Visual3 - A Software Environment for Flow Visualization

Robert Haimes and Mike Giles and Dave Darmofal
Computational Fluid Dynamics Laboratory
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology, Cambridge, MA 02139

Computer Graphics and Flow Visualization in Computational Fluid Dynamics
VKI Lecture Series #10, Brussels, 16-20 September 1991.

Abstract

A graphics environment has been developed for the visualization of three-dimensional engineering and scientific data. Unlike existing packages, it can handle a wide variety of structured and unstructured grids, and deals with generic scalar and vector data, making it suitable for a wide range of physical applications. An important feature of the package is the ability to view unsteady data in either post-processing or co-processing modes. Another unique aspect of **Visual3** is the dimensional windowing approach coupled with cursor mapping which allows efficient pointing in 3D space. There is also a strong emphasis on a wide variety of 'probes' allowing the user to interactively investigate the computational data.

In that this is a discussion of an interactive visualization environment, these notes cannot be considered complete without a live demonstration of **Visual3**.

NOTE: Sections of these notes have been extracted from [1] [2] [3] and [4].

Contents

1 Introduction

Scientific visualization is an area which has grown greatly in size and importance in the last five years. This growth has been driven by a number of factors. One is the change in the types of calculations which are being done by CFD researchers. Ten years ago, the standard CFD calculation would be of steady, 2D inviscid flow, which could be easily understood by straightforward 2D computer graphics (e.g. monochrome contour plots, line plots of surface pressure, etc.). Now, 3D, unsteady, and/or viscous calculations are becoming common, and these are very much more difficult to imagine or visualize in our minds, let alone on a two-dimensional computer screen. Therefore, the development of tools for the visualization of such flows is very important if we are to properly understand the results of our calculations.

A related driving force is the huge increase in computer power over the last decade. A numerical simulation can be split into three phases: pre-processing (defining the geometry and constructing the computational grid), computation and post-processing (trying to understand the computational results). Ten years ago, most of the time and cost was in the computation phase. Due to faster computing speeds and much more complex geometries and flows, most of the time is now spent in the pre- and post-processing phases. Also, decreasing computer costs and increasing salaries mean that a larger fraction of a company's overall computing costs comes from personnel costs. Therefore, for industry which is both cost-conscious and anxious to reduce design times, there is a strong economic benefit to investing in visualization to increase user productivity.

The final most important factor is the emergence in the last few years of reasonably-priced computer hardware (and associated software) for visualization. SGI (Silicon Graphics) was the pioneer in developing custom chips to perform the general 3D transformations, Gouraud-shading and Z-buffering needed for 3D visualization. The competition increased with the formation of the Stellar and Ardent companies which later merged to form Stardent. Three years ago these companies were the only ones to offer true visualization platforms, offering 80,000–120,000 Z-buffered, Gouraud-shaded (smooth shading based on corner values) triangles/second, 250,000–500,000 3D vectors/second and CPU performance in excess of 10 Mflops. Also, these machines cost in excess of $100,000, making it impossible to think of putting one on the desk of every engineer or researcher. Today, such machines are offered by SGI, Stardent, DEC, HP, IBM, SUN and others, with discount prices around $30,000. In five years time it is perfectly reasonable to expect that such machines will have dropped in cost by another factor of four, and will be on the desk of every person who needs it.

The software side of the picture is also very important. In the past there have been numerous efforts to create software standards, but many of these have failed, leaving users with the difficult task of porting complex graphics programs onto different low-level graphics libraries. It appears that this situation is now ending. X-windows, which was developed at MIT as part of Project Athena with the support of DEC and IBM, has gained wide acceptance throughout the computing industry. X is available for almost all UNIX systems as well as PCs and Macs, and is now often a required feature in workstation purchases by governments and industry. A limitation of X is that its graphics is purely two-dimensional, and does not have any mechanism for making use of special hardware features such as Gouraud-shading. Many groups have used SGI's proprietary GL software for 3D visualization of SGI platforms, but the other manufacturers organized a subgroup of the X consortium to formulate PEX (Phigs Extension to X) a 3D extension of X with full support for all special hardware features. The PEX movement is now gathering speed, with PEX software expected in the latter part of 1991 from Stardent, DEC, IBM and HP, and with SGI promising it late in 1992. PEX is therefore the ideal software base on which to build a visualization system which will run on multiple platforms.

The history of visualization programs for three-dimensional CFD results begins with PLOT3D at NASA Ames [5]. This was developed approximately five years ago shortly after the introduction of the first SGI workstations. At the time, block-structured grids and flow solvers were the standard for three-dimensional calculations, and so PLOT3D was designed to handle just this one data type. Similarly, the hardware at the time had limited computing and graphical rendering capability, and so PLOT3D was designed more for the quality of the final images than for interactive speed. More recently, researchers at NASA Ames have developed FAST, a visualization system encompassing several different programs with a greater emphasis on interactivity on higher-performance workstations.

Three years ago at MIT we were involved in the development of flow algorithms for unstructured 'finite-element' grids, using triangles and quadrilaterals in two dimensions, and hexahedra in three dimensions. We were also working on both unsteady flow calculations and grid adaptation in two dimensions. Elsewhere, other people were working on grid adaptation [9, 10] and unstructured flow solvers in three dimensions using tetrahedra [6, 7, 8], and were beginning to do unsteady three dimensional calculations. We realized that there was no visualization software available at that time to view these results, and so we began the development of **Visual2** and **Visual3** our visualization packages for two and three dimensions, respectively. From the start, these were designed to handle unstructured grids with a variety of different cell types, and to view steady or unsteady data. They were also designed on the top Stellar workstation, the GS2000, in the confident expectation that this high-end 'graphics-supercomputer' would be the fore-runner of the graphics workstations which are now available from all of the major vendors. Similarly, Stellar, and subsequently Stardent, was a founder member of the PEX group, and so we based our packages on XFDI, a Stardent proprietary library which has evolved into PEX-IM, an immediate-mode library for interfacing to PEX.

These lecture notes attempt to present a comprehensive view of **Visual3**. A paper by Giles and Haimes [1] discusses the earlier development of the two-dimensional package, **Visual2**. The notes begin by discussing the software architecture which was driven by our design goals and was very carefully considered before any programming was done. In some ways this is the most critical section because mistakes or poor choices in the architecture can severely restrict a package. The third section explains the programming interface, and in particular the method of coupling to user codes for unsteady visualization. The fourth section very briefly covers hard copy output. The fifth and sixth sections present all of the tools which are available to a **Visual3** user in trying to understand and interrogate a flow field, and the final section explains the underlying mathematical algorithms which are used, and have been optimized to achieve very high interactivity.

2 Visual3 Architecture

2.1 Design Goals

The following were the design goals for **Visual3**, together with their motivation and implications.

- High performance

  Visual3 was designed for high-performance graphics workstations with special hardware for 3D visualization. In the initial development stages, this meant that all development was done on Stardent GS series 'graphics supercomputers'. Now that other manufacturers (including DEC, IBM, HP and SUN) are producing comparable machines, **Visual3** will be made avaialable on these too.

  Also, to achieve high performance it was important to use low-level graphics software which was well suited to visualization. PHIGS was considered in the early development stage since it is an accepted standard available on many platforms. However, PHIGS was designed many years ago for CAD/CAM applications and involves certain structures that are inappropriate for interactive, unsteady visualization and incur significant performance penalties. Therefore, as explained earlier, we developed **Visual3** using Stardent's proprietary XFDI software and we are now moving to the emerging PEX standard which has been specifically designed as a highly efficient low-level interface to the new generation of high-performance graphics workstations.

- Highly interactive research tool

  During Computer-Aided Education software development for undergraduates at M.I.T.'s Project Athena, it became clear the interactivity was of paramount importance in conveying information to the user[11]. Having to wait for results of a user action breaks the train of thought and can be frustrating. In order for a visualization package to be a fruitful tool it must be able to answer questions with a minimum delay in time.

  Visual3 was designed primarily as an interactive tool to aid researchers in understanding and interpreting their computational results. The production of glossy presentation graphics was very much a secondary role. To improve its usefulness as an investigative tool, considerable attention was devoted to providing a wide variety of tools and 'probes' which the researcher can use to extract information from the solution. Also, the design emphasized performance in both the graphics and compute portions to maximize the degree of user interactivity.

- Animation of time varying data

Although at present most three-dimensional calculations are steady-state, there is a growing number of unsteady calculations, such as the stator/rotor interaction calculations of Rai [12] and Chen [13]. As computer speeds increase we will reach a point at which such calculations become possible on a more routine basis, and will have to face the need for good visualization tools. Therefore, **Visual3** was designed to handle unsteady data in both co-processing and post-processing modes.

- Support of unstructured grids

  Visual3 was designed to support as many different CFD grids as possible, including both block-structured and unstructured grids. **Visual3** views any grid as a heterogeneous collection of cells of four types: hexahedra, tetrahedra, prisms and pyramids. These four types support almost all current CFD codes. The hexahedra are for structured, block-structured [14] and adaptive hexahedral programs [10]. The tetrahedra are for unstructured tetrahedral programs [6, 7, 8]. The prisms are for viscous boundary layer elements used in conjunction with unstructured tetrahedral elements [15]. Finally, the pyramids (with a quadrilateral base and four triangular faces) can be used for the decomposition of larger cells types; any cell with quadrilateral and triangular faces can be split into pyramids and tetrahedra by forming an additional node at its center, and connecting it to all of the other nodes.

- Generality

  It was decided that **Visual3** should not be limited to simple CFD applications, and should instead be useful in areas such as chemically reacting flows, aeroacoustics, structural analysis, electomagnetics. This was accomplished through designing **Visual3** to handle scalar and vector data in a generic fashion, with the responsibility on the programmer to supply that data.

- Simple programming interface

  Any graphics post-processing package requires some programming, although it may be as simple as converting the flow solver's data to a form that the package can read. A goal for **Visual3** was to keep any programming required as simple as possible, expecting that the typical programmer/user (at least in the CFD community) would be a knowledgable FORTRAN programmer with no desire to learn graphics programming such as X-windows. Accordingly, **Visual3** was designed to be used from FORTRAN, both with a C interface available for generality. **Visual3** itself is written primarily in FORTRAN, with certain crucial parts (particularly graphics, sorting and dynamic memory allocation) in C.

- Data Integrity

In general, the scientific visualization software available seems to be aimed more at generating pictures for presentation than at understanding of fluid dynamics. An important design goal for **Visual3** was that it be a tool in the investigation and interpretation of computational results. When this goal is met, a side benefit is that with better viewing and understanding of the flow results, the visualization software can be used for flow solver debugging. For example, unlike packages that generate "pretty" pictures, **Visual3** performs all shading in a facetted manner. This is part of a design philosophy to give the investigator only the data. Phong shading, or anything that may smooth out blemishes (techniques routinely used in CAD) are always avoided.

