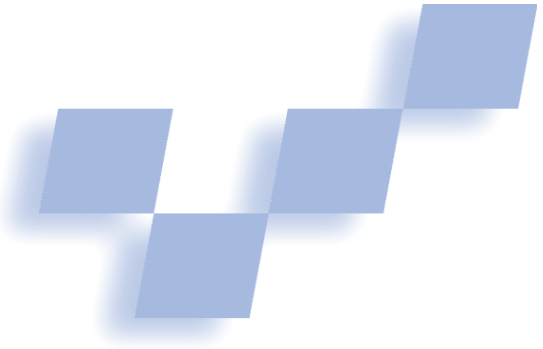
Out-Of-Core Rendering of Large, Unstructured Grids



Ricardo Farias
State University of New York at Stony Brook

Cláudio T. Silva
AT&T

We address the problem of rendering large, unstructured volumetric grids and present a set of techniques that render arbitrarily large data sets on machines with limited memory.

The need to visualize unstructured volumetric data arises in a broad spectrum of applications including structural dynamics, structural mechanics, thermodynamics, fluid mechanics, and shock physics. One of the most powerful visualization techniques is direct volume rendering, a set of rendering techniques that avoids generating intermediary surface representations of the volume data. Direct volume rendering techniques are based on creating optical models that determine how the volume data interacts with light. By changing the modeling, it's possible to render different features of the volume.[1]

Here we address the problem of direct volume rendering of large, unstructured volumetric grids on machines with limited memory. This problem is interesting because such data sets are likely to come from computations generated on supercomputers, while visualization often happens on smaller, desktop machines. Our work also complements the recent trend of developing efficient out-of-core scientific visualization techniques. Given large, unstructured grids, currently several external memory visualization tools exist (such as isosurface computation,[2] streamline computation,[3] and surface simplification[4]) that help scientists visualize their large data sets on machines with limited memory. For instance, by coupling the techniques of Lindstrom[4] and Chiang, Silva, and Schroeder,[2] researchers can compute and simplify isosurfaces of arbitrarily large data sets, effectively visualizing such large data sets on any machine with enough disk space. Our work adds direct volume-rendering algorithms to this already powerful toolbox. (See the "Related Work" sidebar for more background information.)

We present two techniques that vary in rendering speed, disk and memory usage, ease of implementation, and preprocessing costs. The first is a memory-insensitive rendering (MIR) technique that is completely disk-based and requires a small amount of constant main memory. The second technique is based on our ZSweep algorithm. It's more involved in its preprocessing, implementation, and main-memory requirements but can be substantially faster.

Memory-insensitive rendering

In developing efficient external memory algorithms, users must know some characteristics of computer disks and their differences from the in-core main-memory system we're all accustomed to. The basic difference is that disks aren't efficient for random access to locations because "seeks" require a large amount of mechanical movement (of the heads). For sequential access, disks are fast, with a raw bandwidth within a factor of 20 of the main-memory system. Also, we can increase disk bandwidth inexpensively by using several disks in parallel. The appeal of hard drives is that the cost is much lower—on the order of 100 times cheaper than main memory. The need for sequential access when using disks has profound implications for external memory algorithms.

First, the file formats used for out-of-core algorithms must be different and generally more redundant. Indexed mesh formats are common for main-memory techniques. For instance, it's common to save a list of the vertices represented with four floats: the position (x, y, z); scalar field value; and a list of tetrahedra, referenced by four integers that refer to the vertices defining the given tetrahedron. Before we can use such data sets in our algorithm, they must be normalized—a process that dereferences the pointers to vertices. (The Chiang, Silva, and Schroeder paper thoroughly explains this process.[2])

For completeness, we'll briefly explain how to nor-

0272-1716/01/$10.00 © 2001 IEEE

Related Work

The work we describe in this article is mainly related to techniques for rendering unstructured grids and out-of-core visualization techniques. Both are active research areas in scientific visualization. In this sidebar, we briefly review each of these areas.

Unstructured-grid volume rendering

Here we consider existing unstructured-grid volume-rendering techniques from a memory-usage point of view, their applicability to render large grids, and potential extensions for out-of-core rendering. The memory usage of current techniques vary widely, and a straightforward classification of the different techniques isn't possible. Here are some of the various characteristics that generally affect the memory usage of existing techniques:

- the data set's size, in terms of its number and type of cells and vertices. (Given a mesh with t tetrahedra and n vertices, the minimum memory necessary to hold it—assuming uncompressed data and 32 bits for integers and floating-point numbers—is $16(t + n)$ bytes.)
- screen resolution and the data set's image-space depth. (In image space, the memory costs depend on the screen resolution and the data set's thickness along the z direction. Some techniques compute slices along z by intersecting discrete buffers of the same resolution as the screen with the unstructured grid. Assuming 1 byte per color channel, for computing an image of size N-by-N with s slices, we need $4sN^2$ bytes. We note that s should vary with the resolution of the data set in z. That is, if a ray that intersects the data set in s_{max} cells exists, then the closer s gets to s_{max} the more accurate the image we can obtain.)
- the use of mesh connectivity information. Some techniques explicitly use connectivity information, while others use different means of inferring it (such as discrete buffers used for determining depth information) or completely avoid using any kind of connectivity.
- the underlying data structures used for efficiency or accuracy. For instance, some techniques cache extra information per cell or per face of the data set for efficiency.

