Visualization of Transient Finite Element Analyses on Large Unstructured Grids

Donald Dovey

This paper was prepared for submittal to the IEEE Visualization '95 Conference
Atlanta, Georgia
October 30–November 3, 1995

March 22, 1995

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED
DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Visualization of Transient Finite Element Analyses on Large Unstructured Grids

Don Dovey
Lawrence Livermore National Laboratory
P.O. Box 808, L-122
Livermore, California 94550

Abstract

Three-dimensional transient finite element analysis is performed on unstructured grids. A trend toward running larger analysis problems, combined with a desire for interactive animation of analysis results, demands efficient visualization techniques. This paper discusses a set of data structures and algorithms for visualizing transient analysis results on unstructured grids and introduces some modifications in order to better support large grids. In particular, an element grouping approach is used to reduce the amount of memory needed for external surface determination and to speed up “point in element” tests. The techniques described lend themselves to visualization of analyses carried out in parallel on a massively parallel computer (MPC).

1 Introduction

Transient finite element analysis codes developed in the Mechanical Engineering and Electrical Engineering departments at Lawrence Livermore National Laboratory (LLNL) all operate on unstructured grids containing linear elements. These include codes for nonlinear dynamics (DYNA3D and NIKE3D), heat transfer (TOPAZ), fluid dynamics (HYDRA), acoustics (PING), and electromagnetics (DSI3D). In addition, a commercial casting simulation code (ProCAST) is used by engineering analysts in Mechanical Engineering to simulate mold-filling. The visualization methods described in this paper have been implemented in an interactive data visualizer for unstructured grids called GRIZ. GRIZ allows engineering analysts at LLNL to visualize and animate the results of the above analysis codes.

The result data from a finite element analysis is organized into connectivity data and state data. The mesh connectivity consists of a list of nodes (giving the spatial coordinates of each) and a list of elements. Each element is specified by a canonical list of the nodes which make up its vertices and by a material type. Volume elements which may occur in the mesh include hexahedral (brick-shaped) elements and “degenerate” hexahedral elements — wedges, pyramids and tetrahedra (Figure 1). For nonlinear dynamics problems, volume meshes are composed primarily of hexahedral elements. There may be anywhere from several thousand to several million elements in a grid. The state data consists of a series of states in which the result quantities from an analysis are saved for successive instants in time. The result quantities in each state may include scalar data such as effective stress, effective plastic strain, temperature and pressure; vector data such as nodal positions, velocities, accelerations, and electric and magnetic fields; and tensor data such as stress and strain tensors. The particular result data saved depends on the type of analysis being performed.
The solid mechanics codes DYNA3D and NIKE3D simulate nonlinear material deformation. Phenomena which may be modeled include automobile collisions, earthquake response of large structures, metal-forming processes, and the behavior of human tissue such as muscles. In these problems, the mesh nodes change position over the course of the simulation as the material is deformed. This means that surface normals computed for lighting purposes also change. In addition, DYNA3D models material failure by deleting elements when some failure criteria is reached during the course of a simulation [5] — effectively changing the topology of the grid and potentially forcing the visualization program to recompute the visible surface of the grid at each new state. Finally, efforts are under way to create a parallel version of DYNA3D (and several of the other codes listed above) which will run on a 256-node Meiko massively parallel computer. The parallel code is expected to increase by an order of magnitude the number of elements that can be included in an analysis. Currently, a large supercomputer problem might have from 30,000 to 1 million elements. These considerations attest the need for algorithms which will quickly recompute the visible surface of a mesh and the surface normals. Such algorithms are explored in the next sections. In the fourth section, the problem of locating the element which contains a specified point is addressed. The solution to this problem has applications in vector field visualization and other areas.

