SANDIA REPORT

SAND98-1224

Unlimited Release

Printed June 1998

**Dynamic Isosurface Extraction and Level-
of-Detail in Voxel Space****RECEIVED****JUN 15 1998****OSTI**

John M. Linebarger, Peter B. Lamphere, Arthurine R. Breckenridge

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or 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 product, 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, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND98-1224
Unlimited Release
Printed May 1998

Dynamic Isosurface Extraction and Level-of-Detail in Voxel Space

John M. Linebarger
Information Systems Applications Department
Sandia National Laboratories
P.O. Box 5800, MS 1137
Albuquerque, NM 87185-1137

Peter B. Lamphere
Computer Architectures Department
Sandia National Laboratories
P.O. Box 5800, MS 0318
Albuquerque, NM 87185-0318

Arthurine R. Breckenridge
Computer Architectures Department
Sandia National Laboratories
P.O. Box 5800, MS 0318
Albuquerque, NM 87185-0318

Abstract

A new visualization technique is reported, which dramatically improves interactivity for scientific visualizations by working directly with voxel data and by employing efficient algorithms and data structures. This discussion covers the research software, the file structures, examples of data creation, data search, and triangle rendering codes that allow geometric surfaces to be extracted from volumetric data. Uniquely, these methods enable greater interactivity by allowing an analyst to dynamically specify both the desired isosurface threshold and required level-of-detail to be used while rendering the image. The key idea behind this visualization paradigm is that various levels-of-detail are represented as differently sized hexahedral "virtual voxels," which are stored in a three-dimensional *kd*-tree; thus the level-of-detail representation is done in voxel space instead of the traditional approach which relies on surface or geometry space decimations. This algorithm has been implemented as an integral component in the EIGEN/VR project at Sandia National Laboratories, which provides a rich environment for scientists to interactively explore and visualize the results of very large-scale simulations performed on massively parallel supercomputers.

Index terms

Level-of-detail, isosurface extraction, voxel space, structured grids, *kd*-trees, multi-resolution.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

1. Introduction

A terabyte of information has become easy to generate; indeed, data sets of this size are regularly being produced on the massively parallel Intel teraflop supercomputer at Sandia National Laboratories. However, similarly sized data sets have been produced when supercomputers were coupled across the Atlantic Ocean at a 1997 supercomputing conference. More recently, clusters of personal computers have begun to reach this scale of computation. How can a scientist hope to comprehend this volume of data? How can groups of scientists on separate continents jointly work with data sets of this size? Imagine for a moment one scientist examining this information using punch cards or printouts. A globally distributed, collaborative environment with interactive virtual worlds composed of computer generated visualizations is required. However, this is not easy; the revolution in computing enabled by teraflop machines surpasses the capabilities of traditional visualization and data analysis paradigms. To visualize terabytes of information the visualization process will have to move closer to the computation that generated the data.

Currently, most visualization is done by post-processing the data after a simulation has written its results to a disk file, generally in a format convenient for the simulation code, but not for the visualization programs. This approach has numerous limitations and has been shown not to scale in real time. A first step toward a system solution is to avoid unnecessary transforms from one data format into another. Instead, the process should use the data directly in a manner suitable for visualization algorithms. In particular, three dimensional volume data from a simulation should remain accessible as voxel data for the visualizations. This paper documents the first successes in using this new visualization paradigm. This research extended Sandia's existing virtual environment tools by building a system that scales in immersion and in interactivity to meet the needs of teraflop class problems.

