Streaming Simplification of Tetrahedral Meshes

Huy T. Vo, Steven P. Callahan, Peter Lindstrom, *Member*, *IEEE*,
Valerio Pascucci, *Member*, *IEEE*, and Cláudio T. Silva, *Member*, *IEEE*

Abstract—Unstructured tetrahedral meshes are commonly used in scientific computing to represent scalar, vector, and tensor fields in three dimensions. Visualization of these meshes can be difficult to perform interactively due to their size and complexity. By reducing the size of the data, we can accomplish real-time visualization necessary for scientific analysis. We propose a two-step approach for streaming simplification of large tetrahedral meshes. Our algorithm arranges the data on disk in a streaming, I/O-efficient format that allows coherent access to the tetrahedral cells. A quadric-based simplification is sequentially performed on small portions of the mesh in-core. Our output is a coherent streaming mesh which facilitates future processing. Our technique is fast, produces high quality approximations, and operates out-of-core to process meshes too large for main memory.

Index Terms—Computational geometry and object modeling, out-of-core algorithms, streaming algorithms, mesh simplification, large meshes, tetrahedral meshes.

1 Introduction

Simplification techniques have been a major focus of research for the past decade due to the increasing size and complexity of geometric data. Scientific simulations and measurements from fluid dynamics and partial differential equation solvers have produced data sets that are too large to visualize with current hardware. Thus, approximations which maintain a volumetric mesh are necessary to achieve a level of interactivity that is necessary for proper analysis through visualization techniques such as isosurfacing or direct volume rendering.

Although significant work has been done in simplifying triangle meshes, relatively little has been done with tetrahedral meshes. Most of the work in tetrahedral simplification falls into two categories: edge-collapse methods and point sampling methods. These algorithms assume that the entire mesh can be loaded into main memory. However, due to the high memory overhead of storing the mesh connectivity in addition to the geometry, there are limitations on the size of the data set that can be simplified in this manner.

We present an algorithm that *streams* the data from disk through memory and performs the simplification on a localized portion of the entire mesh. Our approach consists of two steps. First, the tetrahedral mesh is arranged in a streaming format that supports coherent sequential access. Then, this streaming mesh is sequentially simplified using a quadric error-based scheme that respects boundaries and fields in the mesh. The resulting mesh is output in the same streaming format and can be used directly in subsequent processing. This allows other streaming algorithms to be used on the simplified mesh such as isosurface extraction [1] or compression [2].

Our streaming algorithm requires only one pass to simplify the entire mesh. Thus, the layout of the mesh is of great importance to producing high quality results. We perform a reordering of the tetrahedral cells and store them on disk using a streaming tetrahedral mesh format. This format provides concurrent access to coherently ordered vertices and tetrahedra. It also minimizes the duration that a vertex remains in-core, which limits the memory footprint of the simplification.

Our tetrahedral simplification incrementally works on overlapping portions of the mesh in-core (see Fig. 1). We use the quadric-error metric to perform a series of edge collapses until a target decimation is reached. By weighting the boundaries and incorporating the field data in our error metric, we can keep the error in the simplified approximation low. This results in a simplification algorithm that can efficiently simplify extremely large data sets. In addition, through the use of carefully optimized algorithms, linear solvers, and data structures we show that significant improvements in speed and stability can be achieved over previous techniques.

The main contributions of this paper include:

- We describe a quadric-based edge-collapse simplification algorithm that operates on portions of the streaming tetrahedral mesh. This operation occurs in a single pass, runs quickly, handles data of arbitrary size, respects field data, and works out-of-core.
- We improve upon previous stream simplification methods by ensuring that the output stream is coherent in order to accommodate further downstream processing. We also introduce optimizations in the data structures and simplification algorithm which dramatically improve the speed and efficiency of tetrahedral simplification.

- *H.T. Vo and S.P. Callahan are with the Scientific Computing and Imaging Institute, 50 S Central Campus Drive, Room 3490, Salt Lake City, UT 84112. E-mail: {hvo, stevec}@sci.utah.edu.*
- *P. Lindstrom and V. Pascucci are with the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 7000 East Avenue, Box 808, L-560, Livermore, CA 94551. E-mail: {pl, pascucci}@llnl.gov.*
- *C.T. Silva is with the School of Computing, University of Utah, 50 S. Central Campus Drive, RM 3190, Salt Lake City, UT 84112. E-mail: csilva@cs.utah.edu.*

Manuscript received 10 Nov. 2005; revised 20 Mar. 2006; accepted 1 May 2006; published online 8 Nov. 2006.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0157-1105.

