Time-Critical Rendering of Irregular Grids

RICARDO FARIAS[1], JOSEPH S. B. MITCHELL[1], CLÁUDIO T. SILVA[2], BRIAN WYLIE[3]

[1]Department of Applied Mathematics and Statistics,
State University of New York at Stony Brook, Stony Brook, NY 11794-3600
{rfarias,jsbm}@ams.sunysb.edu
[2]AT&T Labs-Research, 180 Park Ave., PO Box 971, Florham Park, NJ 07932
csilva@research.att.com
[3]Sandia National Laboratories, PO Box 5800, MS 0318 Org 9215, Albuquerque, NM 87185
bnwylie@sandia.gov

Abstract. Many papers have presented rendering techniques and simplification ideas with the objective of speeding up image generation for irregular grid data sets. For large data sets, however, even the current fastest algorithms are known to require seconds to generate each image, making real-time analysis of such data sets very difficult, or even impossible, unless one has access to powerful and expensive computer hardware. In order to synthesize a system for handling very large data sets analysis, we have assembled algorithms for rendering, simplification and triangulation, and added to them some optimizations. We have made some improvements on one of the best current algorithms for rendering irregular grids, and added to it some simple approximation methods in both image and object space, resulting in a system that achieves high frame rates, even on slow computers without any specific graphic hardware. The algorithm adapts itself to the time budget it has available for each image generation, using hierarchical representations of the mesh for faster delivery of images when transformations are imposed to the data. When given additional time, the algorithm generates finer images, obtaining the precise final image if given sufficient time. We were able to obtain frame rates of the order of 5Hz for medium-sized data sets, which is about 20 times faster than previous rendering algorithms. With a trade-off between image accuracy and speed, similar frame rates can be achieved on different computers.

1 Introduction

Direct volume rendering methods are very useful tools in the visualization of scalar and vector fields. Techniques for volume rendering work primarily by modeling the volume as cloud-like cells composed of semi-transparent material that emits its own light, partially transmits light from other cells, and absorbs some incoming light [8]. In this paper, we address the problem of rendering (non-curvilinear) *irregular grids* (or *unstructured meshes*), having no implicit connectivity. Such structures are effective at representing *disparate* field data. Irregular grid data comes in several different formats; see, e.g., [11]. The introduction of new methods for generating high-quality adaptive meshes has made the general unstructured irregular grids a most important data type to be visualized.

Several papers have presented efficient methods to render irregular grids, including re-sampling techniques, ray-casting techniques [5, 10], sweep-based algorithms [13, 9], and projective methods [12, 3]. This large body of work, mostly done in the past decade, has dramatically increased the efficiency with which we can render irregular grids. In studying the computational complexity of these techniques, one finds a wide range of tradeoffs. Consider an irregular grid composed of n cells (b of the cells being in the boundary), and a given screen (image) of size k-by-k pixels. Projective techniques work by projecting, in visibility order, the polyhedral cells that comprise the mesh onto the image plane, and incrementally compositing the cell's color and opacity into the final image. Regardless of the screen resolution, an image is only *complete* once each of its n cells have been correctly depth-sorted and projected onto the screen. This does not take into consideration the fact that some of the cells may be too small to make a significant contribution by themselves. In contrast, in ray casting techniques, for each pixel potentially only the cells that actually intersect a ray through that pixel need to be touched. This effectively is the case for the technique proposed in [2], which, if r is the average number of cells intersecting a given ray, takes time $O(b + rk^2)$.

Our focus in this paper is on achieving real-time exploration, while possibly trading accuracy for speed. For real-time exploration, there are usually hard bounds on the overall rendering time T; e.g., for 30Hz, $T = \frac{1}{30}$ sec. Here, we explore tradeoffs necessary to design such *time-critical* irregular grid rendering systems. We utilize algorithms, based on extensions to existing techniques, which are scalable, allowing them to run on a wide range of machines. Our goal is to provide essentially the same level of interac-

tivity, regardless of the machine speed, while trading accuracy for speed in a consistent way.

2 The Rendering Algorithm

At the core of our time-critical volume rendering system is a variation of the ray-casting algorithm proposed in [2]. The algorithm is outlined as follows:

(1) We transform each of the n cells into screen space.

(2) For each pixel, we compute a (sorted) list containing the *boundary* cells that intersect the ray through the pixel center.

(3) For each pixel, we perform ray casting incrementally by computing cell intersections, one cell at a time, in front-to-back order along the ray, using a traversal of the cell adjacency information (similar to [5]).

