An Integrated Pipeline of Decompression, Simplification and Rendering for Irregular Volume Data

Chuan-kai Yang Tzi-cker Chiueh

State University of New York at Stony Brook *

Abstract

Very large irregular-grid volume data sets are typically represented as tetrahedral mesh and require substantial disk I/O and rendering computation. One effective way to reduce this demanding resource requirement is compression. Previous research showed how rendering and decompression of a losslessly compressed irregular-grid data set can be integrated into a one-pass computation. This work advances the state of the art one step further by showing that a losslessly compressed irregular volume data set can be simplified while it is being decompressed and that simplification, decompression, and rendering can again be integrated into a pipeline that requires only a single pass through the data sets. Since simplification is a form of lossy compression, the on-the-fly volume simplification algorithm provides a powerful mechanism to dynamically create versions of a tetrahedral mesh at multiple resolution levels directly from its losslessly compressed representation, which also corresponds to the finest resolution level. In particular, an irregular-grid volume renderer can exploit this multi-resolution representation to maintain interactivity on a given hardware/software platform by automatically adjusting the amount of rendering computation that could be afforded, or performing so called *time-critical rendering*. The proposed tetrahedral mesh simplification algorithm and its integration with volume decompression and rendering has been successfully implemented in the *Gatun* system. Performance measurements on the *Gatun* prototype show that simplification only adds less than 5% of performance overhead on an average and with multi-resolution pre-simplification the end-to-end rendering delay indeed decreases in an approximately linear fashion with respect to the simplification ratio.

1 Introduction

An irregular-grid volumetric data set is typically represented as a tetrahedral mesh, which consists of per-vertex geometry and data density information and connectivity information among the vertices. To reduce the storage requirement and the run-time disk access overhead, lossless compression is an effective technique that is more acceptable to the user community. A previous paper [YMC00] showed that it is possible to pipeline the decompression of a losslessly compressed tetrahedral mesh and the rendering of the resulting tetrahedra, thus significantly reducing the memory footprint requirements of rendering tasks whose target data sets are close to or larger than the rendering machine's physical memory. This performance advantage comes from the fact that volume decompression and volume rendering are integrated into a one-pass computation.

To further reduce the rendering delay associated with very large irregular volume data sets, one needs to trade accuracy for performance. Given a rendering hardware and a pre-defined level of interactivity, the goal is to develop a rendering algorithm that can meet the performance requirements while maintaining the highest rendering image quality. The enabling technology that allows making such a tradeoff is tetrahedral mesh simplification, or lossy compression of tetrahedral mesh. Unfortunately, most existing mesh simplification algorithms are implemented as a stand-alone tool rather than as a tightly integrated component of an irregular-grid volume renderer, thus limiting their utility as a dynamic performance adaptation mechanism.

This paper describes a novel mesh simplification algorithm that fits nicely into a decompression-driven volume renderer, thus making it possible to integrate volume decompression, simplification, and rendering into a seamless pipeline that requires only one pass through the compressed input data set. Because of this streamlined structure, the volume renderer can dynamically adjust the rendering accuracy to match user-specified interactivity requirement and/or computation resource availability. Furthermore, simplification and lossless compression together make it possible to represent a tetrahedral mesh as a multi-resolution hierarchy, with the losslessly compressed version corresponding to the finest resolution. Finally, as in the integrated volume decompresser/renderer [YMC00], the proposed integrated decompression/simplification/rendering engine greatly reduces the run-time disk access overhead and peak memory usage for rendering of very large tetrahedral meshes.

We have successfully implemented the integrated decompression/simplification/rendering pipeline in the *Gatun* system. Performance measurements on the *Gatun* prototype show that the proposed mesh simplification algorithm only adds less than 5% overhead on an average compared to an integrated volume decompresser/renderer, thus demonstrating the efficiency of the proposal's implementation simplicity. On the other hand, the proposed mesh simplification can effectively reduce the total rendering time by a factor of up to 2 with a RMSE (root mean square error) as small as 1.32 (on a scale of $0 - 255$) when 90% of the input mesh is simplified away.

The rest of this paper is organized as follows. We review previous related work on tetrahedral mesh compression, simplification, and rendering in Section 2. Section 3 briefly describes the previously proposed integrated tetrahedral mesh decompression and rendering pipeline to set the stage for the discussion of fitting mesh simplification into such a pipeline. In Section 4, we describe the on-the-fly mesh simplification algorithm that is tightly integrated with the mesh decompresser and renderer. In Section 5, we demonstrate how to apply the simplification-capable integrated renderer to *time-critical rendering*. Section 6 reports the performance measurements of the proposed integrated mesh decompression/simplification/rendering pipeline on the *Gatun* prototype for six irregular-grid data sets with the number of tetrahedra ranging from $1.3K$ to $1.2M$. Section 7 concludes this paper by summarizing the main research contribution of this work.

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

2 Related Work

There are a few works on polygonal surface mesh simplification. These methods simplify a 3D surface model by merging neighboring polygons into one if they are "flat" enough [SZL92, HH92, KT96], by re-tiling the triangular mesh through inserting vertices at places with maximal curvature and iteratively remove the old vertices [Tur92], by clustering vertices to obtain a possibly topologically different mesh [RB93], by performing a wavelet-based multiresolution analysis of the input mesh and re-meshing/re-sampling the mesh if necessary [EDD+95], by optimizing the input mesh through addition and/or removal of vertices or collapsing/swapping edges to minimize a certain pre-defined energy function that depends on the mesh [HDD+93], and by simply collapsing edges to form progressive meshes [Hop96]. The last algorithm, in particular, provides a general framework for building view-independent multi-resolution representation of a 3D surface. It uses an error metric that is defined for each simplification primitive (vertex, triangle, tetrahedron, etc.). At run time, the algorithm first computes the error metric for each simplification primitive that is applicable to the input triangular mesh, and then builds a priority queue for these primitives according to their error metrics. The simplification process starts with the application of the primitive with the smallest error metric, collapses edges or patches the resulting mesh, recomputes the error metric for all the simplification primitives that are affected, and proceeds with the primitive with the smallest error metric, etc.

There is another line of research [Hop97, XV96, CVM+96] in this area that focuses on view-dependent simplification, which is able to adapt the extent of simplification for each geometric primitive to view angles and other run-time parameters, and some can also provide error control mechanism. In general this class of algorithms first statically build a hierarchical data structure for the input 3D model and at run time use the viewing angle information to compute simplification parameters and efficiently locate the the necessary part of the hierarchical data structure that is to be used for rendering. Some extended this line of work to support out-of-core view-dependent simplification as well [ESC00].