The result is a set of computer codes that provide dynamic isosurface extraction and multi-resolution (level-of-detail, or LOD) capability in voxel space. Isosurface extraction is an important technique for visualizing surfaces found in volumetric data (*i.e.*, data consisting of 3-D scalar fields usually produced as the result of a computer simulation). The canonical algorithm for extracting isosurfaces is the "marching cubes" algorithm detailed by Lorensen and Cline [1]. Numerous methods for creating levels-of-detail have been proposed, including triangle decimation [2], the use of wavelet functions for surface approximation [3], and more recently, progressive meshes [4] and progressive simplicial complexes [5]. However, in all of these, the set of triangles generated by the marching cubes algorithm is dependent on the chosen isosurface value. For example an isosurface may be requested showing the extent of the volume with a material fraction of at least 0.55, that is, a surface enclosing the regions more than 55% filled with material. In other cases this value may range from near zero up to large powers of ten, for example when analyzing simulations of shock, temperature, or pressure waves. It is always highly desirable to allow the analyst to explore the problem space by dynamically changing the isosurface threshold. However, traditionally, changing the isosurface value requires a complete isosurface recalculation. That time-consuming computation violates a design principle of the EIGEN/VR project, which always seeks to allow arbitrary analysis of data immediately and interactively, *i.e.*, without waiting for a recalculation of the new isosurface.

The NOISE (Near Optimal IsoSurface Extraction) algorithm proposed by Y. Livnat, *et al.* [6] provides the required solution. It allows rapid isosurface extraction from structured and unstructured data grids, such that the user can interactively change the isosurface threshold in a visualization environment and quickly view the resulting surface. A multi-dimensional search tree (called a *kd-tree*; see [7]) is used to store the minimum and maximum values of each voxel in the volumetric data set, and the resulting "span space" is searched for a given isosurface threshold. The output is a set of only those voxels that have potential intersections with the new isosurface, which are then triangulated using the marching cubes algorithm and finally rendered.

However, two problems arose when we first implemented the NOISE algorithm at Sandia National Laboratories. The first was the well-known "ambiguous cases" that occur when using the marching cubes algorithm. Resolution of such ambiguity either adds additional complexity or makes use of the data values in neighboring voxels (which can also have the side effect of improving the shading of the resulting triangles because of the gradient heuristic used; see [8] and [9]). Programs which perform marching cubes triangulation with ambiguity resolution logic are usually run in batch mode. Adding the additional logic to disambiguate the list of voxels returned by the NOISE algorithm during real-time visualization would either have made the frame rate unacceptably slow when an isosurface threshold was dynamically changed, or would have required the addition of additional disambiguation data to each voxel returned from the search (thus ballooning the file size). Fortunately, a straightforward approach which allows unambiguous triangulation of an arbitrary voxel without regard for its neighbors (at the price of occasional surface anomalies) has been published by C. Montani, *et al.* [10], who graciously provided us with a copy of the triangulation table.

The second problem encountered was the sheer number of triangles generated from the ever-increasing growth of our simulation data volumes, which saturated the hardware rendering capabilities even of the largest graphics machines produced in 1997 (SGI Onyx 2), and which compromised the interactive response promised by the NOISE algorithm. Clearly, a dynamic way of specifying *both* the isosurface threshold *and* the level-of-detail is needed to achieve an acceptable rendering frame rate when navigating through large simulation data volumes with many time steps. A particular benefit of such a multi-resolution approach is the ability to display a coarse approximation of the data at interactive speeds while the user navigates rapidly, but which also allows a finer approximation when he/she stops to study the details.

The research algorithms for level-of-detail presentations (some of which were mentioned above) make adjustments to the geometry representations and, in our opinion, do not provide real solutions suitable for large-scale data sets. They operate in geometry space (the surfaces), not in voxel space which is most directly associated with the physical simulation. In other words, each algorithm assumes a stable underlying surface geometry, which is then represented in varying levels of detail. Of course, the ability to dynamically specify the isosurface threshold at visualization time implies that in such an approach there *is* no stable underlying geometry, since the geometry generated depends entirely on the isosurface threshold value chosen. The analyst cannot work with the data in real time if significant delays are introduced while new geometry is created and another algorithm applied to reduce the data representation. Further, surface based level-of-detail representations no longer work with the original data, which is unacceptable for many scientists who must be able to get back to the original data values.