Researchers have developed several efficient algorithms for rendering irregular grids. One class of algorithms is based on adapting ray-tracing techniques for rendering unstructured grids, such as in the works of Garrity,[1] Uselton,[2] and Bunyk, Kauman, and Silva.[3] In general, these techniques require random access to the cells, connectivity information, and in some cases, extra memory to optimize the computation of intersections of rays with faces of the cell complex. Yang, Mitra, and Chiueh's paper[4] proposes an optimization for the technique in the Bunyk paper[3] that attempts to reduce the memory requirements by compositing samples as early as possible, but the proposed view-independent traversal doesn't limit the overall memory use. (The work of Hong and Kaufman,[5] although similar to that in the Bunyk paper,[3] is optimized for curvilinear grids. They used considerably less memory because their system uses the grid structure and doesn't explicitly store cell or connectivity information.)

Researchers have developed other techniques that use scan-line algorithms, which sweep the data with a plane perpendicular to the image plane.[6] Some of these techniques[7] are designed to be memory efficient but still use the mesh's connectivity. Others, such as those proposed by Giertsen[8] and Westermann and Ertl,[9] use discrete buffers to determine the compositing order and completely avoid the need for connectivity information. Using discrete buffers in z potentially lowers the accuracy of these techniques, and the buffers themselves can require a substantial amount of memory.

Some methods[6,10] employ a different kind of sweep algorithm and sweep planes in z. Yagel et al.[11] sample the irregular grid with a fixed number of planes that are later composited together. Their technique doesn't use connectivity, but the space to keep the planes can be substantial because it amounts to computing and caching many images. Farias, Mitchell, and Silva[10] developed ZSweep, which is also based on sweeping a plane in the z direction.

Another approach for rendering irregular grids is using face projection, or feed-forward, methods[12-14] in which the cells are projected onto the screen one by one. Most of these techniques exploit the graphics hardware to compute the volumetric lighting models[13] by first computing a visibility ordering[12,15,16] and incrementally accumulating their contributions to the final image. With respect to memory usage, we can separate the visibility ordering algorithms into two classes: those that use connectivity to compute the ordering[12,16] and those that use some form of power-sorting.[14] The power sorting techniques only require an extra floating-point number per cell, and they don't use connectivity information. In general, those techniques aren't guaranteed to generate correct sorting results for a wide class of grids.

One simple approach[17] is to naively compute all intersections between each ray cast with all the cells and perform a postsorting to compute the image. That is, given an N-by-N image and n cells, for each of the N^2 rays, compute the $O(n)$ intersections with cell facets in time $O(n)$ and then sort these crossing points in $O(n \log n)$ time.

continued on p. 4

continued from p. 3
However, this results in overall time $O(N^2 n \log n)$ and doesn't take advantage of coherence in the data—the sorted order of cells crossed by one ray isn't used in any way to assist in the processing of nearby rays.

Ma and Crockett[18] used this approach in the context of parallel architectures. Their technique distributes the cells among processors in a round-robin fashion. For each viewpoint, each processor independently computes the ray intersections, which are later composited in the algorithm's second phase. To avoid storing many ray intersections, Ma and Crockett cleverly schedule the computation using a *k*-d tree.

Out-of-core scientific visualization

For a general introduction to out-of-core scientific visualization theory and practice of external memory algorithms, readers should see Abello and Vitter.[19]

Cox and Ellsworth[20] propose a general framework for the systems based on application-controlled demand paging. Leutenegger and Ma[21] propose using R-trees[22] to optimize searching operations on large unstructured data sets. Ueng, Sikorski, and Ma[23] use an octree partition to restructure unstructured grids, optimizing the computation of streamlines. Shen, Chiang, and Ma[24] and Sutton and Hansen[25] have developed techniques for indexing time-varying data sets. Shen, Chiang, and Ma[24] apply their technique for volume rendering, while Sutton and Hansen[25] focus on isosurface computations.

Chiang and Silva[26] worked on I/O-optimal algorithms for isosurface generation. Their work assumes that even the preprocessing is performed completely on a machine with limited memory. Although their technique is fast in terms of actually computing the isosurfaces, the disk and preprocessing overhead of their technique is substantial. This led to further research[27] on techniques that can trade disk overhead for time in the querying for the active cells. They developed a set of useful metacell preprocessing techniques. Recently, Lindstrom[28] and El-Sana and Chiang[29] developed external memory algorithms for surface simplification. The technique in Lindstrom[30] simplifies arbitrarily large data sets on machines with just enough memory to hold the output triangle mesh.

