Computing Systems in Engineering Vol. 1, No. 1, pp. 51–62, 1990
Printed in Great Britain.

0956–0521/90 $3.00+0.00
© 1990 Pergamon Press plc

ADVANCED INTERACTIVE VISUALIZATION FOR CFD

M. GILES and R. HAIMES

Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA 02139, U.S.A.

(*Received* 20 *April* 1990)

Abstract—New ideas are presented for the visualization of computational fluid dynamics data. These include both unsteady two-dimensional and steady three-dimensional data on either structured or unstructured grids. In addition to presenting some specific algorithm advances, considerable attention is devoted to innovative interactive probes and the appropriate choice of program architecture and internal data structure.

1. INTRODUCTION

The subject of this paper is the development of new ideas for the visualization of computational fluid dynamics (CFD) data. New approaches are required because rapid advances in basic algorithm development have led to flow codes which produce data that cannot be viewed using established plotting software, such as PLOT3D.[1] PLOT3D was written to visualize steady data on multiple structured grids. However, in the last five years, a major focus of research effort has been unstructured flow codes using grids composed of triangular or quadrilateral cells (in two dimensions), or tetrahedral, prism or hexahedral cells (in three dimensions). This is because of their relative ease of generation for complex geometries, and because of the ease of adaptive grid refinement. Also, much research is being done on unsteady flows, particularly in two dimensions, but visualization tools have been slow to emerge to enable one to visualize and understand the vast amount of data that are generated.

At the same time, there has been a rapid evolution in computer hardware. There are now graphics mini-supercomputers, or super-workstations, which have a floating point capability which is a significant fraction of a CRAY, combined with impressive graphics capabilities. The research work in this paper was performed on a Stardent GS2000 which has, on average, a sustained capability of 15 Mflops and 150,000 Gouraud-shaped triangles per second. This hardware speed makes possible interactive graphics of a kind which was not previously feasible.

There are only a few research groups working on the development of new CFD graphics software. NASA Ames is continuing development of PLOT3D and associated programs. These continue to be based on multi-block grids, and the research emphasis is on improving the supercomputer–workstation links and the extraction of data from the interior of three-dimensional calculations.[2] Weston at NASA Langley has developed a structured three-dimensional program[3] and an unstructured triangular program, and is currently working on an unstructured tetrahedral program. While at the Massachusetts Institute of Technology, Dannenhoffer developed a two-dimensional graphics package called GRAFIC, which is able to handle a variety of grid data structures through a technique described in a later section. Dannenhoffer is continuing the development of GRAFIC at the United Technologies Research Center, and the latest version also treats three-dimensional data.[4] Löhner at George Washington University is developing an unstructured tetrahedral graphics program,[5] and Strid and Rizzi at FFA (the Aeronautical Research Institute of Sweden) have developed a structured three-dimensional graphics program,[6] and are working on an unstructured version.

2. DESIGN OF VISUAL2 AND VISUAL3

2.1. *Design goals*

The first step in developing any software is to carefully define its objectives and intended functionality. The design of VISUAL2 and VISUAL3 was begun a year before the hardware arrived so there was adequate time to refine the software design before coding started. In fact, overall more time was spent on discussing the data structure, software architecture, programming and user interfaces and intended functionality, than has been spent on the programming. The following list gives the design goals of VISUAL2 and VISUAL3, the two-dimensional and three-dimensional programs, respectively, with some discussion of each item.

- Very interactive—as opposed to high-quality glossy videos that take days to produce and are used primarily for presentations, the aim was a practical tool for everyday use by engineers, in which they would be able, instantly, to examine different parameters, probe the flow field and, as thoroughly as possible, interactively investigate their numerical results. This objective came in part from experiences in developing interactive educational software to aid in the undergraduate teaching of fluid dynamics.[7] One consequence of the desire for high interactivity was a decision to sacrifice portability for performance. At the lowest levels, the programs use graphics primitives that are unique to the Stardent hardware, although in the near future these will be converted to the emerging PEX standard. Another consequence was a strong effort to develop innovative "probes" with which the user can interrogate the flow field.

- Animation—VISUAL2 was designed from the beginning to handle unsteady data, in which the flow variables, the grid coordinates and even the basic grid data structure could be changing. It was also designed to allow co-processing, in which the CFD calculation is performed in parallel to the animated visualization, requiring locking procedures to transfer data from the CFD application to VISUAL2. VISUAL3 was also designed to allow animation although at present the computer floating point performance is not sufficient to make this practical in many situations.
- Structured and unstructured grids—an important objective was for the programs to be capable of handling flow data from all of the types of flow codes currently in common use in CFD. This includes multi-block structured grids, and unstructured grids with a variety of different cell types. However, this generality should not be at the expense of performance; instead, if necessary, it should be at the expense of additional programming.
- Flexibility and generality—the intent was that a sophisticated user should be able to modify or tailor the graphics package to some extent. Also, the same graphics package should be able to treat data from a structural analysis program as well as from a CFD program.
- Simple user programming interface—one approach to graphics is to supply the user with a library of graphics functions which they then assemble into a graphics program. At the other extreme is the graphics package which does everything but is not very flexible. As explained in the next section, a middle approach was adopted for VISUAL2 and VISUAL3.

2.2. *Program structure and interface*

The next step in the program development was to decide upon the program structure, or architecture. This is probably the most critical decision since the choice places tight constraints on the functionality and ease-of-use of the final software.

One possibility was to write a library of graphics routines, consisting of perhaps as many as 50 subroutines, which a user could build into a graphics application package. The AVS (Application Visualization System) software developed by Stardent is a very good example of this type of software.[8] AVS uses a visual "network editor" to allow a user/programmer to assemble building blocks into a powerful graphics package. This is excellent as a rapid prototyping tool for quickly developing graphics for entirely new applications, and is used by many people as the final production graphics program. However, it has to make sacrifices in performance and memory usage and currently supports only a limited set of data structures.

At the opposite extreme, one could write a single program (like PLOT3D), which handles structured and unstructured grids and has a preset number of plotting options. PLOT3D has certainly been very successful because it is very easy for a novice user to begin to use it, and this is an attraction of the single program approach. The weakness is the lack of flexibility and generality. If a user wants to plot some variable other than the ones which are preprogrammed into PLOT3D then he must modify the internal code of PLOT3D. If the user wants to use PLOT3D to view structural analysis results, then extensive modifications need to be made. For this reason we chose not to adopt this approach.

Lying between these two approaches is GRAFIC, developed by Dannenhoffer.[4] GRAFIC has a number of pre-packaged options to do straightforward things such as line plots and contour plots on two-dimensional structured grids. In addition, to give more sophisticated users full flexibility, there is a mode in which the user program calls a GRAFIC "control" package, which handles axes, annotation, hard copy, blowups, etc. and calls user-supplied routines to draw all plots. This capability allows users to write custom software for plotting on their particular types of unstructured grids, using a library of low-level routines supplied with GRAFIC. This approach is a little like the graphics library, except that the control program relieves the programmer of a lot of mundane tasks. It was felt that this approach still required too much user programming, but the "control" program architecture was the major influence on the choice of structure for VISUAL2 and VISUAL3.