The approach we have taken with dynamic isosurfacing and level-of-detail representations are powerful extensions of the NOISE algorithm. Different levels-of-detail are

represented as differently sized virtual voxels and the resolution value is stored as a third dimension of the *kd*-tree. Adjacent voxel cells in the volumetric data grid are joined to create larger "virtual voxels." For example, with structured rectilinear grids, the finest level-of-detail is represented by a 1x1x1 voxel; coarser levels can be represented as 2x2x2 voxels, 3x3x3 voxels, etc. Thus the level-of-detail representation is done in voxel space, not geometry space. Spatial coherence of the resulting surface is largely preserved between virtual voxel resolutions because of the interpolative nature of the marching cubes triangulation algorithm.

2. Components of Dynamic Isosurface and Level of Detail (DYNAISOLOD)

The components of DYNAISOLOD can conveniently be divided into two halves: a parallel portion that uses the Message Passing Interface [MPI] standard while creating the LODTree file (a 3D *kd*-tree with associated data), and a visualization-time part that includes search, geometry-creation, and rendering algorithms. Time and space optimizations have been added to each of these two components.

DYNAISOLOD itself is a component of Sandia's EIGEN/VR project for analyzing structured and unstructured grid simulation data sets produced by massively parallel supercomputers. Proof of concept has been applied to the Sandia simulation code, CTH, which produces structured rectilinear grid data sets. An additional goal of this project is to compute and visualize data on the supercomputers and then deliver the results to the desktop monitor rather than a far more expensive visualization machine. As a result, care has been taken at each stage of DYNAISOLOD development to ensure that the resulting data file is readable from a Java program, with the eventual goal of providing a portable Java client rendering for the geometry at the desired isosurface threshold and level-of-detail. This rendering could be served from either a data storage and geometry-creation server or sent to the analyst's desktop for real-time analysis.

3. Parallel Algorithm for Parallel LODTree File Creation

Before describing the file creation algorithm, a brief review of the structure of a *kd*-tree is in order (see [7] for more details). In order to create such an easily searchable, multidimensional (*i.e.*, multikey) data structure (for example, the 3D data structure of our LODTree), a divide-and-conquer approach is used. First, the data set is partitioned into two halves, one of which contains nodes that have a higher key value than the root node, and the other containing nodes that have lower values. These two halves are again partitioned, but this time according to the second key. The four sub-branches are partitioned a third time according to the third key. Then the process repeats itself again for each of the resulting subtrees, starting again with the first key. This structure allows a very fast search, using any combination of criteria involving the three keys. In our case, this can be envisioned as a three-dimensional binary tree where each branch represents a new dimension. For DYNAISOLOD, the three keys are the minimum value, the maximum value, and the level-of-detail of the virtual voxel. This whole partitioning process operates on a set of indexes into the original data set.

Like the NOISE algorithm, DYNAISOLOD uses a Quicksort-like partitioning scheme to find a median node and partition the subtrees. Two significant extensions, however, were made to the NOISE algorithm. The first extension is that the file creation algorithm was parallelized (using MPI) such that at each Quicksort-like recursive call, the top half of the node array is partitioned to another processor until the number of processors in the processor array is reached; at that point all additional recursive partitioning is done locally. A heavily recursive subroutine was written to deterministically predict, based on the number of cells in the simulation data volume and the number of processors in the processor array, the size of the message sent to each processor upon such a recursive split. In addition, it also predicts the size of the resulting sorted message from each processor. Knowing these sizes in advance allows the other processors in the parallel computation to be prepared for work when they receive a message after a recursive array split. It also allows the root processor to determine the correct order in which to receive the partitioned message data from the child processors. Obviously, this strategy implies that the algorithm is most efficient when the number of processors in the processor array is a power of two, but it works correctly for an arbitrary number of processors.

