Interactive Exploration of Large 3-D Unstructured-Grid Data

Kwan-Liu Ma†
Institute for Computer Applications in Science and Engineering

Scott Leutenegger†
Mathematics and Computer Science Department
University of Denver

Dimitri Mavriplis†
Institute for Computer Applications in Science and Engineering

Abstract

Visualizing unstructured-grid data from aerodynamics calculations is challenging because of the associated meshes are typically large in size and irregular in both shape and resolution. This research investigates appropriate data structures and rendering methods to allow interactive exploration of the data.

In conjunction with fast splatting rendering, a multiresolution data representation based on agglomeration is used to make possible interactive visualization on a workstation. That is, data are rendered at a particular resolution according to visualization paramenters as well as the speed and memory capacity of the workstation. Interactive visualization allows the user to quickly determine regions of interest and important visualization parameters such as viewing direction and transfer functions.

We then apply a more accurate, expensive rendering method to the orignal data on the regions of interest. The original data are stored on disk. We show with both analysis and experimental results that *R-tree* is a better data structure for fast retrieval of such disk-resident data.

†This research was supported by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

1 Introduction

In aerodynamics calculations, unstructured grids are used to model objects with complex geometry. Because the grids are typically large in size and irregular in shape and resolution, often special data processing and rendering algorithms are needed to make possible visualization of the simulation results. This research studies the needed software support for conducting the desirable iterative, near-interactive visualization process to analyze very large unstructured-grid data on an average workstation.

The proposed visualization process includes mainly two steps as shown in Figure 1. The first step attempts to derive desirable viewing and rendering parameters and to locate regions of interest, a sub-volume. This may be performed on a workstation using a fast but less accurate rendering algorithm on a coarse representation, thus a much smaller version, of the original data stored in the main memory of the workstation. Consequently, we need a multi-resolution representation of the original data (*mrrd*). The *mrrd* allows interactive exploration of the data. Once a hot spot is identified at a particular resolution, the user may switch to viewing at a higher-resolution for further exploration. This exploration process continues until the region of interest and viewing as well as rendering parameters are completely determined. We use a fast, approximated splatting algorithm for rendering data at resolutions according to visualization requirements, as well as the speed and memory capacity of the workstation.

The second step takes the parameters derived, extracts the selected sub-volume out of the original data, and invokes a more accurate rendering program to produce high quality visualization results. This step may be performed on either a workstation or a high-performance computer. The sub-volume represents a spatial region of interest usually much smaller than the overall grid domain and thus may be rendered more efficiently on a workstation. Note that because of its size, the original data set must be stored on disk. It is then essential to have adequate database support such that a sub-volume can be quickly retrieved from disk. Therefore, we represent the data as an *R-tree* [13], an efficient hierarchical data structure that has been widely adopted by many database applications.

Data exploration is inherently iterative. The visualization process and system architecture developed in this research allow computational scientists to study their data at the highest possible resolution in a more efficient manner, rather than reducing the data or operating at a very inefficient batch-mode. In this paper, we describe the construction of the *mrrd*, demonstrate the effectiveness of the fast splatting rendering method, and show that *R-trees* work better than octrees which have been widely used by the visualization community. Test results were obtained by using a four-million tetrahedra-cell data set on workstations.

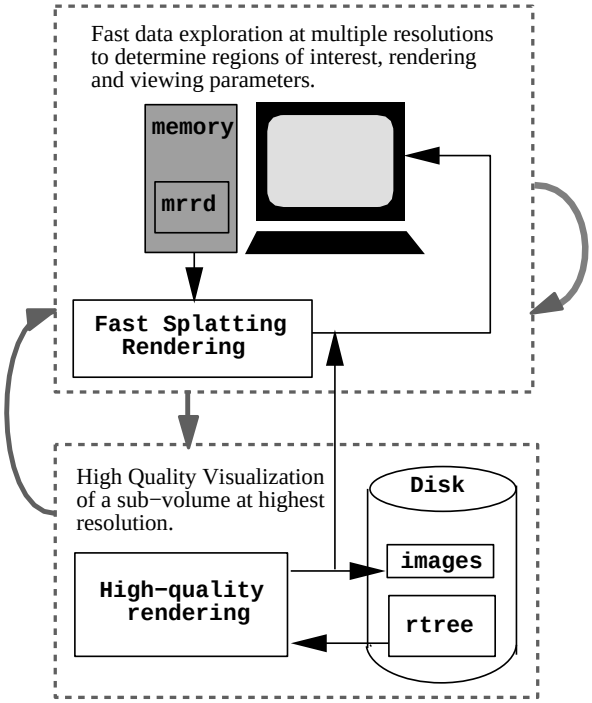

Figure 1: The proposed visualization process and system architecture.

2 Multi-resolution Rendering

Multiresolution data visualization has been an active area of research but most previous work has been concentrated with data on rectilinear grids [19, 2]. In particular, wavelet-based methods have drawn a lot of attention recently [17, 10]. Some multiresolution representations have been designed for triangular surface meshes [14, 3]. For general unstructured grids, only some preliminarily theoretical work exists [1]. In the following sections, we propose a simplified data representation and rendering method to facilitate interactive data exploration.