Following this philosophy the amount of interpolation is allways kept to a minimum.

2.2 Visual3 Structure

The first 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 100 subroutines, which a user could build into a graphics application package. This traditional approach requires much programming and a good understanding of the libraries internals. The AVS (Application Visualization System) software developed by Stardent is a very good example of this type of software that requires no programming [16]. 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 entriely 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.

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 pre-programmed into PLOT3D then they must modify the internal code of PLOT3D. If the user wants to use PLOT3D to view structural analysis results, then extensive modifications need to made. For this reason we chose not to adopt this approach.

Lying between these two approaches is GRAFIC, developed by Dannenhoffer [17]. GRAFIC has a number of pre-packaged options to do straightforward things such as line plots and contour plots on 2D 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 **Visual3**.

The program architecture finally chosen is shown in Fig. 2. **Visual3** contains 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 **Visual3** to concentrate on providing interactivity, animation and overall high performance.

To provide more flexibility than PLOT3D, **Visual3** 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 programmer 'binds' a keyboard key to a particular function, and notifies **Visual3** of the type of the function. For example, a user program may tell **Visual3** that the key 'p' is associated with a scalar function labelled 'pressure'. If the user subsequently presses the key 'p', then **Visual3** loads in the values of 'pressure' by calling a programmer-supplied routine. The programmer-supplied routine can communicate with the top-level user program through COMMON blocks in Fortran or global data in C, 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 **Visual3**. This satisifes the goal of flexibility and generality.

Similarly, **Visual3** obtains the data structure and grid coordinates by calling programmer-supplied routines. If the initialization of **Visual3** states that some or all of the grid and flow data are unsteady, then **Visual3** will keep calling (either synchronously os asynchoronously) the programmer-supplied routines to obtain the latest data. This is the simple mechanism by which animation is achieved.

2.3 Multiple Windows

The next step in the development of **Visual3** was to design what the user would see on the screen.

A multi-windowing setup in concert with reduction of dimensionality is useful in un-

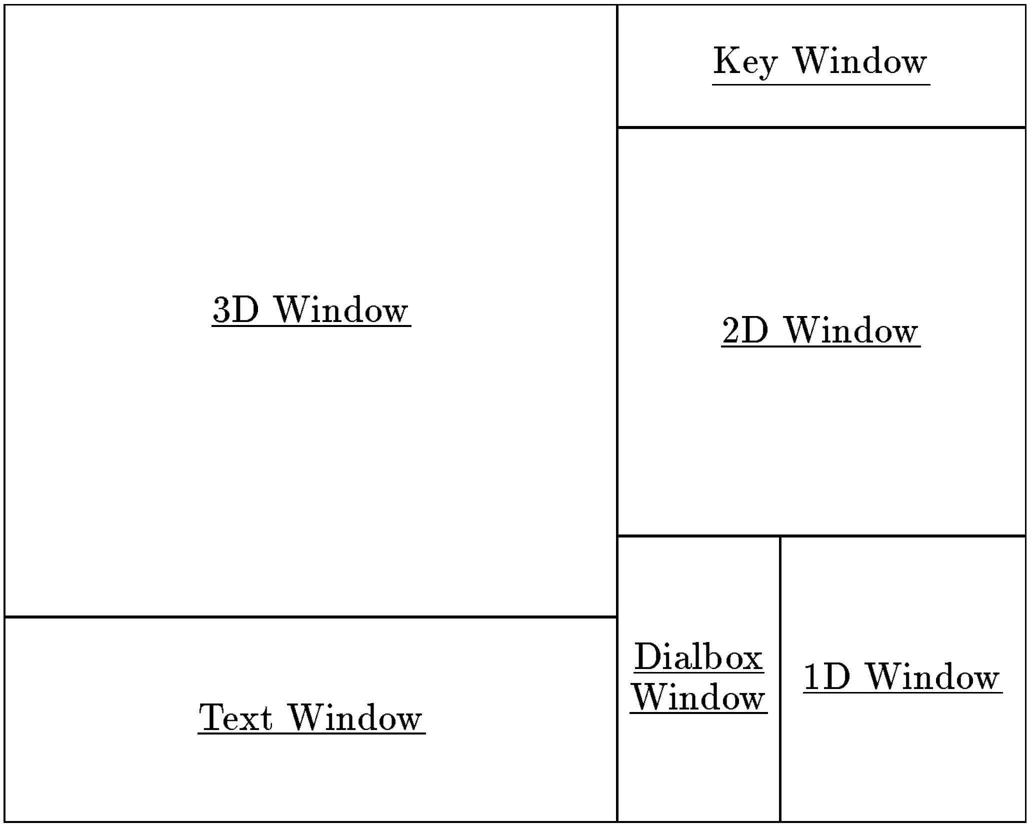

Figure 1: Screen layout

derstanding complex 3D data. Trends are much easier to comprehend if shown on a 2D map. Linear cuts in 2D can be displayed in a normal XY plot. In **Visual3** each type of display has its own window and a function can be active concurrently in the 3D, 2D and 1D windows.

The screen is divided into six different windows as seen in Fig. 1. As is typical of X-window applications, the functions invoked by mouse button, keyboard or dialbox input are dependent upon the position of the cursor. Thus, different functions are available to the user depending on which window is 'active', i.e. contains the cursor. The six different windows are:

- Text Window

  The text window is the window from which **Visual3**, or rather the application program, was started. This window is used to output various messages, including the help menus, and to input filenames, numerical values, etc.

- 3D Window

The 3D window displays data on three-dimensional surfaces, either from the bounding domain or from cuts and iso-surfaces. The user has complete control over the rendering attributes of each individual surface. The 3D window also displays three-dimensional lines such as tufts and streamlines. The view of the 3D object can be rotated, translated and enlarged using the dialbox.

- 2D Window

  The 2D window is used to display data on a mapped domain or cut surfaces (for which there exists a mapping to an (x', y') coordinate system). The user can independently adjust the view in this window. The 2D window is used to initiate particle paths and to perform many of the probes and point in 3-space.

- 1D Window

  The 1D window is used to display one dimensional data which is generated by various probing functions in the 2D window or from mapping streamlines.

- Key Window

  This window displays the color index used in the 3D and 2D windows. When the cursor is in this window, there are many options available. These include the ability to interactively change the color scheme, load in a new color scheme, or change the length of displayed vectors.

- Dialbox Window

  The dialbox window serves two functions. The first is to display the functions associated with each of the dials of the dialbox. If there is no dialbox, the dials can be rotated by putting the cursor on the appropriate dial and holding down the left or right mouse button.

  In this window's secondary role, it displays the state of the surface database. The database lists all of the surfaces, and for each one has four small boxes which give the status of the surface attributes. The attributes are rendering status (on, off or translucent), the second is the grid intersection status (on or off), the third is the gray status (on, color by scalar or gray with line contours) and the fourth is the thresholding status (on or off). The user can change the attributes by pressing any mouse button when the cursor is on one of the boxes.

2.4 Cursor Mapping

Another important concept related to the dimensional windowing approach is cursor mapping. Cursor mapping is the simultaneous display of the current cursor position in the one-, two-, and three-dimensional views. For example, suppose a line plot is displayed in the 1D window. When pointing (with the cursor) in the 1D window, the user is actually pointing in 2-space by reversing the 2D to 1D mapping. Similarly, this point in 2-space corresponds to a point in 3-space. This is a simple and powerful concept. Being able to point at and locate unusual features in the volume (and accessing **Visual3**'s tools) the user can begin to answer questions like 'Is this a numerical anomaly or some interesting feature of the flow?'

This cursor mapping is also extremely useful when used with streamlines. The calculation of a streamline requires an initial 3-space location to begin numerical integration. With structured grids, this is often done by simply selecting a specific computational grid line or node. However, with unstructured grids, this technique is impossible. With cursor mapping, the user simply points to a location in the two-dimensional window which corresponds to a unique 3-space point from which a streamline may be spawned. Without a multi-window system pointing in 3D is, at best, difficult.

2.5 User Interface

Visual3's user interface is also different from most packages which are being developed currently. Other packages are emphasizing a 'point-and-click' Graphical User Interface (GUI) with a hierarchy of pop-up/pull-down menus. This approach was considered and rejected in favor of an interface using mouse clicks and dialbox and keyboard input. There were two reasons for this. The first is our opinion that an experienced user can invoke the options in the visualization package more quickly with single keystrokes than using 'point-and-click' with multiple menus. For the novice user, there is a help menu listing the available options in each window, and printed reference lists are also recommended. The second reason is that pop-up/pull-down menus require considerable amounts of screen space. Often more than half of the area on the screen is dedicated to this input task, whereas in **Visual3** approximately three quarters of the area is dedicated to the output task of displaying images. This is particularly important because of the desire for separate windows to display 3D, 2D and 1D data.

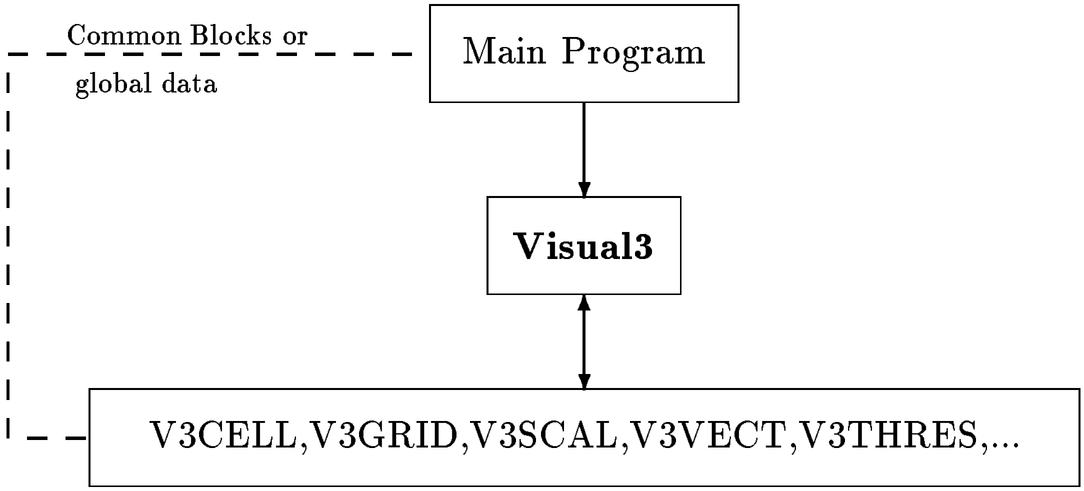

Figure 2: Flow diagram showing use of **Visual3**

3 Programming Interface

3.1 Overview

Fig. 2 shows how the programmer interacts with **Visual3**. The application program first activates and initializes **Visual3** by calling V3_INIT. **Visual3** then gets the data it requires by calling a small number of programmer-supplied subroutines. These subroutines are connected to the main program only through COMMON blocks in FORTRAN or through global data in C.