The program architecture finally chosen is shown in Fig. 1. VISUAL2 and VISUAL3 are subroutines which are called by a user's program, and they perform all graphics functions, including window management, grid plots, contour plots, hard copy, cursor control, "probes", etc. Keeping the user away from all low-level graphics programming achieves the goals of ease-of-use, and allows the VISUAL programs to concentrate on providing interactivity, animation and overall high performance.

To provide more flexibility than with PLOT3D, VISUAL does not deal directly with CFD variables. Instead it only processes generic scalar and vector data. When it is initialized by the user's program, the

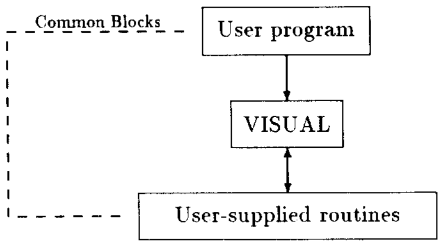


Fig. 1. Flow diagram showing direct use of VISUAL.

user "binds" a keyboard key to a particular user function, and notifies VISUAL of the type of the function. For example, a user program may tell VISUAL that the key "p" is associated with a scalar function labeled "pressure." If the user subsequently presses the key "p," then VISUAL loads in the values of "pressure" by calling a user-supplied routine. The user-supplied routine can communicate with the top-level user program through COMMON blocks, and it knows how to calculate pressure from the user's flow data. In this manner, the user can easily add new functions to be plotted without having to change the internal program of VISUAL. This satisfies the goal of flexibility and generality.

Similarly, VISUAL obtains the data structure and grid coordinates by calling user-supplied routines. If the initialization of VISUAL states that some or all of the data structure, grid and flow data are unsteady, then VISUAL will keep calling (either synchronously or asynchronously) the user-supplied routines to obtain the latest data. This is the simple mechanism by which animation is achieved.

The final comment on the program structure is that the internal data structures used by VISUAL2 and VISUAL3 (which are explained in the next section) may not be the same as the user's chosen data structure. In the most general case, it is the user's responsibility to convert the user's data to VISUAL's internal format. However, a set of CFD application filter programs, called FLOVIZ, is being written for users to use for this purpose. For example, one set of FLOVIZ routines will take multi-block data in the PLOT3D format, convert it into the VISUAL internal format, and define the plotting functions commonly offered by PLOT3D. In this case, the user's program would simply call the top-level FLOVIZ routine, as shown in Fig. 2. The filter programs are easy to write in general, and can be customized for different applications such as structural analysis.

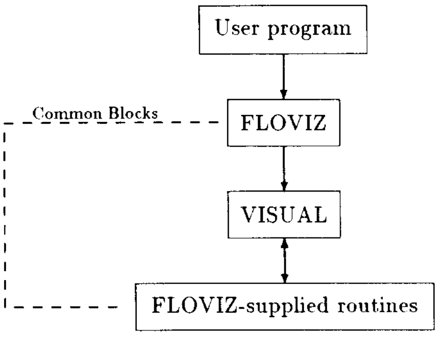


Fig. 2. Flow diagram showing indirect use of VISUAL through FLOVIZ.

2.3. *VISUAL2 data structure*

The choice of internal data structure for VISUAL2 and VISUAL3 was driven by considerations of generality and the maximum possible performance. In two dimensions, every computational cell is being plotted in general, and so the ratio of rendering time (the time needed to do a smooth Gouraud-shaded color fill of a polygon) to compute time (the time needed to evaluate new functions, such as pressure) is large. Therefore, the data structure of VISUAL2 was optimized to obtain the maximum graphics throughput on the Stardent GS2000. This requires a decision which is somewhat machine-dependent. The graphics primitive on the Stardent, which allows a rendering rate of 150,000 Gouraud-shaded triangles per second, uses a structure called a "polytriangle." Other manufacturers use slightly different primitives, but the polytriangle is one of the structures which is the basis of PEX (the Phigs Extension to X windows), which is likely to emerge as the standard of high performance graphics.

The polytriangle is a list of $N+2$ nodes which defines N connected triangles, with nodes 1, 2, 3 defining the first triangle, nodes 2, 3, 4 the second, and nodes N, $N+1$, $N+2$ the Nth. The grid coordinates and function values are defined at the nodes. In this way the polytriangle, for large N, requires approximately one node of coordinate and function data per triangle. This contrasts with using a set of disjoint triangles in which each triangle requires three nodes of data. Thus, the use of polytriangles reduces memory requirements by factor 3, reduces bus transfer requirements by factor 3, and also reduces the triangle processing time because when processing one triangle the low-level routines can take advantage of results from processing the previous triangle.

The primary part of the VISUAL2 data structure is a set of polytriangles which collectively define the entire computational grid. The streamline integrator (which simply integrates any vector function given to VISUAL2) requires connectivity information. The polytriangle structure immediately gives neighbor information for all the common faces internal to it. For the $N+2$ external faces, there is a set of pointers which point to the corresponding cell in another polytriangle with the shared face. If there is no neighboring cell, because it is an edge face, then it points into a separate structure which is a collection of all of the edge faces, grouped into specific edges (e.g. airfoil, inflow boundary, outflow boundary). The edge structure is important because certain functions, such as skin friction, are only defined on an edge, and there are special plotting capabilities to display these.

One complication with the use of polytriangles as the primary data structure is the task of converting user grids into polytriangles. The conversion of structured grids is trivial, since consistently cutting all quadrilateral cells along the same diagonal will naturally produce polytriangles. The harder task is taking a general unstructured grid composed of triangular and quadrilateral cells and decomposing it into polytriangles. A very efficient algorithm has been developed for this. A simpler version which handles only triangular cells is presented in the algorithm section. This is implemented as a routine which can be used either by the FLOVIZ filter routine, or directly by a user's program.

2.4. *VISUAL3 data structure*

In three dimensions, only a small fraction of the computational cells are being displayed in some manner on the screen at one time. Therefore the ratio of rendering time to compute time is small, and the choice of data structure is motivated by the desire to minimize compute time, particularly in some of the more CPU-intensive functions such as volume slicing. In the same way that triangles are the lowest common denominator in two dimensions, tetrahedra are the lowest common denominator in three dimensions, and in principle all other cell types could be split into a number of tetrahedra. Since this would simplify our programming task we considered this approach but rejected it for the following reason.