Fig. 1. Streaming simplification performed on a tetrahedral mesh ordered from bottom to top. The portion of the mesh that is in-core at each step is shown in green.

- We provide a new stable solver for quadric-based simplification that is simpler than the existing algorithms. We also provide both stability and error analysis of the results generated using this technique.
- We show that our streaming algorithm can successfully simplify a data set consisting of over one billion tetrahedra on a commodity PC with low error.

The remainder of this paper is organized as follows: We summarize related work in Section 1.1. In Section 2, we describe our algorithm for arranging the data in a coherent, streaming mesh. Section 3 provides details on our out-of-core simplification, Section 4 contains our stability and error analysis followed by performance measures, Section 5 discusses the benefits of our approach over previous algorithms, and Section 6 provides final remarks and directions for future work.

1.1 Related Work

A common result from scientific computations is a scalar field f in $\mathbb{R}^3$. This scalar field f can be represented over a domain D as a tetrahedral mesh. When it is not possible to achieve interactive visualization of f, it is common to find a tetrahedral mesh with fewer elements and an associated scalar field f' such that the approximation error $\|f' - f\|$ is minimized. Many algorithms have been proposed in an attempt to compute f' quickly and with little error.

Trotts et al. [3], [4] developed a technique that collapses one edge at a time, deciding which edge to collapse next based on an error bound calculated at each step. They provide a bound on the maximum deviation of the field data in the simplified mesh from the original.

Several techniques for simplification have recently been proposed that act on the vertices. Van Gelder et al. [5] remove vertices based on mass and data error metrics. Uesu et al. [6] provide a fast point-based method which works directly on the underlying scalar field. These techniques are more memory efficient than edge collapse methods, but require the addition of Steiner points to handle nonconvex meshes. This requirement makes them difficult to modify for streaming algorithms.

The idea of a progressive mesh for surface level of detail control was proposed by Hoppe [7] and later extended to simplicial complexes by Popović and Hoppe [8]. Staadt and Gross [9] define appropriate cost functions to account for volume preservation, gradient estimation, and scalar data with progressive tetrahedral meshes. Chopra and Meyer [10] propose a fast progressive mesh decimation scheme that is based on the scalar field of the mesh.

Many algorithms have been developed that use different error metrics to perform the simplification via edge collapses. Cignoni et al. [11] use domain and field (i.e., range) error metrics to approximate the original mesh. The use of a quadric error metric for surface simplification was introduced by Garland and Heckbert [12]. Their method uses iterative contractions on vertex pairs and calculates the error approximations using quadric matrices. Natarajan and Edelsbrunner [13] extend the quadric error metric to preserve topological features. Garland and Zhou [14] recently generalized the quadric error metric for simplifying simplicial elements in any dimension.

As model size has continued to increase faster than main memory size in commodity PCs, techniques have been developed to simplify these data sets out-of-core. Lindstrom [15] proposed an algorithm that simplifies triangle meshes of arbitrary size. This algorithm improves upon Rossignac and Borel's [16] vertex-clustering method by using the quadric error metric. The mesh is stored as a redundant list of three vertex positions per triangle. This "triangle soup" is read one triangle at a time and a simplified mesh is constructed incrementally and kept in-core. Lindstrom and Silva [17] improve upon the quality of this algorithm while making the method more memory efficient by storing the simplified mesh out-of-core during processing. They handle boundaries separately to preserve the overall shape of the mesh. Wu and Kobbelt [18] propose an edge collapse-based streaming method for large triangle meshes that requires only one pass to decimate the entire mesh. All three of these methods make use of a redundant on-disk mesh representation that is two times larger than an indexed mesh. However, they are fast because no global indexing is required, which results in better cache coherency. Isenburg et al. [19] present a triangle mesh simplification algorithm based on the processing sequence paradigm. A processing sequence is a mesh representation that interleaves the ordering of the indexed triangles and vertices. This representation provides fast out-of-core access because it arranges the data in a memory-efficient order while requiring no additional storage. We improve upon the method of Isenburg et al. by placing fewer restrictions on the input mesh, making the algorithm directly applicable to a larger class of mesh producing applications.