Volume simplification, on the other hand, attracts relatively less research effort. Cignoni et al. [CFM+97] proposed decimating vertices iteratively and the resultant holes are re-tetrahedralized locally. Renze [RO96] generalized the idea of progressive surface mesh to perform volume decimation for unstructured grids. Staadt [OGS98] proposed techniques for progressive tetrahedralization that tries to avoid some artifacts such as self-intersections due to an improper simplification. Trotts [THJW98] applied a piece-wise linear spline function that is defined over the scalar values over the input tetrahedral mesh as the basis of the error metric for volume simplification. The algorithm associates each tetrahedron with such an error metric and favors the removal of those tetrahedra that causes the least change in the spline function. Removing a tetrahedron is carried out by a sequence of edge collapses. Gelder [GVW99] proposed a less computation-intensive approach for simplification, which aims to minimize the density or "mass" change due to an edge collapse. Boundary vertices come with extra geometry-related error metric, in addition to the so called "data-based" error metric required of internal vertices. The on-the-fly simplification scheme presented in this paper is based on this work.

In the area of lossless tetrahedral mesh compression, there are at least two existing methods. The first one was proposed by Szymczak [SR99]. 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 [GGS99] 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. One of the variants of the compression algorithm used in *Gatun* is similar in spirit to this approach. However, the number of possible cases to consider are much less (4 cases instead of 10), which therefore leads to an easier implementation.

There were several early works on rendering of irregular grids. Wilhelms et. al. [WCA+90] 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 [Fru94] tried to apply the traditional algorithm, originally designed for rectilinear grids, to curvilinear grids by casting rays in the computation domain, or equivalently casting "curved" rays the the spatial domain 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 [Gie92] and later improved or modified by Silva and Yagel [Sil96, YRL+96]. While the algorithm in [YRL+96] needs a large amount of memory and high-end graphics engines, the approaches in [Sil96, FMS00] are more memory efficient and reasonably fast assuming a more moderate graphics engine is available. However, the algorithm proposed by Bunyk et. al. [BKS97] is a simpler and sometimes faster approach based on the work from Garrity, Uselton and Hong [Gar90, Use91, HK99]. 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*.

Probably the work from Farias et al. [RMSW00] is by far the most similar one to us. However, in their work, they applied basically the vertex clustering idea from [RB93] where the topology of a mesh may be changed. *Gatun* applies a simplification approach which can not only preserve the topology of a mesh but also makes it easier to be pipelined with the decompression process. In addition, *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 also be done incrementally and thus can be nicely tied with the mesh decompression process.

3 Integrated Tetrahedral Mesh Compression and Rendering

3.1 Lossless Mesh Compression

Given a tetrahedral mesh, *Gatun*'s tetrahedral mesh compression algorithm starts with the boundary surface as the current surface, and then grows the current surface inwards by enumerating each tetrahedron that is paired with one of the current surface's faces. 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 the current surface for the next iteration. The algorithm then continues with this new surface to visit more tetrahedra, iteration by iteration, until it visits every tetrahedron in the input 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. The boundary surface of a tetrahedral mesh is encoded by a tri-

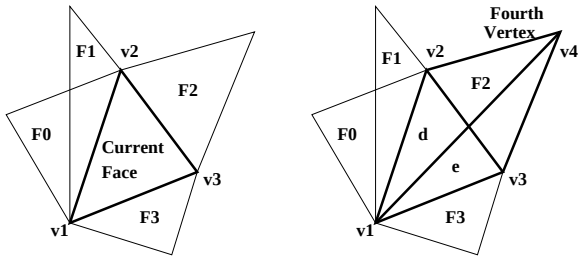

Figure 1: *The fourth vertex of a tetrahedron belongs to a face that is adjacent to the current face*

angle mesh compression algorithm described in [MC98], 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 data 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. A tetrahedron is represented as a vertex (the fourth vertex) that pairs with a triangle face on the current surface. The fourth vertex can be represented either explicitly as a new vertex with its associated geometry and density information, or implicitly as an index to a buffer of previously appeared vertices or to the vertex table.

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 one of its adjoining tetrahedra is not yet visited. Let T be the triangle mesh containing all the partly-used faces as the tetrahedral mesh is being compressed. 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 non-manifold, i.e., an edge of the current face can have more than one adjacent faces. To denote the faces associated with the pairing vertex (fourth vertex), the faces that are edge-adjacent to the current face are ordered, and the vertices of the current face are ordered 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 $0, 1, \ldots, n1 + n2 + n3 - 1$. The ordering among faces adjacent to the same edge is determined by the order in which they are traversed. Thus the ordering of adjacent faces with respect to the current face is guaranteed to be unique and shared by the compressor and the decompresser. Given this ordering, one can uniquely identify the neighboring face that contains the pairing vertex if it exists. Empirical results indicate that the index value of the "pairing face" is $0, 1, 2$ in most of the cases. Figure 1 shows how the fourth vertex can be represented using the ordering discussed above. 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 the face $F2$ and hence, in this case the specification for the pairing vertex is 2. This introduces two new faces, d ($v1$, $v2$, $v4$) and e ($v1$, $v3$, $v4$).

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, the geometry information associated with the vertex is saved to the vertex table. In the second case, we check if the pairing vertex belongs to a face adjacent to a vertex of the current face, as shown in Figure 2. If so, the vertex is specified according to its position in an ordered vertex list that includes all the vertices that belong to faces adjacent to vertices (but not edges) of the current face. The order of

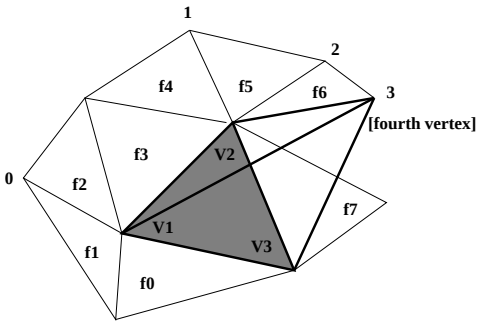

Figure 2: *The fourth vertex belongs to a face adjacent to a vertex (but not edge) of the current face. The shaded triangle is the current face and $f0 \cdots f7$ are the adjacent faces in the current triangle mesh. Only vertices 0, 1, 2 and 3 belong to vertex adjacent faces and vertex 3 is the fourth vertex.*

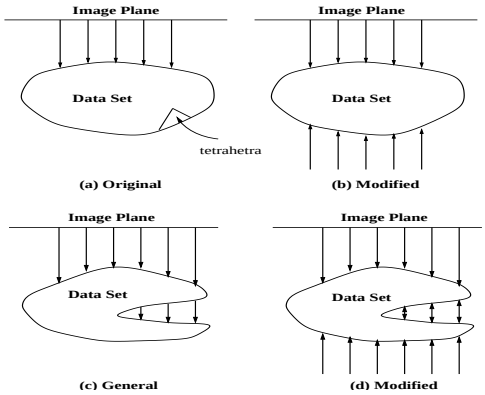

Figure 3: *A 2D example of the original rays versus the modified rays. In (a), all the rays are shooting in one direction, while in (b), for the same data set, rays are duplicated in the opposite directions; (c) represents a more general case where the data set has multiple segments for some rays, while (d) shows our modified version of rays for the same data set.*

the vertices is determined by the order in which the vertex adjacent faces they belong to are visited during compression/decompression. 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, the face is marked as partly-used and included as part of the mesh for the next iteration. Finally, the current face is deleted.