The program control in this interaction between the main program and **Visual3** depends on whether **Visual3** is being used for post-processing or co-processing. In a post-processing mode, the result of some calculation is already known. The main program will call **Visual3** and transfer program control to it. **Visual3** therefore remains in total control; when it calls the programmer-supplied routines, they return the grid coordinates, function values, etc. Control is only returned to the main program when the user exits from **Visual3**.

In co-processing modes, the results are being calculated concurrently by the main program. In this mode, when the main program calls V3_INIT, **Visual3** generates a new process (actually a 'thread', the distinction being that threads share global memory and so data transfer is simpler than using sockets, the standard UNIX method of communication between processes) which runs the graphics, before returning to the main program. At this point there are now two processes (threads) running, one executing the main program, and the other executing **Visual3**. To avoid data conflicts between the two, special locking mechanisms have been set up to ensure that the main program does not update its data at the same time that **Visual3** is fetching it by calling the programmer-supplied routines.

When describing the mesh, the programmer gives **Visual3** a list of unconnected cells of four types; tetrahedra, pyramids, prisms and hexahedra. These four types cover almost all data structures being using in current computational algorithms. Any cell type not covered, or any element using a higher order interpolant must be split up into a combination of these four where a linear assumption is valid.

In order to calculate particle paths from vector fields **Visual3** requires information about which cells are neighbors, i.e. share a common face. Rather than impose on the programmer the burden of computing these connections and passing the information to **Visual3**, instead **Visual3** calculates this information by processing all cells, compiling a list of all of their faces and checking for each face whether it appears on another cell. If it does appear twice, then it is an interior face, and a streamline can pass through from one cell to its neighbor. If it does not appear twice, then it is a surface face on the boundary of the computational domain, and a particle path will terminate when it hits the face. It should be noted that **Visual3** only requires connectivity for the streamline/particle path (and Xray) algorithms.

In principle, all surface faces could be grouped together to form one bounding surface for plotting purposes. However, in many applications it is more useful to split the bounding surface into a number of pieces, referred to as 'domain surfaces'. For example, the outer bounding surface of a calculation of airflow past a half-aircraft (using symmetry to reduce the computation) might be split into four pieces, the far-field boundary, the symmetry plane, the fuselage and the wing. If the programmer specifies each of these as separate 'domain surfaces' by grouping the appropriate faces, then **Visual3** can offer the capability of plotting on just one or two of the surfaces (e.g. the fuselage and wing) and not on the others (far-field and symmetry plane).

3.2 Subroutines

The following is a list of **Visual3** subroutines with a very brief description of their function:

Programmer-called subroutines:

- V3_INIT: initializes **Visual3**, allocates required memory based on number of nodes, cells and surface faces, opens the windows
- V3_STAT: handles synchronization in co-processing applications
- V3_CURSOR: moves the cursor to the text window (and back) when the program asks the user questions

Programmer-supplied routines:

- V3CELL: returns the cell and domain surface data structures

- V3GRID: returns the grid coordinates (XYZ triads) for all nodes
- V3SCAL: returns the requested scalar function values for all nodes
- V3VECT: returns the requested vector function values (optional)
- V3THRES: returns the requested threshold function values (optional)
- V3ZPRIME: defines the programmer-defined cutting plane (optional)
- V3XYPRIME: defines the associated (x', y') values for programmer-defined cutting plane (necessary if V3ZPRIME exits)
- V3SURF: initializes plotting of mapped domain surfaces (optional)
- V3XYSURF: defines (x', y') values for mapped domain surfaces (necessary if V3SURF exists)
- V3SSURF: returns the surface scalar function values (optional)
- V3VSURF: returns the surface vector function values (optional)
- V3EVENT: allows programmer processing of X-events (optional)
- V3EVENTHELP: additional help info associated with V3EVENT (optional)
- V3STRING: additional label for plots - label time in unsteady applications (optional)
- V3UPDATE: updates time and/or data in unsteady post-processing applications (optional)

3.3 Visualization of Steady Data

The simplest steady-state visualization application program is a main routine that reads the data from disk and calls V3_INIT. The application also contains the subroutines, V3CELL, V3GRID, V3SCAL, and if appropriate V3VECT, that are called by **Visual3**. Figure 3 shows the flow diagram for the control of **Visual3**. In the initialization phase V3CELL and V3GRID are called once to obtain the data structure and grid coordinates, which do not change. After initialization, **Visual3** enters a large animation loop. At the beginning of the loop the first step is to process the list of X-events (keystrokes, dial movements, and cursor movements) that have occurred since the beginning of the previous loop. Processing this list can result in changes in the transformation matrix being used to view the objects in the 3D and 2D windows, or a change of desired scalar or vector variable, or the invocation of some new plotting function. The second step is to call V3SCAL if the scalar variable has changed, and V3VECT if the vector variable has changed. The third step is

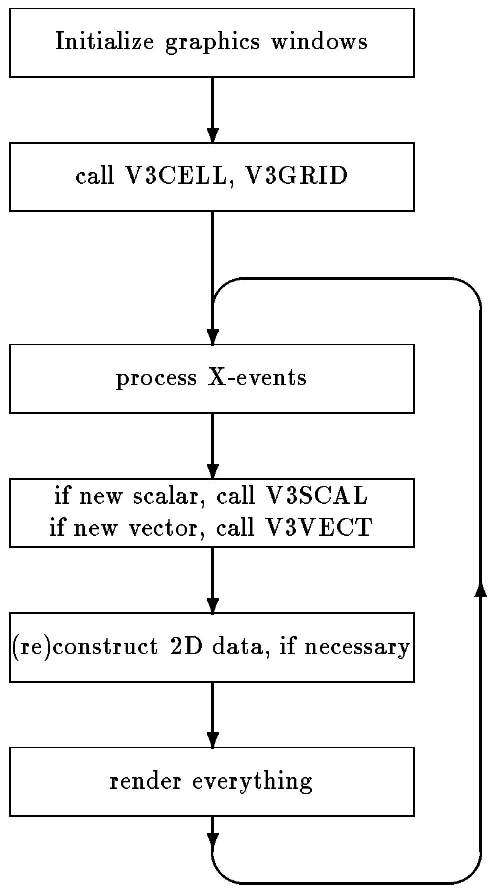

Figure 3: Flow diagram of **Visual3** for steady application

to construct any 2D structures required for cutting planes and iso-surfaces. The final step is to perform the rendering, the drawing of the images in all of the graphics windows. The third and fourth steps are usually the most computationally and graphically demanding, and are optimized by keeping careful note of what has changed since the previous loop.

3.4 Post-processing Unsteady Visualization

In post-processing unsteady visualization, the program control is similar to that in steady visualization. It is assumed that the unsteady data has been pre-computed or needs only a minor level of computational effort to generate it. Again the main program initializes **Visual3** and transfers control to it, and program control is returned to the main program only when **Visual3** terminates.

There are currently two supported types of unsteadiness. The first type is 'data unsteady', in which the grid is fixed but the data changes in time. The second is 'grid unsteady' in which the grid coordinates and the data are time varying. At present the grid

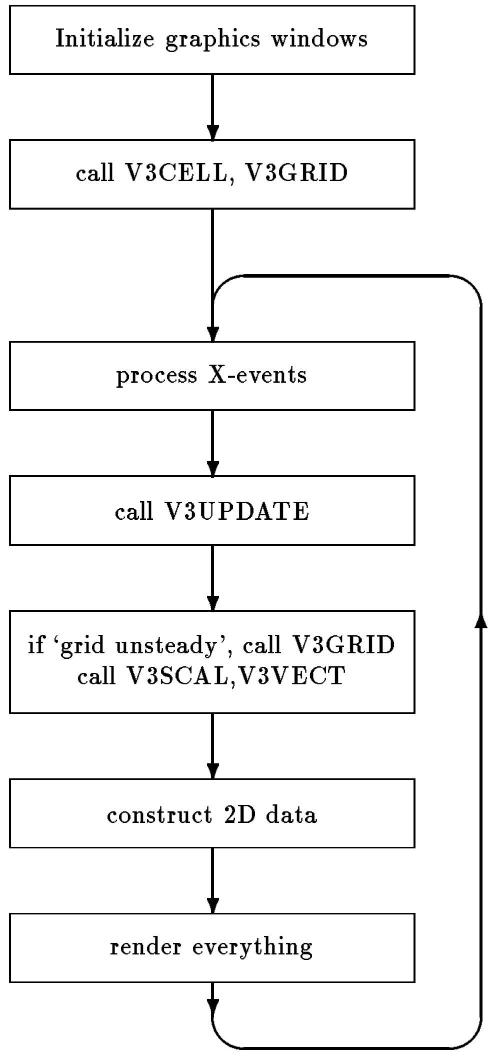

Figure 4: Flow diagram of **Visual3** for post-processing unsteady application

structure is not allowed to be unsteady. If the number of nodes, number of cells or cell connectivity changes then **Visual3** must be re-initialized by terminating and restarting.

The modifications to the control loop in **Visual3** to handle both ‘data unsteady’ and ‘grid unsteady’ modes are very minor, as shown in Figure 4. Before the data collection phase, the programmer-supplied routine V3UPDATE is called. This allows the programmer to update the data as appropriate for the problem. As can be noted by this approach, the time varying data can exist in any form and reside in memory, on disk or may come over a network. During the time that V3UPDATE is executing, **Visual3**’s user interface appears ‘frozen’; all keyboard, dialbox and mouse input is collected for subsequent processing, but there is no visual change on the screen. Thus, for good interactive response, V3UPDATE should require significantly less than a second of computational effort.

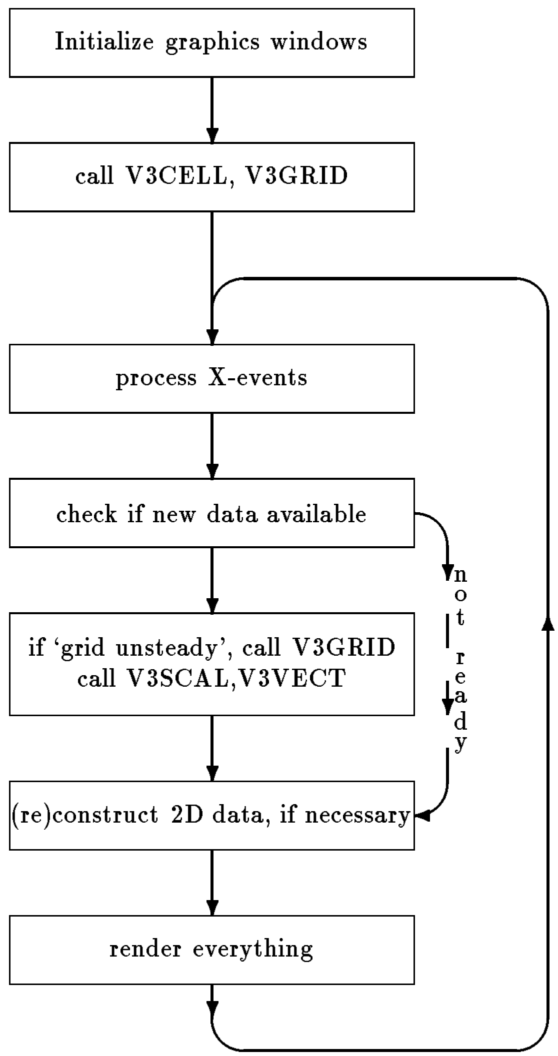

Figure 5: Flow diagram of **Visual3** for co-processing unsteady application

The other two changes are that V3GRID is called if it is 'grid unsteady' and so new grid coordinates must be obtained, and that V3SCAL (and optionally V3VECT and V3THRES) are always called, even if their definition has not changed, because their values will have changed.