With an increase in streaming algorithms, the need for a streamable format that efficiently codes both geometry and connectivity becomes necessary. Isenburg and Lindstrom [20] provide the underlying work for streaming representations of polygonal meshes. They provide metrics and diagrams for measuring the streamability of a mesh and

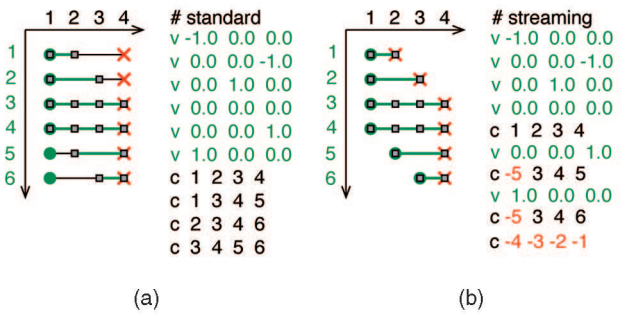

Fig. 2. Layout diagrams with cell indices on the horizontal and vertex indices on the vertical axis. A vertex is active from the time it is introduced until it is finalized, as indicated by the horizontal lines. (a) A standard format for tetrahedral meshes based on the OBJ format. (b) A streaming format that interleaves vertices with cells. Vertex finalization is provided with a negative index (i.e., relative to the most recent vertex).

discuss methods for improving its layout so as to reduce its memory footprint and the time each mesh element remains in-core. Mascarenhas et al. [1] extend this format to handle volumetric grids, which allows for streaming out-of-core isosurface extraction.

2 Mesh Layout

Traditional object file formats consist of a list of vertices followed by a list of polygonal or polyhedral elements that are defined by indexing into the vertex list. Dereferencing such a mesh, i.e., accessing vertices via their indices, requires the whole vertex list to be in memory since elements are generally not assumed to reference vertices in any particular order; an element can arbitrarily reference *any* vertex in the list. Furthermore, streaming such a mesh implies buffering all vertices before the first element is encountered in the stream. A logical progression for large meshes is to store them in a streaming mesh representation that interleaves the vertices and elements and stores them in a "compatible" order. This representation allows a vertex to be introduced (added to an in-core active set) when needed and finalized (removed from the active set) when no longer used.

Isenburg and Lindstrom [20] provide a streaming mesh format for triangle meshes that extends the popular OBJ file format. We use their format with straightforward additions to handle tetrahedral meshes. This includes extending the vertices to four values $\langle x, y, z, f \rangle$ that represent the position in 3D space and a scalar value. In addition, we provide a new element type for a tetrahedral cell that indexes four vertices. This format allows us to finalize a vertex when it is no longer in use by using a negative index into the vertex list, which is backward compatible with the OBJ format. Fig. 2 shows an ASCII example of the streaming tetrahedral format.

An important consideration with streaming meshes is the ability to analyze the efficiency of a mesh layout. Isenburg and Lindstrom developed several techniques for visualizing properties of the mesh to determine the effectiveness of the layout. We use similar tools to measure tetrahedral mesh quality. An important property is the *front width*, or the maximum number of concurrently active vertices. An *active* vertex is one that has been introduced, but not yet finalized. The width of a streaming mesh gives a lower bound on the amount of memory required for dereferencing (and, thus, processing) the mesh. Another important property of the mesh is the *front span*, which measures the maximum index difference (plus one) of the concurrently active vertices. A low span allows a faster implementation because optimizations can be performed that achieve the best performance when the span is similar to the width. Low span makes for efficient indexing in the file format, bounds the width, and, for simplification purposes, ensures that vertices do not become stagnant in the buffer, which would prevent all incident edges from being collapsed. The efficiency of our layout depends on quantifying these properties, thus we provide analysis on different layout techniques so that we can choose the most streamable layout for a given data set.