The problem is how to split a hexahedron into tetrahedra. The smallest number of tetrahedra that the hexahedron can be split into is five, achieved by dividing each face into two triangles in the correct manner to form four tetrahedra with three external faces and one tetrahedron with four internal faces. The difficulty with this is the possibility of an inconsistency between neighboring hexahedra. If the common quadrilateral face shared by two hexahedra is split across one diagonal on one cell and the other diagonal on the other cell, and if the quadrilateral face is twisted (i.e. non-planar), then there will be an overlap and some gaps in the volumetric decomposition into tetrahedra because the common face will be represented differently on either side. In structured grids, one can ensure a consistent splitting, and this approach is used by Strid and Rizzi.[6] In unstructured grids one cannot solve this problem, except by splitting each quadrilateral face into four triangles by inserting a new node at the centroid of the face, and joining this node to all four corners. If each resulting triangle is then connected to another node at the centroid of the entire cell, then 24 tetrahedra are produced. This will clearly greatly increase the computational cost of all operations, and the memory requirements.

Instead, our approach was to use a data structure for VISUAL3 which has four different cell types; tetrahedra, pyramids, prisms and hexahedra. Almost all CFD grids in use today are a combination of one or more of these cell types, and the few that are not can be easily decomposed into these. This data structure keeps memory requirements to a minimum for an unstructured grid, and leads to computationally efficient algorithms tailored for each cell type. The only drawback of this approach was the extra programming that had to be done to handle each cell type.

As in two dimensions, there are also other components to the data structure. There are pointers from each cell to its neighbors, to provide the connectivity information needed by the streamline integrator. There is a list of surfaces (e.g. wing, fuselage, far-field boundary), and for each there is a list of surface faces, which are treated as a set of disjoint triangles because in this case there is no problem about an inconsistency in splitting quadrilateral faces. In the object-oriented computer science terminology, the surfaces are treated as static objects in the object database. "Static" means that the definition of the object (its list of faces) does not change. The "attributes" of the object (whether it is being rendered, whether the grid plotting is on or off) can be changed by the user. A "dynamic" object is created by the volume slicing, either by moving a cutting plane through the volume or by defining an iso-function surface. It is dynamic because as the location of the plane moves, or the value of the iso-function surface changes, the list of cells defining the object will change. To create an image with multiple slicing planes or iso-function surfaces, there is a capability to take the dynamic object at some instant and copy it to a static object in the database. Additional static objects created in this way can be deleted later to free memory.

2.5. *User interface*

Figures 3 and 6 show the screen displays of VISUAL2 and VISUAL3. All graphics are handled through the X-window display system.

VISUAL2 has two main windows. The large one (at top left) is the primary window in which most plotting is performed. The small one (at bottom right) is the secondary window which is used for plotting one-dimensional data, and the "magnifying glass." There is also a window (at top right) with the color map, which defines the color associated with a particular function value. At the bottom left is the text window which displays the help menus and is used to accept numeric input data from the user.

VISUAL3 has three main windows. The largest one (at top left) is the primary window in which all three-dimensional plotting is performed. The slightly smaller one (at middle right) is the secondary window which is used for plotting two-dimensional data, and the smallest one (at bottom right) is used for one-dimensional data. Again there is also a window (at top right) for the color map, and a text window. The last window displays either the functions of the eight dials on the dialbox, or the state of the "objects" in the database. The mouse can be used to edit the database, changing which surfaces are rendered, which have grids displayed, etc.

Our personal opinion is that pull-down menus controlled by a mouse clutter up the screen and are cumbersome to use. Therefore, most user interaction with VISUAL is through the keyboard. Function keys and assorted other special keys control all plotting options. The option invoked also depends upon the window in which the cursor is currently located, as is standard in most X-based applications. Alphanumeric keys are used to define which scalar or vector function is to be plotted, as determined by the key bindings that the user specifies when initializing VISUAL. This provides a great deal of functionality without requiring multiple levels of menus. VISUAL3 also uses the dialbox to input rotations and value changes.

3. INTERACTIVE PROBES AND CAPABILITIES

3.1. *VISUAL2*

VISUAL2 has a set of basic plotting options which generate output in the primary window. These are supplemented by the ability to interactively pan and zoom, and change the color maps.

- Contour plot—a Gouraud-shaded contour plot of the currently "active" scalar variable is generated. This is dynamic if the grid or the function is unsteady. It is shown in Figs 3 and 4.
- Grid plot—the computational grid is superimposed on the current plot.