2 External Face Determination

The most common visualization technique for finite element results is displaying a scalar result value color-mapped on the surface of the grid. To do this efficiently, it’s first necessary to determine which element faces constitute the external surface of the mesh. In [1], Christon and Spelce describe an efficient method for external face determination which quickly recomputes the external faces after elements are deleted. Their algorithm proceeds in two stages. It first determines element adjacency and then uses the adjacency to extract the external faces of the mesh. We summarize the algorithm of Christon and Spelce next, and then introduce improvements to the algorithm.

2.1 Simple Algorithm

In the discussion that follows, we assume the mesh is composed of only hexahedral elements. The extension to other element types is straightforward. We further assume that there are \( N_{hex} \) volume elements in the mesh.

The data structures used in this algorithm are an Element Adjacency Table and a Face Table (Figure 2). The Element Adjacency Table contains, for each element, six numbers which identify the elements adjacent to that element along each of its six faces. The Face Table is a temporary array which is used to determine the element adjacency. It contains \( 6 \cdot N_{hex} \) entries, or one for each element face. Each entry in the Face Table consists of the four nodes of the face, the element number, and the location of the face on the element (1-6). The four node numbers in each face entry are sorted in ascending order and then the whole Face Table is sorted using the node numbers for comparison. In our implementation, the Face Table entries are left in place and an index array is sorted.

If two elements in a mesh are adjacent along a shared face, they will each have an entry for the shared face in the Face Table. The two entries end up next to each other in the sorted table. When the Face Table has been sorted, the algorithm walks through the table. If a face occurs in the table twice, then it is an internal face and the two elements that share the face are set to point to each other in the Element Adjacency Table. If a face has a single entry in the table, then it is an external face and the adjacency pointer in the Element Adjacency Table is set to NULL. After the algorithm has finished stepping through the Face Table, the entries in the Element Adjacency Table are complete and the Face Table is discarded. All of this processing takes place only once, at the start-up of the visualization program.

To quickly determine the external faces at a given state, a third table is introduced. The Visibility Table contains one entry for each element, which specifies

<table>
<thead>
<tr>
<th>Face Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node1  Node2 Node3 Node4 Elem Face</td>
</tr>
<tr>
<td>(6 ( N_{hex} ) entries)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element Adjacency Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adj Elem1 Adj2 Adj3 Adj4 Adj5 Adj6</td>
</tr>
<tr>
<td>(( N_{hex} ) entries)</td>
</tr>
</tbody>
</table>

Figure 2: Tables Used in External Face Determination
whether the element is currently visible or not. The analysis codes output a list of currently "active" and currently deleted elements at each state, and this information is used to update the visibility table. External face determination proceeds by visiting each element. If the element is visible, then the visibility of its six neighbors is checked via the Element Adjacency Table. If the element has no neighbor along a face or if the neighbor element along a face is marked as invisible, then that face is external and must be rendered.

We have modified this algorithm slightly by introducing the notion of explicitly generating the Element Adjacency Table and discarding the Face Table. The algorithm allows the code to quickly recompute the external surface of the mesh at each time step — which is critical to providing interactive animation of analysis results.

The algorithm supports other functionality as well. For example, the user can mark all elements composed of a given material as invisible and then redisplay the mesh without that material. This lets the user examine analysis results at material boundaries. The original visibility is restored by reloading the element “activity” list from the state data. We've implemented a fast "rough" cutting plane using a simple technique. All elements that intersect or lie on one side of a cutting plane are marked invisible — exposing a jagged interior surface of the mesh. This feature allows the user to visually check that elements in the interior of the volume are well-shaped. A third use is in casting simulations. In these simulations, a liquid (hot metal) flows into a pre-meshed cavity. As each element fills with liquid, it is switched from invisible to visible. This makes it possible to animate the advancing front of the liquid (see Figure 6).

The extension to non-hexahedral elements is straightforward. A separate Face Table is created for triangular faces, which is then sorted and traversed in the same manner as the quadrilateral Face Table. An alternative way to handle triangular faces is to load their three nodes into the quadrilateral Face Table and set the fourth node number to -1. This distinguishes triangular faces from quadrilateral faces during the sort.