One simple mesh layout is to sort the vertices on a *spatial* direction, in particular one that crosses the most tetrahedra. Wu and Kobbelt [18] use this technique for triangle mesh simplification. This can be accomplished for large meshes by performing an out-of-core sort on the vertices [17] and writing them into a new file. An additional file is created to contain a mapping of the old ordering to the new one. Next, the tetrahedral cells are written to a new file and reindexed according to the mapping file. A sort is then performed on the file containing the tetrahedral cells based on the largest index of each cell. Finally, the vertex file and the cell file are read simultaneously and interleaved into a new file by writing each cell immediately after the vertex corresponding to the cell's largest index has been written. Spatial layouts work especially well when considering meshes that have a dominant principal direction.

Other techniques may be desirable if the mesh does not have a dominant principal direction, such as a sphere. An approach to handle this type of data is to use a *bricking* method similar to the one proposed by Cox and Ellsworth [21] in which the vertices are ordered into a fixed number of small cubes for better sequential access. A similar approach is to arrange the vertices using a Lebesgue space-filling curve (i.e., *z-order*), which provides better sequential access in the average case. This arrangement can be generated by creating an out-of-core octree [22] of the vertices and traversing them in-order. The interleaved mesh is then written to a file in the same manner described above for spatial sorting. The results of the layout produced by bricking and z-order traversal are similar. When streamed they provide a more contiguous portion of the mesh on average, but the front width and span are typically much larger than sorting spatially.

Another approach used for laying out the mesh on disk is *spectral sequencing*. This heuristic finds the first nontrivial eigenvector (the *Fiedler vector*) of the mesh's Laplacian matrix and was shown by Isenburg and Lindstrom [20] to be very effective at producing low-width layouts. They provide an out-of-core algorithm for generating this ordering for streaming triangle meshes, which we have extended to handle tetrahedra. This method works particularly well for curvy triangle meshes, but tetrahedral meshes are generally less curvy and more compact. Still, in

TABLE 1
Analysis of Mesh Layout

Dataset	Vertices	Tetrahedra	Spatial Sort		Z-Order		Spectral		Breadth-First	
			Width	Span	Width	Span	Width	Span	Width	Span
Torso	168,930	1,082,723	3,118	20,784	7,256	122,174	**2,894**	13,890	5,528	**6,370**
Fighter	256,614	1,403,504	**3,894**	110,881	9,382	215,697	3,916	28,638	16,629	**19,523**
Rbl	730,273	3,886,728	2,814	5,270	10,232	371,269	**2,291**	21,764	3,206	**3,495**
Mito	972,455	5,537,168	19,876	33,524	10,202	642,550	**6,745**	44,190	10,552	**11,498**
SF1	2,461,694	13,980,162	16,898	65,921	48,532	1,958,212	**12,851**	131,152	30,258	**33,378**

most cases, this ordering results in the lowest width, which is ideal for minimizing memory consumption.

A final approach is to create a *topological* layout, which starts at a vertex on the boundary and grows out to neighboring vertices. To grow in a contiguous manner, we use a breadth-first traversal with optimizations to improve coherence [20]. Instead of a traditional FIFO priority queue, we assign priority using three keys. First, the oldest vertex on the queue is used in the same way that it would be in standard breadth-first algorithms. However, if multiple vertices were added to the queue at the same time, a second and third key are used to achieve a more coherent order. The second key is Boolean and gives preference to a vertex if it is the final one in a cell that has not been processed. Finally, the third key is to use the vertex that was most recently put on the queue, which is more likely to be adjacent to the last vertex. These sort keys guarantee a layout that is *compact* [20] such that runs of vertices are referenced by the next cell, and runs of cells reference the previous vertex (e.g., as in Fig. 2b). In practice, this traversal can be accomplished out-of-core by breaking the mesh into pieces. Using this approach, we were able to minimize the front span of the data sets in all of our experimental cases. This is ideal because having a span and width that are similar allows us to exploit optimization techniques described in the simplification algorithm. Recently, Yoon et al. [23] propose a layout based on local mesh optimizations which reduce cache misses. This approach applied to tetrahedra would also give a compact representation as with our breadth-first layout and could be used to achieve similar results.

Table 1 shows the front width and span for four different data sets produced by the layout techniques described above. Spectral sequencing proves to be the superior choice when low width and thus memory efficiency is required. Breadth-first layouts are not as memory-efficient, but as we will see can be processed fast. Note that, unlike [19], we do not require a face-connected order and we do not require that each vertex be finalized before it can be inserted into the in-core mesh. This allows us to avoid local reordering of the tetrahedra or the vertices.

