Techniques for Interactive and Interrogative Scientific Volumetric Visualization

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

Abstract

This paper presents some key aspects of VISUAL3, a highly-interactive visualization environment for the investigation and interrogation of 3D volumetric scientific data.

First, the program architecture and data structure are described. This supports both structured and unstructured grids with minimum memory requirements for large data-sets.

Second, methods are presented for avoiding the need to perform extremely expensive searches over large data-sets. This greatly speeds up the interactivity of critial processes.

Third, an important new rendering idea is introduced for dynamic situations in which the transformation matrix and most of a complex object is static and only a small part of the object is changing.

1 Introduction

VISUAL3[1][2] is a highly interactive 3D volumetric visualization environment. It was originally designed to be able to view the types of data that are currently generated by the Computational Fluid Dynamics (CFD) community but it is not limited to CFD. VISUAL3 deals with scalar and vector field data distributed throughout a volume that has been discretized in either a structured or unstructured manner.

Because of the original focus on CFD needs, VISUAL3 is designed to cope with extremely large data-sets. Large CFD calculations can have more than a million grid cells with a number of scalar and vector variables defined at each grid node, in addition to the grid coordinates. To make matters even more difficult, some 3D CFD codes now produce time-varying (unsteady) data. Designing an interactive system that can view this data required approaches to visualization that are non-traditional.

Another aspect of VISUAL3 which is unusual is the use of multiple windows and a variety of interactive tools (or probes) which have proven to be very useful in the interrogation of volumetric data [3]. VISUAL3 allows the user to view not only the 3D object(s), but it also supplies a 2D window where active surfaces or cuts are drawn. Pointing in the 2D window, by reversing the 3D to 2D mapping, defines a unique place in 3-space. This allows the user to easily extract quantitative data and point to seed or probe locations instead of having to type in coordinates. Also data may be extracted from the 2D window in the form of line plots that are drawn in a 1D window. Cursor mapping is also applied to the 1D data.

2 Architecture

VISUAL3 is an X-Windows application (that also requires some drawing support for 3D Z-buffering and Gouraud shading). Therefore, the life of the VISUAL3 job is defined by an X event loop. There are three parts to the loop that require only small changes depending on whether the data is steady or unsteady[2].

The first phase is event collection. The user input is pulled off the X event queue and internal flags, transformation matrices and other internal data are changed appropriately. The second phase is data collection. If the user has requested a different field variable or the case is unsteady, user portions of the application are called by VISUAL3. Also, any required recalculation of surfaces is performed in this section. The final phase is the drawing portion. When the data in any window has changed the window is redrawn to reflect the new data.

VISUAL3 is always updating the information presented to the user in response to user requests. To ensure that this animation is interactive all data is stored in physical memory, but because of the size of the problems of interest, memory usage must be kept to the minimum required. The functionality which is required includes the following:

- Quick Changes in Field Variables

  There is only one current scalar and vector field variable. To make scalar comparisons 'on the fly' requires that the windows be updated quickly. For cuts through the volume based only on geometric data, the cut surface should not be recomputed. Therefore the color must not be an integral part of the internal data.

- Cursor Mapping

  To facilitate pointing with cursor mapping enough data should be maintained to go back and forth between 3D, 2D and 1D structures.

- Grid Viewing

  The investigator should be able to determine if some feature in the data is grid related. The ability to plot the intersection of the 3D mesh and any surface is important.

The architecture selected to accomplish this is one that allows all internal modules to VISUAL3 access to all the data structures. This is very different from the visualization architectures currently in vogue. Both the 'data flow' and 'filter-like' architectures produce copies (modified or unmodified) of the data. This is fine for smaller data-sets but for the types of data produced by CFD codes, one copy of the data is already huge!

Another problem with 'data flow' and 'filter-like' architectures is that they don't allow feedback, that is, information back up to modules that have already executed. This feature is important for pointing and cursor mapping. In the VISUAL3 paradigm feedback is automatically possible because one module can effect the global data or draw on multiple windows.

The one disadvantage with the architecture of VISUAL3 is that the development and management of the code in VISUAL3 is harder. It cannot be developed in a large team and structured programming techniques do not work. A change in one module or in the data structure can effect the entire program.