2.1 Data Representations

The generation of a multiresolution sequence representation of a given arbitrary data-set is a non-trivial one. For data on rectilinear grids, coarser resolution levels may be constructed simply by removing every n-th point in each coordinate direction. For general unstructured grids, the construction of a sequence of nested coarser grid levels is usually not possible. One approach is to interpolate the data onto a regular cartesian grid which can then be coarsened in the standard fashion. Alternatively, a nested sequence of unstructured grids may be constructed by repeatedly subdividing the cells of an initial coarse unstructured grid, and the data may then be interpolated onto the finest level of this sequence. This approach

has the drawback of introducing interpolation errors into the original data, although this may not be crucial for our application of fast-rendering. On the other hand, when the original data-set contains large variations in spatial resolution, as is most often the case in our applications, the approximating regular or nested grid may be much larger than the original grid itself.

An alternative is to attempt to coarsen the original unstructured mesh through techniques such as decimation [16, 15]. Decimation involves removing a subset of the original grid points and retriangulating the remaining points. One of the difficulties with decimation is the retriangulation phase. For complex geometries, the requirement of reconstructing a valid grid connectivity which conforms to the domain boundaries becomes increasingly difficult for coarser levels. Because we rely on a splatting technique for fast rendering of our coarse level data-sets, a valid boundary conforming grid on the coarse levels is not required. Rather, a set of points defined by their coordinates and a length scale to determine the size of the splat for each point are all that is required. We therefore use an agglomeration technique to construct coarse levels [9].

Given an initial graph (set of points and edges), the agglomeration procedure produces a coarser graph based on a subset of the original set of points. If the original graph approximates a nearest neighbor graph, the agglomerated graphs are also approximations to nearest neighbor graphs of the coarser point sets. Agglomeration consists of picking a fine grid vertices known as a seed points, and deleting their neighboring vertices as depicted in Figure 2. The seed points form the coarse grid points for the next level. In practice, we use a heuristic algorithm to produce coarse grid point sets that are maximal independent sets of the original fine grid points. The deleted points can be thought of as fused or agglomerated into their respective seed point. The inferred graph for the coarse level is then obtained by deleting all edges within an agglomerated group of points, and replacing all edges between neighboring agglomerated groups with a single edge, using a hash table. The size of the coarse level splats can then be determined by considering the average length of all edges incident to each coarse level point in the agglomerated graph.

2.2 Rendering

Splatting was first introduced by Westover [18] and has been used as a fast approximation technique for rendering data on uniformly-spaced rectilinear grids. An image is formed by determining the screen space contribution of each grid point–a footprint–and compositing the footprints on top of each other in the visibility order. For parallel projection, a single footprint table can be pre-calculated and shared by all the voxels.

Applying splatting to unstructured-grid data allows us to ignore the type of computational cells we are dealing with. However, because of the unstructured nature of the grid, a separate footprint must be constructed for each grid point. Using parallel projection, fur-

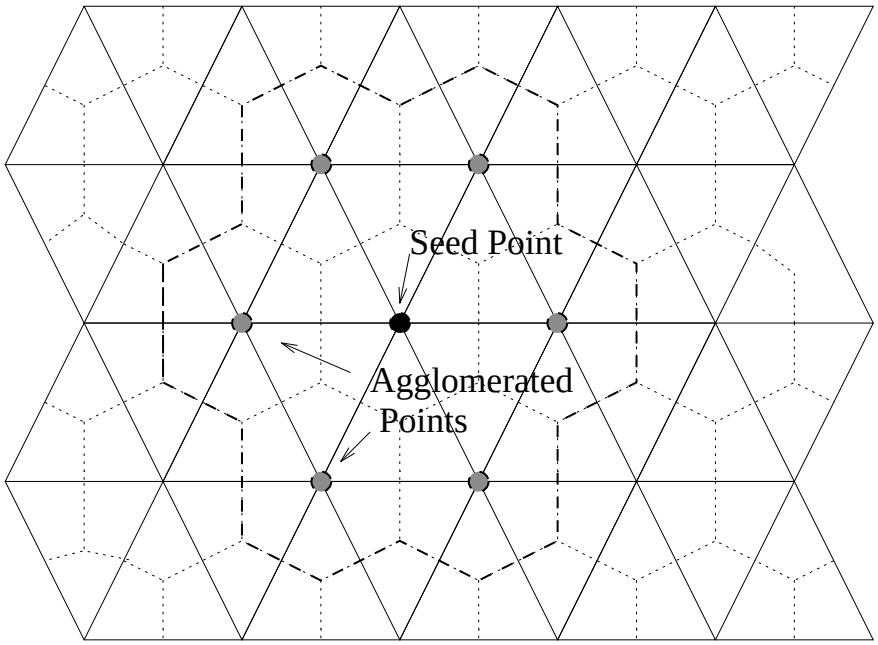

Figure 2: Agglomeration.

ther approximation has been taken by always representing a footprint with a circle. So each footprint is now defined by the scalar value (e.g. density or pressure) and coordinates of the corresponding grid point, and a radius value which is the average distance from the point to all other immediately neighboring points. In this way, we can approximate each footprint, for example, as an octagon, with a set of hardware Gouraud-shaded triangles as described in [6]. Compositing is done with the hardware blending support. In this way, we can achieve good rendering rates. However, in this research, software splatting is used such that high performance general-purpose workstations may be used.