3 Tetrahedral Simplification

3.1 Quadric-Based Simplification

To achieve high-quality approximations, we use the quadric error metric proposed by Garland and Zhou [14]. This metric measures the squared (geometric and field) distances from points to hyperplanes spanned by tetrahedra. The volume boundaries are preserved using a similar metric on the boundary faces and by weighting boundary and interior errors appropriately. The generalized quadric error allows the flexibility of representing field data by extending the codimension of the manifold. Given a scalar function $f : D \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$ defined over a domain D represented by a tetrahedral mesh, we can represent the vertices at each point $\mathbf{p}$ as $\langle x_p, y_p, z_p, f_p \rangle$, which can be considered a point on a 3D manifold embedded in $\mathbb{R}^4$. Thus, by extending the quadric to handle field data, the algorithm intrinsically optimizes the field approximation along with the geometric position.

The quadric error of collapsing an edge to a single point is expressed as the sum of squared distances to all accumulated incident hyperplanes and can, in n dimensions, be encoded efficiently as a symmetric $n \times n$ matrix $\mathbf{A}$, an n-vector $\mathbf{b}$, and a scalar c. It is sufficient to component-wise add these terms to combine the quadric error of two collapsed vertices. Finding the point $\mathbf{x}$ that minimizes this measure amounts to solving a linear system of equations $\mathbf{Ax} = \mathbf{b}$. Once $\mathbf{x}$ is computed, we test whether collapsing to this point causes any tetrahedra to flip [10], i.e., changes the sign of their volume, in which case we disallow the edge collapse. Because $\mathbf{A}$ is not necessarily invertible, it is important to choose a linear solver that is numerically stable. Since quadric metrics are covered in great detail elsewhere (see, e.g., [12], [114]), we here focus only on the numerical issues of robustly minimizing quadric errors (see Section 3.4).

3.2 Streaming Simplification

Combining streaming meshes with quadric-based simplification, we introduce a technique for simplifying large tetrahedral meshes out-of-core. We base our streaming algorithm on [19] and [20], but make several general improvements and provide a list of optimizations that compared to a less carefully engineered implementation results in dramatic speed improvements.

First, unless already provided with streaming input, we convert standard indexed meshes and optionally reorder them for improved streamability. Then, portions of the streaming mesh are loaded incrementally into a fixed-size main memory buffer and are simplified using the quadric-based method. Once the in-core portion of the mesh reaches the user-prescribed resolution, simplified elements are output, e.g., to disk or to a downstream processing module. Thus, input and output happen virtually simultaneously as the mesh streams through the memory buffer (see Fig. 3).

To ensure that the final approximation is the desired size, two control parameters have been added: *target reduction* and *boundary weight*. Target reduction is the ratio between the number of tetrahedra in the output mesh and the number of tetrahedra in the original mesh. Alternatively, this parameter can be expressed as a target tetrahedral count of the resulting mesh. The boundary

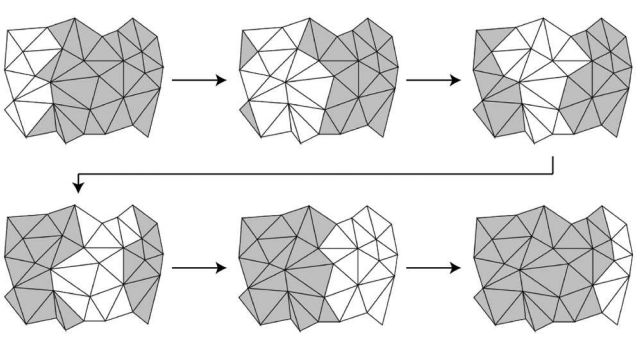

Fig. 3. Example of a buffer moving across the surface of a tetrahedral mesh sorted from left to right. As new tetrahedra are introduced, the tetrahedra that have been in-core the longest are removed from memory.