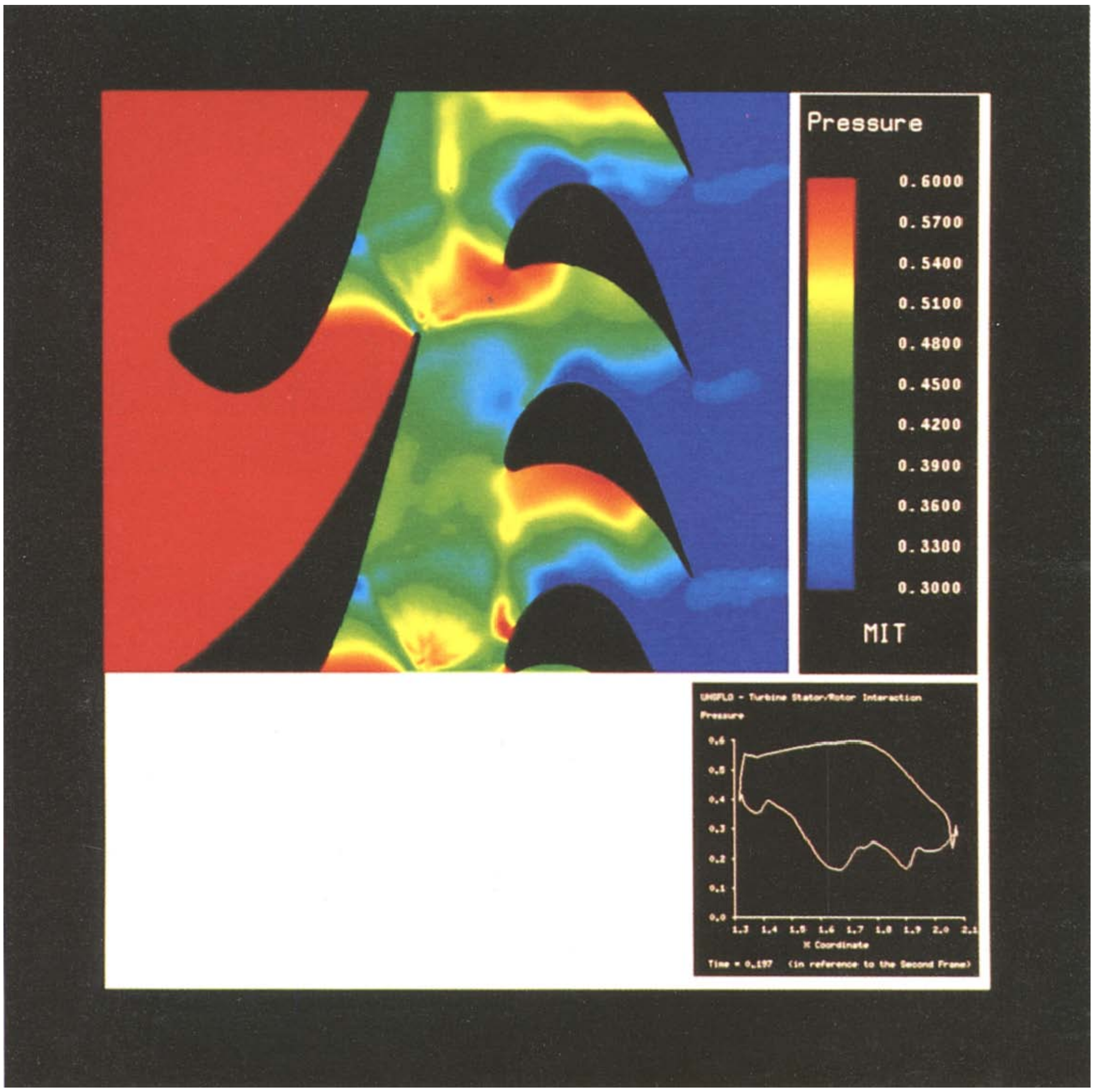


Fig. 3. VISUAL2 with surface line plot.

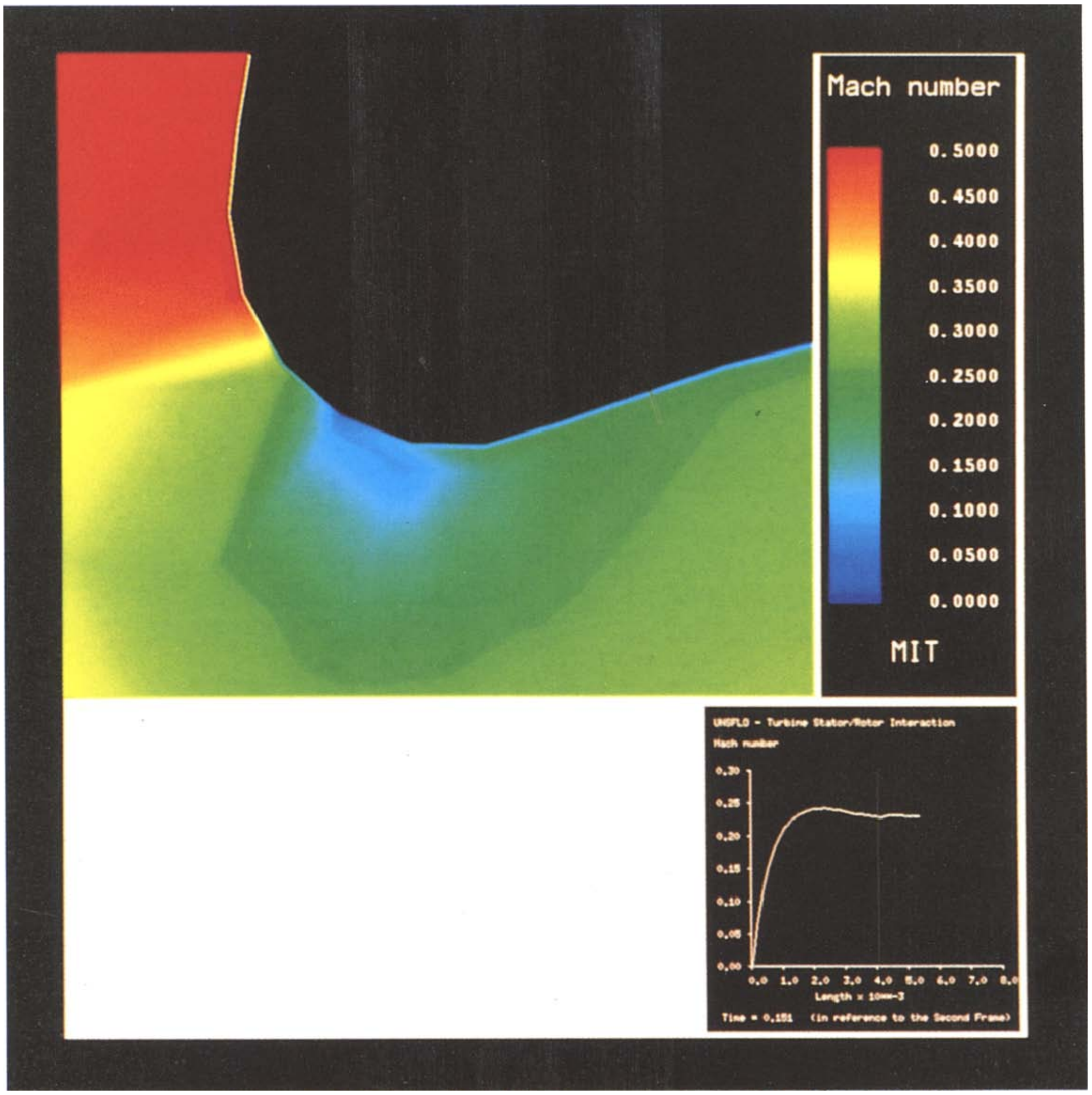


Fig. 4. VISUAL2 with edge normal probe.

- Contour line plot—contour lines corresponding to the current scalar function are superimposed on the current plot.
- Vector plot—vectors corresponding to the current vector function are superimposed on the current plot. This is shown in Fig. 5.

At the screen interface level, the novel feature of VISUAL2 is the large variety of interactive "probes" which can be used by the viewer to interrogate the numerical data. The output from these probes is displayed in the secondary window. An important point is that many of these probes were developed as a direct consequence of suggestions from users who wanted to study some particular aspect of a flow field, and felt that the existing tools were not adequate.

- Point probe—the position of the probe in the primary window is defined by the mouse. The output in the secondary window is a time history of the scalar variable being plotted in the primary window.
- Edge function probe—this is similar to the point probe, but is for plotting edge function data (data like skin friction or heat transfer which is only defined on edges). The probe is defined to be at the edge point closest to the current mouse location.
- Edge plot—this plots the current scalar variable along the edge which is closest to the current mouse location. It is shown in Fig. 3.
- Edge function plot—this is similar to the edge plot, but is for plotting edge function data.
- Line probe—the position of the line is controlled by the mouse. The ouput in the secondary window is the steady or unsteady scalar function