The multiple levels of approximation taken certainly degrade the quality and accuracy of visualization results. The goal is to have a quick view of the data. Although our approximated splatting approach provides a crude picture of the actual physical phenomena, it gives the viewer a pretty good impression about the size, shape, location of the phenomena.

2.3 Visualization Results

To demonstrate the effect of applying the splatting rendering to the multi-resolution representations, we use a data set containing six different resolutions as listed in Table 1. Table 2 shows the performance of a software implementation of the approximated splatting algorithm on an SGI Indigo2 with an R4400 250MHZ processor. Times are in seconds. Image of two different sizes are rendered: 400×400 and 200×200 pixels. Figure 6,7,8,9,10 and 11 display the rendered images. Figure 3 plots the surface mesh of the overall domain explored; the

Table 1: Multi-resolution M6 wing data

resolution	# vertices	# bytes
$r0$	357900	12.8M
$r1$	52656	1.89M
$r2$	8357	300K
$r3$	1669	60K
$r4$	448	16K
$r5$	161	5.8K

Table 2: Timing results of multiresolution splatting

res.	read	sort	render (200^2)	render (400^2)
$r0$	2.1	3.4	5.23	12.9
$r1$	0.3	0.3	2.03	6.4
$r2$	0.1	0.05	1.11	4.0
$r3$	0.02	0.01	1.3	5.2
$r4$	0.02	0.01	2.18	8.6
$r5$	0.02	0.01	2.33	9.4

dark, dense area corresponds to the region of interest which is near the tip of the wing.

From the timing results and corresponding rendered images, we learn that there is always a particular resolution of the *mrrd* with which we can see sufficient details to identify areas of interest at relatively low cost. In this case, both resolution *r1* and *r2* would work well when generating 200×200-pixel images. By taking advantage of the graphics hardware support, sub-second rendering rates can be achieved, making this setup even more attractive. However, note that the rendering time increases dramatically when using much lower resolution data like *r4* and *r5*. This is because each vertex now covers a much large area (the footprint) which becomes expensive to calculate in software. This indicates that very low resolutions like *r4* and *r5* should not be included in the *mrrd* since images from them are too fuzzy and expensive to be useful.

3 Fast Retrieval of Disk Resident Data

To find a subset in a memory resident data set one would normally employ ADT, k-d or quad/oct trees to reduce the search space. These memory based indexing techniques are not appropriate for our disk based data sets since they have poor paging behavior. When dealing

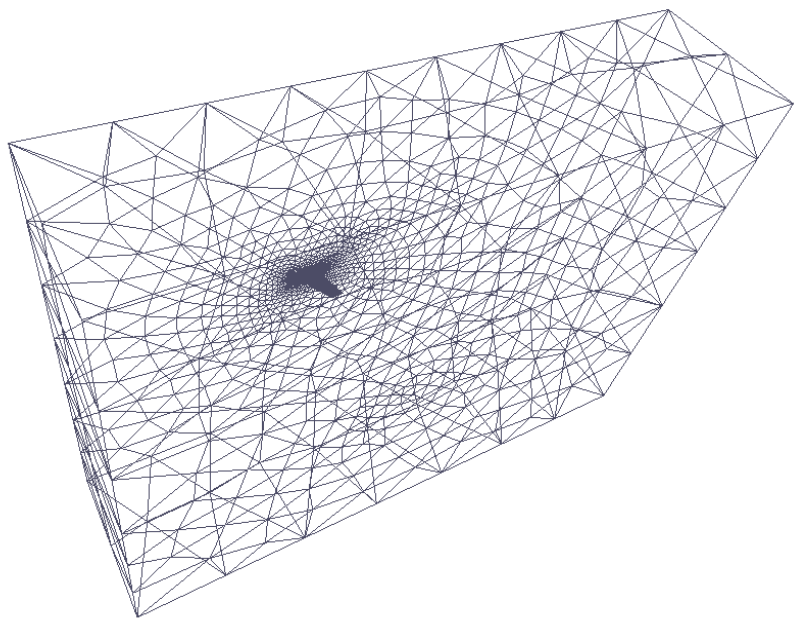

Figure 3: Overall domain as a surface mesh.

with disk based data, the primary objective is to minimize the number of pages read from disk since a disk access is two to three orders of magnitude slower than memory accesses. Of the main memory data structures, the most likely candidates for disk based data would be quad/octrees, but they suffer from the following deficiencies:

- Octrees will be imbalanced for unstructured data. Schemes to preprocess the data to determine where to put partitioning lines could be constructed to provide balance to some degree.
- Packing the octree nodes into disk pages to get good paging behavior is difficult.
- Octrees have a low fan-out of 8. *R-trees* have a fan-out of 146 for 4096 byte pages. If the number of items indexed is N, the number of nodes needing to be accessed for a small search is $O(log_8 N)$ for octree versus $O(log_{146} N)$. Note, how the number of nodes accessed turns into the number of disk pages accessed is dependent on the mapping from nodes to pages for octrees, and is 1-1 for *R-trees.*
- Octrees are designed for point data, not region data like tetrahedra.

