial directions, such as the flow past a thin body.

The second format involves the simultaneous display of multiple planes. As pioneered by Weston [32], the data is thresholded to prevent the planes from obscuring each other, as show in figure 5.4. We prefer this method over the use of transparency since it is less confusing and it eliminates uninteresting uniform regions of the flow from the display. See appendix B for a description of the placement of graphics windows on the screen.

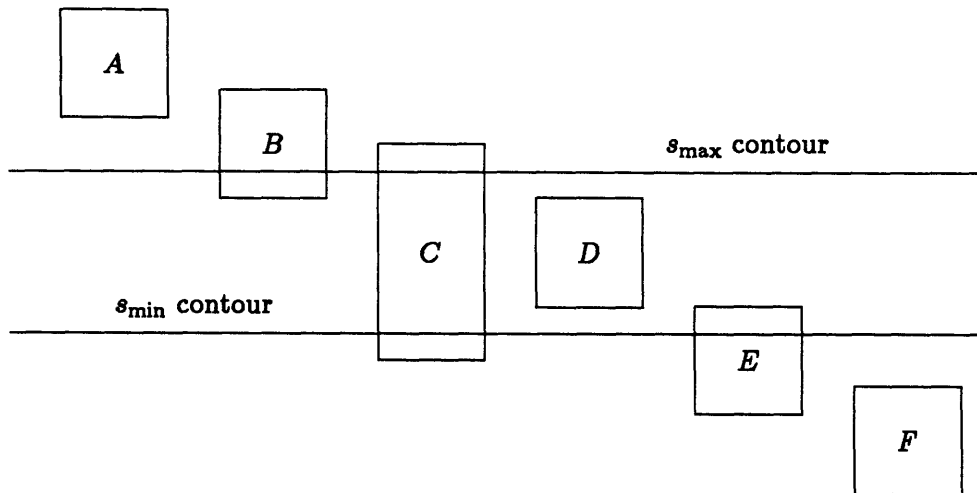


Figure 5.5: Thresholding of polygons.

### 5.3.4 Thresholding

Thresholding is a technique for selecting a subset of an interpolated plane to display, so that it will not obscure details behind it. The basic algorithm is as follows: two limiting scalar values,  $s_{\min}$  and  $s_{\max}$ , are selected. The polygons that comprise the plane, which are generated according to the method described in sections 5.3.1 and 5.3.2, are modified so that only their intersections with the region  $s_{\min} < s < s_{\max}$  are drawn. For clarity, contour lines are drawn at the values  $s_{\min}$  and  $s_{\max}$  as the border of the thresholded region.

Upon generation, each polygon is examined to determine how it relates to the threshold limits, as figure 5.5 illustrates schematically. If, for all corners,  $s_c > s_{\max}$ , like polygon *A*, or  $s_c < s_{\min}$ , like polygon *F*, then the polygon lies entirely outside the threshold region and it is ignored. If, for all corners,  $s_{\min} < s_c < s_{\max}$ , like polygon *D*, the polygon lies entirely within the threshold region and it is rendered in its entirety. If, however, some of the corners are in the threshold region and some are outside, as is the case with polygons *B* and *E*, or if for some corners  $s_c > s_{\max}$  while for others  $s_c < s_{\min}$ , like polygon *C*, then the polygon straddles at least one threshold contour and must be



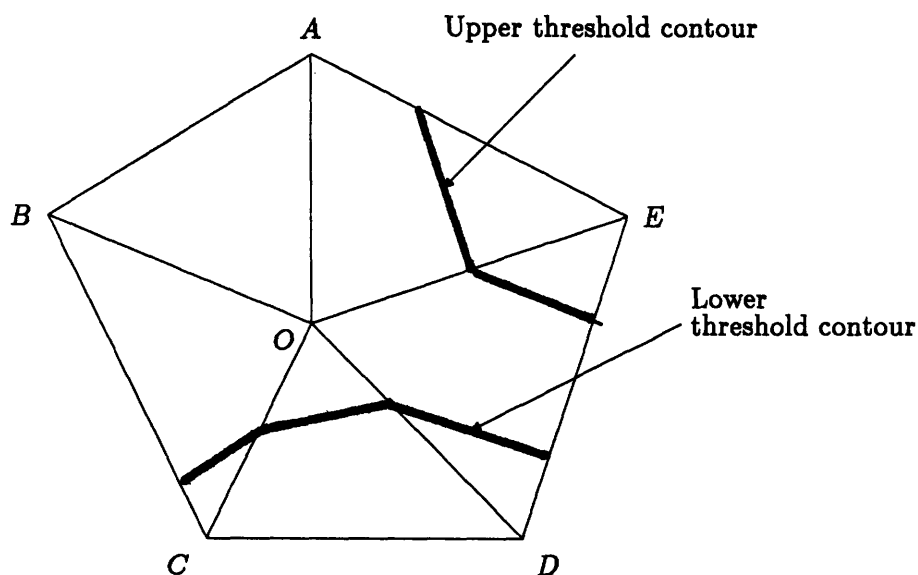


Figure 5.6: Subdivision and thresholding of a five-sided polygon.

modified.

An polygon that is selected for modification must be subdivided into triangles in order to perform thresholding. We choose to divide an  $n$ -sided polygon into  $n$  triangles, each comprised of two adjacent corners of the polygon and the polygon's centroid, as shown in figure 5.6. This method of subdivision yields a unique set of triangles, regardless of the numbering of the polygon corners, and generates smoother threshold contours. The resulting triangles are individually processed to create polygons that are entirely within the threshold region.

The corners of each triangle are sorted according to the value of the scalar  $s$  that is being plotted, with corner 1 having the lowest value and corner 3 having the highest value. The number of corners that have a value of  $s > s_{\max}$  is termed  $n_{\text{gt}}$ , and the number of corners that have a value of  $s < s_{\min}$  is called  $n_{\text{lt}}$ . There are ten possible ways for the two threshold contours to intersect a triangle, depending upon the values of  $n_{\text{gt}}$  and  $n_{\text{lt}}$ . These ten situations, shown in figure 5.7, are described below.