Note that the Element Adjacency Table has other uses than just external face determination. For example, we use it to efficiently track the paths of particles in a flow field. Particle tracking involves numerically integrating the velocity of massless particles (the velocity is obtained by interpolating a velocity field to the particle position) in order to determine their position over time [4]. At any given integration step, a particle may move out of the current element. The face through which the particle left is determined, and the particle tracker moves to the adjacent element along that face by performing a lookup in the Element Adjacency Table. This step is repeated until the particle tracking algorithm reaches the element in which the particle "landed" during the integration step. The Element Adjacency Table greatly improves the speed of particle tracking because it obviates the need to search the full grid for the element that contains the new particle position.

### 2.2 Hashing Function

The difficulty with the previous algorithm for external face determination is that the Face Table takes up a large amount of space. There are \(6 \cdot N_{\text{hex}}\) integers in the table. For a mesh with 300,000 elements, the Face Table occupies 43 Megabytes of memory on a 32-bit workstation. For larger meshes, the table size may lead to memory problems associated with excessive swapping.

Christon and Spelce [1] employed a hashing function to convert the four node numbers of a face to a single hashing key which was then used to sort the Face Table. Assuming that the mesh has a million nodes and is visualized on a workstation with 32-bit integers, the hashing function must map four integers in the range \(1 \times 10^6\) into an integer with range \(4.3 \times 10^9\). In general, it's not possible to define a hashing function which maps the four node numbers into a single integer and which simultaneously guarantees no collisions in the hash entries. This means that each double entry in the sorted Face Table must be checked (by loading the actual node numbers and comparing) to make sure that the entry is an actual internal face rather than the result of a hashing collision. It's not uncommon for ninety percent of the faces in a mesh to be internal, so this requires a small amount of extra processing. And the Face Table size still grows linearly with the number of elements.

### 2.3 Element Grouping

The idea in the grouping algorithm is to group elements into contiguous blocks. External face determination is performed on each block separately. Then, the faces that are shared between blocks are determined.

In parallel implementations of finite element analysis codes, a mesh is partitioned into a series of blocks of adjacent elements (or nodes). Each block of elements
is assigned to a different processor during the simulation. Result values at the boundaries of the blocks must be determined by inter-processor communication. The partitioning algorithms attempt to generate a “good” partition which 1.) balances the amount of work to be done among processors, and 2.) minimizes the amount of inter-processor communication that needs to be carried out. In other words, a partial goal of the partitioner is to minimize the number of external faces for each block. Partitioning schemes are surveyed in [3]. An example partitioning generated by the recursive spectral bisection algorithm is shown in Figure 3.

The partition generated by a parallel version of an analysis code can be used to advantage in the external face determination algorithm. If no such partition is available, the visualization program generates one. Since our mesh generator numbers adjacent elements consecutively, we’ve found that simply grouping the first $N_{\text{block.size}}$ elements as the first block, the second $N_{\text{block.size}}$ elements as the second block, and so forth, yields a reasonably good partition for the purposes of this algorithm. A mesh partitioned using this linear partitioning technique is shown in Figure 4.

The algorithm works as follows. The Face Table is created, but instead of 6-$N_{\text{hex}}$ entries it is allocated to a fraction of that size (say $10\cdot6\cdot N_{\text{block.size}}$). The element faces for the first block of elements are loaded into the table, and the sorting procedure is performed on that block. Then the algorithm steps through the block. If a face has two entries in the table, the appropriate pointers in the Element Adjacency Table are set and both face entries are deleted from the Face Table. If the face has a single entry in the Face Table, it is left untouched. At the end of the step, the Face Table is compressed to eliminate all faces with double entries. The Face Table is used like a stack in this algorithm: new blocks are loaded in after any existing entries. The steps for the next block of elements are performed on the Face Table and the previous steps are repeated for only the faces in the new block. This process continues until there is not enough room in the Face Table to load in the next block of elements. Then the “sort and compress” operation is performed on all entries in the Face Table. This step eliminates faces that are shared between blocks. If there is still not enough room to load the next block of elements, the table is expanded. After all of the blocks have been loaded and compressed, a final “sort and compress” is performed on all of the remaining entries in the Face Table. The faces still in the table after this last step are external faces, and are marked as such in the Element Adjacency Table. Once again, the element adjacency graph has been constructed in the Element Adjacency Table during this process and the Face Table may be discarded.
### Time Required to Generate Element Adjacency Table