References

1. M. Garrity, "Raytracing Irregular Volume Data," *Computer Graphics* (San Diego Workshop Volume Visualization), vol. 24, no. 5, Nov. 1990, pp. 35-40.
2. S. Uselton, *Volume Rendering for Computational Fluid Dynamics: Initial Results*, tech. report RNR-91-026, NASA Ames Research Center, Moffett Field, Calif., 1991.
3. P. Bunyk, A. Kaufman, and C. Silva, "Simple, Fast, and Robust Ray Casting of Irregular Grids," *Scientific Visualization* (Proc. Dagstuhl 97), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 30-36.
4. C.-K. Yang, T. Mitra, and T. Chiueh, "On-the-Fly Rendering of Losslessly Compressed Irregular Volume Data," *Proc. IEEE Visualization 2000*, ACM Press, New York, 2000.
5. L. Hong and A. Kaufman, "Accelerated Ray-Casting for Curvilinear Volumes," *Proc. IEEE Visualization 98*, ACM Press, New York, 1998, pp. 247-254.
6. J. Wilhelms et al., "Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids," *Proc. IEEE Visualization 96*, ACM Press, New York, 1996, pp. 57-64.
7. C. Silva and J. Mitchell, "The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 2, Apr.–Jun. 1997, pp. 104-157.
8. C. Giertsen, "Volume Visualization of Sparse Irregular Meshes," *IEEE Computer Graphics and Applications*, vol. 12, no. 2, Mar. 1992, pp. 40-48.
9. R. Westermann and T. Ertl, "The VSbuffer: Visibility Ordering of Unstructured Volume Primitives By Polygon Drawing," *Proc. IEEE Visualization 97*, ACM Press, New York, 1997, pp. 35-42.
10. R. Farias, J. Mitchell, and C. Silva, "ZSweep: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," *Proc. 2000 Volume Visualization Symp.*, ACM Press, New York, 2000, pp. 91-99.
11. R. Yagel et al., "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *Proc. 1996 Volume Visualization Symp.*, ACM Press, New York, 1996, pp. 55-62.
12. P.L. Williams, "Visibility-Ordering Meshed Polyhedra," *ACM Trans. Graphics*, vol. 11, no. 2, Apr. 1992, pp. 103-126.
13. P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics* (San Diego Workshop Volume Visualization), vol. 24, no. 5, Nov. 1990, pp. 63-70.
14. N. Max, P. Hanrahan, and R. Crawfis, "Area and Volume for Efficient Visualization of 3D Scalar Functions," *Computer Graphics* (San Diego Workshop Volume Visualization), vol. 24, no. 5, Nov. 1990, pp. 27-33.
15. C. Stein, B. Becker, and N. Max, "Sorting and Hardware Assisted Rendering for Volume Visualization," *Proc. 1994 Symp. Volume Visualization*, ACM Press, New York, 1994, pp. 83-90.
16. J. Comba et al., "Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids," *Computer Graphics Forum*, vol. 18, no. 3, Sept. 1999, pp. 369-376.
17. C. Silva, J. Mitchell, and A. Kaufman, "Fast Rendering of Irregular Grids," *Proc. 1996 Volume Visualization Symp.*, ACM Press, New York, 1996, pp. 15-22.
18. K.-L. Ma and T.W. Crockett, "A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data," *Proc. IEEE Parallel Rendering Symp.*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 95-104.

19. J. Abello and J. Vitter, *External Memory Algorithms*, American Mathematical Soc., Providence, R.I., 1999.
20. M. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," *Proc. IEEE Visualization 97*, ACM Press, New York, 1997, pp. 235-244.
21. S. Leutenegger and K.-L. Ma, "Fast Retrieval of Disk-Resident Unstructured Volume Data for Visualization," *External Memory Algorithms and Visualization,* Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) Book Series, vol. 50, American Mathematical Soc., Providence, R.I., 1999.
22. A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Conf. Principles Database Systems*, ACM Press, New York, 1984, pp. 47-57.
23. S.-K. Ueng, C. Sikorski, and K.-L. Ma, "Out-of-Core Streamline Visualization on Large Unstructured Meshes," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 4, Oct.–Dec. 1997, pp. 370-380.
24. H.-W. Shen, L.-J. Chiang, and K.-L. Ma, "A Fast Volume Rendering Algorithm for Time-Varying Fields Using A Time-Space Partitioning (TSP) Tree," *Proc. IEEE Visualization 99*, ACM Press, New York, 1999, pp. 371-378.
25. P.M. Sutton and C.D. Hansen, "Accelerated Isosurface Extraction in Time-Varying Fields," *IEEE Trans. Visualization and Computer Graphics*, vol. 6, no. 2, Apr.–Jun. 2000, pp. 98-107.
26. Y.-J. Chiang and C.T. Silva, "I/O Optimal Isosurface Extraction," *IEEE Visualization 97*, ACM Press, New York, 1997, pp. 293-300.
27. Y.-J. Chiang, C.T. Silva, and W.J. Schroeder, "Interactive Out-of-Core Isosurface Extraction," *Proc. IEEE Visualization 98*, ACM Press, New York, 1998, pp. 167-174.
28. P. Lindstrom, "Out-of-Core Simplification of Large Polygonal Models," *Computer Graphics* (Proc. Siggraph 2000), ACM Press, New York, 2000, pp. 259-262.
29. J. El-Sana and Y.-J. Chiang, "External Memory View-Dependent Simplification," *Computer Graphics Forum*, vol. 19, no. 3, Aug. 2000, pp. C-139–C-150.