NOTE: the calls to V3GRID, V3SCAL, V3VECT and etc. are not made if the call to V3UPDATE does not change the simulation time. Under these conditions **Visual3** assumes that the data has not changed.

3.5 Co-processing Unsteady Visualization

Co-processing is supported on parallel machines with operating systems that have a feature called 'threads', in which a process can be split into two or more parts which run concurrently and share access to COMMON blocks (global data) and main program memory. In this mode, when the main program calls **Visual3**, this performs the initialization and generates a new 'thread' which runs the graphics, before returning to the main program. At this point there are two threads running, one executing the main program, and the other executing **Visual3**.

To avoid data conflicts between the two threads, special locking mechanisms have been set up to ensure that the main program does not update its data at the same time that **Visual3** is fetching it by calling the programmer-supplied routines. The programmer controls when new data is available to **Visual3**, by calls from the main program to V3_STAT. When the main program wants to update the flow variables, at the end of one timestep for example, it calls V3_STAT to notify **Visual3** that the data is being updated and is not currently available. If **Visual3** is in the process of accessing the data, then V3_STAT waits until it is completed before seizing control. Once the main program has finished updating the data, it calls V3_STAT to notify **Visual3** that the data is available while setting the simulation time.

From the point of view of **Visual3**, the modifications to the control loop are shown in Figure 5. After processing the X-event list, and before updating the data, it checks to see if the data is available. If it is not, then **Visual3** jumps to the rendering phase.

4 Hard Copy

There are three types of output files generated by **Visual3** when the user hits the appropriate function keys in different windows. The first is tabular output which is generated from the 1D window. This is an ASCII file suitable for inclusion into most line plotting or spread-sheet software. The default file name is 'visualXYZ.tab'. A hard copy post-processing program 'tab2ps' produces Postscript output.

The second file type is an image dump of the entire screen or an individual window. This file is written in a FORTRAN unformatted manner with the default name of 'visualXYZ.img'. This can be converted into Postscript (using 'img2ps') or Macintosh TIFF files (using 'img2tiff').

The last file type is used for line drawings. This file has the default name 'visualXYZ.lin' and is also written unformatted. A hard copy post-processing program 'lin2ps' produces Postscript output, with hidden line removal.

The source and make-files for the hard copy post-processors are included on the **Visual3** distribution. By making small modifications to the sources, other output devices can be supported with no changes to **Visual3**.

5 Visualization Tools

5.1 Scalar Visualization Tools

One type of data used in **Visual3** is scalar data, which is simply scalar information defined at each node of the computational grid. **Visual3** does not know anything about the data other than an associated function number and label. For example, in fluid dynamic applications, function 1 may have label 'pressure' and function 2 may have label 'Mach number'. An important concept in **Visual3** is the notion of 'active' functions, and the 'active scalar function' is the function number and associated label which corresponds to the scalar data currently stored by **Visual3**. The user can switch to a different scalar function by hitting a particular key on the keyboard which is 'bound' by the initialization procedure to the desired function. For example, key 'p' may be bound to function 1, labelled 'pressure'.

The following is a list of plotting functions available for use with scalar data:

- Surface rendering

  Gouraud-shaded (smooth color shading) surface contours of the scalar function can be rendered on any, or all, of the surfaces. Line contours may also be displayed on any user selected surfaces.

  If the 'active' surface is a mapped domain surface (with an associated mapping to a (x', y') coordinate system), then Gouraud-shaded contours can also be plotted in the 2D window. This allows further probing of surface values as well as initiating surface streamlines.

- Cutting plane

  The cutting plane is a true planar surface cutting through the 3D field. The user may interactively set any orientation for the surface. The resultant planar cut is displayed in both the 2D and 3D windows.

  Once the cutting plane is activated, it is controlled from the 2D window, meaning that it responds to keys and dials that are active when the cursor is in the 2D window. Using dials, the cutting plane can be moved from side to side, up and down, in a direction normal to the plane (scanned) and rotated in its own plane.

  Cut planes are also used for probing and initiating particle paths and streamlines.

- Program-defined cutting plane

  This is similar to the last option, but instead of being a truly planar surface, it is a surface corresponding to $z' = const$, where z' is a programmer-defined function; the 'scan' capability varies the value of *const* interactively. This allows the programmer to define conical, cylindrical or other surfaces not otherwise defined by **Visual3**. The programmer also must define a mapping to (x', y') coordinates so that plotting is possible in the 2D window.

- Iso-surfaces

  An iso-surface is a 3D surface with a uniform value of the current active scalar variable. There is no 2D mapping. The iso-surface value is displayed in the key window and can be varied interactively (in the key window) using the dialbox to 'scan' the value of z'. An exact value may also be set.

5.2 Vector Visualization Tools

The second data type is vector data. This is a set of 3D vector values for each grid node. As with the scalar function, this vector data is associated with an 'active' vector function, which can be changed by pressing a key that is 'bound' to another vector function. There are many plotting functions for vector data:

- Streamlines

  'Streamlines' are curved 3D lines which are everywhere parallel to the local vector field. They are obtained by numerical integration of the vector field along a line starting at some chosen location. Streamlines are only active for steady-state cases or unsteady applications when the user has frozen time (paused the update procedure).

  The starting point is determined by use of a surface plotted in the 2D window. A point in the 2D window maps back to a corresponding point on a surface in the 3D window and so can be used to begin a single streamline; pressing one of the mouse buttons does this and (depending on which button is pressed) produces a streamline going upstream and/or downstream. Alternatively, a line or circle or grid may be defined in the 2D window, which is then used to start a set of streamlines.

  The streamlines can be plotted either as lines of constant color or as lines which are colored according to the value of the current local scalar function. In the latter case, it is helpful to enable the 'grey' status for background surfaces so that instead of being rendered in color they are instead rendered in solid grey, making the colored streamlines clearer.

  A 'streamer' is an interactive streamline. When pointing in the 2D window, the streamline that goes through the cursor position is drawn in the 3D window.

  NOTE: if a domain surface is mapped to the 2D window, then the streamlines which are generated are similar to regular streamlines except that they lie on the surface. This is calculated by taking the projection of the local vector field onto the surface.

- Ribbons

  Stream ribbons are streamlines that have been given some width. One edge is the true particle path, the other edge is constructed by rotating a constant length normal vector about the path tangent according to the local streamwise angular rotation rate. The result is a ribbon whose twist illustrates the streamwise vorticity of the flow.

  When ribbons are invoked, all 3D streamlines are drawn as ribbons. If the streamlines are colored, the ribbon is rendered in the default streamline color, otherwise the ribbon is colored with the current scalar.

  The width of the ribbon may be adjusted by using the dials when the cursor is in the Key window. A specific width may also be entered. Also, the ribbons may be rotated by using the dials with the cursor in the Key Window.

- Tubes

  A tube is a streamline with a circular crossflow area. The radius of the cross-section is derived from the local crossflow divergence. The crossflow divergence measures the local crossflow expansion rate. Thus, the resulting tube displays the local expansion/compression of the current vector field.

  When tubes are invoked, all 3D streamlines are drawn as tubes. If the streamline coloring is on, the tubes will be colored with the current scalar. Otherwise, the default streamline color is used.

  The width of the tube may be adjusted by using the dials when the cursor is in the Key window. A specific width (and maximum radius) may be entered. The maximum radius is useful to limit the size of the tube in stagnation regions of a flow field where the radius can become exponentially large.

- Tubes with Twist

  The rotation and divergence effects can be rendered simultaneously by placing lines on the surface of tubes which twist with the local rotation rate. This effectively combines the functionality of the ribbons and tubes. The final image displays the streamline, the rotation rate, the crossflow divergence, and scalar variations.

  When tubes with twist are invoked, all 3D streamlines are drawn as tubes with twist. If the streamline coloring is on, the tubes will be colored with the current scalar and the lines will be in the default streamline color. Otherwise, the tube is the default color and the lines are the current scalar.

  As with ribbons and tubes, the rotation angle, the tube size, and the tube maximum may all be controlled from the Key window using the dial box or by the appropriate key strokes.

- Bubbles

  Bubbles are unsteady particle paths. Bubbles have essentially the same type of functionality as their steady counterparts - streamlines. Bubbles are active when **Visual3** is being used in an unsteady mode. A single bubble path may be spawned by simply pressing a mouse button in the 2D window which provides an initial point to start the integration in 3D space. If streamline rendering is off, only the current bubble location is displayed. Otherwise, the current and the three previous or ghost bubble locations are rendered. This provides the same effect as hydrogen bubbles in experimental techniques. If streamline coloring is on, the current location will be rendered by the current scalar; if not, the current location is rendered with the default streamline color. Ghost bubbles always appear in grey.

  Several bubble paths may be started from a line or circle or grid in the 2D window.

  Finally, if the streamer is on, bubbles will be continuously spawned from the current cursor location at every time step. This mimics the experimental technique of streaklines where dye is continuously injected at a spot in the flow field.

  Bubbles may also be displayed in steady applications. Once streamlines have been drawn, bubbles may be initiated. The bubbles follow the streamline path showing the time evolution of the integration (this give an indication of the local speed of the flow). These bubbles may be displayed with or without the underlying streamlines. Having the streamlines displayed is useful in vortical flows so they let the user note the particle path without losing the previous history.

- Tufts

  Tufts are similar in concept to streamlines. A regular grid of points in the 2D window map to a corresponding grid of points on the surface in the 3D window. At the points in the 3D window, tufts are drawn which are short lines with magnitude and direction correponding to the local vector field. At the points in the 2D window the tufts correspond to the projection of the 3D vector field onto the 2D plane. One of the dials in the key window allows interactive changes in the scaling parameter which relates the vector magnitude to the tuft size, and a key stroke allows this scaling parameter to be input from the keyboard.