3.2 Decompression-Driven Mesh Rendering

The goal of decompresser-driven mesh rendering is to incorporate the contribution of each tetrahedron into the final image as soon as it is output from the decompresser. This way there is no need to wait for the entire decompression process to complete before rendering starts and the memory allocated to the tetrahedra can be freed as early as possible. *Gatun* uses an object-space ray-casting algorithm to achieve this goal. The rendering algorithm is basically a ray casting algorithm that first attaches all the rays cast from the image plane to the input tetrahedral mesh's boundary surface, and attempts to advance each ray inward as the decompresser enumer-

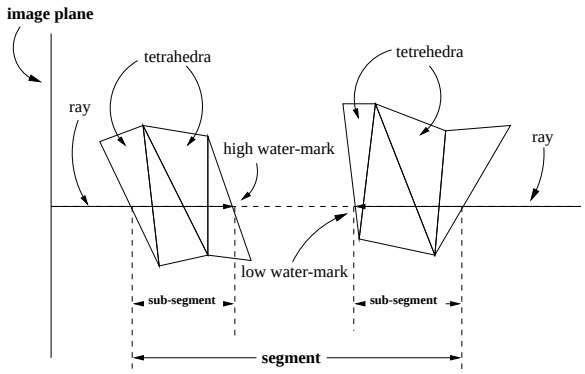

Figure 4: *A 2D example of multiple segments of a cast ray and the two subsegments of each ray segment.*

ates additional tetrahedra. The fundamental design issue is how to identify the set of rays cast that intersect with a given tetrahedron.

Gatun exploits the explicit representation of a tetrahedral mesh's boundary surface to calculate the intersections between the boundary faces 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 3. Note that there are two instances for each ray one for each direction. In addition, each segment is further decomposed into two subsegments, one starting with the end closer to the image plane and having the original ray-cast direction, while the other starting with the end that is further away from the image plane and having the opposite of the original ray-cast direction, as shown in Figure 4. Due to the spirally-inward decompression order, a bi-directional rendering could help to render a decompressed tetrahedron as soon as possible.

Once the intersection points between the boundary faces and the rays are calculated, each ray is "attached" to its corresponding boundary face. Every time a tetrahedron is output from the decompresser, *Gatun* checks if there are any rays attached to any of its four faces, and if so, advances those rays as much as possible. Because the decompresser traverses the tetrahedral mesh in a layer by layer fashion, each cast ray can continue to advance until it samples its corresponding path through the mesh. To determine whether a cast ray has exhausted all the tetrahedra that it can 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 the results of applying the 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 are:

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

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

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.

4 On-the-Fly Tetrahedral Mesh Simplification

To incorporate mesh simplification in the integrated rendering/decompression pipeline described in the previous section, *Gatun* first statically computes a priority list of volume simplification operations, and at run time performs a selective subset of these vertex merge operations based on user requirements and/or available computation resources. The simplification step is inserted between decompression and rendering in a way that is largely independent of the internal working of the renderer and decompresser.

4.1 Static Simplification Algorithm

Gatun makes the following assumptions on the volume simplification algorithm:

- Vertex merge[1] is the only tetrahedral mesh simplification primitive used in the algorithm, and each vertex merge operation is denoted as $V_i \rightarrow V_j$, which means that V_i is merged into V_j, and
- The algorithm can statically compute an error metric for all possible vertex merge operations, and derive a priority order among them based on this error metric.

At run time, *Gatun* simply uses the global priority to determine which vertices and thus which tetrahedral no longer exist after a certain amount of simplification. The volume decimation algorithm [GVW99] described in this subsection is one example of such volume simplification algorithm. *Gatun* can inter-operate with any other simplification algorithms as long as they satisfy the above assumptions. We re-implement [GVW99]'s algorithm for our testing volume simplification algorithm.

This volume decimation algorithm considers two types of errors introduced by simplification: Density-related ($Error_{density}$) and Geometry-related ($Error_{geometry}$). Each vertex of a tetrahedral mesh-based volume data set has an associated data density value. One can compute an average data density value for a tetrahedron based on the data densities associated with its four vertices. Multiplying a tetrahedron's average density by its volume results in its mass. The Density error associated with merging two vertices is the sum of all changes in the mass of all tetrahedra that are affected by this merge operation. As a constraint, a vertex merge operation that causes any affected tetrahedron's volume to become negative after merging is disallowed.

When at least one of the vertices to be merged is a boundary vertex, the volume decimation algorithm considers an additional Geometry error. Each new face that is generated as a result of a vertex merge operation is first paired with all the old faces that share at least one edge with the new face. Then the difference in area between a new face and each of its paired old faces is computed. Each such area difference is then weighted by the ratio between the associated old face's area and the area sum of all affected old faces. The sum of these weighted area differences between new faces and their paired old faces is the Geometry error. In addition to the negative volume constraint, there is a similar negative area constraint. Moreover, boundary vertices can be merged only into

[1] In this paper, the term *vertex merge* is interchangeable with *edge collapsing*.

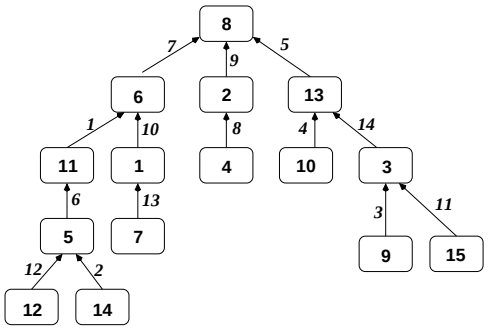

Figure 5: *An example merge tree that shows all the vertex merge operations, as represented as edges, that are being considered, and their relative priorities, as indicated by the weights on the edges.*

boundary vertices to preserve the surface shape of the tetrahedral mesh.

The final error metric associated with a vertex merge operation is $W_{density} * Error_{density} + W_{geometry} * Error_{geometry}$, where $W_{density}$ and $W_{geometry}$ are weighting parameters that are tailored to the needs of individual applications.

After computing the error metric for all neighboring vertex pairs, the volume decimation algorithm selects the vertex merge operation with the smallest error metric, say $V_n \rightarrow V_m$, eliminate all vertex merge operations of the form $V_n \rightarrow V_k$ for some k from further consideration, re-computes the error metric of those vertex merge operations that are affected by the application of $V_n \rightarrow V_m$, and repeats the cycle by picking the one with the smallest error metric from the remaining vertex merge operations, etc. After a vertex merge operation, the geometry of the affected region of tetrahedral mesh is changed. Consequently, the error metric of those vertex merge operations associated with the affected region needs to be recomputed. More specifically, after the application of a vertex merge operation $V_i \rightarrow V_j$, the error metric of all vertex merge operations of the form $V_k \rightarrow V_i$ needs to be recomputed based on the new geometry. Eventually the algorithm ends when all vertex merge operations have been eliminated. The list of selected vertex merge operations are ranked in an ascending order according to their error metric value, the smaller the error metric value is, the earlier an merge operation is performed. The rank value associated with a merge operation represents the order this operation is performed. For example, a operation with rank value 3 means this is the third operation.