malize such a file, with v vertices and t tetrahedra. In an initial pass, we create two binary files: one with the list of vertices and another with the list of tetrahedra. Next, in four passes, we dereference each tetrahedral file index and replace it with the actual position and scalar field values for the vertex. To do this efficiently, we first externally sort the current version of the tetrahedra file in the index we intend to dereference. This takes time $O(t \log t)$ using an external memory merge–sort. Then, we perform a synchronous scan of both the vertex and sorted tetrahedra file, reading one record at a time and appropriately outputting the deferenced value for the vertex. Note that we can do this efficiently in time $O(v + t)$ because all the references for vertices are sorted. When we're done with all four passes, the tetrahedra file will contain t records with the value (not reference) of each of its four vertices.

In our first out-of-core rendering technique, MIR, the algorithm receives a transformation matrix, screen resolution, the normalized tetrahedron file, and associated transfer functions for lighting calculations as input.

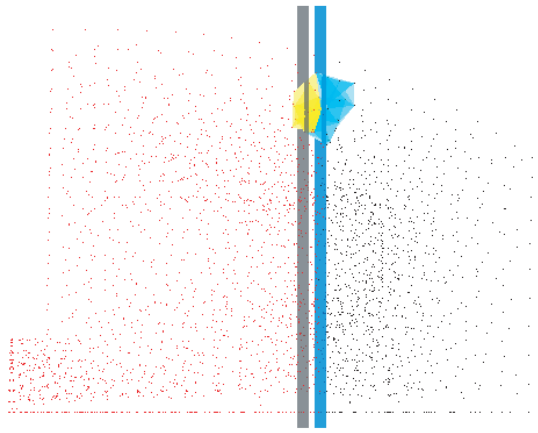
1. The first step in our algorithm is to read each cell (tetrahedron) from the normalized file, transform it with the specified transformation matrix, and compute all its ray intersections. For each pixel ρ_i, which intersects the cell in the interval (z_0, z_1), we output two records (ρ_i, z_0) and (ρ_i, z_1). For color calculations, we also save an interpolated scalar field value. This allows for fast regeneration of images with different transfer functions or (with some changes) the efficient rendering of time-varying data sets. The amount of memory necessary to perform this step is minimal; it's just enough to hold the cell's description and enough temporary storage to compute one intersection, because they're written to disk one by one as they're computed. The amount of disk space required is proportional to the number of actual ray stabbings between rays and cells.
2. The second (and generally, most time consuming) step consists of sorting the file with the ray intersections computed in the previous step, using an appropriate compare function. The compare function we use sorts primarily on the pixel identification ρ_i and secondarily on the depth of intersection z. In other words, after the file is sorted and the records for a particular pixel are together (that is, they appear sequentially in the file), the records are ordered in increasing depth.
3. The third and final step in our scheme is to traverse the ordered file generated in the previous step, use the transfer functions to light, and composite the samples, which are already in the correct order.

Our simple algorithm is essentially an external memory version of a technique previously considered by other researchers.[5,6] One group[5] discarded the technique as too inefficient because it didn't use coherency between rays. Ma and Crockett[6] used this technique for its good load-balancing characteristics. However, to make it practical, they had to optimize it to save space. No space optimizations are necessary for the out-of-core version to be useful. With this scheme, we can render an arbitrarily large image of an arbitrarily large data set if enough disk space exists to save the intersection crossings. It's also simple to implement. It doesn't use any random access to the data set, and its implementation only requires an external sort routine and code to perform ray-cell intersection.