If  $n_{\text{gt}} = 0$  and  $n_{\text{lt}} = 0$ , then the entire triangle is within the threshold region and is plotted. Conversely, if  $n_{\text{gt}} = 3$  or  $n_{\text{lt}} = 3$  then the entire triangle is outside the

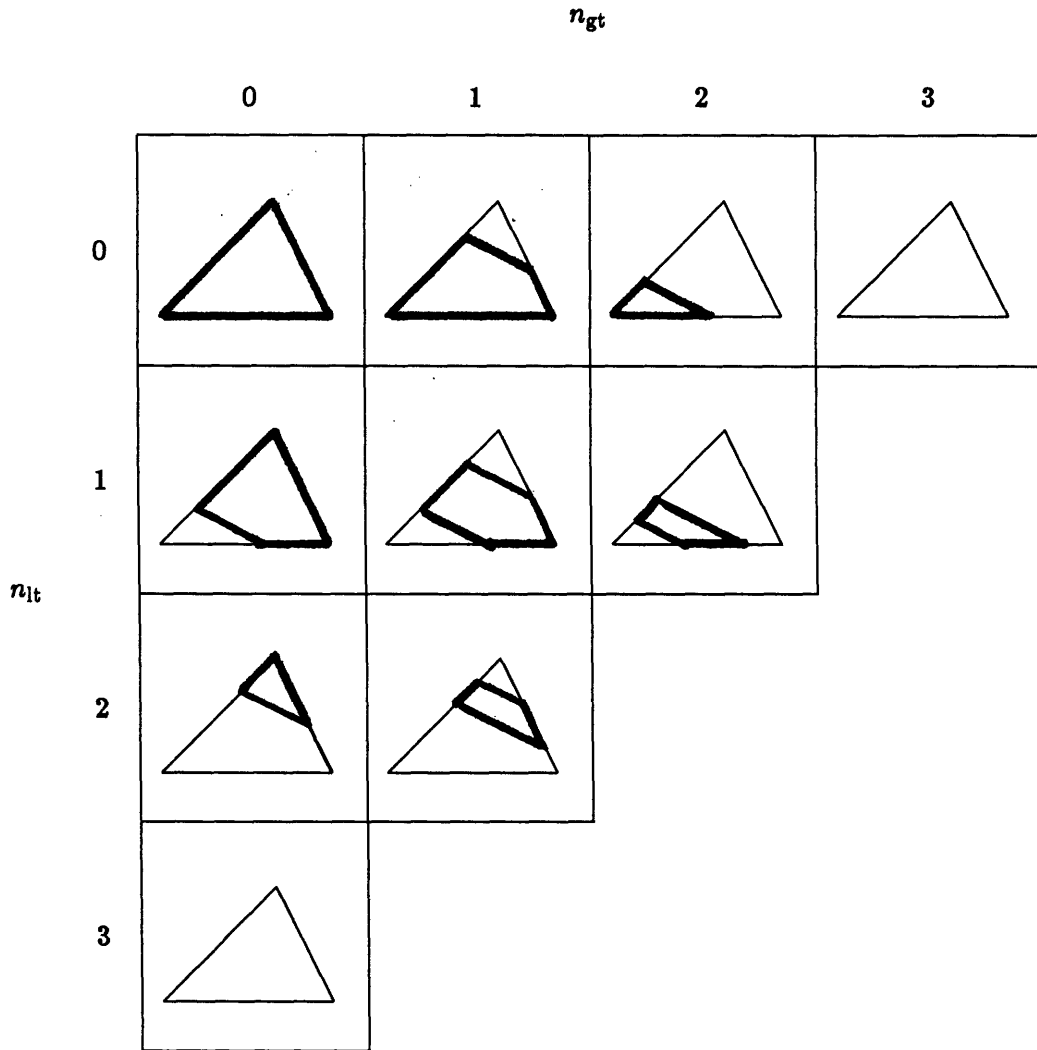


Figure 5.7: Triangle thresholding situations. Bold lines indicate polygons that are to be rendered.

threshold region and is discarded. These two situations are not excluded by the fact that the polygon has already been examined as to whether it lies entirely within or entirely without the threshold region. In figure 5.6, triangle  $\triangle ABO$  has  $n_{gt} = 0$  and  $n_{lt} = 0$ .

If  $n_{gt} = 0$  and  $n_{lt} = 1$ , or if  $n_{gt} = 1$  and  $n_{lt} = 0$ , then one threshold contour intersects the triangle, and the triangle must be clipped. Two new points are defined by interpolation, and a four-sided polygon is plotted. If  $n_{gt} = 0$  and  $n_{lt} = 2$ , or if  $n_{gt} = 2$  and  $n_{lt} = 0$ , again one threshold contour intersects the triangle, but in a different place. Once again the triangle must be clipped, and two new points are defined as before. A new triangle is defined and rendered.

If, however, both  $n_{gt}$  and  $n_{lt}$  are nonzero, then both threshold contours intersect the triangle and four new points must be defined by interpolation. Either a four-sided or a five-sided polygon results, depending on the particular case. These situations are extremely rare compared to the cases in which only one threshold contour intersects the triangle.

## 5.4 Electronic Probes

Existing graphics systems are mostly limited to reading contour level values to extract quantitative information concerning a flow field. We have included a set of electronic probe functions to “measure” flow field data from the display. We describe two such probes here.

### 5.4.1 Linear Profile

Much as the full three dimensional data is interpolated onto a plane, as described in section 5.3, the two dimensional data on one of these planes can be interpolated onto a line segment. The output thus obtained can be easily compared with experimental data produced by a rake or a traversal. The position and orientation of the line segment can

be dynamically controlled by the user. This is a less computationally demanding process than the full planar interpolation since there will be significantly fewer two dimensional cells in the plane than three dimensional cells in the full mesh. Also, the structure of the output is much simpler and requires no preprocessing before it is displayed.

We experimented with a technique whereby interpolation was only performed in polygons that intersect the line segment. When interpolation was performed on a cell, a connectivity table was examined to see which neighboring cell also intersects the line segment. This method was found to be less robust than the brute force method of examining all polygons in the plane. It could not handle a line segment that passes out of the domain and re-enters it. The overhead required to construct the connectivity table was also restrictive, and this technique provided no visible improvement in speed.

#### **5.4.2 Point Probe**

The other electronic probe we have developed allows the user to directly extract numerical data from the flow field. The point probe currently searches for the nearest interpolated node to the mouse cursor and displays the value of the scalar variable at the node. This technique is very simple to implement. The only sophisticated part of the algorithm is the inversion of the viewing transformation matrix (see section 2.4) to find the position of the mouse cursor in the original object coordinate frame.

## Chapter 6

### Results

We have examined the results of several computations as a test of our package's capabilities. No accurate three dimensional unstructured mesh solutions were available, so we made use of calculations performed on structured meshes that we converted to an unstructured data format. We have only had inviscid solutions available for examination.

The first test case was the transonic flow past the ONERA M6 wing, as computed by Roberts [27]. The flow solver was of the cell-centered finite volume type with a blend of second- and fourth-difference smoothing, as initially developed by Jameson [14]. This case was at a Mach number of 0.84 and an angle of attack of 3.03 degrees, and was computed on a grid of 48 by 10 by 10 cells, for a total of 4800 cells and 5929 nodes. This is not adequate resolution to resolve the shocks in the flow.

The largest data set that we examined is the transonic flow past the NTF delta wing, as solved by Becker [3], also using a cell-centered finite volume scheme. At a Mach number of 0.85 and an angle of attack of 15 degrees a strong leading-edge vortex forms above the suction side of the wing. This is precisely the type of flow scenario which our package was designed to visualize, since the interior of the flow contains important features that cannot be fully appreciated by looking at surface effects. The grid used was 30 by 31 by 80 cells in size, for a total of 74 440 cells and 80 352 nodes. Although the GS-1000 has sufficient memory to hold cases several times this size, the rendering and computational times are long enough to cause noticeable delays. The recently announced GS-2000, with twice the speed, should be sufficient to handle these cases.