A vertex that never needs to be merged into any other vertex is called an independent vertex. After applying the above volume decimation algorithm to a tetrahedral mesh, every vertex is scheduled to be merged into some other vertex at a certain priority except one or multiple independent vertices. *Gatun* organizes the list of resulting vertex merge operations into a forest of multiple trees (called a *merge tree*), each of whose root is an independent vertex. In these trees, each child vertex is to be merged to its parent vertex. In addition, every child vertex has a global rank that represents the priority of the corresponding vertex merge operation. Figure 5 shows an example merge tree for a hypothetic tetrahedral mesh. Each node in the tree represents a vertex in the mesh, and each edge represents a vertex merge operation. The weight on an edge represents the global rank of the edge's associated vertex operation. For example, the operation of merging Vertex 1 into Vertex 6 has a rank of 10, or equivalently, this is the 10th operation. At run time, if users ask the system to perform a simplification step that includes only the first 5 vertex merge operations, this operation will be ignored.

4.2 Run-Time Simplification Algorithm

Because the volume decimation algorithm already computes a global rank for all selected vertex merge operations, at run time *Gatun* only needs to perform the first N of these, where N is determined either by users or by the system based on available computation resource. Therefore whether a vertex merge operation needs to be performed for a given N is a local decision. This localness property makes it possible to perform on-the-fly simplification on the resulting tetrahedra stream that the decompresser produces. The only question left is how to effect each eligible vertex merge operation as a compressed tetrahedral mesh is being decompressed. The key insight behind inserting an on-the-fly volume simplification step between decompression and rendering is that all the renderer wants is that all the tetrahedra it uses during the rendering computation are valid tetrahedra that exist in the final mesh after all the selected simplification operations have been applied to the original mesh. That is, as long as every tetrahedron the simplification step passes to the renderer is a valid tetrahedron in the final mesh the rendering result of this integrated decompression/simplification/rendering pipeline is guaranteed to be correct.

Whenever a tetrahedron is enumerated, the renderer in the integrated decompression/rendering engine described in Section 3 advances each cast ray that intersects with the tetrahedron as much as possible, and then stops to wait for more tetrahedra to come so that these rays can move further ahead. As far as the renderer is concerned, it does not care about where and how the tetrahedra are generated. Without simplification, the decompresser outputs all the tetrahedra in the input mesh; With simplification, the simplification module outputs only those tetrahedra that exist in the simplified mesh. Therefore, whenever a tetrahedron from the decompresser arrives, the simplification module needs to check whether the tetrahedron is collapsed after simplification and, if it is not, whether the tetrahedron's vertices change because of vertex merging. Only when a tetrahedron is not going to be collapsed, and the ultimate target vertices of the tetrahedron after application of all vertex merge operations already appear in the decompresser's output stream can the simplification module forward this tetrahedron to the renderer. Because such tetrahedra are valid in the simplified mesh and their data density and coordinate are known, the renderer can safely perform rendering computation based on them and produce provably correct simplified results.

Because what is needed is the set of tetrahedra in the final simplified mesh, one needs an efficient algorithm to summarize the accumulative effect of a set of chosen vertex merge operations. More concretely, one needs to compute the ultimate target vertex into which each vertex is to be merged, based on the input mesh's merge tree data structure and a selective set of verge merge operations. With this per-vertex information, one can easily check whether a given tetrahedron from the decompresser is valid in the final mesh or not, or if the decompresser has traversed all its ultimate target vertices.

Therefore *Gatun*'s on-the-fly simplification algorithm consists of two steps: one before and one during the decompression/rendering process. First, given a threshold N, the on-the-fly simplification algorithm performs a top-down traversal of the input mesh's merge tree and determine which vertex merge operations are eligible by comparing the vertices' rank with N. For each eligible vertex merge operation, which corresponds to an edge in the merge tree whose child vertex's rank is smaller than or equal to N, the child vertex's *ancestor* field is filled with its parent vertex's *ancestor* field. For each non-eligible vertex merge operation, which corresponds to an edge in the merge tree whose child vertex's rank is larger than N, the child vertex's *ancestor* field is filled with its own ID. A root vertex's *ancestor* field is always filled with its own ID. After this traversal, the ultimate target of each vertex is going to be merged into is kept in the vertex's *ancestor* field. If the *ancestor* field points

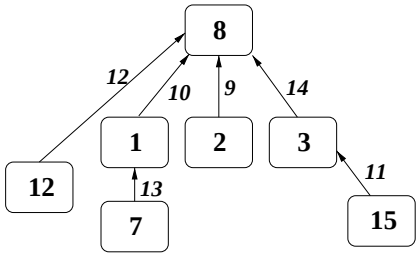

Figure 6: *The result mesh after the first* 8 *merges are carried out.*

to itself, the vertex does not get merged into anyone, and is present in the final mesh. For example, if the simplification threshold N is set to 8 for the merge tree in Figure 5, then after simplification, or equivalently after the first 8 vertex merge operations are done, Vertex 14's ancestor is Vertex 8, Vertex 9's ancestor is Vertex 3, etc., and Vertex 1, 2, 7, 3, 8, 12, and 15 are present in the final simplified mesh, as shown in Figure 6

After decompression/rendering starts, whenever the decompresser passes a tetrahedron, *Gatun* first checks whether the *ancestors* of the tetrahedron's four vertices are distinct. If they are not distinct, i.e., multiple vertices of the tetrahedron collapsed into one, the tetrahedron is not a valid one after all vertex merge operations and thus should be discarded. If they are distinct, it is a valid tetrahedron. However, a valid tetrahedron may not be *usable* yet because not all its vertices' ancestors have already appeared in the vertex stream that the decompresser outputs at that point. A tetrahedron can be forwarded to the rendered if and only if it is both valid and usable. When a non-usable tetrahedron first appears, it is attached to all the ancestor vertices that it waits for (the *vertex table* data structure), and its *waitCount* field is initialized to the number of ancestors that have yet to appear (the *tetrahedron table* data structure). Every time the decompresser enumerates a new vertex, it checks whether there are any tetrahedra waiting for the new vertex, if so decrements the *waitCount* field of each such non-usable tetrahedron by one, and forwards to the renderer those tetrahedra whose *waitCount* field reaches zero.

The on-the-fly volume renderer described in Section 3 needs to identify the input tetrahedral mesh's boundary surface first so that it can compute the intersections between the cast rays and boundary faces. However, the boundary surface may also be affected by the simplification step. For the boundary surface of an input tetrahedral mesh, *Gatun* performs a similar usability check on each boundary face, based on whether the ancestors of each face's three vertices are distinct. Only after all valid boundary faces have been completely identified can the renderer-cast rays be properly attached to the boundary surface. Only after successful attachment of all cast rays can the renderer advance the rays into the simplified tetrahedral mesh to interpolate/composite the data density values at sampling points.

