On-the-Fly Rendering Of Losslessly Compressed Irregular Volume Data

Chuan-kai Yang Tulika Mitra Tzi-cker Chiueh

State University of New York at Stony Brook [*]

Abstract

Very large irregular-grid data sets are represented as tetrahedral mesh and may incur significant disk I/O access overhead in the rendering process. An effective way to alleviate the disk I/O overhead associated with rendering large tetrahedral mesh is to reduce the I/O bandwidth requirement through compression. Existing tetrahedral mesh compression algorithms focus only on compression efficiency and cannot be readily integrated into the mesh rendering process, and thus demand that a compressed tetrahedral mesh be decompressed before it can be rendered into a 2D image. This paper presents an integrated tetrahedral mesh compression and rendering algorithm called *Gatun*, which allows compressed tetrahedral meshes to be rendered incrementally as they are being decompressed, thus forming an efficient irregular grid rendering pipeline. Both compression and rendering algorithms in *Gatun* exploit the same local connectivity information among adjacent tetrahedra, and thus can be tightly integrated into a unified implementation framework. Our tetrahedral compression algorithm is specifically designed to facilitate the integration with irregular grid renderer without any compromise in compression efficiency. A unique performance advantage of *Gatun* is its ability to reduce the run-time memory footprint requirement by releasing memory allocated to tetrahedra as early as possible. As a result, *Gatun* is able to decrease rendering time by a factor of 2 for very large tetrahedral mesh whose size exceeds the amount of physical memory. At the same time, the smaller working set and better access locality of *Gatun* improve the rendering performance by up to 30%, even when the input tetrahedral mesh is entirely memory-resident.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms; E.4 [Coding and Information Theory]: Data Compaction and Compression.

Keywords: Irregular Grids, Tetrahedral Compression, Volume Rendering

1 Introduction

Irregular-grid volumetric data sets represented as tetrahedral meshes are becoming ever more important in volume visualization research because of its natural fit with the way physical systems are modeled, i.e., representing more details only where it is actually needed. However, because of the lack of structure in the irregular grids, the coordinates of the vertices need to be explicitly represented, as well as the connectivity among vertices. As a result, irregular volume grids require relatively large storage space and may incur significant disk I/O overhead during the rendering process. One way to reduce this disk I/O performance overhead is to compress the irregular grid data sets, so that at least the initial data set loading time is reduced. Existing tetrahedral mesh compression algorithms focus mainly on improving the compression efficiency and are completely decoupled from the irregular volume rendering process. Because of this decoupling, a compressed tetrahedral mesh needs to be completely decompressed before it can be rendered into a 2D image. In contrast to this “store and forward” approach, this paper describes a “cut through” approach that integrates the decompression and rendering steps into a streamlined process, and thereby significantly cuts down both the run-time memory footprint size and the disk I/O bandwidth requirement.

The resulting irregular-grid volume rendering system, called *Gatun*, consists of a lossless tetrahedral mesh compression algorithm and an object space-based ray-casting rendering algorithm. The proposed tetrahedral mesh compression algorithm is a generalization of the one originally designed for triangle mesh [9] and provides compression efficiency comparable to [5]. Basically this algorithm can start from a seed tetrahedron or a seed face, and then proceeds in a “breadth first” manner until it covers all the tetrahedra. This breadth first traversal decision plays an important role in *Gatun*’s ability to streamline the decompression and rendering steps of compressed tetrahedral meshes.

The second component of *Gatun* is an irregular grid volume renderer that uses ray-casting, but is object space-based rather than image space-based. This object space orientation allows *Gatun* to incrementally build up the contribution of each tetrahedron to the intersecting rays as it is decompressed. It exploits the tetrahedral adjacency information, which is created and exploited by the tetrahedral mesh compression algorithm, to perform on-the-fly rendering of uncompressed tetrahedra without waiting for the whole data set to be completely decompressed. As a result, the start-up latency for rendering a compressed tetrahedral mesh is minimal. To support incremental rendering, the grid compression algorithm traverses the tetrahedral mesh from the external surface “inwards” towards the core of the data set. As new tetrahedra appear in the decompression process, their contributions to the rays that intersect with them are accumulated. This incremental rendering algorithm includes a new tetrahedron interpolation scheme and a new segment-based compositing formula, and implements unit sampling as in standard regular-grid volume rendering.

A key innovation of *Gatun* is its ability to determine when a tetrahedron is no longer needed and de-allocate the memory allocated to the tetrahedron as soon as it is done. As a result, *Gatun* reduces the maximal memory footprint to as low as one fifth of the entire data set and greatly improves the rendering performance of very large data sets by avoiding paging. In summary, *Gatun* streamlines decompression and rendering of losslessly compressed tetrahedral mesh and significantly reduces the run-time memory and disk bandwidth requirements. In certain cases, *Gatun* makes it possible to render certain data sets on a machine that were not even “runnable”

[*]Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400. Emails: {ckyang, mitra, chiueh}@cs.sunysb.edu

with standard irregular-grid volume renderers.

The rest of this paper is organized as follows. We review related works on tetrahedral mesh compression, as well as on irregular grid rendering in the next section. Section 3 presents the proposed tetrahedral mesh compression algorithm and how it is designed to facilitate integration with the rendering process. In Section 4, we describe the on-the-fly rendering algorithm that is tightly coupled with the mesh decompressor, and the current prototype implementation. Section 5 reports the performance measurements of *Gatun* prototype for six irregular-grid data sets with the number of tetrahedra ranging from $1.3K$ to $1M$. Section 6 summarizes the main research contribution of this paper.

2 Related Work

In the area of lossless tetrahedral mesh compression, there are two existing methods. The first one was proposed by Szymczak [13]. Their representation consists of a tetrahedron spanning tree string, which is obtained by recursively attaching tetrahedra to external faces starting from an arbitrary tetrahedron, and a folding string, which defines the incidence relations among the remaining external faces. Their method requires 7 bits per tetrahedron on an average to represent the topology. The second method, proposed by Gumhold [5] achieves by far the best compression efficiency for tetrahedral meshes. Their cut-border engine starts with the faces of an arbitrary tetrahedron and attempts to add tetrahedra to the external faces through different operations. They require 2.04 bits per tetrahedron on an average. The compression algorithm used in *Gatun* is similar in spirit to this approach. However, the operations we perform and the heuristics we use are much simpler and leads to faster implementation.