There are many structures for indexing disk resident data, for indexing both multi-dimensional point data and region data one of the best structures is the *R-tree* [4]. With static data, *R-trees* can be loaded to 100% disk utilization and provide efficient access.

3.1 R-trees

An *R-tree* is a hierarchical data structure derived from the *B-tree* and designed for efficient execution of intersection queries. *R-trees* can be used for any number of dimensions. For clarity and brevity we limit our discussion to the two-dimensional case but our results are for the three dimension case. *R-trees* store a collection of rectangles which can change over time through insertions and deletions. Arbitrary geometric objects are handled by representing each object by the smallest upright rectangle which encloses the object.

Each node of the *R-tree* stores a maximum of n entries. Each entry consists of a rectangle R and a pointer P. For nodes at the leaf level, R is the bounding box of an actual object pointed to by P. At internal nodes, R is the minimum bounding rectangle of all rectangles stored in the subtree pointed to by P. Note that every path down through the tree corresponds to a sequence of nested rectangles, the last of which contains an actual data object. Note also that rectangles at any level may overlap and that an *R-tree* created from a particular set of objects is by no means unique.

To perform a query Q, all rectangles that intersect the query region must be retrieved and examined (regardless of whether they are stored in an internal node or a leaf node). This retrieval is accomplished by using a simple recursive procedure that starts at the root node and which may follow several paths down through the tree. A node is processed by first retrieving all rectangles stored at that node which intersect Q. If the node is an internal node, the subtrees corresponding to the retrieved rectangles are searched recursively. Otherwise, the node is a leaf node and the retrieved rectangles (or the data objects themselves) are simply returned. See [4] for a more detailed description of R-tree structures and searching.

3.2 Packing Algorithms

R-trees and variants allow for dynamic insertion and deletion at the expense of efficient search times. When the data is all present at load time and non-changing, as in our CFD data sets, preprocessing can be done to create more efficient *R-trees*. Such preprocessing is known as *R-tree packing* [12], and result in well structured trees for efficient queries and 100% disk utilization. Several algorithms exist [5, 7, 12]. We choose to focus on the Sort and Tile Recursive (STR) algorithm [7] since it is the easiest to implement and has been shown to provide more efficient point query support and at least as efficient region query support as the others [7].

In the following text we assume a data file of r rectangles and that each *r-tree* node can hold n rectangles. The general process is similar to building a B-tree from a collection of keys by creating the leaf level first and then creating each successively higher level until the root node is created [11].

General Algorithm:

1. Preprocess the data file so that the r rectangles are ordered in $\lceil r/n \rceil$ consecutive groups of n rectangles, where each group of n is intended to be placed in the same leaf level node. Note that the last group may contain less than n rectangles.

2. Load the $\lceil r/n \rceil$ groups of rectangles into pages and output the pair (MBR, page-number) for each leaf level page into a temporary file. The page-numbers are used as the child pointers in the nodes of the next higher level.

3. Recursively pack these MBRs into nodes at the next level and continue proceeding upward until the root node is created.

The above packing algorithms differ only in how the rectangles at each level are ordered. The STR algorithm orders rectangles as follows:

> "Tile" the data using $\sqrt{r/n}$ rectangular buckets of various sizes so that each bucket contains roughly $\sqrt{r/n}$ input rectangles. Once again we assume coordinates are for the center points of the rectangles. First sort the rectangles based on x-coordinate. Determine the number of leaf level pages $P = \lceil r/n \rceil$ and let $S = \lceil \sqrt{P} \rceil$. Now divide the rectangles into S vertical slices so that each vertical slice contains $S * n$ rectangles. Sort the rectangles from each slice based on y-coordinate and pack them into nodes (the first n rectangles into the first node, the next n into the second node, and so on).

Note the sorting mentioned above is disk based sorting using merge sort if the files are too large to be sorted in main memory. We call an *r-tree* created by STR packing an STR-tree.

3.3 Analysis

In this section we present analysis comparing the number of pages accessed by brute force, octrees, and *R-trees*. To make the comparison of *R-trees* with octrees more convincing we give every benefit of the doubt to octrees. Specifically, we make the following optimistic assumptions for octrees:

- The tree is perfectly balanced.

- The paging behavior is perfect, if N octree nodes can fit in a disk page and we access M nodes, then we only access $\lceil M/N \rceil$ disk pages.

- We assume all data is at the leaf level and there is no tree structure to navigate.