4.3 Discussion

Figure 7 shows the interface between the stages in the proposed decompression/simplification/rendering pipeline. The decompresser forwards to the simplification stage one tetrahedron at a time. However this tetrahedron may be suppressed because it is invalid, or temporarily held up because it is not usable. Therefore, the enumeration of a new vertex in the decompression stage may push one or multiple pending tetrahedra to the renderer. An important advantage of this pipeline is that almost no modification is needed on the

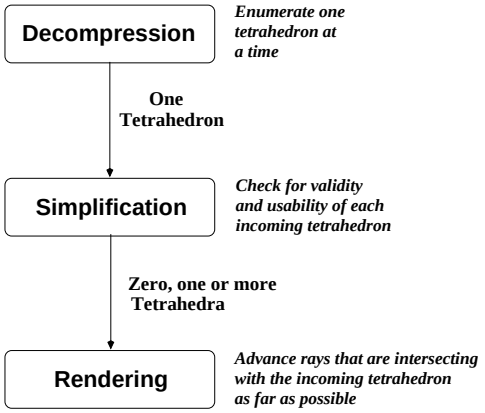

Figure 7: *The interfaces between and the operations performed by the stages in the integrated pipeline.*

decompresser or the renderer. The only change to the decompresser is to add a check for pending tetrahedra or faces on the enumeration of a new vertex. The renderer itself does not need to be modified for correctness.

5 Time-Critical Rendering

The original motivation for incorporating volume simplification into an integrated pipeline is to apply it at run time to trade quality for performance, or more specifically to *time-critical rendering*, where the goal is to maximize the rendering quality for a fixed timing budget by simplifying the input mesh appropriately. Although the integrated decompression/simplification/rendering pipeline described in the last section makes a good starting point, it itself is not sufficient to support time-critical rendering, because the decompression overhead dominates the end-to-end user-perceived delay regardless of the extent of mesh simplification and thus reduction in rendering time. In other words, while simplification does decrease the rendering time, it does not affect the end-to-end delay that much because the system still needs to decompress the *entire* input data set and simplify it to a chosen level. To address this issue, we pre-compute multiple simplified versions of each input data set, each corresponding to a particular simplification ratio, which is defined as the number of vertices that are simplified away divided by the number of all vertices. As shown in Figure 8, an input tetrahedral mesh is pre-simplified at the simplification ratios of $1 - 1/2^i$, and all these simplified versions are independently compressed and stored to the disk. The maximal value of i depends on the available system resource and performance requirements.

At run time, given a rendering time budget, the system first maps it to the corresponding simplification ratio, which may be different for different data sets, and then applies simplification to the pre-computed version of the input data whose simplification ratio is the largest among those whose simplification ratio is smaller than the target simplification ratio. For example, in this Figure, the version with the simplification ratio of 0.5 is selected as the starting mesh for simplification if the target simplification ratio is p, whereas the version with the simplification ratio of 0.75 is selected if the target simplification ratio is q. The basic idea of this approach is similar to mip-mapping used to speed up texture mapping. Although the disk storage cost of this scheme is doubled, the run-time performance is improved significantly as shown in the next section, because both decompression and rendering overheads are now about inversely proportional to to the simplification ratio.

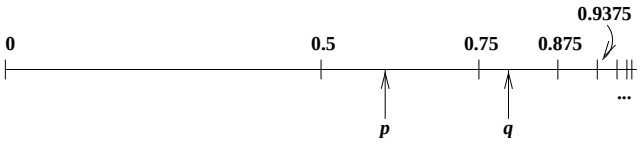

Figure 8: *Given a target simplification ratio, choose the most simplified version of a multi-resolution input mesh that is finer than the target as the starting mesh for simplification.*

6 Performance Evaluation

The input data sets used for the performance evaluation of *Gatun* are listed in Table 1, as ordered by the number of tetrahedra. While the first two data sets are unstructured grids, the remaining four are curvilinear grids converted into tetrahedral grids. The first two grids are included to demonstrate that the proposed integrated pipeline is applicable to general unstructured-grid data sets. Notice the Delta Wing data set has different characteristics than before because we have found that due to its degeneracy in some cells, our previous approach of partitioning each of its curvilinear grid cell into five tetrahedra will result in an inconsistent tetrahedral mesh, therefore we have partitioned each of its curvilinear grid cell into six tetrahedra to resolve the inconsistency, thus increasing the number of tetrahedra therein.

The compression efficiency (2.5 bits/tetrahedron, connectivity only) remains the same as reported previously, because we use the same tetrahedral compression algorithm. In addition, the peak memory usage saving (50% to 70%) also carries over when simplification is integrated into the pipeline, because the additional memory space requirement introduced by simplification is minimal, especially when compared with the rendering stage, whose memory footprint requirement is always dominant. The overall reduction of memory footprint in *Gatun* not only speeds up the rendering process by one to two orders of magnitude when input data sets are too large to fit into physical memory, but also shortens the perceived rendering delay when input data sets are completely memory-resident.

6.1 Simplification Overhead

Run-time simplification consists of two steps: a top-down traversal of the merge tree to adjust each vertex's ancestor field, and the enqueuing and checking for pending tetrahedra or faces that are waiting for their vertices to be decompressed. Table 2 shows the performance overheads for these two steps as measured from the *Gatun* prototype. The rendered image plane is set to 256×256. For each input data set, we also varied the simplification ratio from 0.1 to 1.0. The hardware testbed used to collect performance measurements is a P4 1.5GHz machine with 512 MBytes of memory running RedHat Linux 7.1 . Table 2 shows the simplification overhead in terms of the percentage of the total time. Notice the simplification overhead drops as simplification ratio goes up, which is because more tetrahedra become degenerate and get discarded therefore the associated enqueuing and checking (for pending tetrahedra) overheads are also reduced. In all cases, the run-time simplification overhead is less than 5% on an average for all six test data sets. This result demonstrates that simplification is a low-overhead run-time performance adaptation mechanism for irregular-grid volume rendering.

6.2 Rendering Performance Improvement

A major performance advantage of the proposed integrated pipeline is that each piece of volume data is brought into the main memory exactly once. In contrast, a baseline (Generic) implementation of the same three steps may involve reading the input tetrahedral mesh from the disk, simplifying it, and rendering the simplified mesh. The whole decompressed data is generated first before any simplification step can be applied. Similarly simplification must be completed before rendering can start. Figure 9 shows the end-to-end rendering time comparison between the baseline implementation and *Gatun* for each of the six test data sets. For both implementations, as the simplification ratio increases, the rendering delay decreases as expected. The rendering performance improvement due to simplification is only up to a factor of two (the Delta Wing data set and simplification ratio 0.99). The performance improvement is not linear with respect to the simplification ratio because when the overhead of "touching" the input data set once in order to simplify it dominates the overall performance cost. This result clearly demonstrates why simplification alone does not provide sufficient flexibility for run-time adaptation.