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Number of Blocks</th>
<th>Initial Face Table Size (# faces)</th>
<th>Final Face Table Size (# faces)</th>
<th>Number of Full Table Compressions</th>
<th>Total CPU Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>1267</td>
<td>18960</td>
<td>same</td>
<td>39</td>
<td>80.0</td>
</tr>
<tr>
<td>158</td>
<td>634</td>
<td>18960</td>
<td>same</td>
<td>37</td>
<td>79.8</td>
</tr>
<tr>
<td>316</td>
<td>317</td>
<td>18960</td>
<td>same</td>
<td>33</td>
<td>75.2</td>
</tr>
<tr>
<td>632</td>
<td>159</td>
<td>18960</td>
<td>26544</td>
<td>32</td>
<td>73.7</td>
</tr>
<tr>
<td>1264</td>
<td>80</td>
<td>18960</td>
<td>34128</td>
<td>12</td>
<td>57.6</td>
</tr>
<tr>
<td>2528</td>
<td>40</td>
<td>18960</td>
<td>49296</td>
<td>3</td>
<td>55.8</td>
</tr>
<tr>
<td>79</td>
<td>1267</td>
<td>37920</td>
<td>same</td>
<td>9</td>
<td>60.5</td>
</tr>
<tr>
<td>158</td>
<td>634</td>
<td>37920</td>
<td>same</td>
<td>8</td>
<td>59.9</td>
</tr>
<tr>
<td>316</td>
<td>317</td>
<td>37920</td>
<td>same</td>
<td>6</td>
<td>57.0</td>
</tr>
<tr>
<td>632</td>
<td>159</td>
<td>37920</td>
<td>same</td>
<td>4</td>
<td>54.3</td>
</tr>
<tr>
<td>1264</td>
<td>80</td>
<td>37920</td>
<td>same</td>
<td>3</td>
<td>54.0</td>
</tr>
<tr>
<td>2528</td>
<td>40</td>
<td>37920</td>
<td>same</td>
<td>3</td>
<td>56.1</td>
</tr>
<tr>
<td>100080</td>
<td>1</td>
<td>600480</td>
<td>same</td>
<td>0</td>
<td>71.7</td>
</tr>
</tbody>
</table>

Table 1: Time Required to Generate Element Adjacency Table for Submarine Hull Problem (100,080 Elements)

We can divide the performance of the algorithm into two categories: memory usage and processing time. For the dataset shown in Figure 4 (containing 100,080 elements), the blocking algorithm runs successfully with a Face Table that is 1/32 of its original size (see Table 1). This represents a substantial memory savings. The computational performance of the algorithm is more difficult to characterize, since it is dependent on the topology of the individual grid, the partitioning scheme used, the block size, and the initial Face Table size. Sorting the full Face Table in the naive algorithm requires $O(6N_{hex} \ln(N_{hex}))$ operations. In the grouping algorithm, the sorting of the individual blocks is $O(N_{block}6N_{block.size} \ln(N_{block.size}))$, or $O(N_{hex}6N_{block.size})$. An additional term in the grouping algorithm is the cost of any full table sorts. For a mesh with $N_{hex} = 100,000$ and $N_{block.size} = 316$, the grouping algorithm (ignoring the additional term) is about forty percent faster, but the cost of re-sorting some faces during the full table sorts means that this speedup might not be achieved. In general, a smaller block size will tend to increase the efficiency of the sorts according to the above complexity formula, but this savings may be negated as the number of faces which must be sorted more than once increases. A "good" partition will be more efficient than a "poor" partition, since it will reduce the number of faces which must be re-sorted.