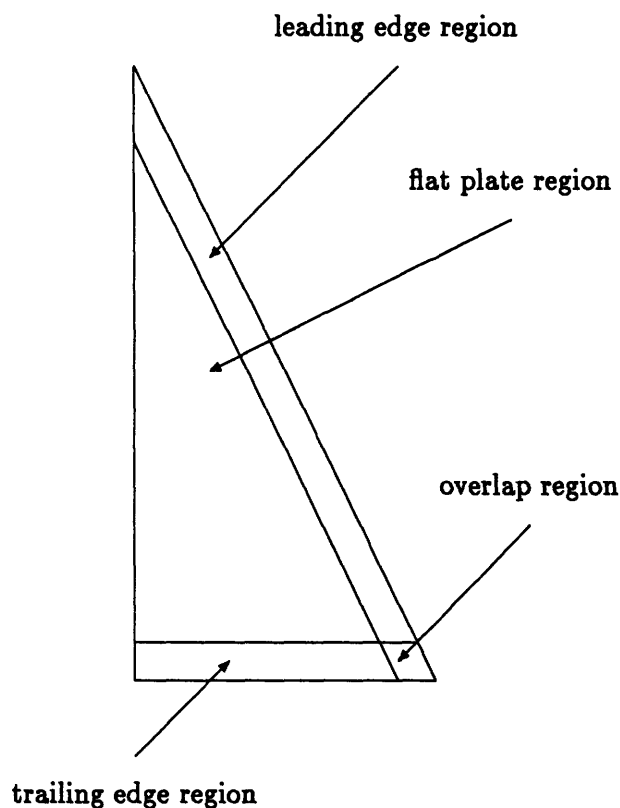


Figure 6.1: NTF delta wing geometry.

## 6.1 NTF Delta Wing Calculation

The NTF delta wing is a blunt leading edge delta wing that will be tested at the National Transonic Facility at NASA Langley Research Center. The wing has a leading-edge sweep of 65 degrees and has no twist or camber. Except for the leading and trailing edges, the geometry of the wing is that of a flat plate. The leading- and trailing-edge geometries are spanwise constant except for a region near the wing tip where the two geometries intersect (see figure 6.1). The test model has a root chord of 2.14 feet [18].

The geometry is normalized for flow calculations so that the wing root chord is unity. Only half the flow domain is solved, with a symmetry plane at the wing root. The flow domain extends one chord forward of the apex of the wing and aft of the trailing edge.

Becker's solution algorithm divides the flow domain into two zones, an outer zone

in which the Euler equations are solved, and an inner zone in which the Navier-Stokes equations are solved. The solution which we examine, however, is a preliminary solution on the outer grid only, with a solid wall boundary condition imposed at the “wing surface” and a periodic boundary condition on the “wake surface.” The periodic boundary condition is inaccurate with this geometry since the wake has thickness.

The results presented here can be qualitatively compared with the results of experiments by Hummel [11] and of higher-order vortex panel calculations by Hoeijmakers [10].

In delta wing flows of this type a strong vortex is expected to form off the leading edge, which rolls up and sits above the suction surface of the wing inboard of the leading edge. If the wing has a conical geometry, conical solutions can be found to model this type of flow [26]. The NTF wing is not quite conical, but it is very close. The pressure at the center of the vortex is reduced below the free stream level, and this pressure drop is transmitted to the wing surface below the vortex.

Figure 6.2 shows the pressure on the upper surface of the wing in the upper left, at the plane at 80% of the chord in the upper right, and along a line normal to the wing surface passing through the center of the vortex, in the lower left. The latter shows that the pressures at the center of the vortex and at the wing surface are nearly identical. Since the flow above the wing is nearly conical, the planar cross-section closely resembles such a cross-section at any axial station, although the pressure drop at the center of the vortex decreases as one moves aft. Figure 6.3 is similar, but instead of the profile it shows that the pressure at the center of the vortex is 0.22184, as compared to a free stream value of 0.71429, a significant drop.

The variation of total pressure has been found [26] to be a more accurate indicator of the extent of the vortex than is the variation of pressure. Figures 6.4 and 6.5 show the total pressure on the upper surface of the wing, on a plane at the 90% axial station, along a line normal to the wing surface through the center of the vortex, and at the center of the vortex. The surface plot shows little variation, except for a small region beneath the vortex at the trailing edge, due to inviscid separation. The plane and the profile show that the total pressure on the wing surface is almost entirely recovered to

the free stream value of 1.14558. Variations on the wing surface are due to numerical error. The total pressure at the center of the vortex, however, is 0.60211, a considerable drop. Figures 6.4 and 6.5 also show that the vortex is flattened in the wing-normal direction, whereas the pressure variations shown in figures 6.2 and 6.3 would indicate a nearly circular vortex cross-section.

The variation of quantities on the mesh surface provide no information on the activities of the vortex after it has passed the wing trailing edge. A good overall representation of this flow is provided by figure 6.6, which is a side view of four particle paths that are entrained into the vortex, and four particle paths that are not (three of them pass through the lower surface of the wake and are lost to the particle path integration algorithm, and one is outboard of the wing tip). The color of the pathlines indicates the pressure, in which the free stream value is yellow, lower pressures appear green, and the lowest pressures are blue. It is clear that the vortex is coherent while above the wing surface, but that it weakens and diffuses after passing the trailing edge, at which point it turns to follow the free stream direction. This fact is very difficult to notice using any other visualization methods. The vortex also moves slightly inboard as it incorporates the bound vorticity of the wing, a fact which is faintly visible in figures 6.7 and 6.8, which show the same set of particle paths viewed from the front and from the top of the wing. The vortex also moves slightly downward from the free stream direction under the influence of the downwash of the opposite vortex.

The motion of the vortex inboard and towards the free stream is also visible in figures 6.9 and 6.10, which show the total pressure on eighteen planes, ranging from the 10% axial station to the 190% axial station, with thresholding values of 0.5 and 1.0. The conical nature of the vortex above the wing is also seen, as well as the diffusion of the vortex past the trailing edge, and a new feature: a counter-rotating vortex appears outboard of the main vortex just as it passes the trailing edge. As a result, the main vortex actually intensifies in strength for a short distance. This has also been seen by Hummel and Hoeijmakers. The counter-vortex is visible in this figure up to one-half wing root chord behind the trailing edge. More detail is visible in figures 6.11, 6.12 and 6.13, which show total pressure at the 120% axial station, with a profile through



the two vortices and the values of the total pressure at their centers. The counter-vortex has about half the strength of the main vortex at this station, which is itself stronger than it was above the wing (compare a total pressure of 0.50118 at this station to 0.60211 at the 90% station).