There were several early works on rendering of irregular grids. Wilhelms et. al. [15] applied a re-sampling technique to reduce the problem to rendering of traditional simpler regular rectilinear grids. However, when accommodating the finest details, the re-sampling overhead may be exceedingly high. Another attempt from Fruhauf [2] tried to apply the traditional algorithm, originally designed for rectilinear grids, to curvilinear grids by casting "curved" rays to fit the curvilinearity of the data set. However this approach can not be readily applied to the unstructured grids. Another school of algorithms is called "sweeping" algorithm, proposed originally by Giertsen [4] and later improved or modified by Silva and Yagel [11, 16]. While the algorithm in [16] needs a large amount of memory and high-end graphics engines, the approach in [11] is more memory efficient and reasonably fast assuming a more moderate graphics engine is available. However, Bunyk et. al. [1] presented a much faster algorithm based on the work from Garrity and Uselton [3, 14]. This algorithm, although still requires a great deal of memory for good performance, does provide a good starting point to derive the rendering algorithm used in *Gatun*. Another completely different approach to render irregular grids is through "projection." Here each tetrahedron is scan-converted and displayed on the image plane through a projection process that takes into account the depth information. The idea was proposed by Max et. al. [8] and similarly by Shirley et. al. [10]. The best property of this approach is that it can take advantage of the standard 3D graphics hardwares. But a major drawback is that it requires depth-sorting of the tetrahedra in order to generate correct composited image. The work from [7] uses a hybrid scheme by "splatting" the sample points obtained from the object base. However, it requires a re-sampling process and a sorting to get the correct result. Ma [6] and Silva [12] have tried to parallelize irregular volume rendering algorithms. In contrast, the target machines of this project are mostly PC workstations which in general are single-processor machines.

Gatun uses an object space-based ray-casting approach. Because of ray-casting, the resulting rendered image quality is high. Because of the object space architecture, rendering can be done incrementally and thus can be nicely tied with the mesh decompression process.

3 Tetrahedral Mesh Compression

3.1 Overview

Given a tetrahedral mesh, the proposed tetrahedral mesh compression algorithm starts with the boundary faces and then grows the surface inwards by visiting the tetrahedron that is paired with each boundary face. After all the tetrahedra that can be paired with the current surface are visited, the set of faces of these tetrahedra, that are not parts of the current surface, form a new surface. The algorithm then continues with this new surface to visit more tetrahedra, iteration by iteration, until it visits every tetrahedron in the mesh.

The input tetrahedral mesh consists of a *vertex array* containing the geometry information associated with the vertices and a *tetrahedron array* containing four vertex indices per tetrahedron. The output of the tetrahedral mesh compression algorithm also consists of two parts: a representation of the boundary surface and a representation of the geometry and connectivity of the tetrahedral mesh. To represent the boundary surface of a tetrahedral mesh efficiently, we use a triangle mesh compression algorithm described in [9], which is based on a similar breadth-first-traversal approach applied to triangle meshes. The geometry information associated with a vertex, such as coordinate and density value, appear in the compressed output **only once**, when either the first boundary face or the first tetrahedron containing that vertex is visited. The first time the geometry information of a vertex appears in the input, it is appended to the *vertex table*. Future references to this vertex can then be an index access to the vertex table. The order of the first appearance of each vertex in the input determines the index of the vertex in the vertex table, which is implicitly agreed upon by the mesh compressor and decompressor. In the proposed tetrahedral mesh compression algorithm, visiting a tetrahedron means denoting the fourth vertex that pairs with a triangle face on the current surface. If the fourth vertex of every visited tetrahedron is explicitly represented by its geometry information in its first appearance and as an index into the vertex table for all subsequent appearances, then it takes $N - V$ indices where N is the number of tetrahedra, V is the number of vertices, and the size of each vertex index is $log(V)$ bits. The challenge of the tetrahedral mesh compression algorithm design is to represent "fourth" vertices implicitly by exploiting connectivity information, so that fewer than $log(V)$ bits per tetrahedron is required.

Let us call the particular face with which to pair a "fourth" vertex as the *current face*. A face is a *partly-used face* if only one of its adjoining tetrahedra is not yet visited. A partly-used face is either a boundary face with no adjoining visited tetrahedron or a non-boundary face with one adjoining visited tetrahedron. This is because a boundary face has exactly one adjoining tetrahedron and a non-boundary face has two adjoining tetrahedra. Let T be the triangle mesh containing all the partly-used faces as the compression algorithm visits the tetrahedral mesh. The current face is an element of T. In most cases, the fourth vertex that pairs with the current face is a vertex that belongs to one of the edge-adjacent faces of the current face in T. Note that the triangle mesh T can be a non-manifold with the implication that an edge of the current face can have more than one adjacent faces. To denote the face with which the pairing vertex is associated, we order the faces edge-adjacent to the current face. First, we order the vertices of the current face according to their indices to the vertex table. Let this order be $(v1, v2, v3)$, and let $n1$, $n2$, $n3$ be the number of faces adjacent to the edges $(v1, v2)$, $(v2, v3)$, and $(v3, v1)$ respectively (excluding the current face). Then the faces adjacent to the current face are numbered

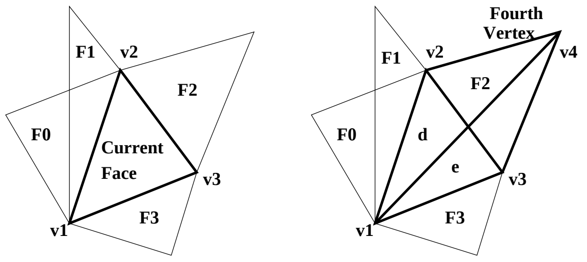

Figure 1: Enumeration of the fourth vertex

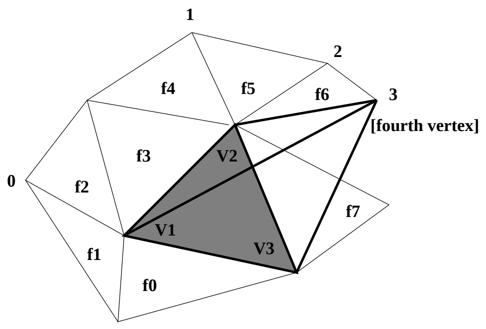

Figure 2: Enumeration of the fourth vertex as a vertex incident to the vertices of the current face. The shaded triangle is the current face. Only vertex 0,1,2 and 3 satisfy the condition to be a candidate for the fourth vertex and vertex 3 is the fourth vertex.