- Arrows

  Arrows are the same as tufts except that they are defined only at the intersection of the surface (in the 2D window) and the computational cell edges. To emphasise that they are different, they are drawn as arrows, whereas tufts are drawn as straight lines with a cross base. They are shown only on cutting planes and are not displayed in the 3D window. Arrows provide vector information that has not been interpolated as much as tufts.

5.3 Grid

The computational grid can be displayed on any, or all, of the surfaces. The grid lines that are displayed correspond to the intersection of the plotting surfaces and the faces of the computational grid.

5.4 Thresholding

A threshold function [18], when enabled, restricts all surface plotting to only those parts of the surface on which the values lie within a certain range. The user can interactively vary the upper and lower threshold bounds. The user can also select, through column 4 of the surface database, which surfaces are to be thresholded.

If the threshold function is chosen to be the same as the scalar function, then this provides a means of plotting the part of a surface on which the scalar function is within certain limits. If the threshold function is chosen to be geometric (e.g. x) then this produces a dynamic 'cutaway', in which the surface is only plotted within a certain geometric volume.

5.5 Probes

There are a variety of 'probes' which are available when plotting in the 2D window or from streamlines:

- point probe

  The point probe is located at the cursor position in the 2D window, and returns, in the text window, the point's coordinates and the value of the active scalar and vector functions.

- strip chart

  The strip chart is similar to the point probe, except that it produces a plot in the 1D window of the current scalar function against time.

- line probe

  When the line probe is invoked the user is asked to input two points using the mouse. These define a line in the 2D window, and the output is a plot in the 1D window showing the variation of the current scalar function along that line.

- edge plot

  The edge plot is similar to the line plot, except that in this case the line in the 2D window is the edge (intersection of a domain surface and the cut) closest to the cursor when this option is invoked.

- surface layer probe

  This option produces a line plot in the 1D window of the current scalar function along a line placed normal to an edge in the 2D window, at the edge position which is closest to the cursor. As the user moves the cursor, the normal line moves accordingly.

- streamline probe

  The streamline probe may be started any time there are streamlines. The last streamline is mapped to the 1D window in a line plot of the current scalar against the length of the streamline. When the cursor is in the 1D window a cross-hair cursor appears in the 3D window. This allows the user to both know which streamline is mapped and probe the streamline. Hitting a key selects another streamline.

6 Feature Identification

A problem with most visualization packages available currently is the lack of feature identification techniques. Thus, the user is typically left to painstakingly scan a cutting plane across an entire domain in search for important flow features. Two novel ideas; clouds and X-rays, are presented which are useful in solving the problem of feature identification in complex 3-D flow fields. The stimulus for the work is a brief mention of the ideas in a paper by R. Löhner[19].

6.1 Xrays

X-rays are exactly analagous to the medical usage of X-rays. In medical imagery, as an X-ray beam travels through a patient, the beam loses energy through interactions with the matter of the patient's body. The ratio of initial to final intensity of an X-ray beam may be modelled by the following simple equation:

$$I(x,y) = I_1/I_0 = \exp\left(-\int_{z_0}^{z_1} \mu(x,y,z)\,dz\right) \tag{1}$$

where μ is an attenuation constant which varies with position[20]. The z-direction is assumed to be the ray direction. If the attenuation constant is zero along the entire ray path, then the ratio is one (i.e., no loss has occured). In the human body, bones have much higher attenuation constants than flesh; thus, an X-ray travelling through a bone suffers a much higher loss than one travelling only through flesh. Similarly, a ray which travels a greater distance through bone will also suffer a greater loss.

In flow visualization, the computational domain is analagous to the human body, and a flow scalar, such as total pressure loss, is essentially the attenuation constant. As an "X-ray" travels through the domain, it loses intensity according to the local attenuation constant, i.e., the local scalar value. Thus, similar to how X-rays in medicine highlight internal portions of the human anatomy, X-rays in visualization highlight internal flow features of the computation. In this implementation of X-rays, the attenuation constant is actually a function of the local scalar:

$$\mu(x,y,z) = f(s(x,y,z))$$

where s is a flow scalar, and f is a mapping function such that $0 \leq f \leq 1$. The initial step of the mapping is:

$$s' = \frac{s - s_{min}}{s_{max} - s_{min}}$$

where (s_{min}, s_{max}) are user-specified scalar limits. At this point, the mapping of s' to μ is performed. In this implementation of X-rays, two pre-set mapping functions are currently available:

1. Linear - this mapping is defined as:

$$f = \begin{cases} 0 & \text{if } s' < 0 \\ s' & \text{if } 0 \leq s' \leq 1 \\ 1 & \text{if } s' > 1 \end{cases}$$

   In the linear mapping, the high scalar values cause the greatest loss of X-ray intensity. Any scalar values above the user-set scalar maximum have an X-ray loss of one; scalar values below the minimum have a loss of zero (i.e., no loss).

2. Midpoint - this mapping is defined as:

$$f = \begin{cases} 0 & \text{if } s' < 0 \\ 1 - 2|s' - 0.5| & \text{if } 0 \leq s' \leq 1 \\ 0 & \text{if } s' > 1 \end{cases}$$

   The midpoint mapping is useful when examining the range around a particular scalar value, i.e., the midpoint, and equal weighting is desired on either side of the midpoint.

The user may also interactively set a function mapping by positioning the cursor on a graph of the attenuation constant versus the scalar to indicate the intensity loss for a specific scalar value. By dragging the cursor over the range of scalar values and controlling the position, an appropriate X-ray function mapping may be created for any situation.

When an X-ray is taken, rays which travel into the screen (i.e. along the positive z-axis in screen coordinates) are issued from a gridded plane. The user may interactively set the view before taking an X-ray. Furthermore, the user also controls the X-ray grid density. The current capability is up to a grid 320 by 320 of X-rays. Also, X-rays may exit and then re-enter the domain analagous to an X-ray exitting and re-entering a body. If this occurs, no loss is accumulated while the X-ray is outside the domain.

Currently, X-rays are still in an experimental stage and therefore not included with the distribution of **Visual3**. Once the speed of generating an image is improved (so that X-rays can be considered interactive) and we can explain the interpetation the output, this technique will be distributed.

6.2 Clouds

Two different types of clouds have been implemented: element and vector. Element clouds are shrunken thresholded cells. If a cell-average scalar quantity falls between a user-controlled minimum and maximum, a percentage of the cell is rendered. The resulting display reveals which areas of the flow have features of possible interest (for example, high total pressure loss). Vector clouds display the local vector field at nodes which meet the threshold criteria. Furthermore, several additive threshold limits may be applied to clouds. Thus, features may be filtered so that the resulting display shows only pertinent information. However, the thresholds must be applied without losing the primary feature of interest in the process. Multiple thresholding for most of the data sets that we have visualized has not been needed. In general, only a single threshold has been necessary to limit the regions displayed to the areas of interest.

Element clouds may also be used to display the quality of an unstructured grid. A fractional size of the cells are rendered according to a user-set parameter between 0 and 1. A value of one renders the cell at its actual size; a value approaching zero causes the volume of the cell rendered to approach zero. Rendering only a fraction of the cell creates the desired cloud effect where the "drops" are shrunken cells. This technique for rendering gives the user information on the quality of the grid by displaying cell size in areas where the flow features are interesting. Clouds can be effective in viewing a grid even before a calculation. Thresholding by a spatial coordinate will display the grid only in the specified range. Thus, an entire unstructured grid may be scanned by simply varying the spatial coordinate thresholds.

The vector cloud technique is extremely powerful because it may simultaneously display scalar and vector information. A major advantage of vector clouds over element clouds is rendering speed. Element clouds for a hexahedral grid require the rendering of six quadrilateral faces per cell. Vector clouds require one line segment per node. In general most workstations render line segments on the order of ten times faster than shaded surface segments. Therefore vector clouds are usually at least 60 times faster than element clouds. For the reason of interactivity only vector clouds have been included in **Visual3**.

7 Algorithms

7.1 Cell Connectivity

Cell connectivity information is required so that the streamline, Xray and bubble integration is efficient. Very early versions of **Visual3** required the programmer to supply connectivity tables along with the cell definition information. For **Visual3** to be robust (that is not abort with the helpful messages 'Segmentation Fault' or 'Buss Error') the connectivity tables needed to be checked. It became clear that in the process of checking the connectivity data that **Visual3** could construct the tables internally. This led to research in further improving the effiency of the connectivity calculations. This section describes the final method used.

At **Visual3**'s initialization and before much of the memory is allocated for the execution of the package the memory for a face table and connection tables for each cell type is allocated. The face table to be constructed has six entries for each face. The first four entries are pointers to (or the indices of) the four nodes that define the face. The cell types supported have either quadrilateral or triangular faces. The fifth contains either a cell number (for the first cell that created the face entry) or a flag indicating that the face has been hit twice and the connectivity data stored. The sixth entry is a pointer to the next face in the list which contains the same minimum node index (a zero indicates the end of the list). This produces a 'threaded list' based on the lowest numbered node of each face. This list can be quickly processed and a face can be found without extensive searching. 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 initalization 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 four to six faces that have pointers to the three or four nodes which define the faces. For each face the process is as follows:

Let I1, I2, I3, and optionally I4 be the indices of the nodes that define the new face. Imin is the minimum of the node numbers. The node table is used to see if there is already a face in the face table which involves node Imin. If there is not, then this new face is added to the face table by setting the first four entries equal to I1, I2, I3, and I4 and the fifth to the cell index with a face number flag. The node table entry for Imin is adjusted to point to the new face.

If there is an entry in the node table, then all the nodes of that entry in the table are compared to I1, I2, I3 and I4. If these match, then the table face is the same as the new face. The appropriate entries in the connectivity tables for the current face/cell and the face/cell stored in the table entry are set to point to each other. The fifth entry of the face table is also set to indicate that the face is complete.

If the table face does not match then the whole process is repeated with the next face in the table which uses the node Imin, by using the pointer in entry six. This continues until one either finds a match for the new face, or runs out of faces which use the node Imin. In the latter case the new face does not currently exist in the face table and so it is added. Also, the sixth entry of the face that used to be the last involving node Imin is set to point to the new face in the table which is now the last entry.

When all the cells have been processed, those entries in the face table that still have cell numbers in the fifth position are faces that bound the volume. These are compared against the programmer-supplied bounding surface face definitions for consistency.

Finally, the face table memory is deallocated and the constructed connectivity data is written to disk. This disk file is used for subsequent invocations of **Visual3** when the same grid structure is used, thereby avoiding the above calculation and providing a quicker initialization.

7.2 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, programmer-defined cutting plane, and iso-function surfaces, which are three of the tools in **Visual3**. In all three cases, the description of the problem can be reduced to the following; given some function z' defined at the computational nodes, and a set of unstructured three-dimensional cells, how does one determine the cells that are crossed by the surface $z' = Z$ (where Z is some constant value), and for each of the crossed cells the surface piece which is to be plotted.