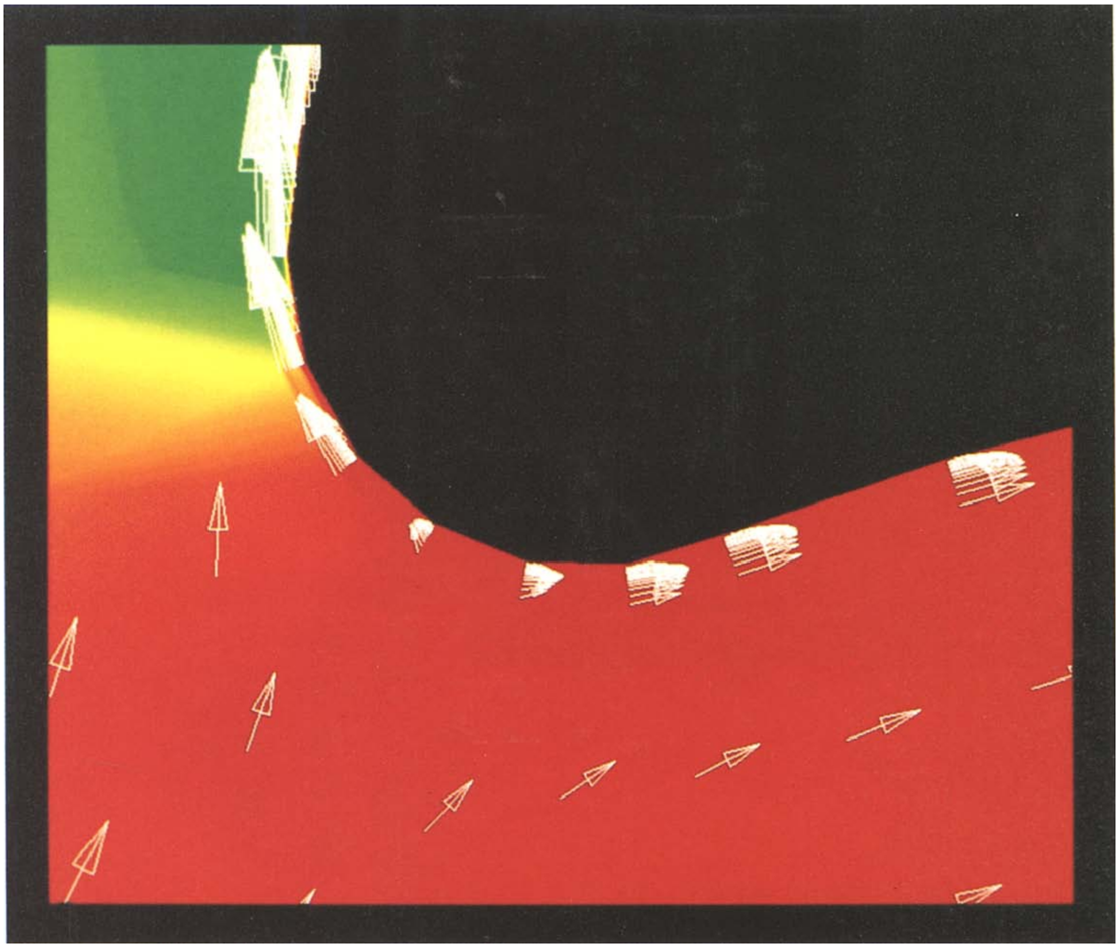


Fig. 5. VISUAL2 with velocity vectors.

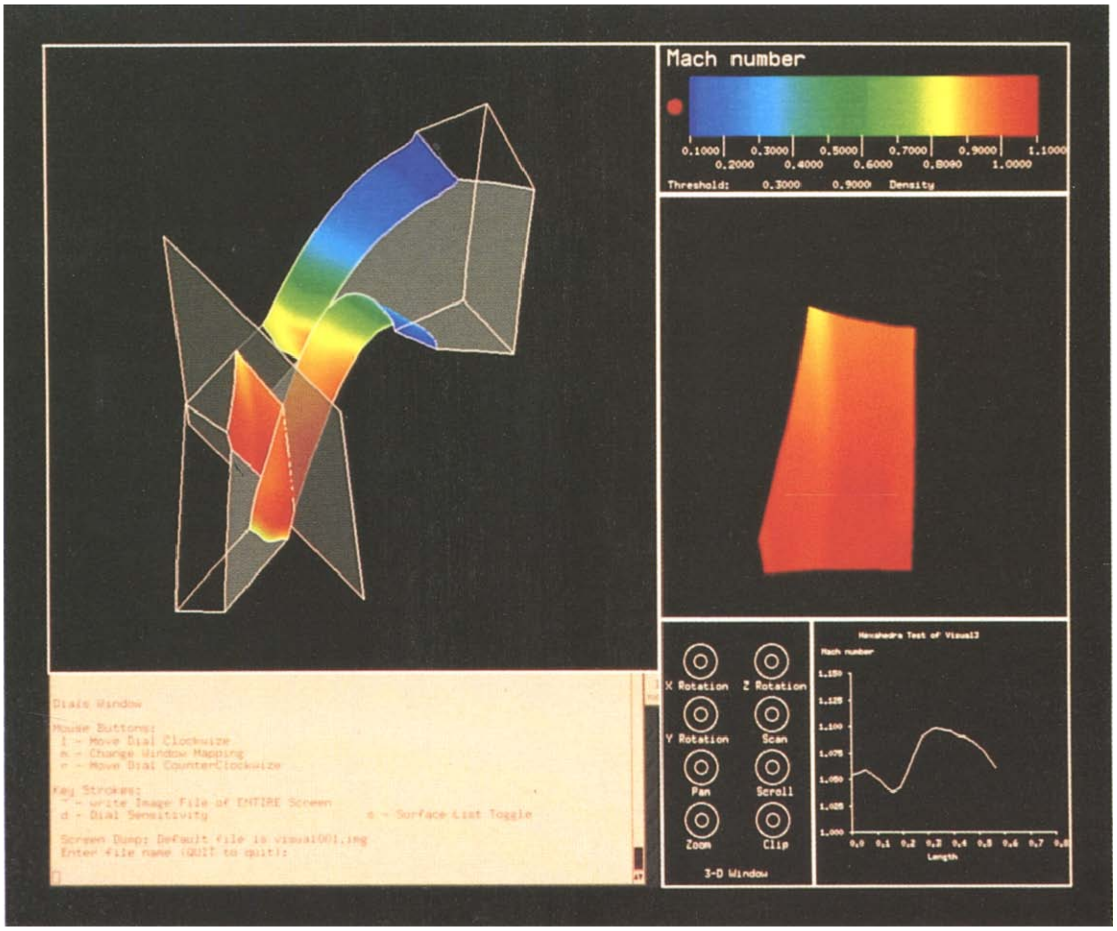
values along the line. A subsidiary option for unsteady cases is to calculate and plot a time-average.

- Edge normal probe—this is similar to the line probe, except that the line is normal to an edge. Like the edge function probe, one end of the line is defined to be at the edge point closest to the current mouse location. The line then extends normally out from the edge into the domain. The initial length is set by the initial distance of the mouse from the edge, but it can be changed by the user. The location of the whole line varies dynamically as the user moves the mouse, allowing the user to quickly scan around an entire surface examining, for example, the boundary layer in a viscous CFD calculation. This is shown in Fig. 4.
- Magnifying glass—this is similar to the interactive pan and zoom, except that the primary window remains unchanged. The output in the secondary window is the magnified region. For efficiency reasons, if the data is unsteady this option "freezes" the action and does not plot dynamically.