Figures 6.14 and 6.15 show the total pressure variation at the 190% axial station, the aftmost station in the flow domain. The vortex is nearly as strong here as at the 90% station, having a total pressure of 0.63333 at its center, but it is much more diffuse. Even this far aft of the wing, the total pressure is not symmetric about the center of the vortex, as the profile in figure 6.14 shows. This might be due to the influence of the opposite vortex.

One can also see, in figure 6.2, a faint planar shock extending along the trailing edge from the symmetry plane almost to the vortex. Figures 6.16 and 6.17 show the Mach number in the symmetry plane about the entire wing, and a close-up of the trailing edge. One can see that this probably is a very faint fishtail shock, which is common at the trailing edges of transonic airfoils. Since this shock is so close to the wake surface, which has an improper boundary boundary condition, it is not clear whether it's structure is accurate. Also, the grid resolution is inadequate to correctly resolve this area.

## 6.2 Practical Guidelines and Experiences

We have found that it is possible to locate a great wealth of flow features in a short period of time because of the interactive nature of our visualization package. For example, the existence of the fishtail shock was not know beforehand. Many other graphics packages require *a priori* knowledge of the flow structure in order to locate flow features in a reasonable amount of time. The standard operating procedure for investigating a new flow is to scan the entire domain with a moving interpolation plane, on which flow features will stand out prominently. Once the investigator has located interesting flow features, they can be analyzed in detail with the other methods. Particle paths and threshold planes are most useful to convey a large amount of information in

a single image, but require some knowledge of the flow structure beforehand to be most effective. Particle paths indicate the vortical nature of a flow where other methods do not, and, when used sparingly, can be used to great advantage in very complex internal flows.

As was discussed in section 4.4, memory storage limitations restrict us to data sets with 450 000 or fewer nodes. Solutions with up to 700 000 nodes can be accommodated by dynamic memory allocation. This is sufficient for three dimensional Euler calculations and for some Navier-Stokes solutions. For such a large data set, computational capabilities are more limiting than memory restrictions. While surface plots, particle paths, and the electronic probes are relatively insensitive to the size of the problem, planar interpolation can become quite cumbersome if thresholding is not in use. It may become necessary to allow the user to omit some lengthy calculations if it is determined that the output will not be immediately needed. Also, it should be noted that the algorithms as they are currently formulated do not vectorize well at all.





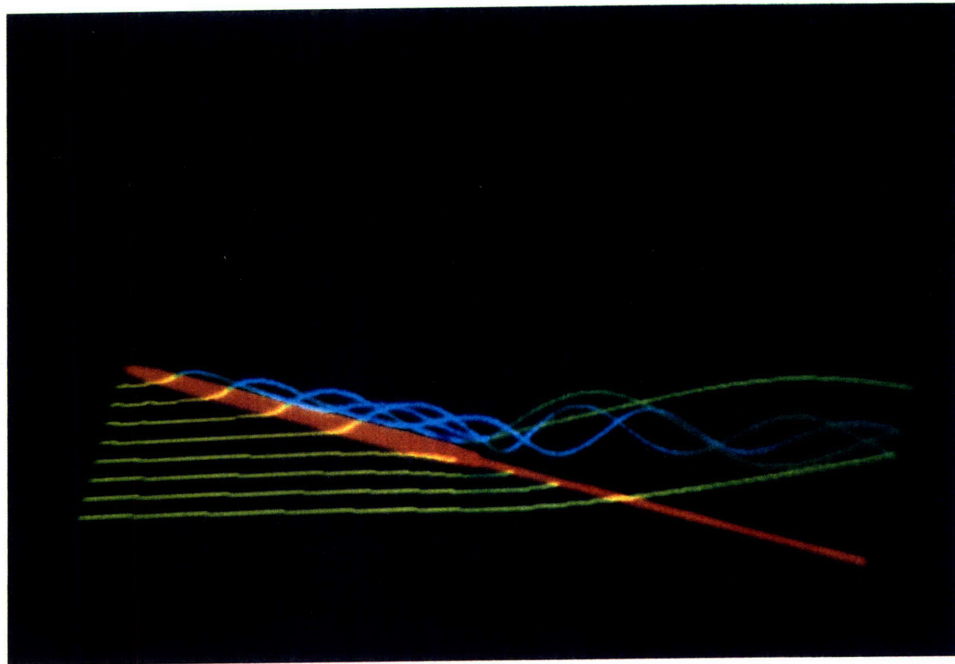


Figure 6.6: Eight particle paths, viewed from the side.

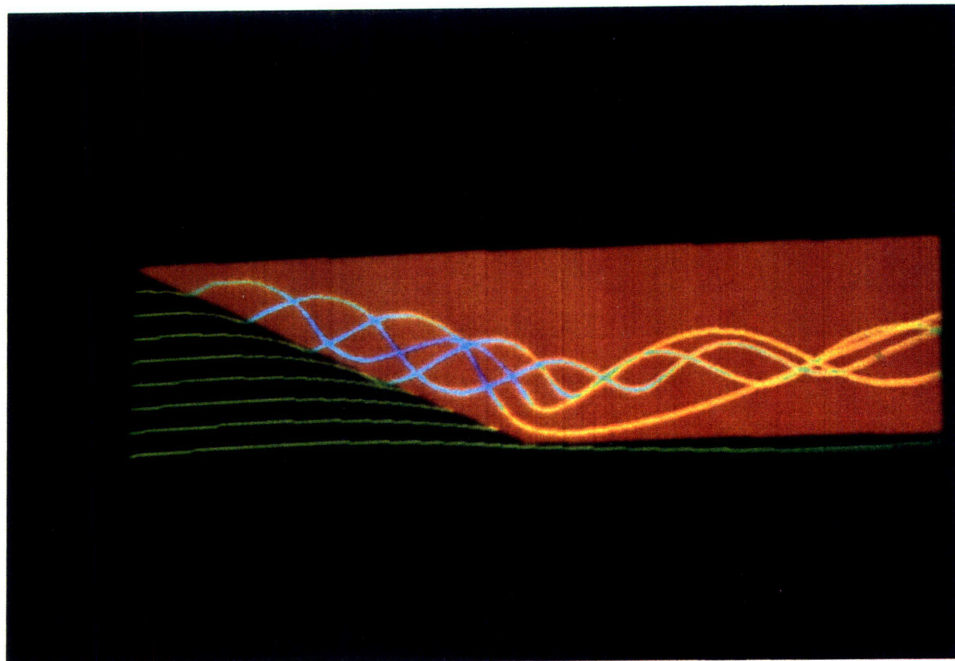


Figure 6.7: Eight particle paths, viewed from the top.