3 Internal Data Structures

The way the data is stored and subsequently handled in VISUAL3 satisfies the requirements for low memory usage and interactive speed. All data is based on either a 3D node or 3D cell numbering scheme. The disjoint cell types supported are tetrahedra, pyramids, prisms and hexahedra. The block cell types are poly-tetrahedra and structured blocks of hexahedral cells. The node numbers are explicitly defined for the disjoint and poly-tetra types and implicitly defined for the block structured type. The cell numbers are explicitly defined (for heterogenous meshes) by all the tetrahedra, pyramids, prisms, then hexahedra, in that order, and implicitly defined by the poly-tetras and then structured blocks.

3.1 3D Cell Structures

The following structures describe the way that the volume has been discretized.

- int cellte[ncellte][4]

  Cell definitions for tetrahedra, where each entry is defined by the 4 nodes that make up the cell.

- int cellpy[ncellpy][5]

  Cell definitions for pyramids, where each entry is defined by the 5 nodes that make up the cell.

- int cellpr[ncellpr][6]

  Cell definitions for prisms, where each entry is defined by the 6 nodes that make up the cell.

- int cellhe[ncellhe][8]

  Cell definitions for hexahedra, where each entry is defined by the 8 nodes that make up the cell.

- int mpolytet[npolyt], polytet[]

  Defines each poly-tetra, where the entry in **mpolytet** is the last node in **polytet** for the strip. **polytet** contains the node numbers that make up the strip and the total length is defined by **mpolytet**[npolyt]. Each strip defines **mpolytet**[i]-**mpolytet**[i-1]-3 tetrahedra.

- int sblocks[nblock][3]

  Defines the size of each structured block in the number of nodes in each direction. Each block defines (**sblocks**[i][0] - 1)*(**sblocks**[i][1] - 1)*(**sblocks**[i][2] - 1) hexahedra and contains **sblocks**[i][0]***sblocks**[i][1]***sblocks**[i][2] nodes.

3.2 Domain Surface Structures

The surfaces that bound the volume are defined as groups of disjoint cell faces. This data is required so that the user may turn on or off the rendering of any of the bounding surfaces of the volume. Also when a domain surface is mapped to the 2D window, enough information exists to allow cursor mapping.

- int msurf[nsurf], surf[][4]

  Defines the domain surface groups, where an entry in **msurf** points to the last surface face in **surf** for that group. Each element of **surf** contains one of the 3D nodes that makes up the surface face. A triangular face only fills the first 3 elements.

- int cellS[nsurf]

  This data points back to the 3D cell that contains the face.

3.3 3D Node Structures

As described above the total number of 3D nodes (**nnode3D**) is the number of nodes used by the disjoint cell types and poly-tetras plus the number of nodes in each structured block.

- float scalar[nnode3D]

  Storage used for the current scalar field variable.

- float threshold[nnode3D]

  Storage used for the current threshold field variable.

- float vector[nnode3D][3]

  Storage used for the current vector field variable.

- float coord[nnode3D][3]

  Storage used for the coordinate triad for each node.

3.4 2D Surfaces

2D surfaces are produced in VISUAL3 by the task of volume slicing through the collection of disjoint 3D cells. Block cell types are viewed by the algorithm as their individual component cells. The procedure is used for the cutting plane, programmer-defined (general) cut surface, and iso-function surfaces, which are three of the scalar function 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 3D nodes, and a set of unstructured 3D cells,

Table 1: Look-up table for tetrahedra

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

how does one determine the cells that are intersected by the surface $z' = Z$ (where Z is some constant value), and then how to generate the facets produced by the intersection.

The process of constructing the surface can be broken into two parts. The first part is to determine the cells in the volume that the surface intersects, the second is to produce the surface facets within the cell. The first part is discussed in Section 4.3.

For those cells that are cut by the surface, an extremely fast method for determining the facet(s) produced is used. This technique is similar to that used by W. Lorensen [4] but no assumption of symmetry is implemented. Full look-up tables are employed for each disjoint cell primitive.

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.

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 there may have more than one polygon produced. 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.

During the process of constructing the surface, no coordinate data or colors are stored, but enough information is retained to allow this data to be constructed. All data stored points back to the 3D structures for nodes and cells. This is very different from traditional and PHIGS storage schemes where an object is defined by the coordinates of new nodes generated on the cut surface, and if color shading is to be performed, the color at each point. Any modification in color mapping requires regenerating the entire surface, even though the geometry of the surface has not changed.

VISUAL3's surface definition allows fast changing of the scalar function or simple changes to the color mapping. Thresholding of the surface can be accomplished by using the threshold variable during the rendering of the data. Also other data can be easily displayed without recomputing the surface. Calculating and displaying the vector field as cut by the surface is trivial.