3.2. *VISUAL3*

VISUAL3 has a set of basic plotting options which generate output in the primary three-dimensional window. Simple dialbox commands allow the user to rotate, pan and zoom, and the user can again interactively change the color maps. For convenience in comparing different solution sets, viewing positions can be stored away, and/or recalled.

- Surface contour plot—a Gouraud-shaded contour plot of the currently "active" scalar variable is generated on all selected surfaces. An additional option is thresholding, in which the contour plot is only given on those parts of the surface on which a thresholding function lies within certain bounds. If the thresholding function is the same as the plotting function this gives the form of thresholding first developed by Weston.[3] If the thresholding function is geometric this produces a "cutaway" view in which part of the surface is removed to enable one to see another part. Other options are to make the rendering translucent (allowing surfaces behind to be partially visible) or to add a simple lighting model (giving valuable cues about



```
Dials Window

Mouse Buttons:
 l - Move Dial Clockwize
 m - Change Window Mapping
 r - Move Dial CounterClockwize

Key Strokes:
 ^ - Write Image File of ENTIRE Screen
 d - Dial Sensitivity                        s - Surface List Toggle

 Screen Dump: Default file is visual001.img
 Enter file name (QUIT to quit):
```

Fig. 6. VISUAL3 with cutting plane.

surface curvature and depth). This is shown in Figs 6–9.

- Surface function contour plot—this is similar to the surface contour plot, except that the scalar quantity is a surface function which is only defined on the surface.
- Surface grid plot—the computational grid is displayed on all selected surfaces. This is shown in Fig. 7.
- Surface vector plot—vectors corresponding to the current vector function are displayed on all selected surfaces.

VISUAL3 has a number of probes with output in either, or both, of the primary three-dimensional window and the secondary two-dimensional window.

- Cutting plane—this is a flat plane whose orientation relative to the computational object is interactively set by the user using the dialbox. On the plane, one can plot the computational grid, or contours of the current scalar variable, or "tufts" corresponding to the current vector variable, or begin streamlines which are the integrals of the current vector function. The output of these options can be plotted in either, or both, of the three- and two-dimensional windows. A rendered cutting plane is shown in Fig. 6, and a cutting plane with streamlines is shown in Fig. 8.
- User-defined cutting plane—this is very similar to the cutting plane, except that the plane is defined by $z' = \text{const}$, where x', y', z' are user-defined functions of the physical coordinates x, y, z. This allows, for example, a user to display contours on an axisymmetric surface which is useful in turbomachinery applications. Using the dialbox, the user can adjust the value of z' to move the cutting plane through the field.
- Iso-function surface—this is similar to the user-defined cutting plane, with z' defined to be the current scalar function value. Since there is no way to define x', y', this option plots the iso-function surfaces only in the three-dimensional window. The iso-surface value can be varied using the dialbox. This is shown in Fig. 9.

When one of the cutting planes is being plotted in the secondary window, many of the probes in VISUAL2 are available to interrogate the two-

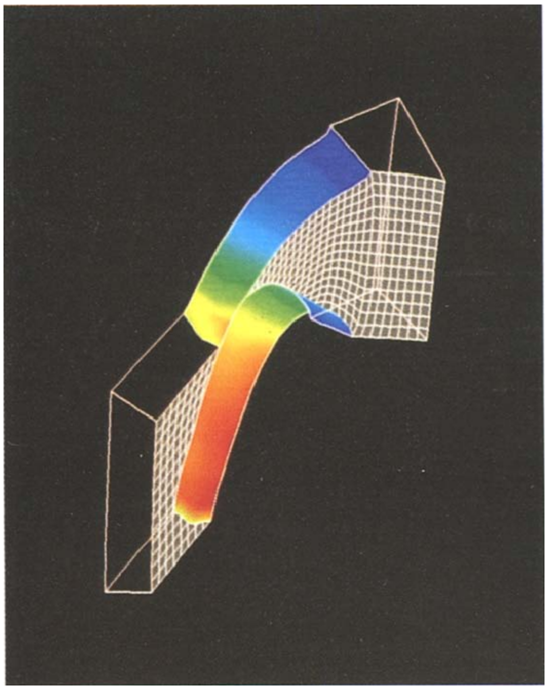


Fig. 7. VISUAL3 with grid plot.

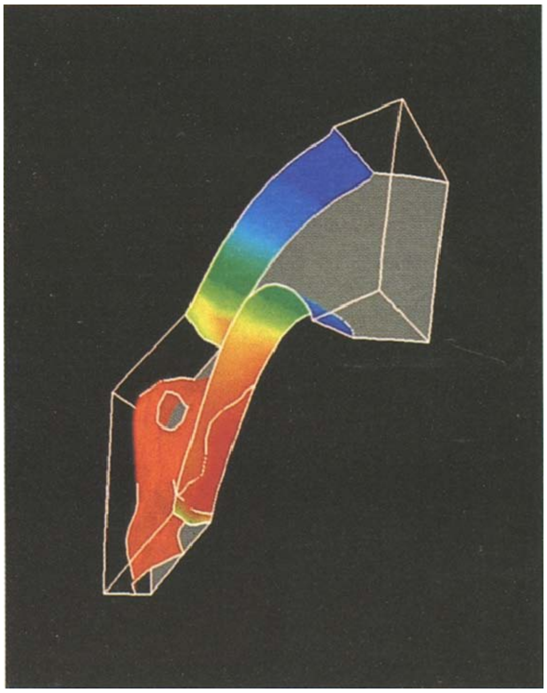


Fig. 9. VISUAL3 with iso-function surface.

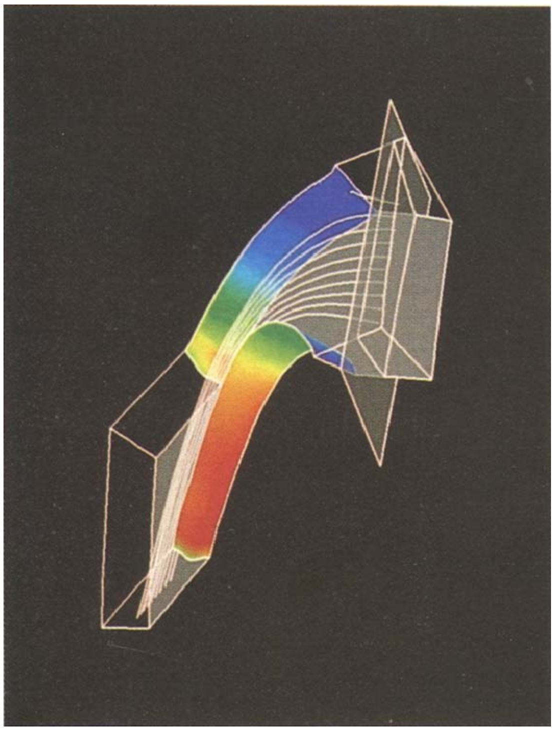


Fig. 8. VISUAL3 with streamlines.

dimensional flow field, with output being displayed in the small tertiary window.