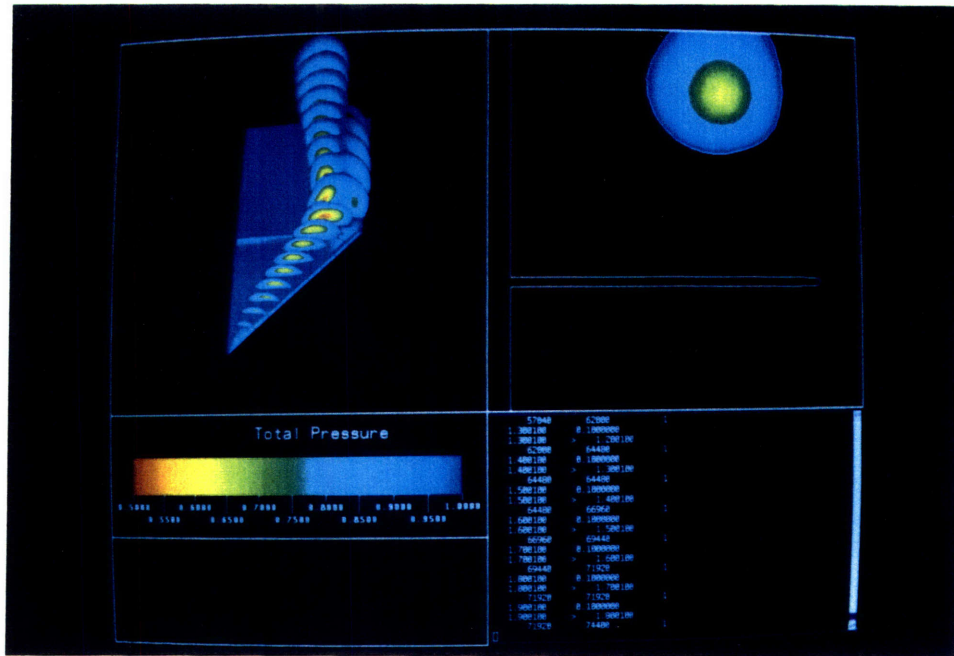


Figure 6.10: Total pressure interpolated onto multiple planes, with threshold values at  $p_0 = 0.5$  and  $p_0 = 1.0$ . View from above, looking aft.

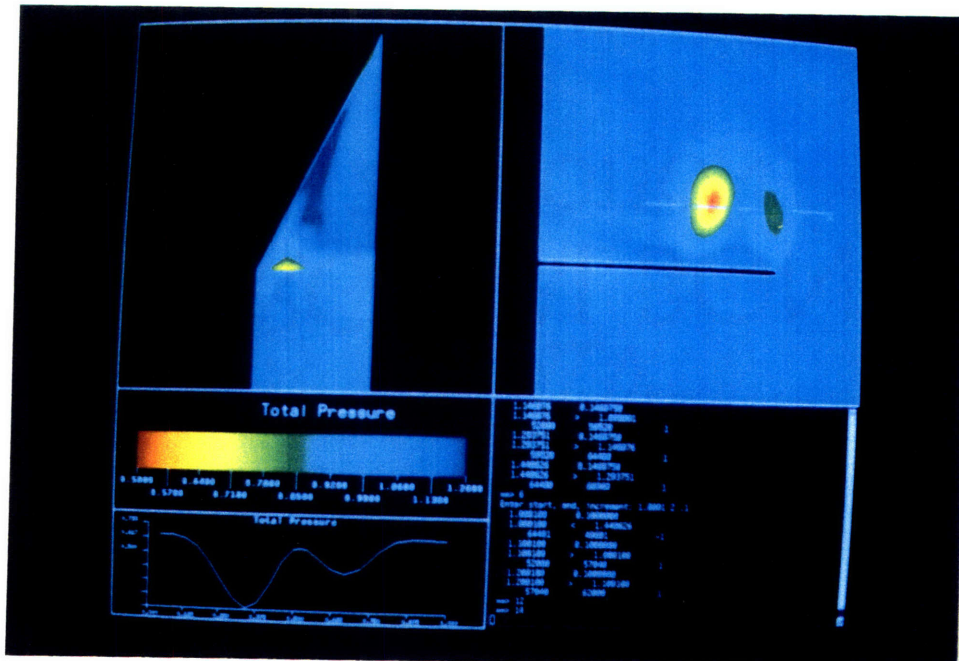


Figure 6.11: Total pressure variation in a plane at the 120% axial station, and on a line through the two vortices.





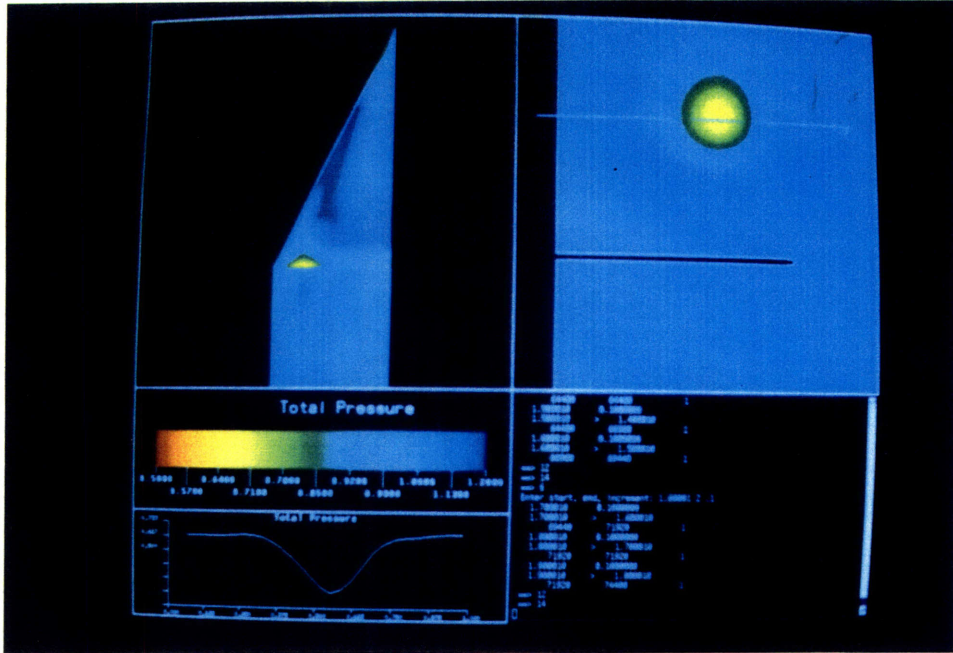


Figure 6.14: Total pressure variation in a plane at the 190% axial station, and on a line through the vortex.