$0, 1, \ldots, n1 + n2 + n3 - 1$. This ordering of adjacent faces with respect to the current face is guaranteed to be unique and shared by the compressor and the decompressor. Given this ordering, specifying the neighboring face that contains the pairing vertex specifies the fourth vertex and visits the associated tetrahedron. Empirical results indicate that this value is $0, 1, 2$ in most of the cases. Figure 1 shows how the fourth vertex can be represented using the ordering discussed. The vertices are ordered as $v1$, $v2$, $v3$. The faces associated with the edges are ordered as $F0$, $F1$, $F2$ and $F3$ respectively. The fourth vertex belongs to face $F2$ and hence, in this case the specification for the pairing vertex is 2. This introduces two new faces, d and e.

If a pairing vertex cannot be represented by an edge-adjacent face, there are two possibilities: (1) the vertex appears for the first time in the input or (2) the vertex appeared earlier. In the first case, we can store the geometry information associated with the vertex into the vertex table. In the second case (which fortunately does not occur frequently), we first check if the paring vertex is incident to any of the current face's vertices, see Figure 2. If so, the vertex is specified according to its position in an ordered vertex list that includes all the vertices that neither belong to the current face nor to any edge-adjacent face of the current face. If the pairing vertex is not incident to any vertex of the current face, it is represented by its index value into the vertex table. Once the fourth vertex of a tetrahedron is determined, the three faces of the tetrahedron other than the current face are examined individually. For each of these faces, if it is marked partly-used, then all the adjoining tetrahedra of that face have been visited and the face can be deleted. If not, we mark the face as partly-used and put it in the next surface for the next iteration. Finally, the current face is deleted.

The compression algorithm as described above does not specify which face of a surface to start with. The choice of the current face has direct impact on compression efficiency since it determines where the fourth vertex falls in the neighborhood of a face. *Gatun* chooses to start with the order of the faces as determined by the triangle mesh traversal and completes the formation of tetrahedra for all of them before moving onto the next iteration. This design decision is based on the belief that peeling the tetrahedral mesh layer by layer will allow the rendering process to complete the processing of a tetrahedron as early as possible. Within an iteration, we choose to follow the generation order of the faces, and empirical results indicate that this heuristic works well.

Figure 3 illustrates how the visit of a tetrahedral mesh starts from the boundary surface. The leftmost figure shows the traversal order of the triangles as determined by the triangle mesh compression algorithm. The rest of the figure shows the formation of the tetrahedra. The solid lines indicate the tetrahedron enumerated for a particular boundary face. Notice that faces 1, 3, 5, 6, and 9 form new tetrahedra with explicitly represented vertices, whereas other tetrahedra are formed by pairing up with a neighboring face. The rightmost figure shows the new faces that will participate in the next iteration.

3.2 Encoding of the Compressed Mesh

The encoding for the boundary faces of a tetrahedral mesh is based on a triangle mesh compression algorithm described in [9]. The way each tetrahedron is represented depends on whether the fourth vertex is represented explicitly, using index reference, or using connectivity information. There are four different commands for the encoding:

1. `EXPLICIT` {`geometry information`}: This is for explicit representation of a vertex when it first appears in the input mesh.

2. `INDEX` {`index value`}: This corresponds to the case when the fourth vertex is represented as an index into the vertex table.

3. `FACE` {`order`}: This is used when the fourth vertex belongs to an incident face of an edge of the current face. The *order* is the order of the face among the adjacent faces as defined earlier.

4. `VERTEX` {`order`}: This is used when the fourth vertex belongs to a vertex-adjacent face of the current face and cannot be represented by face-adjacency. The *order* is the position of the fourth vertex in an ordered vertex list that includes all the vertices that neither belong to the current face nor to any edge-adjacent face of the current face. The *order* of the vertices is determined by the order of the faces they belong to in the partly-used face list described in subsection 3.3.

After each tetrahedron is represented using the above commands, we then apply standard string compression methods such as Huffman encoding on the resultant command sequences to achieve further compression.

3.3 Data Structure

Because *Gatun* streamlines the decompression and rendering steps, the decompression step must be sufficiently simple to reduce its impacts on run-time performance. The mesh decompressor maintains the following data structures:

1. **Vertex Table** contains the geometry information of the vertices and is accessed by both the decompressor and the renderer. The table can be re-created on the fly from the compressed mesh.

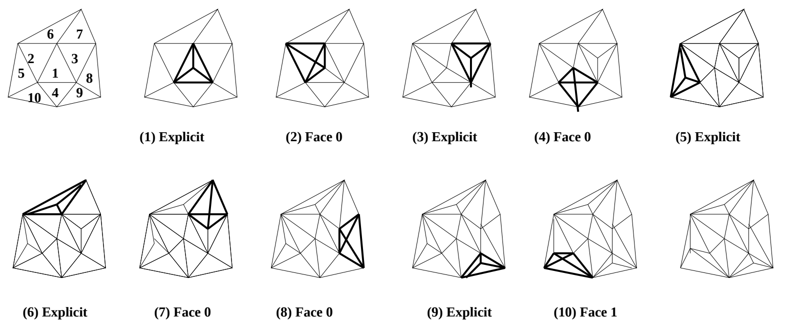

Figure 3: Tetrahedral mesh traversal

2. **Partly-Used Face List** is a list of partly-used faces for each vertex at any point of time. This list changes dynamically as faces are created and deleted. This list is used to determine the particular face/vertex given the `FACE {order}` and `VERTEX {order}` command.