4. ALGORITHMS

4.1. *Two-dimensional polytriangle strip generator*

The objective of the two-dimensional polytriangle strip generator is to take an unstructured grid composed of triangular cells, and convert it into a number of polytriangle strips. This is achieved in two stages. The first stage constructs a table of information about all grid faces, and the second stage uses this to form the strips.

The face table to be constructed has six entries for each face. The first two entries are pointers to (or the indices of) the two nodes that define the face. The third entry is a pointer to the next face in the list which contains the first node. Similarly, the fourth pointer is to the next face which contains the second node. The third and fourth pointers are zero if the current face is the last face which involves the corresponding node. The last two entries are pointers to the cells on either side of the face. If the face is on an edge then there will be only one cell and so the second entry will be zero. In addition to the face table there is a node table with one entry per node which points to the first face which involves that node.

When the initialization process begins, the node and face tables are all zeros. They are filled up progressively by processing all of the cells in the domain. Each cell has pointers to the three or four nodes which define it, and these in turn define three or four faces. For each face the process is as follows.

Let I1 and I2 be the indices of the two nodes that define the new face. The node table is used to see if there is already a face in the face table which involves node I1. If there is not, then this new face is added to the face table by setting the first two entries equal to I1 and I2, and the fifth to the cell index. If there is, then the other node of the face in the table is compared to I2. If these match, then the table face is the same as the new face and so the current cell is the second cell corresponding to that face and the sixth entry is set appropriately. If they do not match then the whole process is repeated with the next face in the table which uses the first node of the new face, using the appropriate third or fourth pointer to find this face. This continues until one either finds a match for the new face, or runs out faces which use the first node. In the latter case the new face does not currently exist in the face table and so it is added to the face table. Also, the third or fourth entry (as appropriate) of the face that used to be the last involving node I1 is set to point to the new face in the table which is now the last.

When the face table is complete, all faces with only one cell are labeled as being edge faces. The construction of polytriangles requires an additional table for the cells. If a cell has not yet been included in a polytriangle then the entry is zero. If it has been used the entry points to the cell's location in the polytriangle structure. The first part of the strip generator starts at edge faces and works into the domain. The edge face points to the neighboring cell; the cell points to its three nodes and so one obtains the index of the interior node, the one not in the edge face; using the node and face tables the node points to the face which is then the starting point for adding the next triangle to the strip. This continues until one reaches another edge face or a triangle which has already been used, which ends the strip. Once all edge faces have been used to start strips, there may still be some cells that have not been used. In this case additional strips are started from each unused cell (extending in both directions) until all cells have been used.

There are some minor complications. As indicated in Fig. 10, a strip that starts at an edge face can grow inwards in two ways, depending on the order in which one takes the two edge nodes. The algorithm checks both possibilities and takes the one which gives the longer strip. Similarly, starting at an unused interior cell, there are three different possible strip orientations, and one chooses the one giving the longest strip. The final operation is to use the information in the node, face and cell tables to construct the connection data between the polytriangles; the cell in one polytriangle points to its nodes, which point to the face, which points to the neighboring cell, which points to its location in the neighboring polytriangle.

This strip generation algorithm is very efficient. Each node is involved in only six faces on average, so chasing through the face table to find and match faces is extremely rapid. For a grid with N triangles, the memory requirements are $O(N)$, and the table generation and the strip generation phases both require $O(N)$ operations. Only 2 s are required on a Stardent GS2000 to generate strips for an irregular triangular grid with 100,000 cells. This is comparable to the disk I/O time for reading in the data set, so it is re-calculated every time instead of being stored on disk.

A very similar procedure is used in three dimensions to take an unstructured collection of cells and compute the neighboring cells and a list of surface faces.

4.2. *Three-dimensional volume slicing*

The task of the three-dimensional volume slicing algorithm is to extract two-dimensional surface information from an unstructured three-dimensional data set. This procedure is used for the planar cutting plane, user-defined cutting plane, and iso-function surfaces listed which are three of the probes in VISUAL3. In all three cases, the description of the problem can be reduced to the following; given some function z', and a set of unstructured three-dimensional cells, how does one firstly determine the cells that are crossed by the surface $z' = Z$, and secondly, for the crossed cells determine the surface piece to be plotted.

Taking the second task first, there is an extremely fast method referred to as the "marching cubes" algorithm, developed by Lorensen and Cline,[9] and used by Löhner, Strid, Dannenhoffer and ourselves. Briefly, the technique for hexahedra is to calculate an eight-bit index for a cell, where each bit is 0 or 1 depending whether the corresponding corner node value of $z' - Z$ is positive or negative. This eight-bit index then points to an entry in a look-up table that gives the logical structure of the surface $z' = Z$. Interpolation of geometry and function values along an edge completes the process.

The harder task is the first task of determining the cells that are crossed. An exhaustive search of all cells is possible but extremely time-consuming. Strid improves the efficiency by performing the exhaustive search for all boundary cells, and then constructing the surface from the edges in, by using cell connectivity information to check neighboring cells to see if they are crossed.[6] Although not vectorizable, this works well for planar cutting planes. However, it does not work for user-defined surfaces and iso-

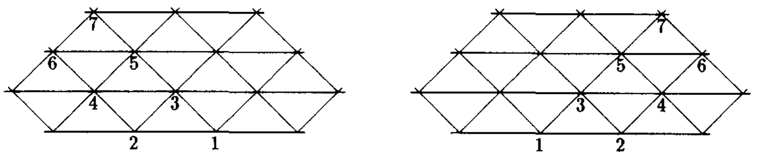


Fig. 10. Alternative polytriangle strips.

surfaces which do not cross the boundaries of the computational region.

Our approach, part of which was published in an earlier paper,[10] begins with an initialization phase when the function z' is first defined. For each cell, j, $z'_{j,\min}$ and $z'_{j,\max}$, the minimum and maximum values of z' of the corner nodes, are calculated. Two ordered lists of cells are formed, $L_{\min}$ ordered by $z'_{j,\min}$, $L_{\max}$ by $z'_{j,\max}$. The final initialization step is to evaluate $\Delta z'$, the global maximum value of $z'_{j,\max}-z'_{j,\min}$, which is the maximum cell "width." If there are N total cells then the initialization requires $O(N\log(N))$ operations using a quick-sort algorithm. For $N = 100{,}000$, the initialization currently takes 15 s on the Stardent GS2000, but it is hoped that this will be reduced in the future by using fully-optimized assembly-level programming for the critical sort algorithm.