Between *Gatun* and the baseline implementation, *Gatun* always wins because *Gatun*'s pipelined structure significantly reduces the amount of disk I/O due to "store and compute," especially for very large input data sets. Although there is plenty of main memory in the test machine, this performance difference is still quite noticeable. When memory resource becomes less abundant, *Gatun* is expected to be far even better than the baseline implementation in the end-to-end rendering time, as shown in the previous work [YMC00].

6.3 Time-Critical Rendering

Figure 9 shows that without pre-computing multiple simplified versions, even at the simplification ratio of 99%, the end-to-end rendering performance is still far from interactive for most data sets. To demonstrate how multi-resolution pre-simplification helps time-critical rendering, we pre-simplified the Blunt-fin data set into 13 different versions at simplification ratios $1 - 1/2^i$ where i ranges from 1 to 13. As shown in Table 3, at the image resolution of 128×128, the end-to-end delay drops to below 0.17 seconds (or 6 frames/sec), when the simplification ratio is greater than 0.984. Figure 10 shows that with pre-computation, the rendering time now indeed becomes approximately linear with respect to the target simplification ratio. It also demonstrates that the proposed multi-resolution pre-computation scheme can also benefit the baseline case as well, as their performances are rather close to each other. The fact that this performance vs. simplification ratio figure is highly linear also means that it could serve as the basis for data set-specific adaptation for time critical rendering. That is, given a desired frame rate, the system could determine the most appropriate simplification ratio from such a figure, retrieves the appropriate pre-computed version, and initiates the simplification process from there. For example, for the Blun-fin data set, if the target frame rate is two frames per second, or equivalently 0.5 seconds per frame, one can determine from Figure 9 that a simplification ratio of 0.905 could support such an interactivity requirement.

6.4 Quality Degradation

To understand how much volume simplification degrades the final rendered image quality we calculated the RMSE (root mean square error) between the resulting images generated from an original data set and from its simplified versions. This RMSE computation excludes those black pixels where there is no actual ray contribution. Figure 11 shows the quality degradation is minimal and noticeable only for very aggressive simplification ratios. Images of Blunt-fin that correspond to the simplification ratio of 0%, 95% and 99% are shown in Figure 12. In this experiment, in addition to standard volume-based rendering, where sample points are uniform along the cast rays, we also tried the face-based rendering method used

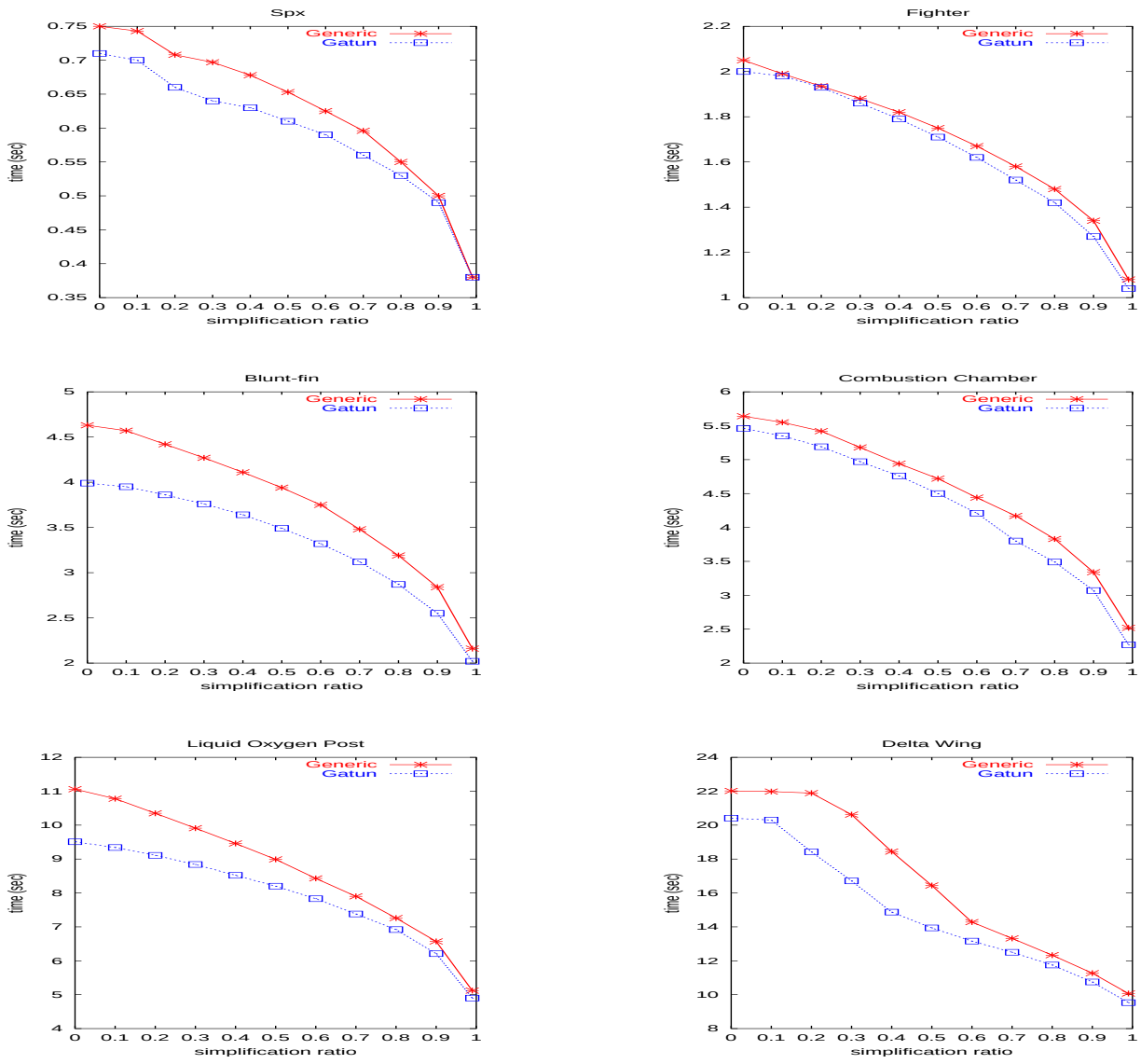

Figure 9: *Comparisons between the generic renderer (the baseline) and Gatun for different simplification ratios. There is only one version of the input data set stored on the disk. Image resolution is set at* 256×256.