The second extension was the extensive use of a bit-mapped representation of the data in each virtual voxel in order to significantly reduce the size of the search tree. To allow the file to be read by a Java program, compiler alignment of the in-memory data structure written out to the file had to be carefully considered, the maximum array size had to be limited to the contents of a 32-bit signed integer, and all numeric fields were defined as signed. (It may be possible to remove this limitation as Java matures). An abbreviated version of the header file for the structure follows.

```
typedef struct lod_file_header_struct
{
    char signature[ SIGNATURE_SIZE ];    // Version signature
    char source[ SOURCE_SIZE ];           // Source of the data
    int width, height, depth;             // Dimensions of the data set
    int num_data_points;                  // Number of data points
    float min, max;                       // Min and Max for the tree
    int size;                             // Number of LOD nodes
    int resolutions;                      // The number of resolutions
} lod_file_header_t;

// A structure defining the whole in-memory LOD tree.
typedef struct lod_tree_struct
{
    lod_file_header_t header;             // The file header information
    int* array_base_indices;              // Array of indices into the data
                                           // for the base cell vertex
    char* min_max_vertex_coords;
                                           // The first 4 bits of this variable contain the
                                           // canonical vertex coordinate of the cell vertex
                                           // that has the minimum data value, and the last
                                           // 4 bits contain the canonical vertex coordinate
                                           // of the cell vertex that holds the max value.
                                           //
                                           // X111X111    <-- bitmap (X is unused bit)
                                           // mxyzaxyz    mxyz = min cell vertex coordinate
                                           //              axyz = max cell vertex coordinate
    char* cell_dimensions;
                                           // The first three bits of this are flags,
                                           // indicating that the cell is not regular in a
```

```

// given dimension. The last 4 bytes contain the
// the regular size of the cell.
//
// X1111111 <-- bitmap (X is unused bit)
// xyzssss x = is width irregular?
//          y = is height irregular?
//          z = is depth irregular?
//          s = regular cell size (2^4 max)
// Irregular cells occur only at the edges of
// the structured grid dataset. Therefore,
// if x, y, or z is irregular, the extreme
// {<x, y, z>} coordinates in the cell are
// equal to the maximum possible width,
// height or depth.
float* data_values; // Array of scalar data values; the dimension
// is width * height * depth
float* data_coords[3]; // Arrays of coordinates associated with the
// data value array, for x, y, z.
} lod_tree_t;

```

Note that the possibility of irregularly shaped virtual voxels at the edges of the grid is taken into account by this representation. Also note that this compressed bitmapped representation works only for structured (rectilinear) grids.

A more detailed account of the flow of the LODTree file creation algorithm follows:

- A. The first processor initializes the MPI communications group, reads in the file, calculates the minimum and maximum data values for all the voxel cells in the file, and then begins its first Quicksort-like partitioning pass through the data. When the median cell is found, this processor schedules out the top half of the array to the next processor in the processor group and then proceeds to perform the same partitioning algorithm recursively on the bottom part. Each processor in turn, when it receives a message that contains its data, will perform the same partitioning algorithm on its message and farm out the top half to the next processor in the processor group, whose number is deterministically predicted by the highly recursive predictive routine mentioned earlier. It then continues to perform the partitioning algorithm on the bottom half of its array. This array subdivision and subcontracting to other processors continues until the limit of the processor array is reached, at which time the recursive partitioning algorithm is performed locally by each process.
- B. For efficiency, messages contain only the minimum amount of virtual voxel information necessary for the partitioning routine to function: the minimum and maximum data values, the level-of-detail, and the index into the full data array contained in processor 0 where all additional information is kept. The index is used by processor 0 to reconstitute the full data later when all of the correctly ordered virtual voxel stub messages have come back from all the processors.
- C. Experimental analysis has shown that the most time-consuming and expensive part of the creation of the *kd*-tree is the Quicksort-like recursive partitioning (usually at least 90% of the serial run time). Thus the initial parallelization strategy chosen was simply to parallelize the recursive partitioning activities, not the initial calculation of the minimum and maximum data values for each voxel.