This algorithm is very simple to implement, and quite fast in practice. Step *(1)* takes $O(n)$ time. Step *(2)* takes $O(\sum_{i,j} b_{i,j} \log b_{i,j})$ time, where $b_{i,j}$ is the number of boundary faces that project onto pixel (i, j). Step *(3)* is an *output-sensitive* step, depending linearly on the total number of ray-cell intersections. We note that a straightforward method for obtaining a time-critical performance is simply to sample a regular subimage (performing l-by-l ray casts instead of k-by-k, for $l < k$), then rescale to the full-size image (which can be efficiently performed in hardware using OpenGL). Another feature of this ray-casting method is the fact that the computation is "embarrassingly parallel" ([10]) in the shared-memory model, allowing for a readily implemented parallel version. In order to understand better how the rendering algorithm works and why our optimizations were necessary, we now discuss in more details steps *(2)* and *(3)*.

Boundary Projection. The algorithm projects each "visible" boundary face onto the screen, creating for each pixel in the projection a list with the intersected "visible" faces. (*Visible* faces are the ones whose outward normal makes an angle greater than $90\,\mathrm{deg}$ with the viewing direction.) Assuming that the boundary is generally not highly erratic, these lists should be short; in practice, we expect that the maximum boundary-list complexity ($\max_{i,j} b_{i,j}$) to be constant (i.e., $O(1)$). Thus, while we could sort the lists (each in time $O(b_{i,j} \log b_{i,j})$) as we create them, we do not bother to do so in practice; instead, each time we need to know the next visible boundary face that occurs along a ray, we simply step through the short list to find the one remaining with lowest z-coordinate.

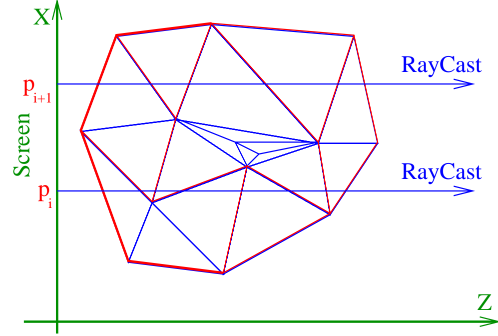

Figure 1: In this cross section of the volume to be visualized, the indices p_i and p_{i+1} represent two neighboring pixels through which two rays are cast. The visible boundary faces are highlighted, while the intersected faces are represented in red. Notice that in the middle of the mesh there is a set of cells, not stabbed by any of the rays, which do not need to be transformed.

Ray Casting. The *current face* is initialized to be the first boundary face intersected along the ray through a given pixel. We then compute where the ray exits the cell (on the *next face*) it just entered, and we compute the scalar value at both the entry point and exit point of the cell (using bilinear interpolation). We then compute the contribution of the current cell to the pixel's color and opacity, adding this to the running sum that represents the integration. If the *next face* is a boundary face, the computation continues only if the remaining list of visible boundary faces is nonempty; the *current face* is then advanced to the next visible boundary face in the list, and we continue along the ray. If the *next face* is an interior face, we determine the neighbor cell on the other side of the *next face*, set the *current face* to the *next face*, and compute the new *next face* based on where the ray exits the neighbor cell. (Each face has pointers to its neighboring cells. A boundary face has only one neighboring cell.) This "walking" along rays is simple, in principle; however, we note that special care must be taken for the degenerate situations when the ray hits an edge or a vertex of the mesh.

Optimizations. In order to use this algorithm in a time-critical setting, we modify step *(1)* to have running time dependent on image-quality. Instead of transforming all the vertices and faces in the mesh, we transform only the ones on the boundary. Then we project them on the screen, and from then on, we *incrementally* transform the interior faces (and their defining vertices) that are intersected by the rays cast. Once transformed, we tag them as such to avoid duplicate transformations. Depending on the image resolution, which determines the number of rays to be cast, and

on the viewing position, only a fraction of the data is actually touched. See Fig. 1. While there is some overhead in testing whether a primitive (vertex or cell) has already been transformed, we have found, in all data sets tested, that this optimization decreases the rendering time.

(Below, we discuss a parallelization of step *(2)*.)

3 Time-Critical Algorithm

Our goal is to achieve the highest frame rate possible by using a highly optimized rendering algorithm (discussed in the previous section), together with both image-space and object-space approximations. Beyond improvements in speed, these techniques will allow us to have greater control over the rendering procedure, giving us the flexibility to trade off between the image generation time and its accuracy. Such a trade-off will heavily depend on the machine's speed and the maximum acceptable simplification, to be controlled by the user.