Table 3: Notational Definitions

T	Number of tetrahedra
N	Number of nodes
NA	Number of nodes accessed by a query
PA	Number of disk pages accessed by a query
BT	Number of bytes per tetrahedra
qx	The x-distance of a query
qy	The y-distance of a query
qz	The z-distance of a query
S	The number of nodes per side of the data cube
β	The average utilization of an octree node

- We assume the tree can be of size cube-root of the number of nodes, we do not restrict it to be a power of 8.

A real octree on unstructured data will have some skew and/or suffer from underutilized nodes. In addition, the mapping of octree nodes into pages will be suboptimal. In order to get good balance for unstructured data a tree structure will be necessary. Ignoring the non-leaf level accesses for both the octree and *R-tree* is favorable to the oct tree since the number of nodes accessed in a point search (find one specific point) by the octree is $O(log_8 T)$ versus $O(log_{146} T)$ for the *R-tree*, where T is the number of tetrahedra being indexed. Given these optimistic assumptions, we now derived the worst case number of nodes accessed for a region query of size $qx * qy * qz$. Table 3 lists the notations used.
The number of disk pages accessed for a query of region $qx * qy * qz$ for the octree is derived as follows:

$$N_{oct} = T/(8 * \beta) \tag{1}$$

$$S_{oct} = \lceil \sqrt[3]{N_{oct}} \rceil \tag{2}$$

$$NA_{oct} = (qx * S_{oct} + 1) \cdot (qy * S_{oct} + 1) \cdot (qz * S_{oct} + 1) \tag{3}$$

$$PA_{oct} = \frac{NA_{oct}}{4096/(8 \times BT)} \tag{4}$$

The number of disk pages accessed for a query of region $qx * qy * qz$ for the STR-tree is derived as follows:

$$N_{str} = T/\lfloor(4096/BT)\rfloor \tag{5}$$

$$S_{str} = \lceil\sqrt[3]{N_{str}}\rceil \tag{6}$$

$$NA_{str} = (qx * S_{str} + 1) \cdot (qy * S_{str} + 1) \cdot (qz * S_{str} + 1) \tag{7}$$

$$PA_{str} = NA_{str} \tag{8}$$

$PA_{str} = NA_{str}$ since one *R-tree* node is one disk page.
The number of disk pages accessed for the brute force method is simply:

$$PA_{brute} = \lceil\frac{T * BT}{4096}\rceil \tag{9}$$

In Figure 4 we present the number of disk accesses versus the percent of the data size assuming 10,000,000 tetrahedra where each tetrahedra occupies 28 bytes (x y z v1 v2 v3 v4). Page size is 4096 bytes. If you assume all the optimistic assumptions and 100% node occupancy for the octree, the octree is slightly better than the STR-tree. Once the occupancy is reduced to more realistic values, the octree requires up to a factor of two more page accesses. We hypothesize that once the skew and poor paging characteristics of an octree are considered, the performance will be significantly worse. In the future we intend to implement octree methods to compare experimentally with the STR-tree.

3.4 Tests Using a Local Disk

To compare subset retrieval time using our STR-tree based method versus brute force, we have performed tests on an SGI Indigo2 with an R4400 250MHZ processor, 128 MB main memory, and a dedicated disk.

3.4.1 Methodology

The data set used as a test case is an unstructured 3-d CFD grid consisting of 804,056 nodes and 4,607,888 tetrahedra. The data is stored in two binary files. The node file stores vortex coordinates, 5 CFD solution values and a node id. The tetrahedra file stores the values that delimit the lower and upper points of the smallest upright 3D region enclosing the tetrahedra and the indices of the nodes that make up the vertices of the tetrahedra. A more compact format omitting the region bounding information may be use, but this would disallow the use of R-trees or octrees and necessitate use of the brute force method and its resultant abysmal performance.

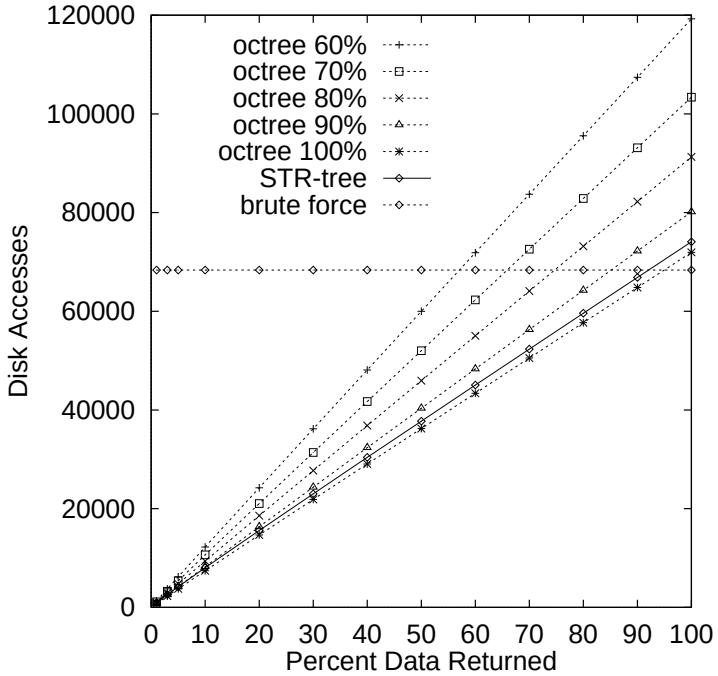

Figure 4: Analysis using 10 million tetrahedra.