If the constant Z is being set for the first time, or if it has changed from its previous value by more than $\Delta z'$, then an active cell list L_{active} is formed by taking the section of list $L_{\min}$ with cells whose $z'_{j,\min}$ lie in the range $[Z-\Delta z', Z]$. This requires only $O(\log(N))$ operations to find the limits, and in general the active cell list has $O(N^{2/3})$ cells. Purging the active list of the few cells that are not crossed by $z' = Z$ gives the final list of crossed cells.

If Z has changed by less than $\Delta z'$ then a different procedure is used. If $Z_{\text{new}} > Z_{\text{old}}$, the old active cell list is supplemented by the cells in list $L_{\min}$ that have values of $z'_{j,\min}$ in the range $[Z_{\text{old}}, Z_{\text{new}}]$. If $Z_{\text{new}} < Z_{\text{old}}$, the old active cell list is supplemented by the cells in list $L_{\max}$ that have values of $z'_{j,\max}$ in the range $[Z_{\text{new}}, Z_{\text{old}}]$. The active cell list is then purged of all cells that are not crossed. For small changes in Z, this procedure is extremely efficient since it involves the addition and removal of just a few cells. This enables a rapid animation rate when interactively varying the value of Z smoothly to sweep through the entire computational domain.

4.3. *Three-dimensional streamlines*

The algorithm for integrating a vector function to produce streamlines is very similar to the techniques used in GRAFIC and PLOT3D, and differs only slightly from the method used by Strid and Eliasson.[6] The algorithm can be broken into two parts, a top-level part which performs the time-integration of the vector function, and a low-level part which is responsible for evaluating the interpolated velocity at an arbitrary point in space.

The top-level task is relatively straightforward. Assuming that somehow one knows the velocity field $\vec{u}(\vec{x})$, then the streamline, defined parametrically as a function of time t by the equation

$$\frac{d\vec{x}}{dt} = \vec{u}(\vec{x}), \qquad (1)$$

is integrated numerically by a fourth-order Runge–Kutta method. To obtain accurate streamlines at the lowest possible cost, an adaptive time-step is used, with the time-step being halved if the velocity direction changes too much over one time-step, and doubled if it changes sufficiently less.

The low-level task is to calculate $\vec{u}$ for an arbitrary $\vec{x}$ on an unstructured grid. The explanation will be solely for hexahedral cells, but the extension to the other cell types is quite natural.

For each hexahedral cell there is a trilinear mapping from a unit cube ($0<\xi_1<1$, $0<\xi_2<1$, $0<\xi_3<1$) in computational space to the hexahedron in physical space. This can be represented as

$$\vec{x} = \sum_{j=1}^{8} f_j(\xi_1, \xi_2, \xi_3)\vec{x}_j, \qquad (2)$$

where the sum is over the eight corner nodes, $\vec{x}_j$, is the coordinate vector of the jth node, and f_j is a trilinear function (linear in each of ξ_1,ξ_2,ξ_3) which is equal to 1 at node j and 0 at all other nodes. Using an isoparametric representation, the velocity field can also be expressed as

$$\vec{u} = \sum_{j=1}^{8} f_j(\xi_1,\xi_2,\xi_3)\vec{u}_j. \qquad (3)$$

Thus the task for the low-level routine has been reduced to finding the cell in which the desired $\vec{x}$ is located, and then finding the corresponding value of $\vec{\xi}$ and hence $\vec{u}$. The second part of this is accomplished by a Newton–Raphson iterative solution of Eq. (2). The Newton–Raphson update is clipped to ensure that $\vec{\xi}$ does not go outside the computational cell. If $\vec{\xi}$ lies on the computational cell boundary, and the update still wants to go outside the cell, then this proves that the desired $\vec{x}$ lies in another cell. Using the cell connectivity information the search switches to the neighboring cell and the Newton–Raphson procedure is restarted. The initial cell used for the search is taken to be the cell corresponding to the last point calculated in the top-level Runge–Kutta integration. Thus the searching procedure will typically only need to move through one or two cells before finding the correct cell, and then converges quadratically to the correct value of $\vec{\xi}$.

5. CONCLUSIONS

This paper has discussed the development of two graphics programs, VISUAL2 and VISUAL3, for the visualization of scalar and vector data in two and three dimensions, respectively. A strong effort was made to explain the reasons behind the choice of data structure and program architecture that was employed, and to present the pros and cons of alternative approaches. This is a critical part of the design of a graphics package and deserves considerable thought before proceeding to the programming stage. The best approach is to first formulate a clear set of design goals for the functionality of the software. Together with the capabilities and limitations of the hardware, this then gives one a good basis on which to optimize the program design.

The original scientific contributions of this paper lie in two areas. The first is a variety of innovative "probes" for interrogating the numerical data. Although most of these probes are not difficult to implement, and so do not represent major breakthroughs, collectively they provide a set of visualization tools which is much more powerful than existing graphics programs. The second area is in algorithms for visualization on unstructured grids; the generation of two-dimensional polytriangle strips from an unstructured set of triangular cells, and the calculation of iso-function surfaces and arbitrary cutting planes in three dimensions.

Acknowledgements—Professor Earll Murman, Mr David Modiano, Mr David Darmofal and Dr John Dannenhoffer have all contributed greatly to this research through critical review of the software design at an early stage, many suggestions for improvements at later stages, and ideas on new "probes" and interactive capabilities. DARPA and Stardent Inc. largely financed the purchase of the GS2000 superworkstations without which this work would not have been possible.

REFERENCES

1. P. Buning and J. L. Steger, "Graphics and flow visualization in computational fluid dynamics," AIAA Paper 85-1507, 1985.
2. T. Lasinski, P. Buning, D. Choi, S. Rogers, G. Bancroft and F. Merritt, "Flow visualization of CFD using graphics workstations," AIAA Paper 87-1180, 1987.
3. R. F. Weston, "Color graphics techniques for shaded surface displays of aerodynamic flowfield parameters," AIAA Paper 87-1182, 1987.
4. J. F. Dannenhoffer, "GRAFIC—an interactive graphics package for scientific applications," Technical Report 89-39, United Technologies Research Center, East Hartford, CT, 1989.
5. R. Löhner, P. Parick and C. Gumbert, "Some algorithmic problems of plotting codes on unstructured grids," AIAA Paper 89-1981, 1989.
6. T. Strid and A. Rizzi, "Development and use of some flow visualization algorithms," VKI Lecture Series on Computer Graphics and Flow Visualization in CFD, 1989.
7. E. M. Murman, A. R. LaVin and S. C. Ellis, "Enhancing fluid mechanics education with workstation based software," AIAA Paper 88-0001, 1988.
8. C. Upson *et al.*, "The application visualization system: a computational environment or scientific visualization," *IEEE Computer Graphics and Applications* (1989).
9. W. E. Lorensen and H. E. Cline, "Marching cubes in a high resolution 3D surface construction algorithm," *ACM Computer Graphics* **21(4)**, 163–169 (1987).
10. D. L. Modiano, M. B. Giles and E. M. Murman, "Visualization of three-dimensional CFD solutions," AIAA Paper 89-0138, 1989.