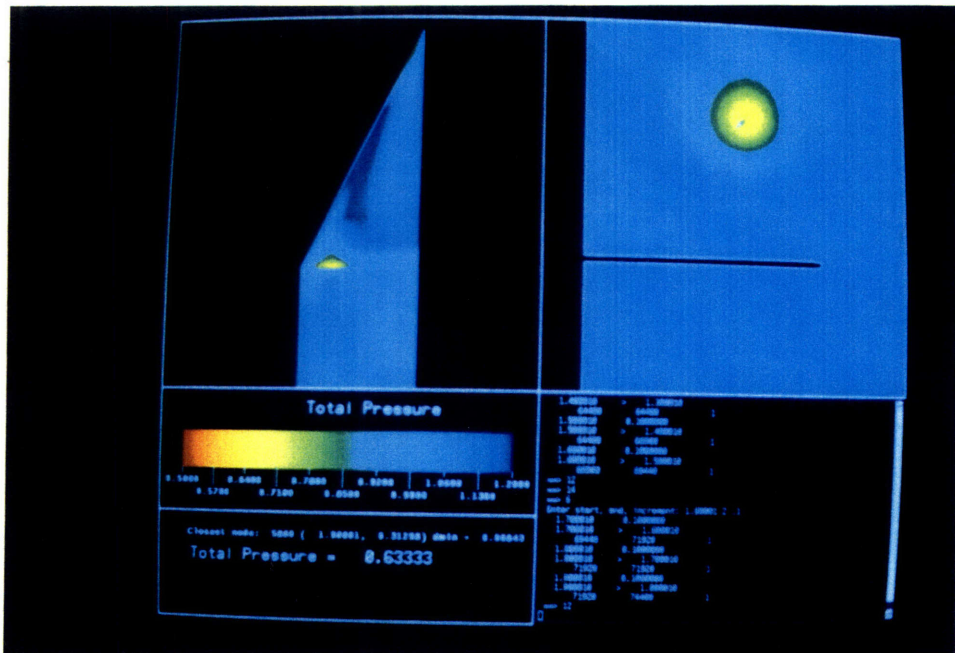


Figure 6.15: Total pressure variation in a plane at the 190% axial station, and at the center of the vortex.

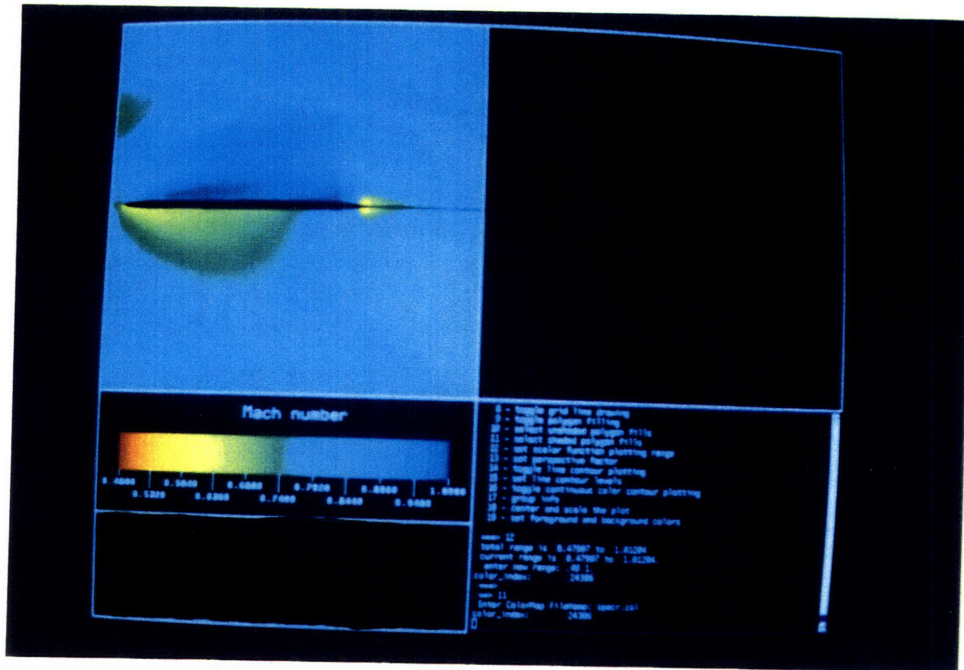


Figure 6.16: Mach number in the symmetry plane.

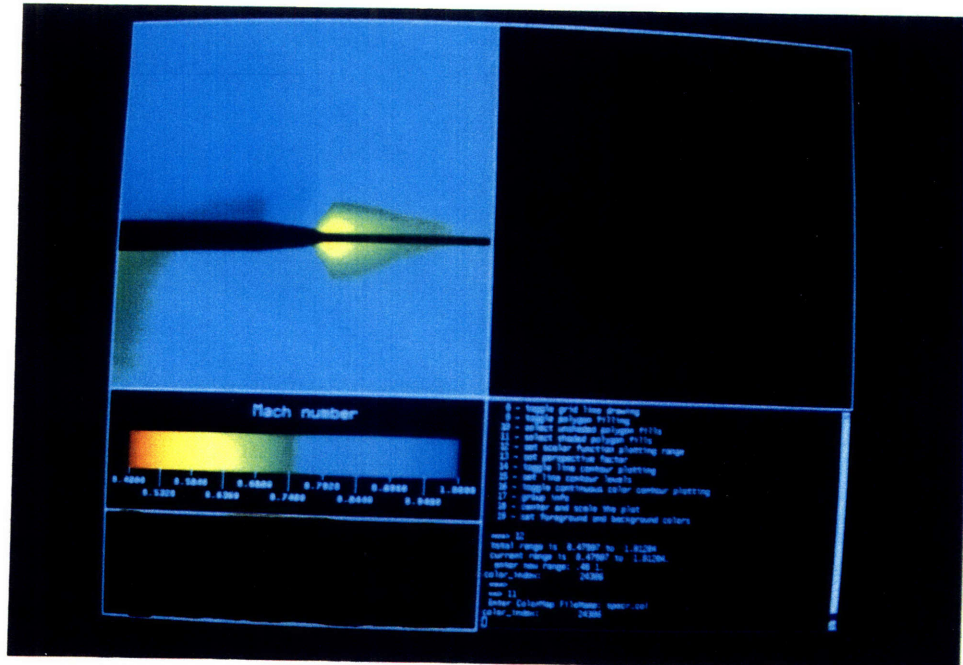


Figure 6.17: Mach number in the symmetry plane. Trailing edge detail.

## Chapter 7

# Summary and Conclusions

### 7.1 Summary

We have developed a graphics system that allows the interactive investigation of three dimensional CFD data that is stored on an unstructured mesh, and have implemented it on a graphics supercomputer, the Stellar GS-1000. Although the algorithms we use are often much more complicated than structured grid visualization methods, the added complexity does not significantly reduce the size of the problems we can visualize, either in terms of added computation that is performed or of increased memory storage for an equivalent number of nodes.

We use four methods to visualize 3D flow fields. First, some flow quantity is plotted on a subset of the boundary of the mesh, as selected by the investigator. This is useful for comparison with experiment or evaluation of surface loading, but it provides no information about the interior of the flow field. This is the simplest and least computationally intensive method that we use. Because of its simplicity, it has been the most widely used three dimensional visualization method.

Another method is to calculate particle paths by numeric integration. Pathlines are useful for visualizing external flows, and can also provide a comparison with experimental data. Calculation of pathlines is numerically intensive and cannot be done in real time on ordinary graphics workstations. Particle path integration required much modification to be useful with unstructured mesh data.