Table 1 shows the effect of varying the block size and Face Table size for the model in Figure 4. The performance of the algorithm without element grouping is shown in the bottom row of the table. All times in Table 1 are for an SGI Indigo workstation and were obtained with calls to the system routine getrusage().

The most important conclusion that can be drawn from the table is that variations in the block size and initial Face Table size did not have much effect on the performance of the algorithm for this example, although a small amount of tradeoff between space and time efficiency is visible in the table. Therefore, the block size and Face Table size could reasonably be selected to minimize memory use.

### 3 Smooth Shading and Crease Detection

Once the external faces in the mesh have been determined, it is necessary to compute vertex normals for lighting purposes. Flat facettet shading can help emphasize the discrete nature of the model, but the overall shape of an object is better conveyed with smooth vertex normal-averaged shading. Unfortunately, once the mesh has reached the visualization program, the location of surface tangent discontinuities (creases) in the original geometry is no longer known. To avoid smoothing across creases in the model, a scheme for node normal averaging that incorporates crease detection is needed. We compare two schemes — a new one and an old one — here.

The first method is to take each of the four edges of each external face of the mesh, and put it into a table. The sorting algorithm of the last section is applied to the edges in order to determine the adjacency between external faces along their edges. Finally, an arbitrarily chosen threshold angle is used to decide whether to
smooth across each edge which is shared by two adjacent faces. This scheme is acceptable in cases where it is performed only once at startup, but it is too slow for interactive animation of simulations where the mesh is deforming and the normals must be recomputed at each state.

A faster but rougher method was described in [1]. In this method, all faces sharing a node contribute their face normal to the average node normal. Then, the rendering code compares the face normal of each face to the average node normal of each of the face's vertices and uses a threshold angle to decide whether to use the average node normal or the face normal at that vertex. Since the average node normals can be calculated trivially, this algorithm is much faster than the previous one but still yields acceptable results.

Surface normals are recomputed when the mesh is deformed and also when the surface topology is changed — which happens whenever element visibility is modified.

Explicit detection of crease edges, performed at start-up, serves an important purpose in our software. Crease edges, drawn as wireframes, are used for fast rendering while the user is rotating the model and are used to outline the model in volume displays. Winget had previously reported in [6] a technique for "visible edge detection" which essentially implemented the second method above, marking individual nodes as continuous or discontinuous based on the difference between the average node normal and the surrounding face normals. All edges that extended between two discontinuous nodes were then marked as "visible edges." The first method provides a more accurate way to explicitly detect these crease edges.

Surface normals are recomputed when the mesh is deformed and also when the surface topology is changed — which happens whenever element visibility is modified.

4 Point in Element Determination

This section addresses the following problem: given a result quantity which is stored at the nodes of a grid and given an arbitrary point in the interior of the grid, how does one calculate the result value at the specified point? The solution can be divided into two steps: 1.) find the element that contains the point and 2.) interpolate the result values from the nodes of the element to the point. There are a number of applications for this procedure, including initializing particle traces, displaying vector "hedgehogs" in a vector field, mapping result values from one grid to another grid, and interactive 3D probing of a result field. Efficiency is an issue in all of these applications.

For isoparametric finite elements, a set of shape functions or interpolation functions both defines the geometry of an element based on the element nodes and interpolates quantities from the nodes to the interior of the element. Points in the interior of an element are defined by their natural coordinates $(\xi, \eta, \zeta)$ on the interval $[-1, 1]$ (Figure 5). For an 8-node trilinear element,

$$P(\xi, \eta, \zeta) = \sum_{i=1}^{8} N_i(\xi, \eta, \zeta) P_i$$

where

$$N_i = \frac{(1 + \xi_i)(1 + \eta_i)(1 + \zeta_i)}{8}$$

Here, the natural coordinates at each corner node $i$ of the element are given by $(\xi_i, \eta_i, \zeta_i)$, $P_i$ are the global coordinates of the corner nodes and $N_i$ are the shape functions. (See, for example, [7].)