- D. Once all the processors have finished their partitioning chores, processor 0 interrogates each process in turn, in the order that was also dynamically predicted by the recursive prediction routine, and retrieves the message stub back from each processor in correct *kd-tree* order. That message stub order is then used to write out the full *kd-tree* data; the result is a LODTree file, a 3D *kd-tree*. (Optionally, the integrity of the *kd-tree* can then be checked to ensure that the LOD-tree is in fact a true *kd-tree* [i.e., is structured correctly as a three-dimensional *kd-tree*]).

A decision was made to test the proof of concept of voxel-based visualization on the SGI machines before moving the work to the Intel teraflop machine. However, a current weakness in the DYNASOLOD algorithm is that the first parallel process (processor 0 in MPI parlance) must be able to fit all of the simulation data in memory at one time, in order to create the initial array partitioned around the median voxel. Thus the size of the file being processed is limited by the amount of memory available to the initial processor. For shared memory machines (like Sandia's current 32G Silicon Graphics Onyx2) this poses much less of a problem than for distributed memory machines (like the 128M per node Intel teraflop supercomputer), which have a limited amount of memory available at each processor node.

4. File Creation Options and Optimizations

Numerous run-time options and optimizations for the parallel file creation program are available. These options include:

- The maximum number of levels-of-detail desired
- The virtual voxel size increment, in each dimension, for each level-of-detail (defaults to 1)
- A post-processing check of the structural integrity of the LODTree file (*i.e.*, to verify that it is indeed a three-dimensional *kd-tree*)
- The "virtual voxel overlap factor" that specifies by how much an edge voxel can expand in order to prevent rows of small voxels at the edges of the grid.
- An optimization option for shared memory implementations of MPI, which synchronizes memory allocations and message passing associated with recursive splits between processors in order to reduce the overall memory footprint; this option essentially trades memory space for run time.

In addition, the run-time environment provides a parameter that specifies the number of processors in the processor group.

5. Run-time Search and Rendering Algorithm

At run-time we see how the effort spent creating the searchable tree results in much greater efficiency and flexibility for the user. The rendering component of the NOISE algorithm starts with a search in span space, returning the set of cells which contain the isosurface value

(i.e., whose minimum value is less than the isovalue, and whose maximum value is greater.) Our multi-resolution DYNAISOLOD algorithm thus becomes a search with an extra parameter, the desired resolution (or level-of-detail) of the resulting isosurface. Like the NOISE algorithm, our LODTree file is searchable in $O(\log N)$ time [6].

After the LODTree file has been loaded, the user provides an isosurface threshold value (an isovalue) and a desired level-of-detail. The level-of-detail may be furnished indirectly, perhaps as some function of current navigation speed. The data set is then searched for cells that contains this isovalue and are of the appropriate size to create the correct level-of-detail. Recall that the cells in the data set have been sorted according to the three keys: minimum data value present in the cell, maximum data value present in the cell, and the size of the cell (resolution, or level-of-detail). The cells that we want to find are those with a minimum data value below the isovalue and a maximum data value above that threshold. These cells contain data points that intersect that isovalue, and therefore will contain the approximation of the isosurface.

Each node of the tree is interrogated in terms of the current search key, and each descent into a lower level of the tree causes the current key to cycle to the next key. For example, the first level (root node) of the tree is searched by the minimum vertex value of the voxel (cell). If the minimum value is higher than the desired isovalue, the left child of the node is searched, if lower, both children are searched.

For the next layer, the next key (cell maximum) is queried. If the cell minimum is less than the isovalue, we must search all cells (nodes) along the right child. Otherwise, all of the left children are searched.