The first task of determining the cells that are crossed is actually the harder of the two tasks. An exhaustive search of all cells is possible but is extremely time-consuming and is only used for unsteady applications in which the data can change from one time level to the next. For steady applications a much more efficient method is used.

Strid [22] avoids the complete search by performing a search for all boundary cells that are crossed, and then constructing the surface from the edges in, by using cell connectivity information to check neighboring cells to see if they are crossed. Although not vectorizable, this works well for planar cutting planes. However, it does not work for programmer-defined surfaces and iso-surfaces which do not cross the boundaries of the computational region.

Our approach 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 then 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

Table 1: Look-up table for surface contruction in tetrahedra

4-bit index	surface boundary
$1 \equiv 0001$	$1/2 \to 1/3 \to 1/4$
$2 \equiv 0010$	$1/2 \to 2/3 \to 2/4$
$3 \equiv 0011$	$1/3 \to 2/3 \to 2/4 \to 1/4$
$4 \equiv 0100$	$1/3 \to 2/3 \to 3/4$
$5 \equiv 0101$	$1/2 \to 2/3 \to 3/4 \to 1/4$
$6 \equiv 0110$	$1/2 \to 1/3 \to 3/4 \to 2/4$
$7 \equiv 0111$	$1/4 \to 2/4 \to 3/4$
$8 \equiv 1000$	$1/4 \to 2/4 \to 3/4$
$9 \equiv 1001$	$1/2 \to 1/3 \to 3/4 \to 2/4$
$10 \equiv 1010$	$1/2 \to 2/3 \to 3/4 \to 1/4$
$11 \equiv 1011$	$1/3 \to 2/3 \to 3/4$
$12 \equiv 1100$	$1/3 \to 2/3 \to 2/4 \to 1/4$
$13 \equiv 1101$	$1/2 \to 2/3 \to 2/4$
$14 \equiv 1110$	$1/2 \to 1/3 \to 1/4$

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 less than 15 seconds on the Stardent GS2000.

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 $Z_{new} > Z_{old}$, the old active cell list is supplemented by the cells in list L_{min} that have values of $z'_{j,min}$ in the range $[Z_{old}, Z_{new}]$. If $Z_{new} < Z_{old}$, the old active cell list is supplemented by the cells in list L_{max} that have values of $z'_{j,max}$ in the range $[Z_{new}, Z_{old}]$. 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.

Now that an active list of crossed cells has been formed the second step is the construction for each crossed cell of the corresponding surface piece. There is an extremely fast

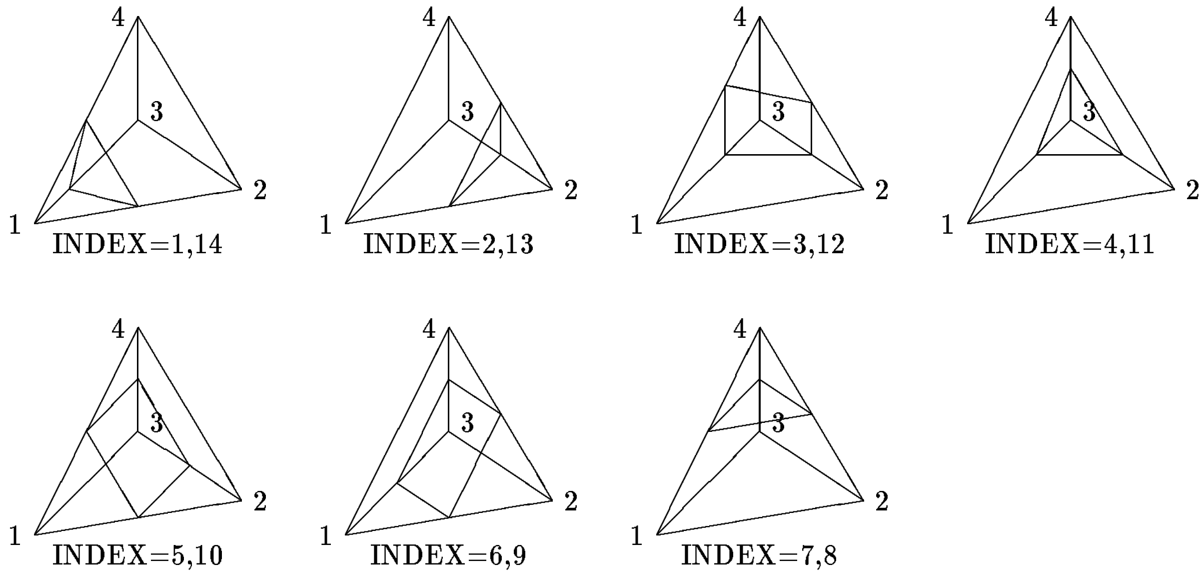

Figure 6: Cut surface shapes for tetrahedra

method for doing this that is similar to that used by W. Lorensen [21].

For illustration we consider a tetrahedral cell. A four-bit index for the cell is constructed by setting the nth bit to 0 or 1 depending whether the value of $z' - Z$ at the nth node is negative or positive. Edges along which $z' - Z$ changes sign clearly must have a point at which it is zero, and which is therefore a corner of the surface $z' = Z$. This index is therefore used as an entry into a lookup table which gives the logical structure of the surface $z' = Z$, by listing the edges which it intersects; see Table 1 in which 1/2 means the edge between nodes 1 and 2. Figure 6 shows the shape of the cut surfaces. Note that for some, the slicing surface is a non-planar quadrilateral which must later be split into two triangles for rendering purposes.

The construction of the surface is completed by using linear interpolation along the edges to obtain the values of the grid coordinates and other variables at the points at which $z' = Z$. The same approach is used for pyramids, prisms and hexahedra. The only complications are that there are 5-bit, 6-bit and 8-bit indices in the three cases respectively, and the iso-surface may have more than one piece. This makes the lookup table larger and more complicated but it is still extremely rapid and avoids almost entirely the use of conditional branching in the code.

7.3 3D 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 [22]. 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

$$\frac{d\vec{x}}{d\tau} = \vec{u}(\vec{x}), \tag{2}$$

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 timestep, and doubled if it changes sufficiently little.

The low-level task is to calculate $\vec{u}$ for an arbitrary $\vec{x}$ on an unstructured grid. For each cell type there is a mapping from some standard cell of the same type in computational space (ξ_1, ξ_2, ξ_3). The standard cells are illustrated in Figure 7. The mapping can be expressed as

$$\vec{x}(\vec{\xi}) = \sum_j f_j(\vec{\xi})\, \vec{x}_j \tag{3}$$

where the sum is over the nodes of the cell, $\vec{x}_j$ is the coordinate vector of the j^{th} node, and f_j is a function which is equal to 1 at node j and 0 at all other nodes.

For tetrahedra the functions f_j are:

$$\begin{aligned} f_1 &= 1 - \xi_1 - \xi_2 - \xi_3 \\ f_2 &= \xi_1 \\ f_3 &= \xi_2 \\ f_4 &= \xi_3 \end{aligned} \tag{4}$$

For pyramids the functions f_j are:

$$\begin{aligned} f_1 &= \tfrac{1}{4}(1-\xi_1-\xi_3)\,(1-\xi_2-\xi_3) \\ f_2 &= \tfrac{1}{4}(1+\xi_1-\xi_3)\,(1-\xi_2-\xi_3) \\ f_3 &= \tfrac{1}{4}(1+\xi_1-\xi_3)\,(1+\xi_2-\xi_3) \\ f_4 &= \tfrac{1}{4}(1-\xi_1-\xi_3)\,(1+\xi_2-\xi_3) \\ f_5 &= 1 - (1-\xi_3)^2 \end{aligned} \tag{5}$$

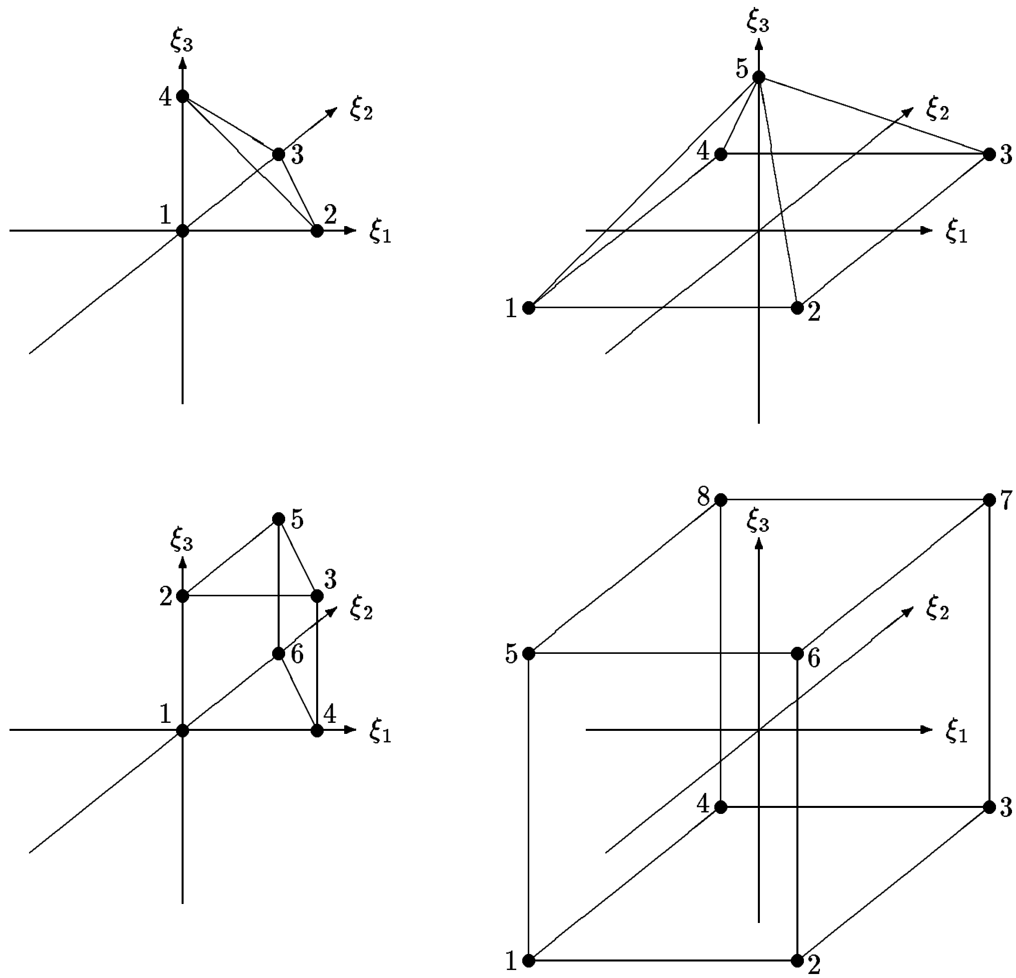

Figure 7: Four standard cells in computational space

For prisms the functions f_j are:

$$
\begin{aligned}
f_1 &= (1-\xi_1-\xi_2)\,(1-\xi_3) \\
f_2 &= (1-\xi_1-\xi_2)\,\xi_3 \\
f_3 &= \xi_1\,\xi_3 \\
f_4 &= \xi_1\,(1-\xi_3) \\
f_5 &= \xi_2\,\xi_3 \\
f_6 &= \xi_2\,(1-\xi_3)
\end{aligned}
\tag{6}
$$