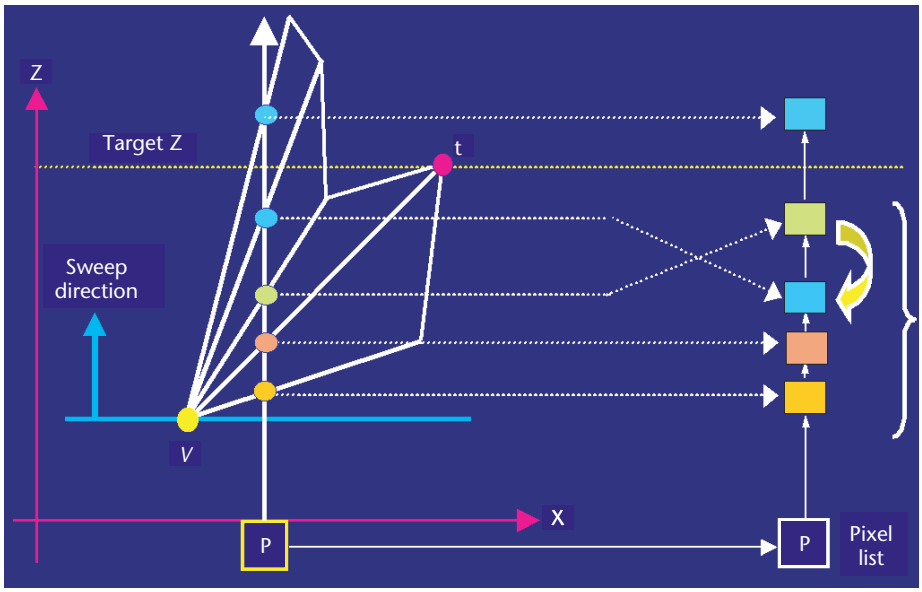
Out-of-core ZSweep

Our second technique is slightly more complex but is often a more efficient out-of-core unstructured grid renderer. It's based on our ZSweep algorithm[7] (see Figure 1, next page, for an overview).

The in-core ZSweep algorithm is based on sweeping the data with a plane parallel to the viewing plane (see the blue plane in Figure 1a) in order of increasing z, pro-



(a)



(b)

1 The in-core ZSweep algorithm. (a) 3D sweep portion of ZSweep. In blue, we show the sweep plane. The swept points are in black, and points that haven't been touched yet are in red. We highlight the tetrahedra incident on the current event point. The newly found faces (which generate new intersections) are in yellow, and the old faces are in cyan. (b) ZSweep compositing in 2D (that is, along a plane perpendicular to the viewing direction) for clarity. The current event point v is in yellow. We also show the newly found faces and the intersections along a general ray. Each intersection contributes a color and has to be composited in the correct order. The ordering computed with an insertion sort is on the right.

jecting the faces of cells that are incident to vertices as they're encountered by the sweep plane. ZSweep's face projection differs from the ones used in other projective methods.[8] During face projection, we compute the intersection of the ray emanating from each pixel and store their z-value and other auxiliary information in a sorted list of intersections for the given pixel. Our data structure for keeping the intersections is similar to an A-buffer.[9] We defer the lighting calculations[1] to a later phase (see Figure 1b). The algorithm performs compositing when it reaches the target Z plane (see the gray plane in Figure 1a). The efficiency arises because the algorithm exploits the implicit (approximate) global ordering that the vertices' z-ordering induces on the cells that are incident on them. This leads to only a few ray intersections that must be processed out of order. The efficiency also arises from using early compositing, which makes the algorithm's memory footprint quite small. The key properties for ZSweep's efficiency is that given a mesh with v vertices and c cells, the amount of sorting ZSweep does is $O(v \log v)$ in practice. Depending on the number of ray intersections, this is substantially lower than the amount necessary to sort all the intersections for each pixel.

ZSweep has two sources of main-memory usage: the pixel intersection lists and the actual data set. The data-set storage requirements represent our largest memory use. Besides the storage for the actual vertices and cells, we must also keep each vertex's use set—that is, the cells incident to each vertex.

The basic idea in our out-of-core technique is to break the data set into chunks of fixed size that we can render independently without using more than a constant amount of memory. To further limit the amount of memory necessary, we subdivide the screen into tiles, and for each tile, we render the chunks that project into it in a front-to-back order. This gives us the same optimizations as the in-core ZSweep algorithm where we've shown that image tiling leads to substantial performance improvement because of better cache coher-

ence.[10] Subdividing the screen into tiles and the data set into chunks that are rendered independently has successfully been applied to a parallelization of ZSweep.

We divided our algorithm into two parts: a view-independent preprocessing phase, which must be performed only once and generates a data file on disk that we can use for all rendering requests, and a view-dependent rendering algorithm.