For the next layer of the tree, the LOD key is searched, and the search cycle continues for all subsequent levels of the tree: the minimum key, followed by maximum key, and so on until tree has been exhausted. If at any point during the search, the cell minimum is less than the isovalue, the cell maximum is greater than the isovalue, and the LOD of the cell is equal to the desired LOD, then that cell is added to the list of cells that contain the isosurface. Pseudocode of the search algorithm follows:

```
void search_minimum {
    if (minimum value in cell > isovalue) // isosurface does not
        search_maximum (left branch);    // intersect cell
    else {
        if (maximum value in cell > isovalue && LOD of cell ==
            desired LOD)
            add cell;
        search_maximum (right branch);
    }
}

void search_maximum {
    if (maximum value in cell < isosurface) // isosurface does not
        search_lod (right branch);        // intersect cell
    else {
        if (minimum value in cell < isosurface && LOD of cell ==
            desired LOD)
            add cell;
        search_lod (left branch);
    }
}

void search_lod {
    if (LOD of cell > desired LOD)
        search_minimum (left branch);
}
```

```

else
  if (LOD of cell < desired LOD)
    search_minimum (right branch);
  else {
    if (minimum value in cell < isovalue &&
        maximum value in cell > isovalue)
      add cell;
    search_minimum (right branch);
    search_minimum (left branch);
  }
}

```

Various optimizations can improve the performance of even this efficient algorithm. For example, once we know that the minimum data value for the cell is less than the isovalue, we know that this will be true for all cells on the left branch of the tree (because all cells on the left branch have a smaller key than the parent node). Therefore, we no longer have to test the minimum key past that point. The same thing can be done for the maximum key. For the resolution (level-of-detail) key, it is possible to optimize the search only if we are aware that the level-of-detail desired is either the greatest or the least possible. Then we can simply stop testing the LOD keys if the search has passed on to the left (i.e., smaller) subtree of an already minimum LOD node. Since the search is most time-consuming at the lowest level-of-detail, this is a worthwhile optimization.

Once the cells containing the isosurface are found DYNAISOLOD generates triangles on a cell by cell basis. First, it finds which corners of the cell are underneath or above the isovalue to determine which edges of the cell the isosurface intersects. From this we can decide how to triangulate the isosurface in the cell. Using the modified triangulation table mentioned above, we generate a set of triangles whose vertices are linearly interpolated along the cell edges.

This process results in redundant calculation of vertices, as many triangles share vertices. So as we calculate each vertex we store it in a hash table, keyed to the edge that it lies on (since the isosurface approximation only intersects each cell edge at most once). When we encounter the same vertex again, we can simply copy the vertex coordinates from the hash table. After the coordinates of the triangles have been calculated, normals are computed by a cross-product, and then stored (via the same hash table) for later averaging with normals of other triangles that have the same vertex.

6. Rendering Options and Optimizations

In keeping with DYNAISOLOD as a component of a scientific data analysis toolkit, several rendering options are provided and can be user-specified. The first option toggles the direction of the normals generated for the isosurface triangles. Other data formats may need normals pointed in the opposite direction, so the facility to choose the normal direction was provided. Second is the ability to render wireframe, flat shading, or smooth (Gouraud) shading versions of the generated triangles. The third involves the use of display lists and hash tables both to increase the speed of the rendering and to improve the shading of the resulting surface. The hash table is used to blend normals together for triangles that share vertices. The goal is to provide user-specified tradeoffs between speed and accuracy: for ultimate speed, the user might choose the highest level-of-detail and wire frame rendering in order to navigate quickly through

the data set; for greatest accuracy, the user might turn the hash table on and specify smooth shading in order to render the surface in the most precise way possible.

7. Results

The observed results are divided into the same two categories as DYNAISOLOD itself: parallel file creation and run time search, geometry creation, and rendering. For parallel file creation, the results are very dependent on CPU(s) used, networking available, and disk speeds. For example, a 27 million cell data set can be transformed in parallel into an LOD-tree in one minute using a four-processor SGI Onyx2. The overhead added to the base simulation volume data is approximately 1.77 times the size of the original simulation volume data set. Heavy use of bit-mapped cell data representation was required in order to achieve this result. When comparing the overhead file size, you also need to compare functionality obtained. Traditional methods would have a geometry file for each threshold and level-of-detail which obviously does not scale for large data sets. The overhead for DYNAISOLOD is constant.