In this section we discuss the image-space and object-space approximations we employ in our system. An alternative technique for the simplification of irregular grids is presented in [6].

Multi-Resolution Images

To generate multi-resolution images, we choose the simplest image-space simplification algorithm possible, to avoid spending time with both expensive computations and boundary constraints. We render the exact color for one pixel and duplicate it over a p-by-p pixels square, for a small value of p. (In our tests, we allow p to range from 1 to 9.) As will be shown later, the multi-resolution approximation has a narrow limit of its effectiveness for both speed-up and inversely for its error. For a 3-by-3 resolution, the gain in the rendering speed is high, while the visual impact is acceptable, still allowing the user to distinguish small details in the approximated image. Depending on the data set, larger values of p will only decrease the render time by a small amount, while resulting in very crude images.

Mesh Simplification Algorithm

To further improve the rendering time, we made use of object-space levels of detail, creating simplified meshes that are cheaper to render. We employ a method that is relatively simple, based on ignoring mesh connectivity, simplifying the point data (scalar values at mesh vertices), and then reconstructing an approximating mesh by re-triangulating the simplified point data. Special care is given to approximating the mesh boundary, while ensuring that the retriangulation of the interior point data does not induce artifacts from concavities in the boundary. We now elaborate on the steps of the algorithm: (1) interior mesh simplification; (2) boundary mesh simplification; (3) preserving concave boundary regions; (4) re-triangulating the simplified mesh; and elimination of transparent cells.

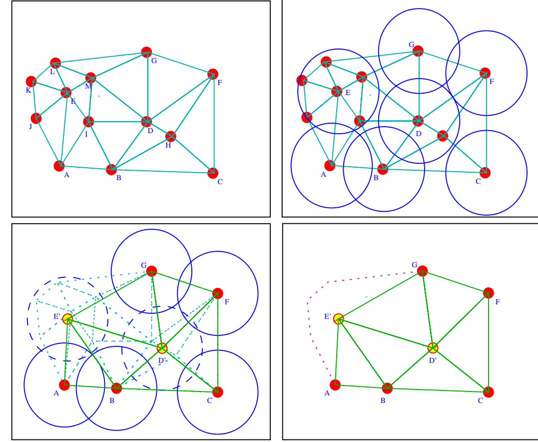

Figure 2: Loss of boundary information if the kd-tree is used to simplify the entire mesh. (a) Original mesh. (b) All points are sent to the kd-tree. (c) The averaged center for each region, or the result points for each region. (d) The final simplified mesh, where the dotted line represents the original contour detail that was lost.

Interior Mesh Simplification. We propose a simple and very fast algorithm for volumetric data simplification based on the use of a *kd-tree*. The criteria for internally arranging the vertices inside the kd-tree is as follows. A vertex inserted into the kd-tree will lie inside an existing region if its distance to the *center* of a region is smaller than a given value, called the *radius* of the region. (The radius is obtained from the user-specified simplification rate; see below.) If the current vertex does not lie inside any existing region, it will define a new region and its coordinates will determine the center. After inserting all vertices into the kd-tree, the new (simplified) mesh will have one vertex corresponding to each region of the final kd-tree. These vertices will have coordinates and scalar values equal to the averaged coordinates and scalar values of all vertices inserted into the region. See Fig. 2.

One way around the problem of loss of boundary information if the kd-tree is used to simplify the entire mesh is to send only the interior points to the kd-tree and at the end, merge the simplified set of points with the boundary points. See Fig. 3.

The algorithm computes the radius for the regions in the following way. It computes the number of vertices equivalent to the percentage of simplification input by the user (say S). Now it remains to be found the radius for the re-

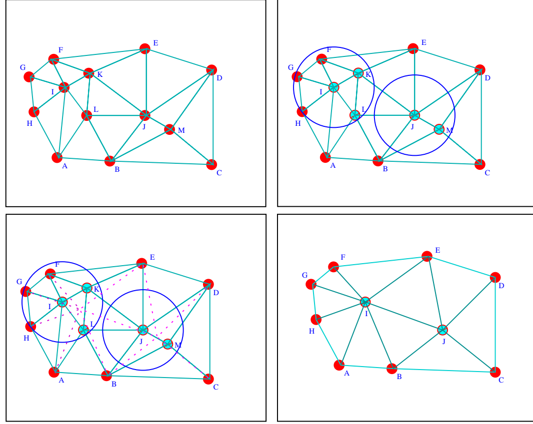

Figure 3: (a) The original mesh. (b) Only interior points are inserted into the kd-tree. Note that the sequence in which the points are sent to the kd-tree is the same regardless of whether or not we choose to preserve the boundary. That is true once the code scans the points and just skips the points labeled as boundary. (c) The averaged center for each region, or the result points for each region. (d) The final simplified mesh.