The third method is to interpolate three dimensional data onto a plane that intersects the flow field. This provides more information about the interior dynamics of a

flow field than either of the other methods. It is simple to implement if the data is stored on a structured mesh by limiting oneself to using grid coordinate surfaces. The procedure is more complicated and more computationally intensive when data is stored on unstructured meshes, but the results are of a higher quality. Data must be interpolated between nodes onto the plane. The plane can have an arbitrary orientation, and can be moved in real time to allow the researcher to quickly locate flow features.

Finally, the data on the plane can be further interpolated onto a line segment to produce a profile of a flow quantity. The location and orientation of the line segment can be controlled dynamically by the investigator. In addition, the state of the flow at a point can be continuously polled with a point probe.

We examined, as a test case, the transonic flow past the NTF delta wing, as solved by Becker. The near-conical nature of the leading-edge vortex while it is above the wing is easily identifiable. We also identified the formation of a counter-rotating vortex aft of the trailing edge that strengthens the main vortex. The behavior of the main vortex in this region is also clearly seen. In addition, we located a small inviscid separation region below the vortex at the trailing edge, and a fishtail shock at the trailing edge of the wing inboard of the vortex.

## 7.2 Conclusions

We have found that it is possible to locate a great wealth of flow features in a short period of time because of the interactive nature of our visualization package. For example, the existence of the fishtail shock was not known beforehand. Many other graphics packages require *a priori* knowledge of the flow structure in order to locate flow features in a reasonable amount of time. We have found that the technique of planar interpolation is effective in initially locating the major features of a new flow. Once some knowledge of the flow structure is gained, particle paths and threshold planes can be used to convey a large amount of information in a single image. The electronic probes were found to be very useful to help understand the details of the flow. We

found that the time required to perform planar interpolation becomes excessive when examining very large solutions, unless thresholding is in use. The other visualization methods are less numerically demanding, and are not expected to cause problems with large solutions. The amount of work required for planar interpolation is proportional to  $N_C^{\frac{2}{3}}$ , where  $N_C$  is the number of cells in the solution. The newly announced GS-2000, with twice the speed of the GS-1000, should realize the same performance with data sets three times the size. The upper bound on solution size at present is 450 000 nodes, which is adequate for Euler calculations and some Navier-Stokes solutions. This limit can be increased to 700 000 nodes with improvements in memory management.

### 7.3 Recommendations for Further Work

This work is a first-generation graphics package, and as such is far from complete. All the algorithms we used are derived from previously implemented methods, and we expected all of them to provide a useful representation of the flow field. Now that we have a basis upon which to build, further work is needed to develop new visualization methods that can provide more information than the ones currently in use. For example, a surface can be constructed in three dimensions upon which a flow quantity has a constant value. Such contour surfaces could be produced with a modification to our planar interpolation algorithm. This algorithm essentially locates a surface of constant  $z'$ , which is a plane. By replacing  $z'$  with the desired scalar quantity in the algorithm, surfaces of constant value can be produced. A problem with these surfaces is that they often are either too simple in structure to provide useful information, or they are so complex that determining what one is seeing, and interpreting it, is a difficult task.

Another intriguing visualization method is an extension of particle path integration, called streamtapes. A streamtape is a two dimensional ribbon one edge of which is a true particle path. The other edge is constructed in a manner similar to a particle path, but with the restriction that it always remain at a constant normal distance from the other edge. The result is a constant-width ribbon whose twist illustrates the streamwise vorticity of the flow. It is necessary to visually distinguish the two sides of

the streamtape, with different colors, for example.

In addition to developing advanced visualization methods for steady flows, it is necessary to formulate strategies to display unsteady three dimensional flows. All of our methods, in one way or another, use time to help understand the three dimensional structure of steady flows. With surface plots and particle paths, it is simply a matter of interactively changing the direction of view, whereas planar interpolation and the electronic probes rely on a changing geometry. How these methods can be modified to work well with time-varying data is unclear. Possibly the best method to visualize unsteady flows will be interpolation onto multiple planes with thresholding, since it displays a large amount of data concerning the interior of the flow field statically. Also, the distinction must be made between particle paths, streamlines and streaklines, and one must determine what information is provided by each.

One step beyond traditional methods of flow visualization is feature detection, in which algorithms are developed to automatically locate interesting flow features for the investigator. Shocks can be located by inspecting the Mach number resolved in the direction of the pressure gradient. When this quantity has a value of unity and is decreasing, a shock is indicated. The shock can be plotted as a surface within the flow domain. It should be possible, in principle, to develop such strategies to locate other flow features, such as shear layers, vortices, and slip surfaces.

## Bibliography

- [1] *XFDI Reference Manual*. Stellar Computer Inc., 1988.
- [2] D. A. Anderson, J. C. Tannehill, and R. H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Hemisphere Publishing Corporation, 1984.
- [3] T. M. Becker. *3-D Calculations of Transonic Vortex Flows over the NTF Delta Wing*. Master's thesis, M.I.T., August 1989.
- [4] M. O. Bristeau, O. Pironneau, R. Glowinsky, J. Periaux, P. Perrier, and G. Poirier. "Application of Optimal Control and Finite Element Methods to the Calculation of Transonic Flows and incompressible Viscous Flows." In *Proc. IMA Conference on Numerical Methods in Applied Fluid Mechanics*, Academic Press, 1980.
- [5] M. D. Brown. *Understanding PHIGS. TEMPLATE*, The Software Division of Megatek Corporation, 1985.
- [6] D. A. Chapman, H. Mark, and M. W. Pirtle. "Computers vs. Wind Tunnels for Aerodynamic Flow Simulations." *Aeronaut. Astronaut.*, 35:22-30, 1975.
- [7] D. Edwards. "Three-Dimensional Visualization of Fluid Dynamic Problems." AIAA Paper 89-0136, January 1989.
- [8] J. D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Publishing Company, 1982.
- [9] J. L. Hess and A. M. O. Smith. "Calculations of Potential Flow About Arbitrary Bodies." *Prog. Aeronaut. Sci.*, 8:1-139, 1966.
- [10] H. W. M. Hoeijmakers. *An Approximate Method for Computing Inviscid Vortex Wake Roll-Up*. NLR TR 85 149 U, November 1985.