3. **Surface Queues** are two queues of faces corresponding to the triangle meshes associated with the current iteration and the next iteration. The decompressor dequeues the first face from the current iteration queue and adds the newly generated faces to the next iteration queue.

4 On-the-Fly Compressed Mesh Rendering

4.1 Baseline Algorithm

The baseline tetrahedron mesh rendering algorithm is based on the work described in [1]. Although this algorithm was designed for parallel projection, it can be readily extended to support perspective projection.

The algorithm has a "view-independent" preprocessing step that identifies the adjacency information and some normal vector processing associated with each face of the input tetrahedral mesh. The adjacency information includes which vertices are used in which faces, and which faces are used in which tetrahedra.

Given a view angle, the baseline algorithm first identifies the intersections between the input mesh's boundary faces and all the cast rays. More concretely, the algorithm back-projects each boundary face to the image plane, finds a bounding box for the projected footprint, and tests every ray inside the bounding box to check whether it falls within the boundary face's projected footprint. An optimization to this step is to perform this intersection computation only for boundary faces whose projected footprint is not completely occluded. Once the set of intersecting boundary faces are identified, the algorithm sorts the intersection points with respect to a ray according to their distance to the origin of the ray on the image plane. Note that a ray may have multiple such intersection points if the input tetrahedral mesh is not convex.

The final phase of the baseline algorithm is a ray-casting process. For each ray, the algorithm retrieves the first boundary face that it intersects, then the face's associated tetrahedron, and then the next face of the same tetrahedron that this ray passes through. If this next face is a not a boundary face, then the "next" tetrahedron that shares this new face is retrieved. Through the same process, a ray can go through faces of neighboring tetrahedra until it hits another boundary face and exits the data set. The adjacency information computed by the pre-processing step plays an important role in allowing the algorithm to quickly identify which faces and tetrahedra to retrieve for a given ray at run time. As the baseline algorithm traverses the sample points on a cast ray, it performs interpolation of density values, maps density values into color and opacity values, and accumulates the contributions of sample values through standard compositing formulas. The sampling approach used here is unit-distance sampling.

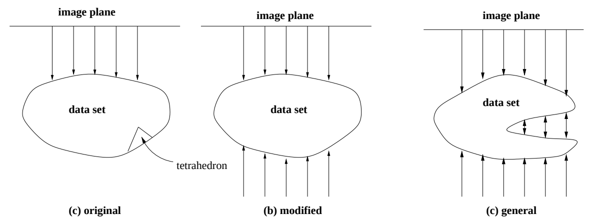

Figure 4: A 2D example of the original rays versus the modified rays.

4.2 The Gatun Approach

The goal of on-the-fly mesh rendering is to accumulate the contribution of each tetrahedron as it is output from the decompressor. This way there is no need to wait for the entire decompression process to complete and the memory allocated to the tetrahedra can be freed as soon as possible. *Gatun* uses an object-space ray-casting algorithm to achieve this goal. The fundamental problem is to identify the set of rays cast from the image plane that intersect with a given tetrahedron. Compared to the baseline algorithm, which assumes that an uncompressed tetrahedral mesh is already available, *Gatun* does not have all the adjacency information immediately available, and ray-casting process is dictated by the order of the tetrahedra that the decompressor outputs.

By employing an inward approach towards tetrahedral mesh compression, *Gatun* exploits the explicit representation of boundary surfaces to calculate the intersections between the boundary surfaces and the cast rays. Then a ray is decomposed into a set of one or multiple segments, each corresponding to a contiguous section of the ray that intersects with the input data volume, as shown in Figure 4. Moreover, each segment is further decomposed into two subsegments, one starting with the end closer to the image plane and having the original raycast direction, while the other starting with the end that is further away from the image plane and having the opposite of the original raycast direction, as shown in Figure 5.

Once the intersection points between the boundary faces and the rays are available, each ray is "attached" to its corresponding boundary face. Every time a tetrahedron is output from the decompressor, *Gatun* checks if it has some rays attached to any of its four faces, and if so, advances those rays as much as possible. To determine whether a ray has exhausted all the tetrahedra that it can

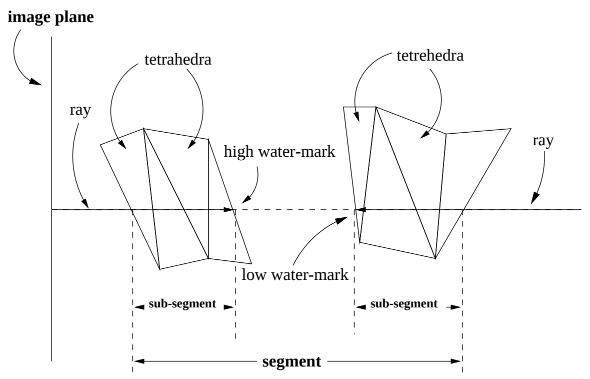

Figure 5: A 2D example of the segment and subsegments for a given ray.

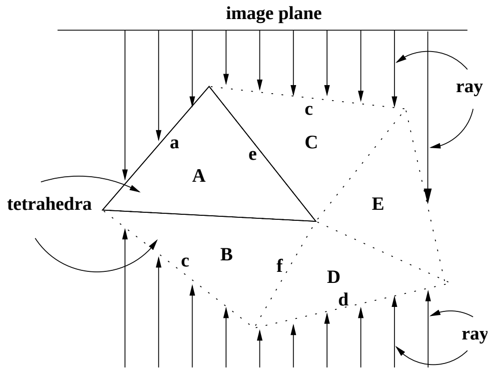

Figure 6: This figure shows a 2D case where a tetrahedra traversal order may lead to the late arrival of some rays.

possibly intersect, the renderer maintains a *water mark* that represents the current progress of each subsegment, and concludes that a segment is "done" when the water marks of these co-locating subsegments meet.