gions that will result in a number of regions approximately equal to the number S. The algorithm starts by computing the diagonal (D) of the data set and then generates the *kd-tree* with half this value, or $R = D/2$. If the resulting number is greater than S the algorithm updates R as $R = R + R/2$; otherwise, if the number of regions is less than S, the algorithm updates R as $R = R - R/2$ and regenerates the *kd-tree* for the new region radius. This procedure is repeated until the desired number S of regions is obtained. Note that this search for the radius is a binary search, requiring at most $O(\log(n))$ steps. Once we find a mesh with the desired number of points, the algorithm proceeds to simplify the boundary.

This algorithm can be used to simplify the whole mesh, but it can cause undesired loss of boundary information. We choose alternatively to use a surface simplification algorithm to obtain the approximation of the boundary of the data set. This is discussed in the next section.

Boundary Mesh Simplification. In our first approach, we did not consider surface simplifications. However, we noticed that the surfaces of some data sets can contain a significant fraction of the total number of the data set vertices, so we devised a way to make it available as an option. This flexibility is necessary because some data sets in our tests presented problems with the surface simplification even if the rate of simplification was very small, of the order of 10%.

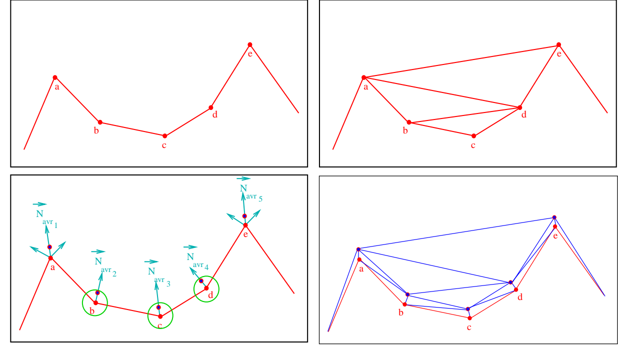

Figure 4: (a) The vertices from a to e are tagged as belonging to a concave region of the boundary. (b) This is the final triangulation if we naively triangulate the data set without taking precautions to preserve concave regions. (c) Ghost vertices are inserted in the direction of the averaged normal for each concave vertex. (d) After the triangulation we can retrieve the concavity of such regions by eliminating any face that contains at least one ghost vertex.

As we mentioned above, the simplification of the boundary requires a different approach because care must be taken to avoid destroying the form of the data set and still preserve its topology. Changes in the shape will be noticed more immediately then errors introduced to the color of the data set. In our current system we use the popular algorithm presented in [4], which allows not only surface simplification taking into account an expected error, but also allows the simplification to be performed over surface meshes whose vertices possess colors[1].

In the preprocessing phase, the algorithm tags all faces and vertices belonging to the boundary and make sure that each face has its vertices in counterclockwise order with respect to the exterior of the data set. (It is mandatory that the face normals are pointing outwards.) We then send these faces and vertices, with the desired rate of simplification, to *QSlim*[2].

All vertices of the new simplified mesh will be retrieved from the *kd-tree* and from *QSlim*. We expect that the vertices retrieved from the *kd-tree* will still be interior vertices. If any vertex retrieved from the *kd-tree* becomes exterior due to the boundary simplification, it will be eliminated in a future step; we note that this can cause *holes* in the simplified data set.

Preserving Concave Boundary Regions. After retrieving all vertices from the *kd-tree* and from *QSlim*, we have

[1]Remember that each vertex has a scalar value associated with it, requiring the simplification algorithm to be able to handle colored surfaces.

[2]QSlim code is available at Michael Garland's Home Page: graphics.cs.uiuc.edu/~garland/

Data Set	Vertices	Faces	Tetrahedra	Memory
SPX	2,896	27,252	12,936	11M
Blunt Fin	40,960	381,548	187,395	75M
Combustion	47,025	437,888	215,040	86M
Post	109,744	1,040,588	513,375	191M
Delta	211,680	2,032,084	1,005,675	370M

Table 1: Data sets, and their characteristics, used in our experiments. The memory usage is reported to render an image at resolution of 128^2.

to rebuild the mesh connectivity, which will be explained in the next section. We use a Delaunay triangulation for rebuilding the mesh connectivity.