Preprocessing

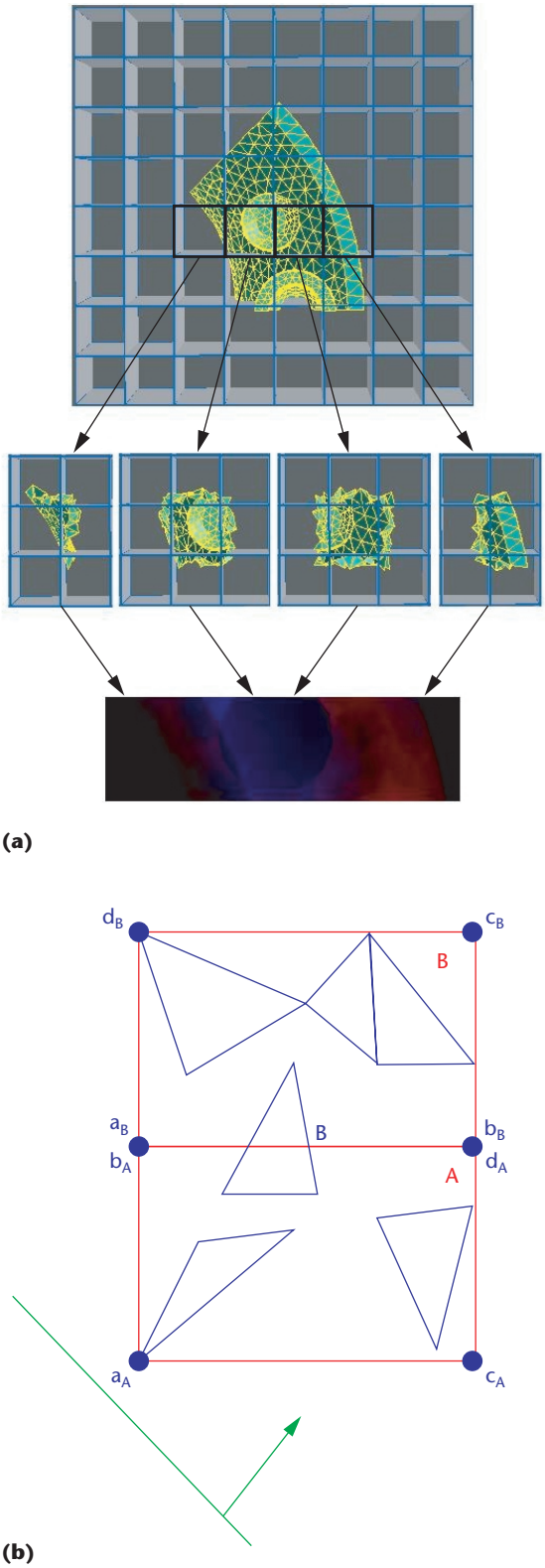
Our preprocessing is simple, and it resembles the metacell creation in the Chiang article.[2] Basically, we break the data-set file into several metacells of small, roughly fixed size. (The metacells and their construction are slightly different in Chiang,[2] because each cell belongs to a single metacell. In our case, a cell belongs to as many metacells as it spatially intersects. This isn't a substantial difference, and the normalization techniques described there still apply.) Given a target number of vertices per metacell m out of v total vertices, we first externally sort all vertices by the x-values and partition them into $\sqrt[3]{v/m}$ consecutive parts. Then, for each such chunk, we externally sort its vertices by the y-values and partition them into $\sqrt[3]{v/m}$ parts. Finally, we repeat the process for each refined part, except that we externally sort the vertices by the z-values. We take the final parts as chunks. This is the main step in constructing the chunks because it determines their shape and location in space. Chunks might differ dramatically in their volumes, but their numbers of vertices are roughly the same.

In general, the number of metacells is relatively small, so we can safely assume they fit in the memory. To render a metacell, ZSweep must have all the cells that spatially intersect that metacell and all the vertices that belong to those cells. These computations can be efficiently computed in external memory. (For full details, see the Chiang article.[2]) Our preprocessing outputs two files. The small one is a high-level description of the metacells, including their bounding box, number of vertices, number of cells, and a pointer to the start of the data for the metacell in the main data file. The larger data file is a list of the vertices and cells for each metacell. Note that several vertices and cells are repeated (possibly multiple times) in this data file, because each metacell is a self-contained unit.

Rendering algorithm

Our rendering algorithm is simple. Basically, we divide the screen into tiles and render the image tile by tile. For each tile, we compute the metacells that intersect that tile, sort the metacells in a front-to-back order, and render it using the ZSweep algorithm.

Figure 2 shows the details. For each tile, we find M, the set of the metacells that project into it. Then, we sort the vertices of the bounding boxes of M in front-to-back order by inserting them on a queue Q. The queue is used for sweeping the vertices, which have several marks. In particular, we tag vertices based on whether they're bounding-box or data-set vertices. When the sweep plane touches the first bounding-box vertex of a metacell m, we retrieve all the vertices and cells of m from disk, transform the vertices, and insert them on Q, tagging them as data-set vertices. Out-of-core ZSweep processing is essentially the same as the in-core algorithm, but it performs reading operations lazily. As it reaches vertices, it projects faces; Figure 1 shows the overall operation. As the algorithm touches bounding-box vertices, we keep track of the number of bounding-box ver-



2 The rendering portion of out-of-core ZSweep, which is performed in (a) tiles. (b) After reaching eight bounding-box vertices of a given metacell, we can safely deallocate the metacell.

Table 1. Main data sets we used for benchmarking.

Data Set	Number of Vertices (1,000)	Number of Cells (1,000)	Metacell File (Kbytes)	Metacell Data (Mbytes)	Normalized File (Mbytes)
Blunt Fin	41	187	40	26	12.7
Combustion Chamber	47	215	40	23	14.6
Oxygen Post	109	513	110	82	34
Delta Wing	212	1,005	254	205	68
SPX	2.9	13	2.6	1.2	0.8
SPX1	20	103	15	12	8
SPX2	150	830	63	110	71
SPX3	1,150	6,620	56	706	641

Table 2. Rendering times (in seconds) for our memory-insensitive irregular grid rendering algorithm.