Because of the use of segments and subsegments, the compositing process associated with a ray is necessarily hierarchical: a standard front-to-back compositing algorithm is used within a subsegment, and a slightly modified front-to-back compositing algorithm is used between subsegments and between segments. The sample-to-sample front-to-back compositing formulas are:

$$C_{out} = C_{in} + (1 - O_{in})C_v O_v \quad (1)$$
$$O_{out} = O_{in} + (1 - O_{in})O_v \quad (2)$$

where C_{in} and O_{in} are the input color and opacity values, C_{out} and O_{out} are the output color and opacity values, C_v and O_v are the color and opacity values of the sample point v, which are results of applying desired color and transfer functions to the interpolated density values of v. It can be shown that to maintain the same sample-by-sample compositing semantics the subsegment-by-subsegment compositing formula is:

$$C_{total} = C_{front} + (1 - O_{front})C_{back} \quad (3)$$
$$O_{total} = O_{front} + (1 - O_{front})O_{back} \quad (4)$$

where C_{front} and O_{front} are the color and opacity values of the front subsegment, C_{back} and O_{back}, are the color and opacity values of the back subsegment, and C_{total} and O_{total} are the color and opacity values of the encompassing segment.

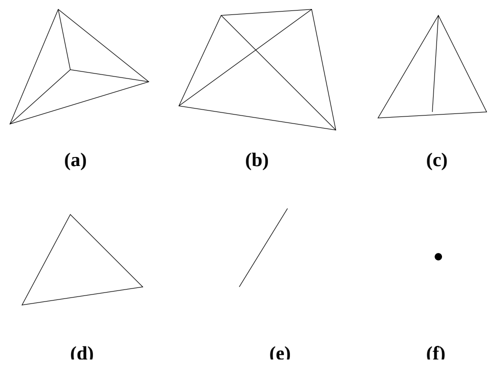

Figure 7: The possible geometrical projection shapes of a tetrahedron.

Because *Gatun* uses an object-space rendering algorithm that processes the rays that intersect with each tetrahedron, it is conceivable that the memory allocated for a tetrahedron can be freed as soon as it is done, thus significantly reducing the memory footprint requirement of the rendering process. However, this requires *Gatun* to maintain the following invariant: whenever a tetrahedron is processed, all the rays that can intersect with this tetrahedron have already been attached to its faces. This invariant does not hold in general, because when the decompressor outputs a tetrahedron, some of the rays that intersect with this tetrahedron may not arrive at its faces yet. For example, in Figure 6, if tetrahedron A is output first, then only the rays for its face a have arrived. Those rays that are to be attached to face e through face c have not come yet.

To address this problem, *Gatun* checks whether a tetrahedron is *ready* before processing it. A tetrahedron is ready if: *the projection of "processed" faces of this tetrahedron covers or forms a superset of the projected footprint of this tetrahedron.* This check is called the *readiness* check. A face is *processed* if all the rays that can be attached to this face have already been attached. By default faces are *unprocessed*. There are only two cases in which a face can become processed. First, this face is a boundary face, for which *Gatun* uses back projection to identify all the intersecting rays. Therefore by definition it is a processed face. Second, this face belongs to a tetrahedron that is ready. By definition, a tetrahedron is ready if all the rays that can be attached to it are already attached, therefore after the processing of the tetrahedron, all its constituent faces must be processed. After a tetrahedron is processed, the memory allocated to it is freed.

Gatun uses a *tetrahedron classification* scheme to classify the projection of a tetrahedron into a fixed number of cases in order to decide if the projected footprint of the processed faces of a tetrahedron is the same as the projection of the entire tetrahedron. There are 6 different possible shapes for a tetrahedron's projection image, as shown in Figure 7. The algorithm first tries to decide which case it is for a given tetrahedron. It first checks if there exists any pair of vertices whose projections are exactly the same. If yes, then the projection shape must be either case (d), (e) or (f) in Figure 7; otherwise, the projection shape is either case (a), (b) or (c). Let p_0, p_1, p_2 and p_3 represent the projected 2D points of the tetrahedron's four vertices on the image plane, and $\vec{ij}$ the vector pointing from p_i to p_j. For cases (a), (b) and (c), the algorithm performs four 2D cross products: $\vec{01} \times \vec{02}$, $\vec{01} \times \vec{03}$, $\vec{12} \times \vec{13}$ and $\vec{02} \times \vec{03}$, and classifies each tetrahedron according to the sign of these results: positive, negative or zero. For cases (d), (e), and (f), the algorithm performs three 2D cross products: $\vec{01} \times \vec{02}$, $\vec{01} \times \vec{03}$ and $\vec{02} \times \vec{03}$, and classifies them according to the signs of these cross products. If all of the three cross products are zero, then the projection of the tetrahedron must be a segment or a point, for which further distinction is not necessary because practically such a tetrahedron would not intersect rays. To speed up tetrahedron classification, we use a radix-3 table look-up scheme to map the results of cross products to the classification decision. On a Pentium II 300MHz machine, this algorithm takes less than 1.5 secs to classify $1M$ tetrahedra.

After a tetrahedron's projection is classified, *Gatun* performs the *readiness* check on the tetrahedron. For example, if an tetrahedron's projection is case (a) in Figure 7 with p_3 in the center, then there are two ways to "cover" this projection: $\triangle 012$ or $\triangle 013 \cup \triangle 123 \cup \triangle 203$, where $\triangle ijk$ is a face formed from p_i, p_j and p_k. So in this case, if $\triangle 012$ is already a processed face, then the entire tetrahedron is covered and therefore ready. Scenarios in which two processed faces are required to cover a tetrahedron's projection can be found from group (b), (c) and (d) in Figure 7.

It should be noted that a tetrahedron must be ready when more than two of its faces are processed, and a tetrahedron cannot be ready when none of its faces is processed. Therefore the readiness check only needs to be applied to a new tetrahedron when it has one or two processed faces. When the decompressor outputs a tetrahedron, if the renderer cannot conclude that it is ready, then the renderer puts this tetrahedron to the waiting field of all its unprocessed faces. If on the other hand the new tetrahedron is determined to be ready, then the renderer processes all the attached rays by accumulating the contributions from the tetrahedron to the rays from the processed faces, advancing these rays according to their casting direction, and eventually attaching these rays to other faces of this tetrahedron. As for unprocessed faces, they turn processed as more rays are attached to them during the processing of the tetrahedron.