Data set	# of points	# of tetrahedra	# of faces	# of boundary faces
Spx	2896	12936	27252	2760
Fighter	13832	70125	143881	7262
Blunt-fin	40960	187395	381548	13516
Combustion Chamber	47025	215040	437888	15616
Liquid Oxygen Post	109744	513375	1040588	27676
Delta Wing	207970	1195839	2408702	34048

Table 1: *Characteristics of input data sets used in this performance study.*

	Simplification Ratio										
Dataset	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.99
Spx	2.81%	2.85%	2.57%	2.34%	2.06%	1.99%	1.44%	1.17%	0.91%	0.61%	0.26%
Fighter	5.00%	5.05%	4.66%	4.30%	3.91%	3.51%	3.08%	2.63%	2.11%	0.78%	0.19%
Blunt-fin	7.01%	6.58%	6.73%	5.85%	5.21%	4.87%	3.91%	3.20%	2.78%	1.56%	0.34%
Combustion	5.86%	5.60%	5.20%	5.03%	4.83%	4.22%	3.80%	3.42%	2.57%	1.62%	0.44%
Post	7.99%	7.60%	7.13%	6.66%	6.22%	5.48%	4.85%	3.92%	2.74%	1.93%	0.41%
Delta Wing	8.08%	7.88%	7.60%	7.18%	7.26%	6.59%	5.62%	4.64%	3.48%	2.13%	0.31%

Table 2: *The run-time simplification overhead expressed as the relative percentage of the total rendering time for different simplification ratios.*

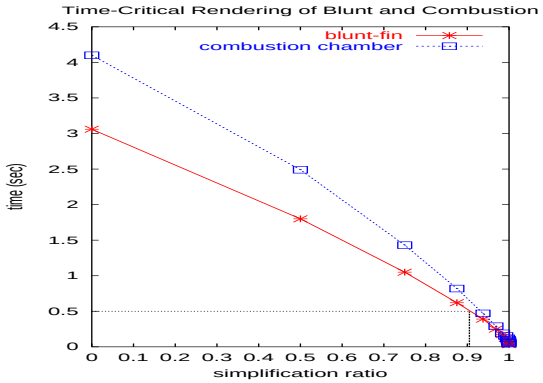

Figure 10: Time-critical rendering of the Blunt-fin and Combustion Chamber data sets for different simplification ratios. Image resolution is set at 128×128.

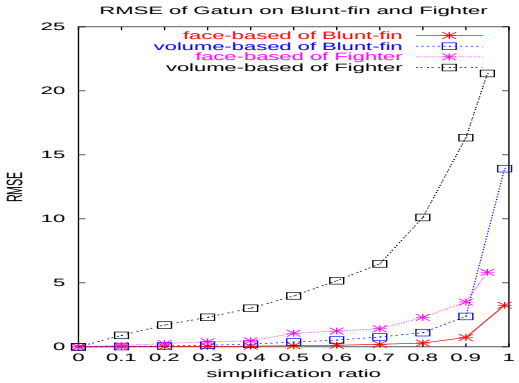

Figure 11: *The RMSEs of the images generated from Blunt-fin and Fighter data sets for different simplification ratios, using face-based and volume-based rendering. Image resolution is set at* 256×256.

by Bunyk et al. [BKS97], where sample points on the cast ray are required to reside on the faces of tetrahedra. Figure 12 shows that with volume-based rendering, noticeable artifacts only appear when more than 99% of the input tetrahedral mesh is simplified away. With face-based rendering, the artifacts are still difficult to discern even when the mesh is more than 99% simplified. Because the absolute image quality of face-based rendering is worse than volume-based rendering, face-based rendering is more "tolerant" of additional errors introduced by simplification. Accordingly, the quantitative difference in RMSE between rendered results of original and simplified sets is smaller for face-based rendering than for volume-based rendering, as shown in Figure 11.

We want to emphasize again that this work does not claim any new contribution to volumetric mesh simplification algorithms per say. Rather, it focuses on how to integrate a class of simplification algorithms that satisfy the assumptions listed in Section 4.1 into a decompression-driven renderer. Therefore, the goal of this subsection is to show that integrating a simplification algorithm with decompression and rendering is as effective as the original simplification algorithm when it is implemented separately.

7 Conclusion

Although lossless compression is an effective technique to reduce the storage requirement and run-time disk access cost for very large irregular volume data sets, it cannot reduce the rendering computation overhead because the number of tetrahedra that a renderer needs to process remains unaffected with or without lossless compression. Lossy compression of irregular volume data, or volume data simplification, on the other hand, provides a volume rendering system the additional flexibility to trade off rendering time and quality. Unfortunately, most previous research on volume data simplification focused on the development of stand-alone simplification tools that are never integrated into the renderer. As a result, unlike in surface rendering, where a polygonal renderer uses simplification as an effective control tool to adjust rendering accuracy and performance at run time, volume simplification has never been

value of i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
time (sec)	3.06	1.80	1.05	0.62	0.39	0.25	0.17	0.12	0.10	0.08	0.07	0.05	0.04	0.03

Table 3: *Rendering time required for the Blunt-fin data set for different simplification ratios, where the simplification ratio is defined by* $1 - 1/2^i$. *Image resolution is set at* 128×128.