Screen Resolution 512 x 512				
Data Set	Blunt fin	Combustion chamber	Oxygen post	Delta wing
Projection time	45	10	81	103
Time to order	213	19	386	412
Compositing time	44	6	75	79
Total time	302	35	542	594
Screen Resolution 1024 x 1024				
Data Set	Blunt fin	Combustion chamber	Oxygen post	Delta wing
Projection time	171	24	291	338
Time to order	1,030	82	1,747	1,965
Compositing time	180	26	316	322
Total time	1,381	132	2,354	2,625
Screen Resolution 2048 x 2048†				
Data Set	Blunt fin	Combustion chamber	Oxygen post	Delta wing
Projection time	254	52	435	496
Time to order	589	190	922	1,062
Compositing time	233	55	422	430
Total time	1,076	297	1,779	1,988

† We obtained the times for the 2,048 × 2,048 on a SGI R12K 400-Mhz system, with a fast SCSI disk array. Faster disks on the SGI lead to substantially improved times.

tices of a given metacell that we've seen so far. When this number reaches eight, we can safely deallocate the metacell (see Figure 2b). When we reach vertex d_a, we can free the memory from metacell a.

Experimental results

Here we report results for our two out-of-core rendering techniques and the in-core ZSweep algorithm. When not indicated, we obtained our results on a PC-class machine equipped with an AMD K7 Thunderbird 1-GHz processor, one IDE disk, and 1 Gbyte of main memory running Linux. To limit the amount of main memory available for testing purposes, we used the Linux kernel to indicate the amount of main memory to use by specifying the boot parameters directly into Linux Loader (lilo)—for example, specifying linux mem=32M at the boot prompt. (Chiang, Silva, and Schroeder use a similar methodology.[2] Simply limiting the amount of memory generally isn't enough because the operating system is likely to perform aggressive caching if enough memory is available, thus effectively transferring the data set into memory implicitly.) Table 1 has information about the data sets we used in our tests. The first four are tetrahedralized versions of the well-known NASA data sets. SPX is an unstructured grid (see Figure 1a and 2a) composed of tetrahedra. We subdivided each tetrahedron into eight for each version of the last three—that is, SPX3 is 512 times larger than SPX.

MIR

We've generated several images of the benchmark data sets using our MIR rendering algorithm. Theoretically, MIR shouldn't depend on the amount of main memory available (see Table 2). The four columns in Table 2 for each image dimension show the time it took to project the cells on the screen, the time to order the projection file, the time to compose all intersections, and the total render time.

In all our experiments, our code never used more than 5 Mbytes of main memory. It takes the normalized file as its input. Given a new point of view, it rotates the cells one by one and projects their faces on the screen with a scan conversion that's directly saved in the projection file. The projection file's size depends on the image's dimension and also on the number of segments generated for each pixel. It can get large, but the algorithm works the same. Note that the cost of the algorithm's last step, the compositing, also depends on the average

Table 3. Rendering times (in seconds) for the in-core ZSweep code running with 1 Gbyte of RAM.

Data Set	512^2	1024^2	2048^2
SPX	7	26	118
SPX1	14	46	203
SPX2	29	93	383
SPX3	107	238	834

length of segments. Depending on the data set and image size, MIR can use a lot of disk space. For example, for the Delta, the projection file has 304 Mbytes for a 512×512 image, 1.2 Gbytes for a 1024×1024, and 4.8 Gbytes for a 2048×2048.

Large images

We ran some tests with a large data set (not included in Table 1) containing roughly 1.5 million vertices and 8.5 million cells. Generating a 5000×5000 image (which takes up more than 70 Mbytes of disk) took MIR 224 seconds on a SGI Origin 3000 equipped with R12K 400-Mhz processors and a fast SCSI disk array. This is faster than our other data sets because the number of ray intersections is small. We also generated a $10{,}000 \times 10{,}000$ image from the same data set that took 824 seconds. In this case, the image occupies 300 Mbytes of disk.

Out-of-core ZSweep

Tables 3 and 4 show some results for our out-of-core ZSweep code. Out-of-core ZSweep has constant memory usage per data set, irrespective of the size of the images being generated, and can generate images that the original in-core ZSweep couldn't. For a 2048×2048 image of the Delta, the in-core ZSweep would need more than 380 Mbytes of memory, but the out-of-core ZSweep only needs about 24 Mbytes.

Our experiments show that MIR and out-of-core ZSweep are practical techniques we can use under different conditions. Out-of-core ZSweep is usually more efficient than MIR, sometimes by a factor of 10 or more, but it requires that we preprocess the files with the metacell technique before rendering. However, out-of-core ZSweep uses more memory than MIR. For generating a few high-resolution images of large data sets, MIR might be a good choice.