After a tetrahedron is processed, the renderer examines each constituent face. If a face is previously processed, then it is freed because all its adjoining tetrahedra have been visited. If a face is unprocessed, *Gatun* changes the status of this face to processed, removes the tetrahedron in its waiting field, and performs the readiness check on the tetrahedron just removed to see if it, with the addition of this processed face, is ready or not. If the tetrahedron indeed turns ready, *Gatun* continues to process this tetrahedron. Whenever a tetrahedron is found to be ready, it is put into a tetrahedra working queue and *Gatun* returns control to the decompressor to output new tetrahedra only when this working queue is empty, which means either currently all the non-ready tetrahedra are waiting for some faces to become processed, or all the tetrahedra have been processed already. Because a non-ready tetrahedron is put into the waiting field of all its unprocessed faces, there is a bit in the tetrahedron data structure to indicate that a given tetrahedron is already put in the tetrahedra working queue and to avoid duplicated insertion from multiple faces.

With the tetrahedron classification scheme for efficient readiness check, *Gatun* is able to free a large percentage of tetrahedra as soon as their processing is done, thus greatly reducing the memory requirement at run time. Moreover, the smaller working set also leads to better overall performance in many cases, as will be shown in the performance evaluation section.

5 Performance Evaluation

The data sets we are using for testing purposes are shown in Table 1, ordered by the number of tetrahedra. While the first two data sets are unstructured grids, the remaining four are converted into tetrahedral grids from originally curvilinear grids. The intent to include the first two grids into our experiments, is to demonstrate that our algorithm can be applied to general unstructured grids.

5.1 Compression Efficiency

Table 2 presents the compression performance. The command sequence generated by the compression code is run through a Huffman encoder to arrive at the final compressed output. The first and second column indicate the cost in bytes to represent the boundary faces and the tetrahedra. This cost is only for topology. We ignore the issue of compressing geometry information in this work. The **Total Cost** column indicates the sum of boundary and tetrahedra representation. The fourth column shows the average number of bits required to encode the topology of a tetrahedra. Column five and six show the percentage savings in terms of bytes in topology and topology plus geometry, compared to the input representation. The input representation is a vertex array followed by the tetrahedra represented as four indices into the vertex array. Per-vertex geometry costs 16 bytes, 12 for coordinates and 4 for density. Finally, the last column shows the decompression speed in tetrahedron per second. This decompression speed is for topology only and does not include the disk I/O cost. On an average, our encoding requires 2.31 bits per tetrahedron and can be decompressed at the rate of 162K tetrahedra/sec. Our result is comparable to that of [5], which is the fastest tetrahedral compression algorithm developed so far and requires 2.04 bits on an average, and our encoding can be decompressed 1.5 times as fast.

Data set	Points	Tetrahedra	Faces	Bound. Faces
Spx	2896	12936	27252	2760
Fighter	13832	70125	143881	7262
Blunt	40960	187395	381548	13516
Combustion	47025	215040	437888	15616
Post	109744	513375	1040588	27676
Delta	211680	1005765	203208	41468

Table 1: Description of our testing data sets.

Dataset	0	1	2	>2
Spx	73.49	8.30	7.16	11.05
Fighter	75.71	7.75	7.48	9.06
Blunt	54.69	28.12	16.12	1.07
Combustion	54.78	28.09	15.88	1.25
Post	53.67	28.85	16.67	0.81
Delta	52.79	29.46	17.12	0.63

Table 3: Face order distribution

Table 3 represents the distribution of the `order` values in `Face {order}` command among four cases: 0, 1, 2, and greater than 2. Across all data sets, more than 88% of the time, the `order` value is either 0, 1, or 2. In fact, for all curvilinear data sets, the pairing vertex can be covered with these three values for 99% cases.

5.2 Rendering Performance

To evaluate the performance of *Gatun* , we first modified the baseline renderer (called *generic renderer* hereafter) so that it reads the compressed data set from the disk, uncompresses the entire data set, and then starts the rendering. To show that *Gatun* can improve the rendering performance in many cases with a much smaller memory footprint, we conduct experiments on a Pentium II 300MHz machine with 320M memory. With the help of LILO (Linux Loader), we can configure the available physical memory capacity to 160M and 80M as well.

Figures 9, summarizes the performance comparison of *Gatun* and the generic renderer. There are totally three different main memory settings: (1)320 MB, (2) 160 MB and (3) 80 MB and four image resolutions: (1) 128×128, (2) 256×256, (3) 512×512 and (4) 1024×1024. The total execution time of generic renderer

Dataset	Boundary Rep Cost	Tetra Rep Cost	Total Cost	Cost in Bits per Tetra	Topology Savings in %	Total Savings in %	Tetra Decoded per Sec
Spx	606	3983	4590	2.84	97.78	79.89	185K
Fighter	1587	19268	20855	2.38	98.14	81.97	184K
Blunt	2569	48488	51058	2.18	98.30	80.66	164K
Combustion	2971	55772	58743	2.18	98.29	80.65	146K
Post	5240	132964	138204	2.15	98.32	81.00	149K
Delta	7846	263279	271125	2.15	98.32	81.22	148K

Table 2: Compression performance. The cost is in bytes

and *Gatun* corresponding to four different image resolutions for all data sets are shown. When the system has 320MB of main memory, the entire working set can be memory resident for all the data sets. However, a system with 160MB main memory fails to hold the data set for *Liquid Oxygen Post* at resolution 1024×1024 and *Delta Wing* at all resolutions, and a system with 80MB main memory fails to hold the data sets for *Blunt-fin* and *Combustion Chamber* at resolution 1024×1024 and *Liquid Oxygen Post*, *Delta Wing* at all resolutions. The most important point to notice from this figure is that sometimes for very large data set such as *Delta Wing*, the generic renderer fails to complete the rendering process due to excessive memory requirement, for example, when the main memory size is 160MB. However, *Gatun* can complete the rendering for this data set due to its smaller memory footprint. In general, performance of *Gatun* is much better than the generic renderer when the working set of the generic renderer cannot fit in the main memory, resulting in considerable virtual memory paging. In general, *Gatun* achieves around 8.2% performance improvement for smaller resolution and around 40% for higher resolution. Even when the entire working set can be resident in the main memory, as in the case of 320MB memory setting, *Gatun* can improve the rendering performance by as much as 30% for large data set and high image resolution.