The equation for $P$ above is actually separated into equations for $x$, $y$, and $z$. Testing whether a point is in the interior of an element requires solving the inverse of these equations. One computes the inverse Jacobian of the shape functions and then uses an iterative technique such as Newton iteration to solve for the natural coordinates of the point. The iteration converges very quickly — usually in two or three steps. If the resulting natural coordinates are not in the range $[-1, 1]$, then the point is outside the element. If the point is inside the element, the result value can be interpolated to the point by

$$R = \sum_{i=1}^{8} N_i(\xi, \eta, \zeta) R_i$$
where \( R_i \) is the result value at corner node \( i \).

In order to avoid applying the expensive inclusion test above to each element in the grid, we first test the candidate point against a bounding box computed from the nodes of an element. Even this simple bounding test is too expensive, however, if it must be applied to (on average) half of the elements in a large grid.

To improve the performance of the algorithm, it's useful to look at techniques used in ray-tracing applications. The categories of most interest are spatial partitioning methods (e.g. octree) and hierarchical bounding volume methods (e.g. bounding boxes.) The hierarchical bounding technique is a logical candidate, since the element partitioning that was used to facilitate external surface determination can be utilized again in this context.

Hierarchical bounding boxes are implemented with a minimal amount of extra data. A bounding box is calculated for all elements in each element block and the candidate point is tested against the bounding box of a block before being tested against the elements in the block. The result is a substantial performance improvement for the point inclusion test. For our example problems, the hierarchical test was five to fifteen times faster than tests which didn't use element grouping. Testing the point against a bounding box for the entire grid before testing against each block is also a good idea.

A traditional method for displaying a vector field on an unstructured grid is to draw oriented line segments at the nodes of the grid. This approach can lead to confusing images because of nonuniform grid density. We prefer instead to resample the unstructured grid data on a regular grid (see [2]). Since resampling requires solving the “point in element” problem numerous times, the increase in performance provided by hierarchical bounding boxes can be essential for maintaining interactivity.

5 Examples

Example applications of the previous techniques are shown in Figures 6 and 7. The sequence in Figure 6 shows three frames from a nonlinear dynamics problem in which a solid penetrator is fired at high velocity into a barrier material. Both mesh deformation and element failure occur in this problem as elements in the projectile and barrier fail during the impact event. The colormap shows effective plastic strain, with red areas being the areas that undergo the most deformation.

The sequence in Figure 7 shows a metal casting problem in which hot copper is injected into a mold. Once again, the surface topology of the mesh varies from state to state as the elements in the mesh fill with fluid. The colormap in this example shows the temperature of the injected material, with cooling occurring over time. The mold material has been made invisible, but crease edges of the model are shown.

6 Conclusions and Future Work

A two-stage approach to external surface determination for unstructured grids has been described. The algorithm lends itself to interactive animation of analysis results from nonlinear dynamics, casting and fluid flow problems. An element grouping technique was developed which substantially reduces the amount of memory needed for external surface determination (thereby allowing larger datasets to be handled) and which speeds up “point in element” testing.

As transient datasets generated on massively parallel computers become larger, it becomes less feasible to transfer the whole dataset to a graphics workstation for viewing. We need to develop distributed visualization techniques which support interactive animation of analysis results for datasets which reside on multiple processors of a massively parallel computer.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48.

The author would like to thank Mark Christon and Roger Crawfis for their helpful comments on this paper. Richard Procassini and Michael Loomis assisted with the images. GRIZ was written by Don Dovey, Tom Spelce, and Doug Speck.

References


Figure 6: Three Snapshots from a Penetrator Simulation

Figure 7: Three Snapshots from a Metal Casting Simulation