The brute force method is as follows: a) read through the node file and return all nodes that are contained in the specified subset region; b) read thought the tetrahedra file and return all tetrahedra that are contained in the specified subset. The code has been optimized to read in 16K chunks at a time. Note, we ran some experiments with different blocking factors and found buffer of 16K to minimize input time.

The STR-tree based method creates two STR-trees, one for the node file and one for the tetrahedra file. The trees are then used for the searches, hence only the relevant data plus a small portion of overhead nodes are read in from memory. Note, the STR-trees are stored using the normal UNIX file system, no attempt to tune performance by clustering the nodes on raw disk was made.

The absolute and relative performance of the two policies is highly dependent on many factors, the most important being main memory size, whether the disk is local or across a network (such as when using NFS), and disk speed versus processor speed. Main memory size is especially relevant since the file system will cache pages of files in main memory between runs. Thus if main memory is sufficient ally large, subsequent runs will incur no actual disk I/O, only *soft* page faults.

Our primary comparison metric was the wall clock time (obtained from the *rusage* system call) to run a subset retrieval. Note, CPU time is not sufficient since most of the time is spent in disk retrieval. We also present the number of disk reads for the experiments. There was some variance in measured response times. All data points presented are the average of

40 runs for the same subset region. No attempt at generating confidence intervals was made.

3.4.2 Local Disk Results

In the first set of experiments we alternate running the brute force method with the *R-tree* method. This has the effect of the brute force method flushing some of the remaining pages from the buffer pool from the previous STR-tree retrieval. Thus, performance of the STR-tree method is worse than it would be in a setting of repeated queries. Figure 5 presents results for retrievals of subsets where the regions are cubes centered on the origin and increasing in size.

For a region boundary containing 164,251 tetrahedra (about 3.6% of the overall data) or less, response time is less than 5 seconds for the STR-tree method but 44 seconds or greater for the brute force method. Note that the STR-tree method results in access times that are 9.7 to 16.4 times faster than the brute force method.

As the query region increases, the response time for the brute force method is constant whereas that of the STR-tree method increases. The brute force method reads in the entire data set each time, regardless of the region size. The STR-method only brings in data for the desired region plus some extra data from around the surface of the subset region. According to our test results, the number of actual page reads for the STR-tre method is significantly smaller than the brute force method until retrieving more than 40% of the data.

So far we assume the entire memory contents from the previous STR-tree query are flushed before re-running. In a real interactive setting the user would likely select sub-regions repetitively. Since the file cache or virtual memory system will buffer some of the data between runs, we would expect better performance due to memory hits. How much better depends on how much of the data is re-used in subsequent queries. We would certainly expect the top levels of the STR-tree to be in memory [8].

Devising a string of query regions as a benchmark is not easy to do in an unbiased way. Instead, we rerun the same query 41 times and remove the response time of the first run. This provides a lower bound on the expected response time since we would expect only some of the data to be reused, not all of it. Figure 5 also includes the results for these repetitive optimistic bounds. Note, the brute force method does not benefit since it must still read in the same number of disk pages, but the STR-tree based method experiences a substantial reduction in response time. The actual response times would fall somewhere between these two extremes and be dependent on the query string. Finally, if splatting rendering is used, we only need to bring back the node data. In this case, the retrieval time is very small and could support interactive visualization.

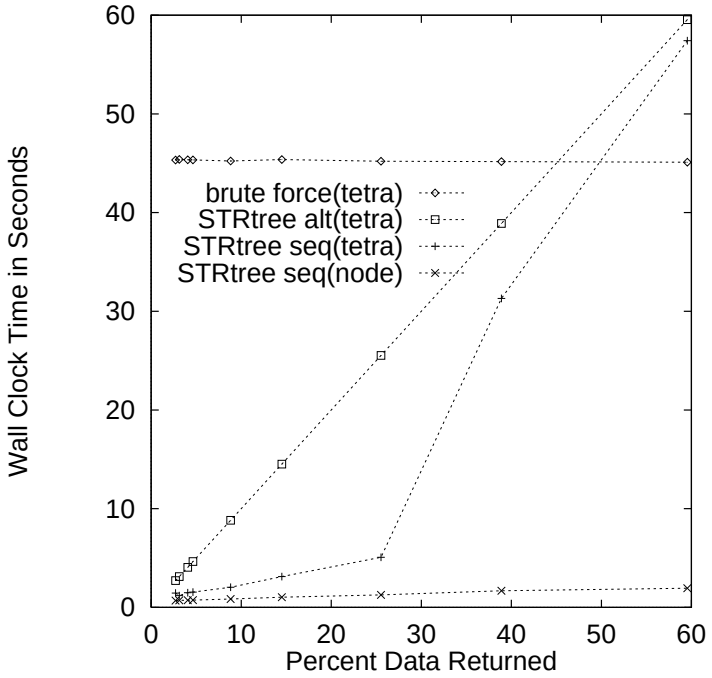

Figure 5: Experimental Results, Local Disk

3.4.3 Anticipated Effect of Larger Data Sets

The data set used in these experiments is much smaller than anticipated future data sets. When a data set is 10-100 times larger, searches using the brute force method will require 10-100 times more time, whereas we expect the STR-tree method to only require a modest amount (0-1%) more time. The reason is that with a branching factor of 146, one can increase the data size by a factor of 146 and only add one level to the STR-tree, and hence one more page access, Thus, we expect for near future data sets the STR-tree method will be yet another 10-100 times faster relative to brute force.