For some data sets and for small image resolutions, *Gatun* might perform worse than the generic renderer. The reasons behind this performance loss are the following. First, *Gatun* needs to attach rays from face to face, an overhead that does not appear in the baseline approach where each ray is dealt with one at a time. The generic approach therefore does not involve any intermediate bookkeeping cost. Second, *Gatun* needs to maintain the invariant that each tetrahedron, when being processed, have already had all the rays attached to its faces. This involves the readiness checks and possibly some queuing/dequeuing of tetrahedra to/from their faces. These overheads turn out to be non-negligible in some low resolution cases. The fundamental trade-off we face here is between performance and memory. De-allocating the memory for a tetrahedron requires performing the classification. In the generic renderer, it can be shown that for *Delta Wing* at 256×256 image resolution , only about one third of the tetrahedra would intersect the rays. Therefore in *Gatun* about two third of the effort to "classify" and thus releasing the allocated memory for tetrahedra will end up as redundant work. Thus the effort to reduce the memory usage results in more computation in this case, which in turn may hurt the performance. As can be seen from this figures, as the image resolution increases, and higher percentage of tetrahedra get "touched", higher percentage of the above effort will become useful, and therefore the advantage of using smaller memory begins to emerge. Another subtle point worth pointing out is the following. Since we are dealing with one tetrahedron at a time, it seems it should bear better memory locality than that of the generic renderer where the memory access pattern is considerably random, determined by the traversal of the ray along the data set. Figure 8 shows the peak memory requirement for *Gatun* and generic renderer corresponding to different image resolutions. In general, the peak memory requirement of *Gatun* is around 50-70% of the generic renderer. As the image resolution increases, the difference becomes more significant. However, at 512×512 and 1024×1024 image resolutions for the *Spx* data set, *Gatun* uses a little more peak memory than that in the generic renderer. This is because *Spx* has holes and far from being convex, and therefore the average number of segments per ray is slightly higher than in the other data sets. This results in extra storage and processing overhead for the segments. However, we should point out that most of the time the memory usages of *Gatun* are below the corresponding peak memory usage because of the early deallocation scheme we employ, whereas for generic renderer peak memory requirement in constant for the entire duration of the rendering process.

Figure 10 shows some images rendered from our system. We applied the linear color transfer function obtained from [1] and a linear opacity transfer function by mapping the highest density value to 1 and the lowest to 0. No directional shading is applied. The left on in Figure 10 is an image rendered from the blunt fin data set, while the middle one and the right one are from the combustion chamber and the fighter data set.

6 Conclusion

Very large irregular-grid volume data sets are becoming more or more popular in the scientific computing community because it is a natural fit for modeling the physical world. Rendering of irregular or unstructured grids could incur significant disk I/O overhead due to both initial data loading and virtual memory paging. Compression is an effective technique to reduce the memory requirements at run time. While existing rendering systems require the input data set to be completely uncompressed before rendering starts, the system described in this paper called *Gatun* supports a novel on-the-fly rendering method for losslessly compressed tetrahedra mesh. To further reduce the memory usage, *Gatun* features a unique "garbage detection" technique that can quickly recognize the tetrahedra that have been processed completely and thus can be safely discarded without affecting the correctness of rendering programs.

Performance measurements on the first *Gatun* prototype shows that the lossless compression algorithm used in *Gatun* achieves a compression efficiency of 5 times on an average, the run-time memory requirements are reduced by up to a factor of 7, and the average performance improvement is around 20%, when the data set size is larger than the host memory size. Even when the whole data set is memory resident, *Gatun*'s integrated rendering algorithm is up to 30% better than a "store and forward" approach because it exhibits better data access locality.

7 Acknowledgement

Thanks to Paul Bunyk, Claudio Silva, Lichan Hong and Kwan-liu Ma for providing some helpful discussions and the NASA testing data sets, which have been out of service for several months. This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, NSF EIA-9818342, NSF ANI-9814934, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, USENIX student research grants, as well as funding from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc.

References

[1] P. Bunyk, A. E. Kaufman, and C. T. Silva. Simple, Fast and Robust Ray Casting of Irregular Grids. Technical report, Center for Visual Computing, State University of New York at Stony Brook, 1997.

[2] T. Fruhauf. Raycasting of Nonregularly Structured Volume Data. *Computer Graphics Forum (Eurographics '94)*, 13(3):294–303, 1994.

[3] M. P. Garrity. Raytracing Irregular Volume Data. *Computer Graphics (San Diego Workshop on Volume Visualization*, 24:35–40, Nov 1990.

[4] C. Giertsen. Volume Visualization of Sparse Irregular Meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.

[5] S. Gumhold, S. Guthe, and W. Straβer. Tetrahedral Mesh Compression with the Cut-Border Machine. In *Proceedings of the 10th Annual IEEE Visualization Conference*, October 1999.

[6] K-L. Ma. Parallel Volume Rendering for Unstructured-Grid Data on Distributed Memory Machines. In *Proc. IEEE/ACM Parallel Rendering Symposium '95*, pages 23–30, 1995.

[7] X. Mao, L. Hong, and A. E. Kaufman. Splatting of Curvilinear Grids. In *IEEE Visualization '95*, pages 61–68, 1995.

[8] N. Max, P. Hanrahan, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3d Scalar Functions. *Computer Graphics (San Diego Workshop on Volume Visualization*, 24:27–33, Nov 1990.

[9] T. Mitra and T. Chiueh. A Breadth-First Approach to Efficient Mesh Traversal. In *Proceedings of the 13th ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 31–38, September 1998.

[10] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:63–70, Nov 1990.

[11] C. Silva. *Parallel Volume Rendering of Irregular Grids*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, 1996.

[12] C. Silva and J. Mitchell. The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), June 1997.