This scheme works fine for convex data sets; however, for data sets that possess concave regions, further precautions must be taken. When the vertices are sent to the *qhull*[3] code, to generate the Delaunay triangulation, all concavities in the boundary disappear, since we obtain a triangulation of the convex hull of the points. In order to avoid losing this important information, in the preprocessing phase we identify all vertices belonging to boundary concavities. This identification is done by (a) marking all points belonging to the boundary; (b) using qhull to create the convex hull of the boundary; and (c) tagging the points that belong to the boundary but are not on the convex hull.

Each concave point will have an associated *ghost* vertex, very close to it, but just outside the boundary (in the direction of the averaged outwards normal). These ghost vertices will have a special associated scalar value that indicates to our ray casting function its transparency. See Fig. 4.

The distance between the *ghost* vertex and its related concave vertex is another parameter that can be controlled by the user (by default, we use 3% of the length of the data set's diagonal).

Re-triangulating the Simplified Mesh. After the simplification, the new set of points (which may contain up to $2b$ points in addition to the vertices in the simplified set) is sent to *qhull* [1], which returns a (Delaunay) tetrahedralization. The problem now is that any face that contains (at least) one ghost vertex must be considered to be transparent; there can be a significant number of such faces.

Eliminating Transparent Cells. At first, our code treated transparent faces individually, disregarding the contribution for any pair of faces in which at least one of them was transparent. This, however, was very inefficient and slowed the rendering function. Instead of thinking about transparent faces, one can think about transparent cells. Any tetrahedron which contains at least one transparent face, can be considered transparent and can be completely eliminated from the tetrahedra set. This criterion enormously reduces the number of tetrahedra in the resulting simplified mesh, restoring the shape of the original mesh very accurately, while simplifying our rendering code, since faces no longer require special treatment for been transparent.

[3] Qhull code is available at www.geom.umn.edu/software/download/qhull.html

Data Set	Resolution	Bunyk et al.	Optimized
	128^2	2s	1s
Blunt Fin	256^2	8s	4s
	512^2	27s	13s
	1024^2	104s	50s
	128^2	4s	2s
Combustion	256^2	10s	5s
	512^2	37s	14s
	1024^2	141s	52s
	128^2	5s	3s
Oxygen Post	256^2	19s	8s
	512^2	72s	27s
	1024^2	271s	100s
	128^2	4s	2s
Delta Wing	256^2	13s	6s
	512^2	43s	23s
	1024^2	157s	72s

Table 2: Our optimized version is consistently faster than the previous implementation.

4 Experimental Results

We report our results on an SGI machine (with a single 300MHZ MIPS R12000 processor and 512 Mbytes of memory). Table 1 lists the data sets we used in our experiments and measurements and all its relevant information, such as number of vertices, faces and cells. In Table 2 we compare the times obtained by the original algorithm [2] with the times we obtained with our optimized version. Our optimized version of Bunyk et al's algorithm, is, by itself, a factor of two improvement. Our changes to step *(1)*, utilizing a lazy transformation of the vertices, is shown to be very effective. See table 3.

The first approximation we use is to render the data set at a lower resolution and use *OpenGL* efficient interpolation to show the image in a larger resolution. In Fig. 5 we show the exact image of *Liquid Oxygen Post* at the resolution of 300^2 (Fig. 5(a)) and the image interpolated (Fig. 5(b)) from 128^2 to 300^2. The error, measured as the mean difference for all three color components of the *RGB* equal $(3.53\%, 2.43\%, 23.38\%)$. Note that even for a difference of 23% on the blue component, the interpolated image looks just a little bit brighter than the exact one.

Data Set	Resolution	Vertices Transformed	Cells Transformed
	128^2	25K	160K
Blunt Fin	256^2	30K	215K
	512^2	35K	270K
	1024^2	39K	319K
	128^2	47K	433K
Combustion	256^2	47K	437K
	512^2	47K	437K
	1024^2	47K	437K
	128^2	53K	315K
Oxygen Post	256^2	66K	431K
	512^2	83K	588K
	1024^2	96K	770K
	128^2	83K	450K
Delta Wing	256^2	114K	708K
	512^2	148K	1040K
	1024^2	169K	1398K

Table 3: Compare the number of vertices and cells transformed for each resolution; it becomes clear the source of the speed-up.