Again the drawback of this approach is complexity. The routines that extract surface information require rendering routines that do more that just plot surfaces. The code that actually draws to the window must collect all the appropriate data, compute and construct the surface as it is being passed to the graphics subsystem.

The 2D structures created by the volume slicing contain:

- int node[nnode2D][2]

  Defines the '2D' node based on the two 3D node indices. Because only linear interpolation is used, the surface will only cut the edge of a cell at one point. The '2D' node is that point.

  During the construction of the surface, a '2D' node will be hit by all the cells that share an edge. To avoid making the number of '2D' nodes needlessly large, an efficient algorithm is used to determine if a node already exists (see Section 4.2).

- float interp[nnode2D]

  A number between 0.0 and 1.0 that defines the distance from the first 3D node in **node**.

- int tri2D[ntri][3]

  All 2D surfaces are defined as disjoint triangles. Each of the entries contain a '2D' node number. Cutting a hexahedra can produce up to a 7 sided polygon. All polygons are reduced to the appropriate number of triangles.

- int cell2D[ntri]

  This data refers back to the 3D cell that produced the disjoint triangle. This allows cursor mapping to function back into the 3D window.

- int grid[ngrid][2]

  The intersection of the surface and the underlying mesh is defined by disjoint lines. Each entry contains the 2 '2D' node end-points. The lines are produced by storing the outline of the original polygon produced from the look-up table.

  Like the '2D' nodes, during the construction of the surface, a grid segment will be shared by neighboring 3D cells. To half the number of grid segments, a similar algorithm is used to determine if a line already exists (see Section 4.2).

- int edge[nedge][3]

  The intersection of the surface and the domain surfaces (that bound the volume) is also defined by disjoint lines. Each entry contains the 2 '2D' end-points and the bounding surface index. This is used to outline the intersection in the 3D window (so visually the surfaces don't blend into each other) and allow 1D plotting of the scalar at the intersection of a particular domain surface.

3.5 1D Structures

Most simple lines that are created, by probes, for the 1D window are constructed 'on the fly' from the information in the 2D structures. Integrated information [5] such as streamlines, that require much computational processing, are stored in the following 1D structure. Again, like the 2D surfaces all data refers back to the 3D structures.

- int cell1D[linelen]

  Each entry is a point in the line. This data refers back to the 3D cell that contains the data point.

- float weight1D[linelen][8]

  The weights (0.0 - 1.0) of each of the nodes that make up the cell. For cell types with fewer than 8 nodes, only the appropriate entries are filled.

  These weights are used to reconstruct the data point. Therefore, the coordinate triad, scalar and vector function at the point can be computed. This storage scheme allows quick color map changes and also mapping the line directly to the 1D window. This structure also facilitates cursor pointing from the 1D window directly to the 3D window.

4 Searching

When attempting to interactively visualize 3D volumetric data as dense as the problems routinely computed in CFD, searching the entire data-set must always be avoided. Even techniques such as oct-tree have problems, especially when the data is unstructured. During the development of VISUAL3, the approach to searching was "Just say NO!".

Following is a short description of some alternative approaches to the 'search' problem.

4.1 Connectivity

During the process of moving through the volume along a line (such as with ray casting, streamline and particle path integration) one cell is exited, so to continue it is necessary to know which is the neighboring cell that is entered. In a volume discretized in a simple block manner, the answer is obvious. In the general case this might require searching for the next cell.

VISUAL3 requires cell connectivity, which tells the line integration procedure, what cell is entered when leaving a particular face of the neighboring cell. This produces fast integrations but a large penalty is paid in memory usage. Implicit cell connectivity in the block primitives is maintained, but each disjoint primitive requires four, five, five, and six face pointers per cell for tetrahedra, pyramids, prisms and hexahedra respectively. The option exists in VISUAL3 to deallocate the connectivity tables if the user does not wish to use the tools that require this information.

4.2 Threaded Lists

Threaded lists were used during procedures in VISUAL3 that produce certain structures, such as the 2D surfaces and the connectivity tables. These threaded lists avoid searches. For example, consider the calculation of the connectivity tables:

A face table is constructed with five entries for each face. The first three entries are the indices of three of the four nodes that define the face. The cell types supported have either quadrilateral or triangular faces. The fourth 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 fifth 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 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 disjoint cell type 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 three entries equal to I1, I2, I3, and I4 (leaving out Imin) and the fourth 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 fourth 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 five. 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 fifth 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 fourth position are faces that bound the volume. Finally, the memory for the face table is deallocated.