4 Conclusions

We have demonstrated that coupling *mrrd* with the fast splatting rendering method allows near-interactive visualization on an average workstation. A more compact *mrrd* would allows higher resolution representations resident in main memory. The current splatting method produces semi-transparent cloud like rendering. We are developing techniques for marking features, such as the geometric outline of an airplane, onto the volume rendered image to provide additional visual cues, which help select viewing position and identify correlations.

We have also shown that the *R-tree* is a more efficient data structure for retrieving disk-resident data. We intend to conduct more extensive comparison tests of the data retrieval

methods involved. First we plan to implement well balanced octrees for comparison. Next we intend to test the techniques on much larger data sets, 20-500 million tetrahedra. For these larger data sets we expect the brute force method will be 3 orders of magnitude slower than STR-trees. We should also conduct tests using a remote disk and use multiple disks to distribute an extremely large data set.

References

[1] Abgrall, R., and Harten, A. Multiresolution Representation in Unstructured Meshes. I. Preliminary Report. Tech. Rep. UCLA CAM Report 94-20, July 1994.

[2] Cignoni, P., De Floriani, L., Montani, C., Puppo, E., and Scopigno, R. Multiresolution Modeling and Visualization of Volume Data Based on Simplicaial Complexes. In *Proceedings of the 1994 Symposium on Volume Visualization* (1994), pp. 19–26.

[3] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., and Stuetzle, W. Multiresolution Analysis of Arbitrary Meshes. In *SIGGRAPH '95 Conference Proceedings* (1995), pp. 173–182.

[4] Guttman, A. R-trees: a Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD* (1984), pp. 47–57.

[5] Kamel, I., and Faloutsos, C. On Packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management (CKIM-93)* (November 1993), pp. 490–499.

[6] Laur, D., and Hanrahan, P. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Proceedings of SIGGRAPH '91* (1991), pp. 285–288.

[7] Leutenegger, S., Edgington, J., and Lopez, M. STR: A Simple and Efficient Algorithm for R-Tree Packing. Tech. Rep. Technical Report # 96-1, University of Denver Computer Science, 1996.

[8] Leutenegger, S., and Lopez, M. The Effect of Buffering on the Performance of R-Trees. Tech. Rep. Technical Report # 96-2, University of Denver Computer Science, 1996.

[9] Mavriplis, D., and Venkatakrishnan, V. Agglomeration Multigrid for Viscous Turbulent Flows. Tech. Rep. ICASE Report No. 94-62, Institute for Computer Applications in Scinece and Engineering, 1994.

[10] Muraki, S. Multiscale 3D Edge Representation of Volume Data by a DOG Wavelet. In *Proceedings of the 1994 Symposium on Volume Visualization* (1994), pp. 35–42.

[11] Rosenberg, A., and Snyder, L. Time and Space Optimality in B-Trees. *ACM Transactions on Database Systems 6*, 1 (March 1981).

[12] Roussopoulos, N., and Leifker, D. Direct Spatial Search on Pictorial Databases Using Packed R-trees. In *Proceedings of the ACM SIGMOD* (May 1985).

[13] Samet, H. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, 1989.

[14] Schroder, P., and Sweldens, W. Spherical Wavelets: Efficiently Representing Functions on the Sphere. In *SIGGRAPH '95 Conference Proceedings* (1995), pp. 161–172.

[15] Schroeder, W., Zarge, J. A., and Lorensen, W. E. Decimation of Triangle Meshes. In *SIGGRAPH '92 Conference Proceedings* (1992), pp. 65–170.

[16] Turk, G. Re-Tiling Polygonal Surfaces. In *SIGGRAPH '92 Conference Proceedings* (1992), pp. 55–64.

[17] Westermann, R. A Multiresolution Framework for Volume Rendering. In *Proceedings of the 1994 Symposium on Volume Visualization* (1994), pp. 51–58.

[18] Westover, L. Footprint Evaluation for Volume Rendering. In *Proceedings of SIGGRAPH '90* (1990), pp. 267–276.

[19] Wilhelms, J., and Van Gelder, A. Multi-Dimensional Trees for Controlled Volume Rendering and Compression. In *Proceedings of the 1994 Symposium on Volume Visualization* (1994), pp. 27–34.