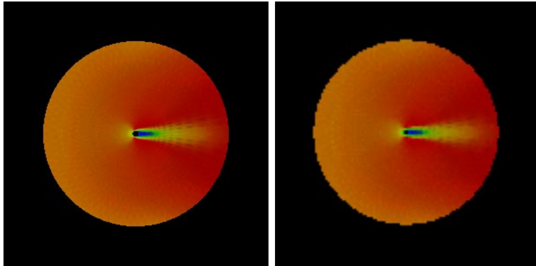

Figure 5: (a) Exact image of Liquid Oxygen Post rendered at 300^2 pixel. (b) Image rendered at 128^2 and interpolated to 300^2 pixels.

A two-time speed-up in running time is not enough, our goal is to achieve much faster frame-rates in a scalable framework. We basically explored two different ways to achieve this goal: multi-resolution image generation and hierarchical mesh simplification.

Multi-resolution Image Generation

Table 4 summarizes the rendering times to generate images for the four bigger data sets, for different image resolutions, obtained by running the code on a PC computer. We can see that the running time continues to drop as we effective increase the pixel size from 1-by-1 to 9-by-9. Unfortunately, the larger the pixel size, the smaller the speed up, and the improvement is negligible after 7-by-7. As the pixel size increases, the image quality decreases accordingly. In Fig. 6,

Pixel size	1^2	2^2	3^2	5^2	7^2	9^2
Blunt Fin	2.8s	1.6s	1.2s	0.7s	0.5s	0.4s
Combustion	5.1s	2.9s	2.0s	1.2s	0.9s	0.8s
Oxygen Post	11.9s	5.1s	4.1s	3.0s	1.9s	1.7s
Delta Wing	34s	13s	14s	14s	11s	4.4s

Table 4: The times are all in seconds. The top row has the effective pixel size used.

we show a typical set of images computed under these different approximations.

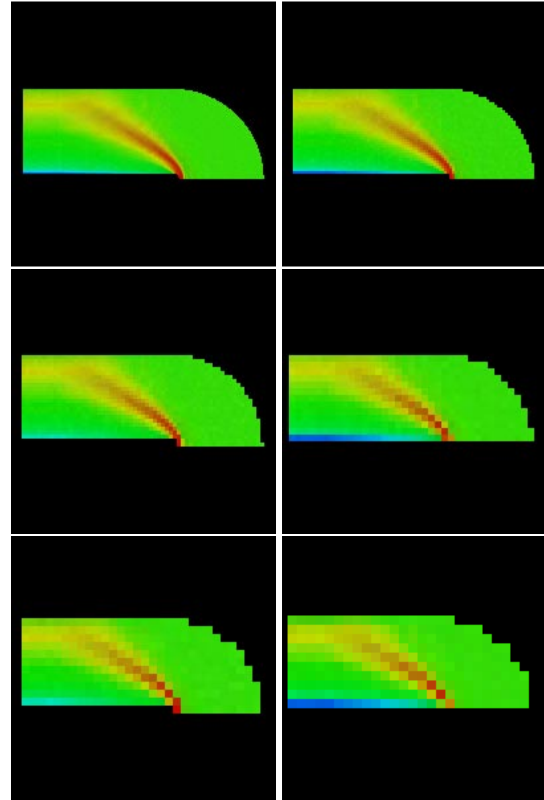

Figure 6: Renderings of the Blunt Fin under different image resolutions. (a) exact image (1-by-1 pixel size); (b) 2-by-2; (c) 3-by-3; (d) 5-by-5; (e) 7-by-7; (f) 9-by-9.

Mesh Simplification

By introducing mesh simplification in our algorithm, we ended up with a large number of possible combinations between all these approximation algorithms; we include only a sample of the results here. Taking into account the optimization we made to the original Bunyk et al's algorithm and by combining all approximations we introduced in this work, we were able to raise the frame rate from 0.35 Hz

(original data set at full resolution) to 3.5 Hz (with 90% of mesh simplification at 9^2 pixel resolution. A speed up factor of about 20 (remember that our optimized version of the render algorithm is twice as fast as Bunyk's original algorithm). To conclude this section we include in Fig. 7 some pictures of the *Oxygen Post* data set in full resolution and mesh simplification of 0%, 25%, 50%, 75% and 90%. Each row of image shows the image generated using each of these simplification. On the right column the image was generated on full pixel resolution, while on the right column we show the image for 9×9 pixels approximation. The error noticed on the edges the image is due to losses of detail of the boundary of the original data set. This can be avoided if one trades the simplification of the boundary that is performed separately from the interior simplification. For instance, see Fig. 8. The loss of information on the edges is due to the fact that for larger simplification rates the surface simplification algorithm generates meshes that contain some faces whose normals point inwards. Thus, depending on the data set characteristics and the desired rate of compression, one must trade between between opting or not for boundary simplification. Leaving the boundary unsimplified, even for 75% simplification of the interior points, the image looks almost the same. The errors measured between the image of *Oxygen Post* with simplification in (Fig. 7(a)) and its image for 75% of simplification only for the interior points in (Fig. 8(b)) led to an error, for *(r,g,b)* colors, respectively equal to (1.86%,1.18%,9.06%). Note that the error for the *blue* color is the only one to present a considerable value. But as the predominant color in the image depends on *red* and *green*, the error introduced by the simplification, led just to a slightly brighter image.