Sample timings of the run-time algorithm are given in the following tables (note that a triangle strip is a more compact representation of a set of triangles that form a surface):

Small Data Set of 27 Million Cells

LOD	Number of Triangles	Search time in Seconds	Geometry Creation in Sec.	Rendering Time in Seconds
1	560,688	0.35	10.29	4.95
2	137,212	0.10	2.05	2.42
3	60,164	0.03	0.96	0.98
4	33,760	0.02	0.42	0.54

LOD	Number of Triangle Strips	Search time in Seconds	Geometry Creation in Sec.	Rendering Time in Seconds
1	67,264	0.32	16.29	2.66
2	13,538	0.09	3.28	1.89
3	7,754	0.03	1.48	0.83
4	3,522	0.02	0.70	0.44

Medium Data Set of 54 Million Cells

LOD	Number of Triangles	Search time in Seconds	Geometry Creation in Sec.	Rendering Time in Seconds
1	465,856	0.25	6.16	3.01
2	110,528	0.08	1.32	1.65
3	48,158	0.03	0.56	0.68
4	26,704	0.02	0.30	0.38

LOD	Number of Triangle Strips	Search time in Seconds	Geometry Creation in Sec.	Rendering Time in Seconds
1	51,660	0.26	10.45	1.48
2	12,406	0.08	2.16	0.58
3	6,044	0.03	0.90	0.25
4	3,371	0.02	0.48	0.13

Large Data Set of 100 Million Cells

LOD	Number of Triangles	Search time in Seconds	Geometry Creation in Sec.	Rendering Time in Seconds
1	276,180	0.23	3.68	1.44
2	61,796	0.09	0.69	0.27
3	18,748	0.01	0.23	0.07
4	8,460	0.01	0.08	0.04

LOD	Number of Triangle Strips	Search time in Seconds	Geometry Creation in Sec.	Rendering Time in Seconds
1	60,584	0.22	5.81	1.05
2	14,268	0.08	1.03	0.16
3	4,686	0.01	0.32	0.05
4	2,243	0.01	0.14	0.02

We can conclude from the quite acceptable run-time performance at lower levels of detail despite increasingly larger problem sizes that dynamic isosurfacing and level-of-detail can scale to be interactive and immediate. By using a lower LOD for navigation, the rendering times are reduced by an order of magnitude, while still maintaining the availability of the original data for exact analysis. Note that DYNAISOLOD is targeted for the working scientist who wants to get an overview of structured grid simulation volume data in as flexible and rich an analysis environment as possible. It is not meant to replace traditional marching cube methods, but instead should be used in conjunction with those methods. Indeed, Sandia's EIGEN/VR combines both dynamically-specified isovalues and levels-of-detail for data sets that need such rich analysis, with statically generated geometry and perhaps statically generated levels-of-detail for data sets that don't need such analysis capability. For example, in simulations where the data volume consists of materials, a static isosurface threshold of 0.55 has proven to be very satisfactory, such that dynamic isosurfacing capability is simply not needed; the geometry can be statically generated in that case, and a fixed number of levels-of-detail provided via triangle decimation. Indeed, static geometry creation yields higher quality triangulation because neighboring cell information can be taken into account to resolve ambiguity and to generate better normals (via blending at shared vertices) for shading purposes. Static geometry also has a very low impact on disk storage and main memory requirements. However, for simulation data objects with a much greater range of values, such as pressure fields and shock waves, the rich analysis capability

provided by DYNAISOLOD has proven to be very helpful. The slider control for the dynamic isosurface threshold, provided by the visualization tool user interface, can range through any data value present in the simulation data. Likewise, the dynamic level-of-detail slider control can range independently through all the levels-of-detail present. Another application is to use the dynamic features of DYNAISOLOD early in the analysis process to discover appropriate isosurface threshold values to use for static geometry creation in subsequent stages of the analysis process. A further application exploits the fact that once you load the DYNAISOLOD kd-trees you actually get all thresholds without any additional memory requirements. So the renderer could theoretically display all isosurface thresholds, which provides a powerful quantitative analysis tool.