The MIR code is considerably slower because it performs more sorting and disk I/O. MIR might be particularly useful when trying to render a data set from the same viewpoint with a different transfer function. Because the mapping from scalar values to color (as specified in the transfer function file) is performed during compositing, we can effectively generate images with different classifications efficiently. Also, it would be efficient to render time-varying data sets because the expensive ordering doesn't need to be redone.

Conclusions

We presented two out-of-core volume techniques, which we implemented and tested against one another, and compared their rendering times and memory requirements against the in-core ZSweep algorithm.[7]

Table 4. Rendering times for the out-of-core ZSweep using 128 Mbytes of RAM. We show the time (in seconds) to generate the image and the cost per cell (in μs).

Data Set	512^2		1024^2		2048^2	
SPX	8	615	34	2,615	154	11,846
SPX1	24	233	72	699	305	2,961
SPX2	78	93	160	192	595	716
SPX3	289	43	418	63	1,157	174

The simplest technique, MIR, is useful when the amount of memory available is highly limited or only a few images of a given data set are necessary. We can also use MIR to compute several images of a given data set from the same viewpoint with different classifications (such as transfer functions). For using our out-of-core ZSweep, it would be best if the data's metacell representation is already available. Because such representations are useful for other purposes, such as isosurface generation,[2] we believe this scheme will prove beneficial.

We are currently exploring several extensions of our work. One of the simplest is using prefetching and multithreading to speedup the rendering further in out-of-core ZSweep, especially when multiple processors are available. For real-time rendering, it would be interesting to develop a time-critical version of out-of-core ZSweep,[11] which trades accuracy for speed during rendering. ■

Acknowledgments

We thank Peter Williams and Will Schroeder for interesting data sets and NASA for the Blunt Fin, Liquid Oxygen Post, and Delta Wing data sets. Ricardo Farias acknowledges partial support from CNPq-Brazil under a PhD fellowship. This work was made possible by the generous support of Sandia National Labs and the US Department of Energy Mathematics, Information, and Computer Science Office.

References

1. N. Max, "Optical Models for Direct Volume Rendering," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 2, June 1995, pp. 99-108.
2. Y.-J. Chiang, C.T. Silva, and W.J. Schroeder, "Interactive Out-of-Core Isosurface Extraction," *IEEE Visualization 98*, ACM Press, New York, 1998, pp. 167-174.
3. S.-K. Ueng, C. Sikorski, and K.-L. Ma, "Out-of-Core Streamline Visualization on Large Unstructured Meshes," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 4, Oct.–Dec. 1997, pp. 370-380.
4. P. Lindstrom, "Out-of-Core Simplification of Large Polygonal Models," *Computer Graphics* (Proc. Siggraph 2000), ACM Press, New York, 2000, pp. 259-262.
5. C. Silva, J.S.B. Mitchell, and A.E. Kaufman, "Fast Rendering of Irregular Grids," *1996 Volume Visualization Symp.*, ACM Press, New York, 1996, pp. 15-22.
6. K.-L. Ma and T.W. Crockett, "A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimen-

sional Unstructured Data," *Proc. IEEE Parallel Rendering Symposium*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 95-104.
7. R. Farias, J. Mitchell, and C. Silva, "ZSweep: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," *Proc. 2000 Volume Visualization Symp.*, ACM Press, New York, 2000, pp. 91-99.
8. P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics*, vol. 24, no. 5, Nov. 1990, pp. 63-70.
9. L. Carpenter, "The A-buffer, An Antialiased Hidden Surface Method," *Computer Graphics* (Proc. Siggraph 1984), ACM Press, New York, 1984, pp. 103-108.
10. R. Farias and C. Silva, "Parallelizing the ZSweep Algorithm for Distributed-Shared Memory Architectures," to be published in *Proc. Int'l Volume Graphics Workshop*, 2001.
11. R. Farias et al., "Time-Critical Rendering of Irregular Grids," *Proc. SIBGRAPI 2000* (Brazilian Computer Graphics Conference), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 243-250.

Ricardo Farias *is a PhD student in operations research in the Applied Math Department at the State University of New York at Stony Brook. His primary research is on visualization of large volumetric data sets and high-performance computing. He has a BS in physics from Fluminense Federal University (Rio de Janeiro, Brazil) and an MS in computer vision from the Graduate School and Research in Engineering Institute (COPPE) of the Federal University of Rio de Janeiro (UFRJ).*

Cláudio Silva *is a senior member of the technical staff in the Information Visualization Research Department at AT&T Labs–Research. His main research interests are in graphics, visualization, applied computational geometry, and high-performance computing. His current research focuses on architectures and algorithms for building scalable displays, rendering techniques for large data sets, 3D scanning, and algorithms for graphics hardware. He has a BS in mathematics from the Federal University of Ceará, Brazil. He has an MS and a PhD in computer science from the State University of New York at Stony Brook. He is an ACM, IEEE, and Eurographics member.*

Readers can contact Silva at AT&T Labs–Research, 180 Park Ave., Room D265, Florham Park, NJ 07932, email csilva@research.att.com.