For hexahedra the functions f_j are:

$$
\begin{aligned}
f_1 &= \tfrac{1}{8}(1-\xi_1)\,(1-\xi_2)\,(1-\xi_3) \\
f_2 &= \tfrac{1}{8}(1+\xi_1)\,(1-\xi_2)\,(1-\xi_3) \\
f_3 &= \tfrac{1}{8}(1+\xi_1)\,(1+\xi_2)\,(1-\xi_3) \\
f_4 &= \tfrac{1}{8}(1-\xi_1)\,(1+\xi_2)\,(1-\xi_3) \\
f_5 &= \tfrac{1}{8}(1-\xi_1)\,(1-\xi_2)\,(1+\xi_3) \\
f_6 &= \tfrac{1}{8}(1+\xi_1)\,(1-\xi_2)\,(1+\xi_3) \\
f_7 &= \tfrac{1}{8}(1+\xi_1)\,(1+\xi_2)\,(1+\xi_3) \\
f_8 &= \tfrac{1}{8}(1-\xi_1)\,(1+\xi_2)\,(1+\xi_3)
\end{aligned}
\tag{7}
$$

Given an arbitrary point $\vec{x}$, and a good guess of the cell in which it is located, a Newton-Raphson procedure is used to calculate the corresponding $\vec{\xi}$. The initial guess $\vec{\xi}^{(0)}$ is taken to be in the center of the cell, and then an improved value is given by

$$
\vec{\xi}^{(n+1)} = \vec{\xi}^{(n)} + \left(\frac{\partial \vec{x}}{\partial \vec{\xi}}\right)^{-1} (\vec{x} - \vec{x}^{(n)})
\tag{8}
$$

If the correct cell is being searched, this will converge quadratically to the correct point. If the cell being searched does not contain the point, then the new value $\vec{\xi}^{(n+1)}$ will lie outside the standard cell in computational space. The search is then switched to the neighboring cell in the direction of $\vec{\xi}^{(n+1)}$. When doing streamline integrations, the best guess for the cell location is the cell which was used for the last streamline point. In this case, the search seldom has to move through more than one or two cells.

Having found the correct cell and the value for $\vec{\xi}$, the velocity is given by

$$
\vec{u} = \sum_j f_j(\vec{\xi})\,\vec{u}_j
\tag{9}
$$

7.4 Streamribbons

Streamribbons are streamlines that have been given some width and rotation based on the local streamwise vorticity. Many implementations of streamribbons calculate two streamlines with the second line constrained to remain a constant distance away from the original line[24][25][26]. This method can be troublesome because as the second line diverges from the initial pathline, the resulting ribbon becomes a less accurate representation of the local twist. Furthermore, in a highly sheared flow, neighboring streamlines may have large velocity differences causing the slower streamline to require many more time steps before completing the integration. The construction of triangles forming the ribbon surface will also be extremely difficult in a shear flow because the slower streamline will contain many more points. The end result is the need for a complicated algorithm to build ribbon surfaces. The approach used in **Visual3** is slightly different. One edge is the true streamline, the other edge is contructed by rotating a constant length normal vector about the line according to the local streamwise angular rotation rate [27]. The result is a ribbon whose twist reflects the streamwise component of the curl of the vector field of the flow. The equation which describes the rotation of the streamline normal is:

$$\begin{aligned} \frac{d\theta}{d\tau} &= \frac{1}{2}(\vec{\omega} \cdot \vec{s}) \\ \vec{s} &= \frac{\vec{u}}{|\vec{u}|} \\ \vec{\omega} &= \nabla \times \vec{u} \end{aligned} \tag{10}$$

where θ is the current angle of the normal with respect to its initial direction. The factor of $\frac{1}{2}$ is needed because the angular rotation rate is half the curl of the vector. Equations 2 and 10 are then integrated simultaneously using the same Runge-Kutta integration algorithm. The curl of the velocity vector field is evaluated numerically using the same parametric representations in computational space as used for the streamline algorithm.

$$\begin{aligned} \frac{\partial \vec{u}}{\partial \vec{\xi}} &= \frac{\partial \vec{u}}{\partial \vec{x}} \frac{\partial \vec{x}}{\partial \vec{\xi}} \\ \Longrightarrow \qquad \frac{\partial \vec{u}}{\partial \vec{x}} &= \frac{\partial \vec{u}}{\partial \vec{\xi}} \left(\frac{\partial \vec{x}}{\partial \vec{\xi}}\right)^{-1} \\ &= \left(\sum_j \vec{u}_j \frac{\partial f_j}{\partial \vec{\xi}}\right) \left(\sum_j \vec{x}_j \frac{\partial f_j}{\partial \vec{\xi}}\right)^{-1} \end{aligned} \tag{11}$$

A possible interpretation difficulty arises because of the 360 degree periodicity of the current normal angle. To eliminate this difficulty, consecutive normal angles are forced to be at most 30 degrees different. If a separation of greater than 30 degrees occurs, the pseudo-time step is simply decreased and the current integration step is re-calculated. This constraint on the change of angle also results in the rendering of smoother ribbons.

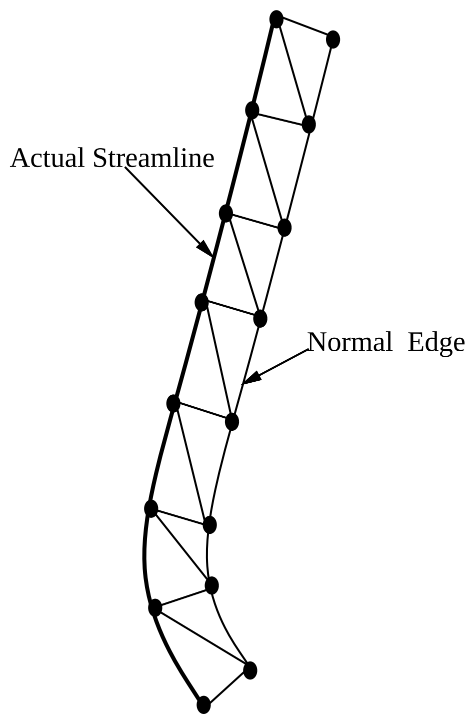

Figure 8: Polytriangle strip for a streamribbon

The final step is choosing a normal direction for the first node of the streamline. Initially, this is done automatically; however, the user may rotate this position interactively and thus rotate the entire ribbon. This rotation feature is extremely helpful in determining exactly how and in which direction a ribbon twists. Construction of the ribbon surface into polytriangle strips can be done straightforwardly by simply joining a streamline node to its normal node and then connecting this to the next streamline node. In this manner, the entire ribbon becomes a single polytriangle strip (see Figure 8) allowing faster drawing. Furthermore, this method of ribbon calculation avoids the problem of constraining a second streamline and is extremely easy to implement in comparison.

7.5 Streamtubes

The precise definition of a streamtube is the surface formed by all the streamlines passing through a closed contour[28]. The construction a streamtube involves the same difficulties as the construction of a streamribbon. To avoid these problems, a slightly different approach is used. The streamtube is considered to have a circular crossflow area with the radius derived from the local crossflow divergence. The crossflow divergence measures the local crossflow expansion rate. Specifically, we have:

$$\nabla_T \cdot \vec{u} = \nabla \cdot \vec{u} - \frac{du'}{dx'}$$

where $\nabla_T \cdot \vec{u}$ is the crossflow or transverse divergence and the primed derivative represents the change of the x' velocity component along the x'-axis aligned with the local velocity vector. The equation governing the expansion of the flow in the transverse direction is:

$$\frac{1}{r}\frac{dr}{d\tau} = \frac{1}{2}\nabla_T \cdot \vec{u}$$

which may be rewritten as:

$$r = r_0 \exp\left\{\frac{1}{2}\int_{\tau_0}^{\tau} \nabla_T \cdot \vec{u}\, d\tau'\right\} \tag{12}$$

Thus, r, the crossflow radius, is an exponential function of the streamwise integral of the crossflow divergence. The integral in Equation 12 is integrated forward in pseudo-time along with Equation 2 and Equation 10. The velocity gradients are obtained numerically in the same way as for the streamribbons. The initial crossflow radius, r_0, is controlled by the user.

To render the tube, each circular crossflow section is divided into equal arcs. We have found that 30 degree arcs provide adequately smooth tubes. For most streamtubes, rendering poly-triangle strips in the streamwise directions results in longer strips than if crossflow sections were rendered; thus, rendering tubes in the streamwise direction will increase the algorithm performance. A user-set radius limiter is also needed to insure the width of a streamtube does not become so large so as to obscure an entire image. This rapid growth in the streamtube width is especially difficult near stagnation points where the radius becomes extremely large.

7.6 Streamtubes with twist

The rotation and divergence effects can be rendered simultaneously by placing lines on the surface of the streamtube which twist with the local rotation rate. The only additional difficulty which arises in rendering the twist lines is that the lines must lie on a facet between poly-triangle strips. This guarantees that the lines will lie on top of the streamtube surface. However, to place the twist lines on a facet requires that the poly-triangle strips must also twist as they travel downstream. Thus, the entire rendering algorithm consists of:

- Render twisted streamwise poly-triangle strips
- Render twisted lines along facets of poly-triangle strips

7.7 Unsteady Particle Paths

The final variation of streamlines is unsteady particle paths. This involves an integration in time and space. Specifically, the position of a particle is given by:

$$\frac{d\vec{x}}{dt} = \vec{u}(\vec{x}, t) \tag{13}$$

In choosing a numerical scheme to integrate Equation 13, three important factors must be considered:

1. The velocity vector field, $\vec{u}$ is changing in time. If the numerical scheme requires several time planes of the velocity field, either all time planes will remain in memory or each time plane will be read from disk when needed. Thus, for large data sets, this type of scheme will place extreme demands either on memory usage or memory bandwidth. Thus, optimally, the integration should require only one time plane to be active in memory.

2. The integration should have good accuracy to make certain the numerical particle paths are a reasonable representation of the true paths. Thus, simple schemes such as forward or backward Euler integration, although requiring only one time plane of the velocity field will be inadequate. Furthermore, adaptive time stepping cannot be performed because the time step between new positions is set by the time step of the user's data.