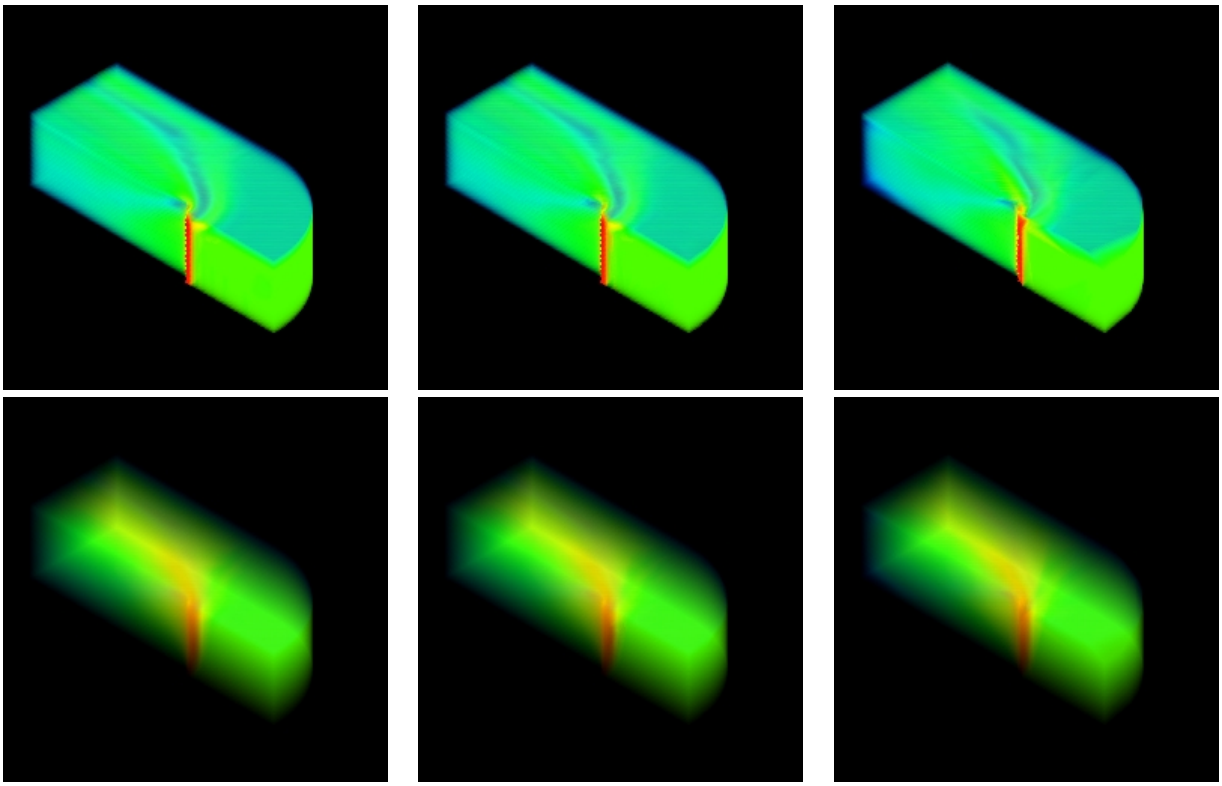

Figure 12: *Rendered images from Blunt-fin with different simplification ratio. From left to right: No simplification, 95% simplified, 99% simplified. Images on the top row are rendered by volume-based rendering, while images on the bottom row by area-based rendering.*

an integral part of a volume renderer as a dynamic adaptation mechanism. This paper describes the first irregular-grid volume rendering system that integrates not only volume simplification with rendering, but also volume decompression, into a single seamless pipeline. With this integrated pipeline, simplification becomes an active element of a volume rendering system, and each piece of volume data is brought into the main memory only once as it travels through the decompression, simplification, and rendering steps. We have successfully implemented the proposed integrated decompression/simplification/rendering pipeline on the *Gatun* system. Empirical measurements on the *Gatun* prototype show that the additional performance overhead associated with run-time simplification is less than 5% on an average compared to the same pipeline without simplification. However, with simplification in place, the rendering performance can be improved only by a factor of up to 2 because the entire data set still needs to be touched at least once regardless of the simplification ratio. To address this problem, we propose a multi-resolution pre-simplification scheme similar in spirit to mip-mapping, which effectively reduces the end-to-end rendering time to be about inversely proportional to the simplification ratio, and makes a powerful building block for *time-critical rendering*.

In the future, we plan to integrate not only view-independent simplification, but also view-dependent simplification into our framework, given that the latter should provide more room for more aggressive simplification during run-time data browsing. We are also investigating other simplification primitives other than vertex merge or edge collapse which can be integrated into our framework.

References

[BKS97] 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.

[CFM+97] P. Cignoni, L. D. Floriani, C. Montani, E. Puppo, and R. Scopigno. Multiresolution representation and visualization of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 3(4), December 1997.

[CVM+96] J. Cohen, A. Varshney, D. Manocha, G. Turk, and H. Webber. Simplification Envolopes. In *SIGGRAPH '96*, pages 119–128, August 1996.

[EDD+95] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution Anal-

ysis of Arbitrary meshes. In *SIGGRAPH '95*, pages 173–182, August 1995.

[ESC00] J. El-Sana and Y. Chiang. External Memory View-Dependent Simplification. *Eurographics '2000*, 19(3), 2000.

[FMS00] R. Farias, J Mitchell, and C. Silva. Zsweep: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering. In *ACM/IEEE Volume Visualization Symposium 2000*, 2000.

[Fru94] T. Fruhauf. Raycasting of Nonregularly Structured Volume Data. *Computer Graphics Forum (Eurographics '94)*, 13(3):294–303, 1994.

[Gar90] M. P. Garrity. Raytracing Irregular Volume Data. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24:35–40, Nov 1990.

[GGS99] 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.

[Gie92] C. Giertsen. Volume Visualization of Sparse Irregular Meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.

[GVW99] A. V. Gelder, V. Verma, and J. Wilhelms. Volume Decimation of Irregular Tetrahedral Grids. In *Computer Graphics International 1999 Conference Proceedings*, 1999.

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh Optimization. In *SIGGRAPH '93*, pages 19–26, July 1993.

[HH92] Paul Hinker and Charles Hansen. Geometric Optimization. In *IEEE Visuallization '93*, pages 55–64, 1992.

[HK99] L. Hong and A. Kaufman. Fast Projection-Based Ray-Casting Algorithm for Rendering Curvilinear Volumes. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):322–332, October 1999.

[Hop96] H. Hoppe. Progressive Meshes. In *SIGGRAPH '96*, August 1996.

[Hop97] H. Hoppe. View-dependent Refinement of Progressive Meshes. In *SIGGRAPH '97*, August 1997.

[KT96] Alan D. Kalvin and Russell H. Taylor. Superfaces: Polygonal Mesh Simplification with Bounded Error. *IEEE Computer Graphics and Applications*, 16(3):64–77, May 1996.

[MC98] 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.

[OGS98] M. H. Gross O. G. Staadt. Progressive Tetrahedralizations. In *IEEE Visualization '98*, pages 397–402, October 1998.

[RB93] J. Rossignac and P. Borrel. Multi-resolution 3d approximation for rendering complex scenes. *Geometric Modeling in Computer Graphics*, pages 455–465, 1993.

[RMSW00] R.Farias, J. Mitchell, C. Silva, and B. Wylie. Time-Critical Rendering of Irregular Grids. In *Proceedings of the XIII SIBGRAPI International Conference*, 2000.

[RO96] K. J. Renze and J. H. Oliver. Generalized Unstructured Decimation. *IEEE Computer Graphics and Applications*, 16(6):24–32, 1996.

[Sil96] C. Silva. *Parallel Volume Rendering of Irregular Grids*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, 1996.

[SR99] 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.

[SZL92] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of Triangle Meshes. In *SIGGRAPH '92*, pages 65–70, July 1992.

[THJW98] I. J. Trotts, B. Hamann, K. I. Joy, and D. F. Wiley. Simplification of Tetrahedral Meshes. In *IEEE Visuzalization '98*, pages 287–295, October 1998.

[Tur92] Greg Turk. Te-tiling Polygonal Surfaces. In *IEEE Visuallization '92*, pages 55–64, 1992.

[Use91] S. Uselton. Volume Rendering for Computational Fluid Dynamics: Initial Results. Technical Report RNR-91-026, Nasa Ames Research Center, 1991.

[WCA+90] 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.

[XV96] Julie C. Xia and Amitabh Varshney. Dynamic View-dependent Simplification for Polygonal Models. In *IEEE Visuallization '96*, pages 327–334, 1996.

[YMC00] C. Yang, T. Mitra, and T. Chiueh. On-the-Fly Rendering of Losslessly Compressed Irregular Volume Data. In *IEEE Visualization 2000'*, October 2000.

[YRL+96] 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.