8. Future Research Areas

Several future research areas have been identified. One goal includes rearchitecting the algorithm to make better use of the distributed memory limitations (*i.e.*, the amount of memory on each node) of massively parallel supercomputers such as the Intel teraflop machine, or of a network of personal computers. Another goal is to link the DYNAISOLOD rendering engine to a number of different visualization application uses, such as automatically generating triangles at a lower level-of-detail when the user is navigating through the data set in the visualization environment, sensing when the user stops navigating, and doing progressive refinement from there until the highest resolution in the file is reached.

Because this proof of concept was so successful, a future research area is the extension of this approach to unstructured data volumes. In order to keep the resulting file size as small as possible, parallel compression and run-time decompression of voxel vertex data may need to be performed, such that the file representation of structured and unstructured grid data may indeed be different. Another goal is intelligent level-of-detail, in which the voxel size at any given level-of-detail could vary depending on the rate of change of the data values in the cell. The goal is file size reduction, accomplished by allowing arbitrarily large virtual voxels at any given level-of-detail in areas of the grid where the data values were not changing very much, and smaller virtual voxels at the same level-of-detail in areas of the grid where the data was changing rapidly.

Such an approach forms a tradeoff between detail and space; the canonical voxel size for that LOD is maintained when detail is necessary (*i.e.*, in voxels where the data values are changing rapidly) and can be expanded in voxels where the data was relatively static. A final future goal is to support tetrahedral data as well as hexahedral data.

9. Acknowledgements

This work was supported by Sandia National Laboratories, a multi-program laboratory operated by Sandia Corporation (a Lockheed Martin company) for the United States Department of Energy under contract DE-AC04-94AL85000. The assistance of Tom Anderson and Patricia Crossno in understanding the problem space and brainstorming possible solutions is also gratefully appreciated.

10. References

- [1] W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163-169, July 1987.
- [2] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen, "Decimation of Triangle Meshes," *Computer Graphics*, vol. 26, no. 2, pp. 65-70, July 1992.
- [3] M.H. Gross, R. Gatti, and O. Staadt, "Fast Multi-resolution Surface Meshing", *IEEE Visualization '95 Proceedings*, pp. 135-142, Nov. 1995.
- [4] H. Hoppe, "Progressive Meshes," *ACM SIGGRAPH '96 Proceedings*, pp. 99-108, Aug. 1996.
- [5] J. Popovic and H. Hoppe, "Progressive Simplicial Complexes," *ACM SIGGRAPH '97 Proceedings*, pp. 217-224, Aug. 1997.
- [6] Y. Livnat, H.-W. Shen, and C.R. Johnson, "A Near Optimal Isosurface Extraction Algorithm Using the Span Space," *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73-84, March 1996.
- [7] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 8, no. 9, pp. 509-517, September 1975.
- [8] G.M. Nielson and B. Hamann, "The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes," *IEEE Visualization '91 Proceedings*, pp. 83-91, October 1991.
- [9] A. Van Gelder and J. Wilhelms, "Topological Considerations in Isosurface Generation," *ACM Transactions on Graphics*, vol. 13, no. 4, pp. 337-375, October 1994.
- [10] C. Montani, R. Scateni, and R. Scopigno, "A modified look-up table for implicit disambiguation of Marching Cubes," *Visual Computer*, vol. 10, no. 6, pp. 353-355, 1994.

Distribution:

6534	MS 1137	Ken E. Washington
6532	MS 1137	John M. Linebarger
9215	MS 0318	George Davidson
9215	MS 0318	Arthurine R. Breckenridge
9215	MS 0318	Peter B. Lamphere
1	MS 9018	Central Technical Files, 8940-2
2	MS 0899	Technical Library, 4916
2	MS 0619	Review & Approval Desk, 12690
		For DOE/OSTI

M98005688

Report Number (14) SAND-98-1224

Publ. Date (11)

1998 06

Sponsor Code (18)

DOE/DP, XF

UC Category (19)

UC-700, DOE/ER

19980702 050

DTIC QUALITY INSPECTED 1