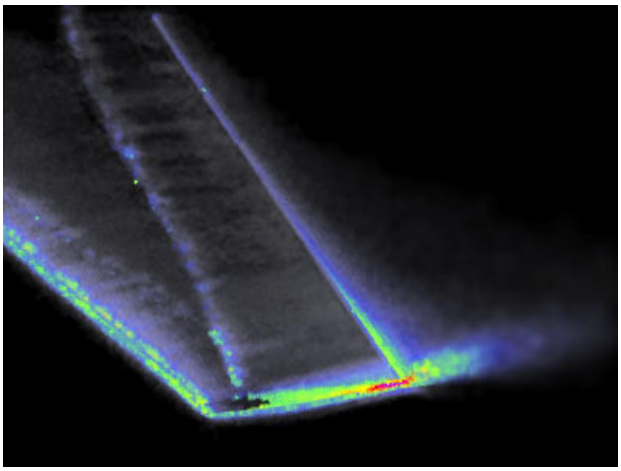

Figure 6: Highest resolution: $r0$.

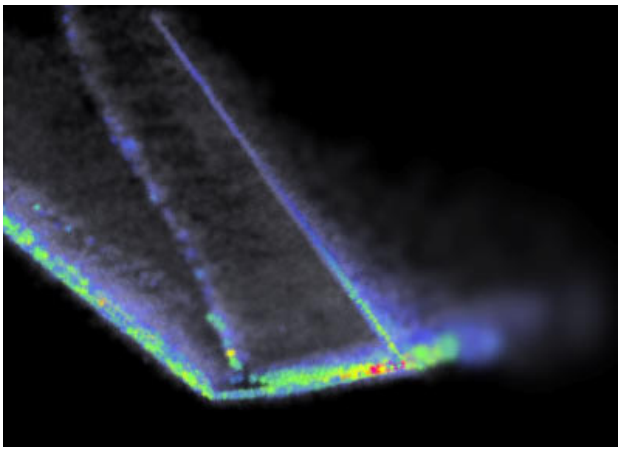

Figure 7: Resolution: $r1$.

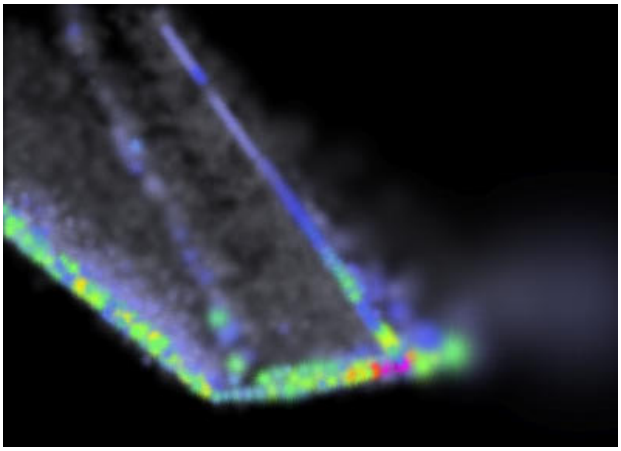

Figure 8: Resolution: $r2$.

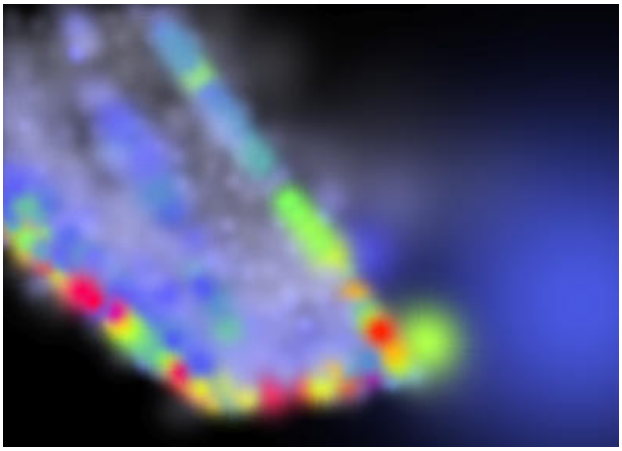

Figure 9: Resolution: $r3$.

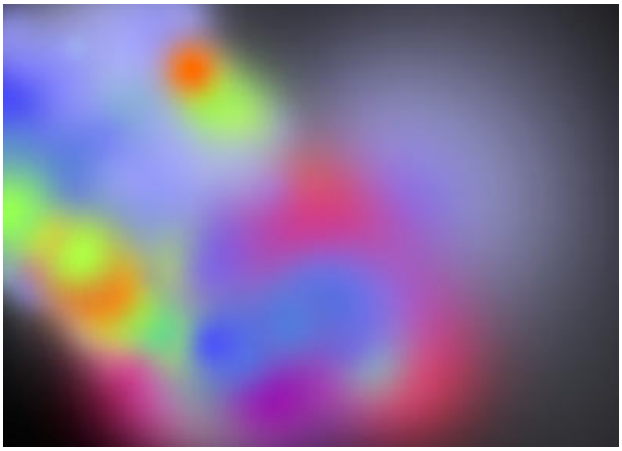

Figure 10: Resolution: $r4$.

Figure 11: Lowest resolution: $r5$.