[13] A. Szymczak and J. Rossignac. Grow & Fold: Compression of Tetrahedral Meshes. In *Proceedings of the 5th ACM Symposium on Solid Modelling and Applications*, pages 54–64, June 1999.

[14] S. Uselton. Volume Rendering for Computational Fluid Dynamics: Initial Results. Technical Report RNR-91-026, Nasa Ames Research Center, 1991.

[15] J. Wilhelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri. Direct Volume Rendering of Curvilinear Volumes. *Computer Graphics (San Diego Workshop on Volume Visualization*, 24:41–47, Nov 1990.

[16] R. Yagel, D. Reed, A. Law, P-W. Shih, and N. Shareef. Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. *IEEE-ACM Volume Visualization Symposium*, pages 55–62, Nov 1996.

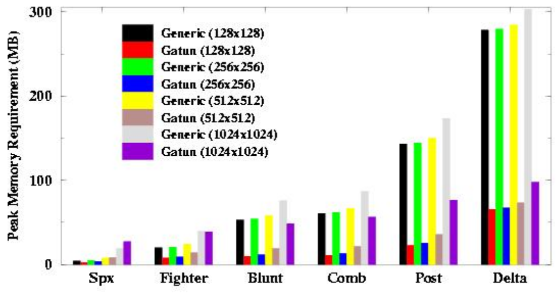

Figure 8: Peak memory requirement for different data sets.

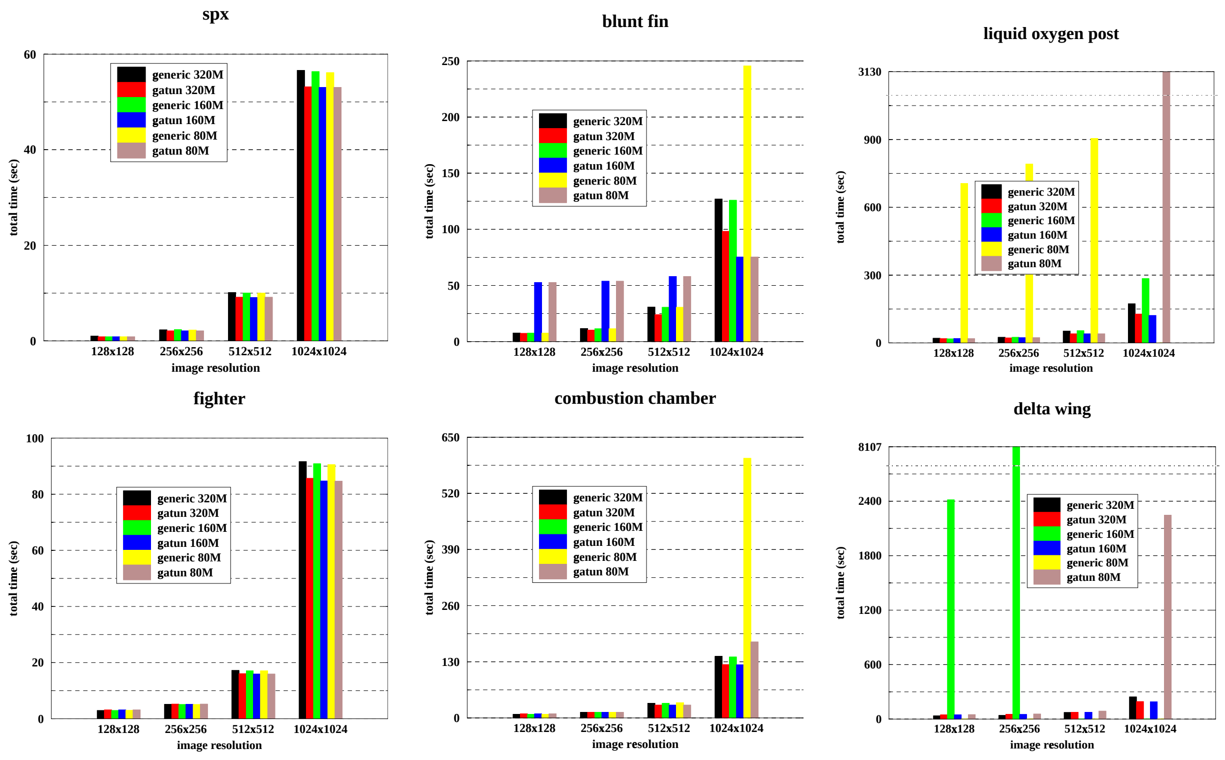

Figure 9: *Execution time for all memory configurations.*

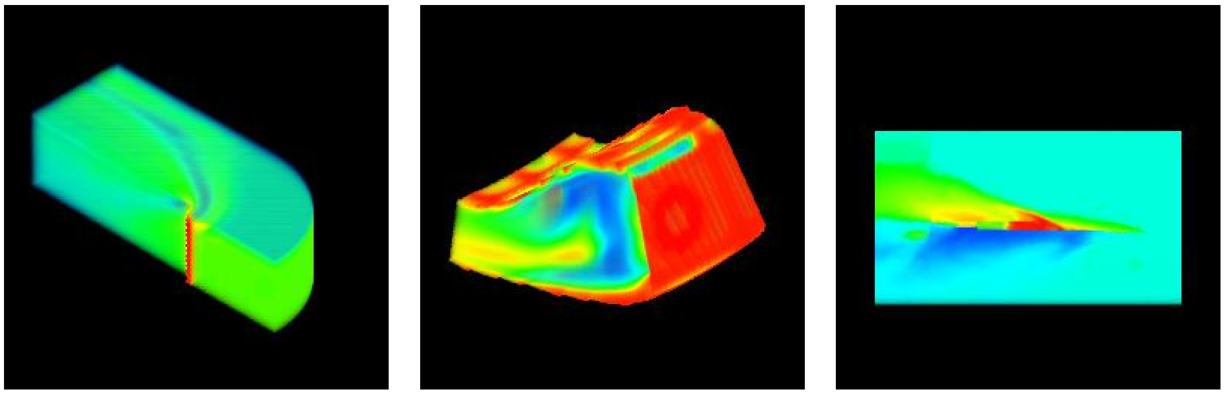

Figure 10: *Some rendered images from our system.*