5 Conclusion

In this paper, we started exploring time-critical techniques for rendering irregular grids. We are primarily interested in developing techniques that are scalable, in the sense of being able to trade accuracy for rendering time, while achieving acceptable image quality in most reasonable cases. We have proposed a variation of the algorithm of [2]. Our technique differs from his in that it performs lazy transformations, it is able to generate images at multiple resolutions, and it works in parallel, using both an image-space and object-space technique. Our results are preliminary and expected to continue to improve. We have developed a simple GUI for our rendering code which allows the user to rotate moderately large data sets, including the Blunt Fin and Combustion Chamber, and even the Delta Wing. See Fig. 9.

Exploiting pixel coherence, we were able to obtain a frame rate of 3.5 Hz; this is to be compared with 3.5 seconds per frame to generate the exact image. Even though this result is far from the desirable 10-30 frames per second,

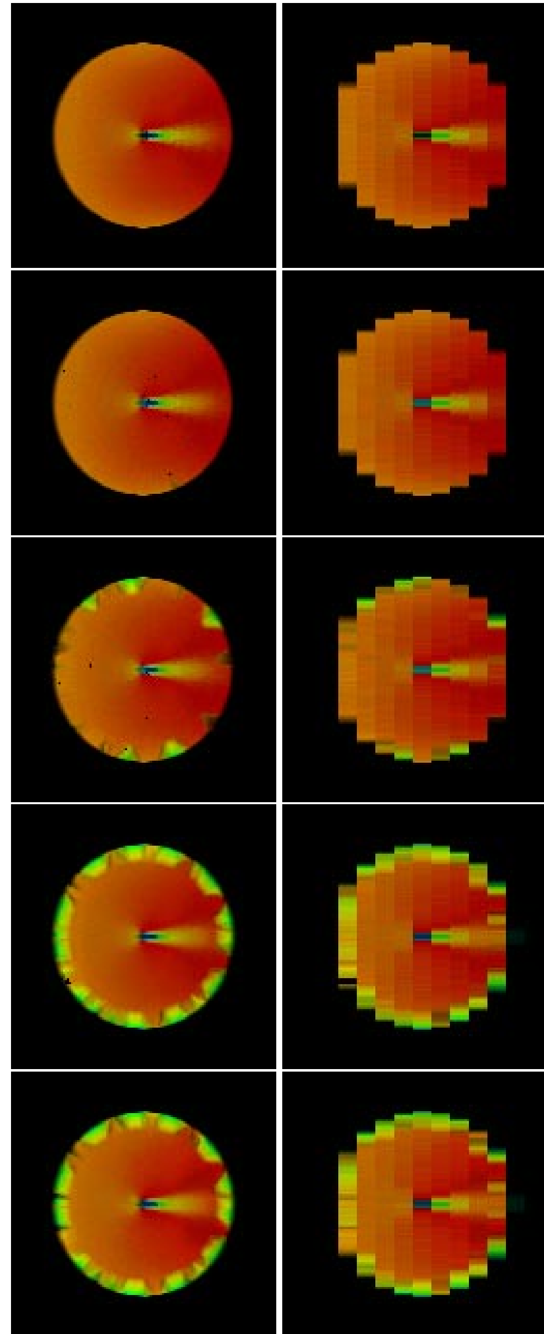

Figure 7: In the right column we show one pixel (full resolution) images for mesh simplification of 0%, 25%, 50%, 75% and 90%. In the left column we show for the same mesh simplification, but for 9×9 pixels image resolution.

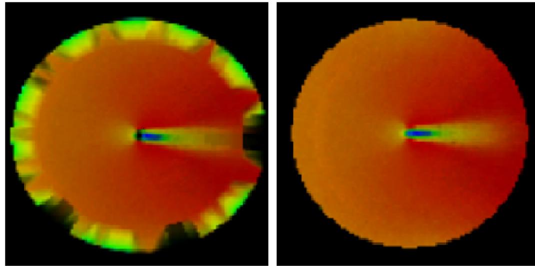

Figure 8: *Liquid Oxygen Post*. (a) Both interior and boundary points were simplified resulting in 26K points and 328K faces. (b) Only interior points were simplified, resulting in 37K points and 494K faces.