The process used for '2D' nodes and grid segments is much simpler. The table used for this processing consists only of two entries. The first entry is the other (higher numbered) node and the second entry is the 'thread'. If the '2D' data exists, the position of the match in the table indicates which index to use for the '2D' node (or grid segment), if not, a new element is added. When a new element is added the position in the table is used as the index.

4.3 Volume Slicing

The first step in volume slicing is to determine which cells are cut by the surface of interest. 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 [6] 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. 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 volume.

The approach used by VISUAL3[1] 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 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 quicksort 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 $Z_{new} > Z_{old}$, 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 volume.

When an iso-surface is the scalar tool, the sorted lists of cells implicitly contain scalar field topology information. Currently this data is not being exploited. Future work may involve abstracting and visualizing topological changes at critical values of Z.

5 Snap-Shot Rendering

What has been discussed to this point are techniques that speed up the compute portion of the volume visualization task. There is one important method that allows much faster rendering of complex objects. This is 'snap-shot' rendering which involves saving away the pix-map (and Z-buffer for the 3D window) at some point during the scene rendering process. This snap-shot is later used as the starting point for the next scene in the animation.

Watching users work with VISUAL3 it became obvious that most of the time spent viewing the data is not rotating, panning, scrolling, zooming or otherwise changing the transformation matrix. Instead, time is spent using the tools such as moving a cut-plane, changing the iso-surface value, or simply moving the cursor and looking at the cursor tracking in the other windows. In these instances the entire picture need not be redrawn; only those parts that have changed require re-rendering.

To emphasize this point, consider an airplane viewed in the 3D window. The investigator has finished changing the viewing position and starts a cutting plane. As the cutting plane is moved through the domain the airplane surface doesn't change. The rendering performance in VISUAL3 can therefore be improved by saving the airplane surface, both the pixmap and the Z-buffer, and then using this as the base on which to render the latest cutting plane.

The cost of this 'snap-shot' rendering is the time to copy the saved pix-map and Z-buffer into the appropriate locations before rendering the dynamic part. The savings are the time which would otherwise be spent re-rendering the saved static part. Clearly, the net savings depends on the complexity of the static object(s). For example, an SGI 4D/310-VGX can render about 160,000 disjoint 3D Gouraud shaded triangles/second (my benchmark!), and restore a pix-map and Z-buffer of a window 760x760 pixels (the size of the VISUAL3 3D window) in less than 1/20 of a second. Therefore the break-even point is about 8,000 triangles. It is clear that in the airplane example (which may require 60,000 triangles just to define the surface) 'snap-shot' rendering is a **BIG** winner.

For this type of rendering to work, it is necessary to order the scene drawing so that any static surfaces are rendered first, the snap-shot taken, then the dynamic or potentially changing part of the scene drawn. Also, depending on the type of transparency used, it may be necessary to render transparent surfaces after the dynamic portions, therefore even if static, transparent surfaces should not be saved in the snap-shot.

This technique makes cursor mapping to the 3D window very interactive. The pseudo-cursor drawn on the surface mapped to the 3D window quickly tracks because very little drawing is required.

Obviously this technique can only work with immediate mode systems. Again PHIGS is not a good model and even though the PHIGS system contains the entire data suite, any change in the scene requires a complete redraw to the window. The current PEX servers, even though they support immediate mode, don't allow any access to the Z-buffer. The philosophy of PEX is to separate the programmer from the hardware. The process of 'snap-shot' rendering does not require modification of the Z-buffer, only saving it (with the drawable) at a specific time during scene generation. The required data could be saved in the PEX server, left unmodified, then used to start the next scene and 'snap-shot' rendering would function. This has been proposed to the PEX committee for inclusion in a future version of PEX. Without this feature, using SGI's GL rendering (which does allow access to the Z-buffer) will always outperform using PEX for VISUAL3 on equivalent hardware.

6 Conclusions

Many performance issues had to be solved during the design and development of VISUAL3 in order to achieve the desired interactivity and allow interrogation of large scale 3D volumetric data-sets. Handling the types of data produced by CFD codes required some re-thinking of the standard approaches to scientific visualization.

The use of an architecture that provided all the data to each module within VISUAL3 as well as 2D and 1D data structures that reference back to the 3D volume were found to be essential. During many of the calculations required to perform the tools and probes, searching was carefully avoided. Finally, 'snap-shot' rendering greatly improves drawing performance in almost all cases when the transformation matrix is not changing.

References

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

[4] 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.

[5] D. Darmofal and R. Haimes. Visualization of 3-D Vector Fields: Variations on a Stream. AIAA Paper 92-0074, 1992.

[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.