VERTEX	
float[10] **A**	quadric matrix
float[4] **p**	position and field value
float ε	quadric error at **p**
int *idx*	index to input/output stream
int *corner*	vertex-to-corner index
bool *deleted*	
bool *written*	
TETRAHEDRON	
int[4] *vidx*	vertex indices
int[4] *link*	corner links
bool *deleted*	
int *idx*	position in input stream

Fig. 4. Data structures used for our quadric-based simplification.

weight prevents the shape of the mesh from changing throughout the simplification. We use a fixed value of 100 times the maximum field value in the data for the weight in our experiments. Similarly to [14], the scalar field is always normalized to the geometry range before actual simplification begins.

Because we only keep a small set of tetrahedra in memory, we do not know the entire mesh connectivity. Thus, we keep the boundary between the set of tetrahedra that are currently in memory and all remaining elements —tetrahedra that have not yet been read or that have been output—fixed to ensure that the simplified mesh is crack-free. We call this boundary the *stream boundary*, which consists entirely of faces from the interior of the mesh. We can identify the stream boundary faces as they are read in by utilizing the finalization information stored in the streaming mesh. A face of the current in-core mesh is part of the stream boundary if none of its three vertices are finalized. We disallow collapsing any edge that has one or both vertices on the stream boundary.

Due to the stream boundary constraint, if we read in one portion of the mesh, simplify it, and write it out to disk in one phase, our output mesh will have unsimplified areas along the stream boundaries. This results in an approximation that is oversimplified in areas and undersimplified in others. To avoid this problem, we follow the algorithm proposed by Wu and Kobbelt [18]. Their algorithm consists of a main loop in which READ, DECIMATE, and WRITE operations are performed in each iteration. The READ operation introduces new elements until it fills the buffer. Next, DECIMATE simplifies the elements in the buffer until either the target ratio is reached or the buffer size is halved. Finally, in their method the WRITE operation outputs the elements with the largest error to file.

As in [20], we improve upon [18], [19] by ensuring, to the extent possible, that the relative order among input elements is preserved in the output stream, with the caveat that tetrahedra whose vertices have not yet been finalized (i.e., are on the input stream boundary) must be delayed. Therefore, the output typically retains the coherence of the input. An error-driven output criterion, on the other hand, can considerably fragment the buffer and split off small "islands" that remain in the buffer for a long time without being eligible for simplification and, thus, unnecessarily clog the stream buffer. Furthermore, such an output stream generally has poor stream qualities, which affects downstream processing. The front width (i.e., number of active vertices), for example, is particularly important for tetrahedral meshes for which each active vertex affects on average four times as many elements as in a triangle mesh and, therefore, more adversely affects memory requirements and processing delay. Furthermore, we relax the requirement that the stream of tetrahedra (triangles) advance in a face (edge) adjacent manner [19], as this is of no particular value to us, and we allow any coherent ordering of mesh elements. Finally, using the more streamable layouts and simpler streaming mesh formats and API from [20], we gain considerably in performance and memory usage over [19].

3.3 Implementation Details

Since our method processes different mesh portions of bounded size sequentially, a statically allocated data structure is more efficient than dynamic allocations, which collectively increase the memory footprint. The size for this buffer should be $O(width)$ depending on the width of the input mesh. However, in practice, we are able to simplify even a 14 million tetrahedra data set using only 20MB of RAM (see Section 4).

In our implementation, we extended Rossignac's corner table [24] for triangle meshes to tetrahedral meshes. The original corner table requires two fixed-size arrays V and O indexed by corners (vertex-cell associations) c, where $V[c]$ references the vertex of c and $O[c]$ references the "opposite" corner of c.

In the case of tetrahedral simplification, the most common query is to find all tetrahedra around a vertex. Therefore, we replace the O array with a link table L of equal size, which joins together all corners of a given vertex in a circular linked list. We additionally store with each vertex an index to one of its corners.