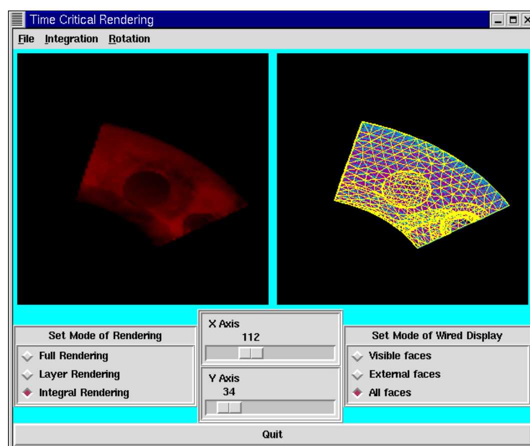

Figure 9: A screen shot of the GUI. On the left window is shown the rendered image and on the right window is shown the grid of the data set. The user can rotate up, down, left and right, and zoom in and out.

it already allows us to rotate the data sets while keeping a decent amount of detail.

Much work remains, particularly on the parallelization. We are working to apply coherence to accelerate the ray casting step, noting that neighboring pixels are likely to be similar. Hence, if there is little time to compute a ray (say, during a fast rotation), it is reasonable to assume a filtered down-sampled version of the image might be a visually accurate representation. We are currently exploiting extensions of these ideas further, in particular as it relates to time coherence. For instance, in an environment where the image is being computed at a lowered resolution, it might not be necessary to perform step 2 (boundary face projection) all the time. The set of visible boundary cells computed in the frame i may still be visible in frame $i+1$, albeit they might not intersect the ray emanating from the middle of the pixel from which it was originally visible. But, if the resulting image will be filtered anyway, simply a reprojection of those faces, and the ray integration from them might give a reasonably accurate picture. In fact, it might even be possible to reuse the intersection calculations.

Because of space limitations, we were forced to shorten the paper to a great extent. Relevant references and comparisons with previous works have been omitted and results abridged. A more comprehensive description of this work will be available in [7].

Acknowledgements

We thank Paul Bunyk (Stony Brook) for access to and help with his code. NASA has gracefully provided the Blunt Fin, Liquid Oxygen Post, and Delta Wing datasets. The Combustion Chamber dataset is from the Visualization Toolkit (Vtk).

This work was made possible with the generous support of Sandia National Labs and the Dept of Energy Mathematics, Information and Computer Science Office. R. Farias acknowledges partial support from CNPq-Brazil under a PhD fellowship. J. Mitchell acknowledges support from HRL Laboratories, the National Science Foundation (CCR-9732221), NASA Ames Research Center, Northrop-Grumman Corporation, Sandia National Labs, Seagull Technology, and Sun Microsystems.

References

[1] C. Bradford Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.

[2] P. Bunyk, A. Kaufman, and C. Silva. Simple, fast, and robust ray casting of irregular grids. In H. Hagen and H. Rodrian, editors, *Scientific Visualization*.

[3] J. Comba, J. Klosowski, N. Max, J. Mitchell, C. Silva, and P. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum*, 18(3):369–376, September 1999.

[4] M. Garland and P. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. *IEEE Visualization '98*, pages 263–270, October 1998.

[5] M. Garrity. Raytracing irregular volume data. In *Computer Graphics*, pages 35–40, November 1990.

[6] A. Van Gelder, V. Verma, and J. Wilhelms. Volume Decimation of Irregular Tetrahedral Grids. In Computer Graphics International, June 1999.

[7] R. Farias. Techniques for Rendering Unstructured Volumetric Grids. PhD thesis (in preparation), SUNY, Stony Brook, 2000.

[8] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[9] C. Silva and J. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), April–June 1997.

[10] S. Uselton. Volume rendering for computational fluid dynamics: Initial results. In *Tech Report RNR-91-026, Nasa Ames Research Center, 1991*.

[11] J. Wilhelms. Pursuing interactive visualization of irregular grids. In *Visual Computer, vol. 9, no. 8, 1993*.

[12] P. Williams. Visibility ordering meshed polyhedra. *ACM Transaction on Graphics*, 11(2):103–125, April 1992.

[13] R. Yagel, D. Reed, A. Law, P.-W. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *1996 Volume Visualization Symposium*, pages 55–62, 1996.