3. Since the time step is not determined by **Visual3**, the numerical technique must be stable for all step sizes. In other words, if the local linearized problem predicts an exponentially decaying change in position, then the numerical scheme should also have a decaying solution. If the local linearized problem has an oscillatory solution then the numerical scheme should at least not predict an exponentially growing solution.

With these three factors in mind, the obvious choice for the integration scheme is backwards differentiation. The general form of an order s accurate backwards differencing formula is:

$$\sum_{i=1}^{s} \frac{1}{i} \nabla^i \vec{x}^{n+1} = k \; \vec{u}^{n+1} \tag{14}$$

where k is the time step and ∇ is the usual backwards difference operator:

$$\nabla \vec{x}^{n+1} = \vec{x}^{n+1} - \vec{x}^{n}$$

This scheme is stable up to the 6th order formula. We use the fourth order scheme which matches the accuracy of the Ringe-Kutta integration used for the streamlines. Only the new time plane of the velocity vector field is needed for the integration. The calculation does require the storage of the s previous positions but this is simply 3 floating points per path per time step and this information is generally kept for rendering purposes.

Equation 14 can be re-written as

$$(1+\tfrac{1}{2}+\tfrac{1}{3}+\tfrac{1}{4})\,\vec{x}(\vec{\xi}^{(n+1)}) - k\,\vec{u}(\vec{\xi}^{(n+1)}) = \text{function}(\vec{x}^{(n)}, \vec{x}^{(n-1)}, \vec{x}^{(n-2)}).$$

This equation is nonlinear in $\vec{\xi}^{(n+1)}$ and is solved by a minor variation of the Newton-Raphson algorithm used in the streamline calculation to determine the computational coordinates $\vec{\xi}$.

7.8 Xrays

The X-ray algorithm is a complicated calculation which, when considering a single ray, is very similar to streamline integration. The algorithm consists of tracing a straight line through the domain and integrating the attenuation constant encountered during the trace. Before describing the complete algorithm, the following terminology is necessary. The computational coordinate system is simply (x, y, z). The screen coordinate system is denoted by primes, (x', y', z') with the x'-axis tangent to the screen plane and pointing to the right, the y'-axis tangent to the screen plane and pointing upward, and the z'-axis directed into the screen. This defines a left-handed coordinate system which is standard in 3-D graphics. Boundary faces are the cell faces which make up the computational domain boundaries. The X-ray grid is set-up such that the origins of the X-rays form a square Cartesian grid centered on the z'-axis, having a unit length side, and aligned with the x'- and y'-axes (Figure 9).The number of rays on a side is N_{rays} and the total number of rays is N_{rays}^2. The X-ray grid is simply a two-dimensional structured grid of the form, $xr(i, j)$ where $1 \leq i, j \leq N_{rays}$. Thus, a simple algebraic mapping exists relating (i, j) to the X-ray origin in the screen coordinate system. Finally, because **Visual3** allows tetrahedra, pyramid, prism, and hexahedra cells, it was necessary to handle both triangular and quadrilateral cell faces. This problem was solved by breaking all quadrilateral faces into triangles. Thus, calculations are performed only on triangular faces.

For a single ray, the X-ray algorithm is as follows:

1. Determine the boundary faces which the X-ray intersects and sort them by distance from the X-ray plane (i.e., the z' coordinate at the intersection).

2. Determine the first cell which the ray enters from the sorted boundary face list.

3. Calculate the cell face from which ray exits and perform the numerical integration of Equation 1 using a trapezoidal approximation from cell entrance to cell exit.

4. If the ray has entered another cell repeat the previous step.

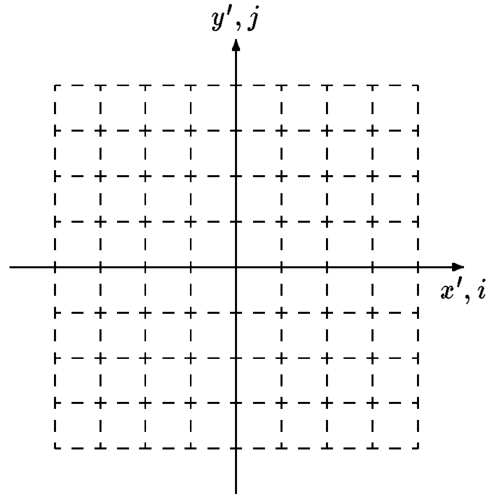

Figure 9: X-ray grid

5. If the ray has left the domain, determine if it re-enters by scanning the sorted boundary face list for an intersection beyond the current location. If so, continue the cell-to-cell integration process. If not, the integration is done.

Upon trying this algorithm the speed was too slow to allow adequate interactivity. A profiling of an X-ray calculation revealed that much of the time was being spent in determining which boundary faces a ray intersects. The problem with the algorithm is that for each X-ray a calculation must performed over the list of all boundary faces.

In order to increase the speed, it was necessary to alter the boundary face calculations. Instead of calculating which boundary faces intersect a given ray, the problem was reversed; in other words, the rays which intersect a given boundary face were calculated. Thus, the new boundary algorithm contains one loop over all boundary faces. The result is a 'linked list' of all the faces which intersect each X-ray (i, j). Specifically, the algorithm for boundary face k is:

1. Calculate the screen coordinates, $\vec{x'}(i)$, of the triangular boundary face k where $i = 1, 2, 3$ for each node of the triangle.

2. Determine the minimum and maximum y' values. The minimum and maximum points are now called $\vec{x'}_{bot}$ and $\vec{x'}_{top}$; the remaining point of the triangle becomes $\vec{x'}_{mid}$.

3. Determine the j range of X-rays which face k spans from y'_{bot} and y'_{top}. Loop over the range from (j_{bot}, j_{top}).

4. Determine x'_{left} and x'_{right} for y'_j; if $y'_j < y'_{mid}$ then sides a and b are used; else sides a and c are used.

5. Determine i_{left} and i_{right} from x'_{left} and x'_{right}.

6. Add boundary face k to the linked list for X-rays (i_{left}, j) to (i_{right}, j).

7. Continue loop over j-range.

8. Boundary face k is completed; continue loop over boundary faces.

This algorithm significantly improves the speed. The first step of the original algorithm is now completely eliminated; otherwise, the general procedure remains exactly the same. A final gain in speed may be obtained by parallelizing the integration of each ray. This is possible because the calculation for each ray is parallel certain.

References

[1] M. B. Giles and R. Haimes. Advanced Interactive Visualization for CFD. *Computing Systems in Engineering*, 1(1):51–62, 1990.

[2] R. Haimes and M. Giles. VISUAL3: Interactive Unsteady Unstructured 3D Visualization. AIAA Paper 91-0794, 1991.

[3] D. Darmofal and R. Haimes. Visual Feature Identification for 3-D Data Sets. AIAA Paper 91-1583, 1991.

[4] R. Haimes and D. Darmofal. Visualization in computational fluid dynamics: A case study. IEEE Computer Society, Visualization '91, October 1991.

[5] P. Buning and J. L. Steger. Graphics and flow visualization in computational fluid dynamics. AIAA Paper 85-1507, 1985.

[6] T. J. Baker. Automatic mesh generation for complex three-dimensional regions using a constrained delauney triangulation. *Engineering with Computers*, 5:161–175, 1989.

[7] R. Löhner, K. Morgan, J. Peraire, and O. C. Zienkiewicz. Finite element methods for high speed flows. AIAA Paper 85-1531, 1985.

[8] J. Peraire, J. Peiro, L. Formaggia, K. Morgan, and O. Zienkiewicw. Finite element euler computations in three dimensions. *International Journal for Numerical Methods in Engineering*, 26:2135–2159, 1988.

[9] J. F. Dannenhoffer III and J. R. Baron. Robust grid adaptation for complex transonic flows. AIAA Paper 86-0495, 1986.

[10] S. Samant, J. Bussoletti, F. Johnson, R. Burkhart, B. Everson, L. Melvin, D. Young, L. Erickson, M. Madson, and A. Woo. Tranair: A computer code for transonic analyses of arbitrary configurations. AIAA Paper 87-0034, 1987.

[11] E. M. Murman, A. R. LaVin, and S. C. Ellis. Enhancing Fluid Machanics Education with Workstation Based Software. AIAA Paper 88-0001, 1988.

[12] M. M. Rai. Unsteady three-dimensional Navier-Stokes simulations of turbine rotor-stator interaction. AIAA Paper 87-2058, June 1987.

[13] Y. S. Chen. 3-d stator-rotor interaction of the ssme. AIAA Paper 88-3095, 1988.

[14] L.-E. Eriksson, R. Smith, M. Wiese, and N. Farr. Grid generation and inviscid flow computations about cranked-wing airplane geometries. AIAA Paper 87-1125, June 1987.

[15] K. Nakahashi. A finite-element method on prismatic elements for the three-dimensional navier-stokes equations. Proceedings of the 11th International Conference on Numerical Methods in Fluid Dynamics, June 1988.

[16] Stellar Computer Inc. Application Visualization System Users Manual, 1989.

[17] J. F. Dannenhoffer. GRAFIC - an interactive graphics package for scientific applications. Technical Report 89-39, United Technologies Research Center, 1989.

[18] R. F. Weston. Color graphics techniques for shaded surface displays of aerodynamic flowfield parameters. AIAA Paper 87-1182, 1987.

[19] R. Löhner, P. Parick, and C. Gumbert. Some algorithmic problems of plotting codes on unstructured grids. AIAA Paper 89-1981, 1989.

[20] S. Webb. *The Physics of Medical Imaging.* IOP Publishing Ltd., 1988.

[21] W. E. Lorensen and H. E. Cline. Marching Cubes in a High Resolution 3D Surface Construction Algorithm. *ACM Computer Graphics*, Vol. 21(4):pp. 163–169, 1987.

[22] 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.

[23] D. L. Modiano, M. B. Giles, and E. M. Murman. Visualization of three-dimensional CFD solutions. AIAA Paper 89-0138, 1989.

[24] G. Volpe and R. E. Melnik. The design of transonic airfoils by a well-posed inverse method. Proceedings, International Conference on Inverse Design Concepts in Engineering Sciences, October 1984.

[25] J.P.M Hultquist. Interactive Numerical Flow Visualization Using Stream Surfaces. *Computing Systems in Engineering*, 1(2-4):349–353, 1990.

[26] M. Siclari, M.J. Mandel and A. Scuderi. Flow Visualization Research at Grumman's Computational Aerodynamics Laboratory, 1989.

[27] D. Darmofal. Hierarchal Visualization of Three-Dimensional Vortical Flow Calculations. Master's thesis, M.I.T., March 1991.

[28] G. K. Batchelor. *An Introduction to Fluid Dynamics.* Cambridge University Press, 1967.