- [11] D. Hummel. *On the Vortex Formation Over a Slender Wing at Large Angle of Incidence*. AGARD CP-247 Paper 14, 1979.
- [12] J. F. Dannenhoffer III and J. R. Baron. "Robust Grid Adaptation for Transonic Flows." AIAA Paper 85-0484, January 1985.
- [13] A. Jameson, T. J. Baker, and N. P. Weatherhill. "Calculation of Inviscid Transonic Flow over a Complete Aircraft." AIAA Paper 86-0103, 1986.
- [14] A. Jameson, W. Schmidt, and E. Turkel. "Numerical Solutions of the Euler Equations by a Finite Volume Method Using Runge-Kutta Time Stepping Schemes." AIAA Paper 81-1259, June 1981.
- [15] T. Lasinski, P. Buning, D. Choi, S. Rogers, G. Bancroft, and F. Merritt. "Flow Visualization of CFD Using Graphics Workstations." AIAA Paper 87-1180, June 1987.
- [16] Y. Levy, A. Seginev, and D. Degani. "Graphical Representation of Three-Dimensional Vortical Flows by Means of Helicity Density and Normalized Helicity." AIAA Paper 88-2598, June 1988.
- [17] R. Löhner, K. Morgan, J. Peraire, and O. C. Zienkiewicz. "Finite Element Methods for High-Speed Flows." AIAA Paper 85-1531, June 1985.
- [18] J. D. Luckring. *NTF Delta Wing Geometry*. NASA Langley Research Center, Hampton, Virginia.
- [19] R. Magnus and H. Yoshihara. "Inviscid Transonic Flow over Airfoils." *AIAA Journal*, 8:2157-2162, 1970.
- [20] J. Miyakawa, S. Takanashi, K. Fujii, and K. Amano. "Searching the Horizon of Navier-Stokes Simulation of Transonic Aircraft." AIAA Paper 87-0524, January 1987.
- [21] D. Modiano, M. Giles, and E. Murman. "Visualization of Three-Dimensional CFD Solutions." AIAA Paper 89-0138, January 1989.



- [22] E. M. Murman and J. D. Cole. "Calculation of Plane Steady Transonic Flows." *AIAA Journal*, 9:114-121, 1971.
- [23] R.-H. Ni. "A Multiple-Grid Scheme for Solving the Euler Equations." AIAA Paper 81-1025R, June 1981.
- [24] A. Nye. *Xlib Programming Manual for Version 11, Volume One*. O'Reilly & Associates, Inc., 1988.
- [25] R. A. Plastock and G. Kiley. *Schaum's Outline of Computer Graphics*. McGraw-Hill Book Company, 1986.
- [26] K. G. Powell. *Vortical Solutions of the Conical Euler Equations*. PhD thesis, M.I.T., July 1987.
- [27] T. W. Roberts. *Euler Equation Computations for the Flow Over a Hovering Helicopter Rotor*. PhD thesis, M.I.T., November 1986.
- [28] S. Robinson and K. Hu. "Stereo Image Visualization of Numerically Simulated Turbulence." AIAA Paper 89-0141, January 1989.
- [29] R. A. Shapiro. *An Adaptive Finite Element Solution Algorithm for the Euler Equations*. PhD thesis, M.I.T., May 1988.
- [30] M. Smith, F. Dougherty, W. Van Dalsem, and P. Buning. "Analysis and Visualization of Complex Unsteady Three-Dimensional Flows." AIAA Paper 89-0139, January 1989.
- [31] T. Theodorsen and I. E. Garrick. *General Potential Theory of Arbitrary Wing Sections*. NACA TR 452, 1933.
- [32] G. Volpe. "Streamlines and Streamribbons in Aerodynamics." AIAA Paper 89-0140, January 1989.
- [33] R. P. Weston. "Color Graphics Techniques for Shaded Surface Displays of Aerodynamic Flowfield Parameters." AIAA Paper 87-1182, June 1987.

## Appendix A

# Disk File Format

The format for data storage in disk files was chosen to minimize the disk space required. The data file is divided, like Gaul, into three parts. The first part is a list of the three dimensional coordinates of each node of the mesh. The second part is a list of the mesh nodes that make up each individual hexahedral cell. This list has the form of an array with  $8N_C$  elements, where  $N_C$  is the number of the cells in the mesh. The final part of the data file is a description of the boundary of the mesh. It is in the form of an array with  $5N_B$  elements, where  $N_B$  is the number of quadrilaterals that compose the boundary. The fifth data element present for each quadrilateral is an integer that identifies which to which boundary group the quadrilateral belongs.

The data are read with the following FORTRAN statements:

```
c
c size of the object.
read(1) nvert, ncell, nbound
c
c read the nodes.
read(1) (x(ivert), y(ivert), z(ivert), ivert = 1, nvert)
c
c read the cells.
read(1) (con(icon), icon = 1, 8*ncell)
c
c read the boundary polyogns.
read(1) (con(icon), icon = 1, 5*nbound)
```

where `nvert` is the number of nodes, `ncell` is the number of cells, and `nbound` is

the number of boundary polygons. The array `con` is a work array so that the cell and boundary polygon data can be read with an implied do-loop, which reduces the size of the disk file. This array is a concatenation of the individual cell or boundary polygon data. When reading the cells, `con(1), ..., con(8)` would contain the node numbers of the corners of the first cell, `con(9), ..., con(16)` would contain the node numbers of the corners of the second cell, and so on. When reading the boundary polygons, `con(1), ..., con(4)` would contain the node numbers of the corners of the first polygon and `con(5)` would indicate to which group it belongs, `con(6), ..., con(9)` would contain the node numbers of the corners of the second polygon and `con(10)` its group number, and so on.

## Appendix B

# Screen Layout

The screen display contains two main graphics windows, as shown in figure B.1. In one window, referred to as the 3-D window, portions of the mesh boundary surface are displayed as described in section 5.1. Any interpolated plane or set of planes is also displayed in this window. In the other window, called the 2-D window, the current plane is displayed in a perpendicular view, using the coordinate frame of equations 5.9 and the results of section 2.4.6 to define a viewing transformation relating this frame to the object coordinate frame.

Three other windows are also present on the screen. The key window contains the colorbar that relates the colors displayed to the value of the scalar being plotted. The output of the linear profile (see section 5.4.1) and of the point probe (see section 5.4.2) appear in the status window. Standard text input and output are handled through the terminal emulation window in the lower right-hand corner.

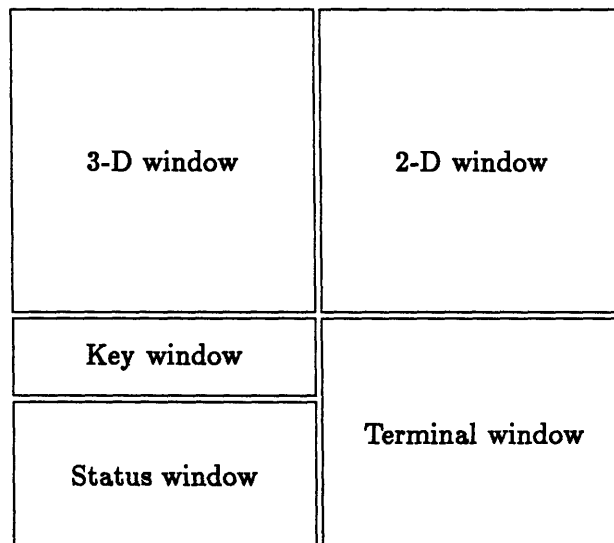


Figure B.1: Arrangement of windows in screen display.