We store the mesh internally as three fixed-size arrays of vertices, tetrahedra, and corners (i.e., the links L). Each vertex contains a pointer to one corner and the quadric error information $(\mathbf{A}, \mathbf{p}, \varepsilon)$ using a parameterization that explicitly represents the vertex geometry and scalar data in $\mathbf{p}$ (see Section 3.4 and Fig. 4). In Fig. 4, only $\mathbf{p}$ and $vidx$ are stored out-of-core, while the rest are computed on-the-fly. This data structure employs 69 bytes per vertex and 37 bytes

TABLE 2
Priority-Queue (P) versus Randomized (R) Approach

Model		Mean	RMS	Max	Time
Torus	P	0.08%	.0.13%	0.56%	0.31s
	R	0.06%	0.10%	0.59%	0.13s
SPX	P	1.37%	2.56%	22.60%	0.52s
	R	1.53%	2.19%	15.96%	0.30s
Torso	P	0.00%	0.02%	1.12%	55.98s
	R	0.00%	0.00%	0.01%	24.11s
Fighter	P	0.06%	0.13%	2.28%	79.03s
	R	0.08%	0.16%	3.46%	29.89s

TABLE 3
Implementation Improvements

Improvements	Time (sec)	Memory (MB)
Initial Implementation	212.95	310
QEF + CG	132.68	282
Multiple Choice + Corner Table	38.35	130
4D Normal, Floats	35.99	78
Final / Binary I/O	22.10	78

per tetrahedra. The quadrics for the tetrahedra are calculated when we read in a new set of tetrahedra and are then distributed to the vertices. For each finalized vertex, we compute boundary quadrics for all incident boundary faces (if any) that have no other finalized vertices and distribute these quadrics to the boundary vertices.

Garland and Zhou [14] use a greedy edge collapse method and maintain a priority queue for the edges ordered by quadric error. Forsaking greediness, we obtain comparable mesh quality by using a multiple choice randomized approach [18], [25] with eight candidates per collapse. There are several advantages of using randomized selection. One is that we no longer need a priority queue or explicit representation of edges. Instead, an edge can be found by randomly picking a tetrahedron and then randomly selecting two of its vertices. Another advantage is that the randomized technique can be further accelerated by exploiting information readily available through our quadric representation (see Section 3.4). Table 2 illustrates the performance of the randomized approach over the priority-queue-based approach. Both algorithms simplified the models to 10 percent of their original resolutions. The randomized results were collected as an average of three runs on the same input with different random seeds. Intuition would suggest that the results generated by the priority-queue approach would have smaller error because it always picks the edge with the smallest error to collapse at each step. However, this is not optimal in many cases. A series of minimal edge collapses can lead to a locked state where the edges with the smallest error cannot be collapsed without flipping tetrahedra. In practice, the problem is more likely to occur in the homogeneous regions of a data set, which contains edges with zero error. The priority-queue approach will greedily simplify this region as much as possible first, leaving it in a locked state. As a result, many neighbor regions (containing edges with small error) may not get simplified because it would violate the flipping constraint. On the other hand, the randomized approach tends to spread the simplification over the whole model, resulting in significantly less locked state. This explains why the maximum error of the Torso data set using the priority queue approach was very high compared to the randomized approach. In general, the randomized approach produces comparable quality to a priority queue, while demonstrating superior performance. Error measurement are explained in greater detail in Section 4.

Before we output a tetrahedron, we must ensure that its four vertices are output first. Once a vertex is output, we mark it as not being collapsible in future iterations. To enhance the performance, we use a lazy deletion scheme, where all vertices and tetrahedra to be deleted are initially marked. At the end of each WRITE phase, we make a linear pass through all vertices and tetrahedra to remove marked elements and compact the arrays. Since we do not allocate additional memory during simplification, keeping deleted vertices and tetrahedra does not increase the memory footprint.

Storing large data sets on disk in ASCII format can adversely affect performance because converting ASCII numbers to an internal binary representation can be surprisingly slow. We have extended the ASCII stream format in Fig. 2 to a binary representation. Because our program spends more than 30 percent of the time on disk I/O, this optimization results in a nonnegligible speedup. For example, on the SF1 data set it improves overall performance by 17 percent.

Since we only maintain a small portion of the mesh in-core, we require a way of mapping global vertex indices to in-core buffer indices. Usually a hash map is used, but, with our low-span breadth-first mesh layout, this hash map can be replaced by a fixed-size array indexed using modular arithmetic. We move occasional high-span vertices that cause "collisions" in this circular array to an auxiliary hash [20].

With all of the optimizations described above, our simplifier can run at high speed without any dynamic memory allocation at runtime. The performance and memory summary can be found in Table 3. The results are for simplifying the Fighter data set (1.4 M tetrahedra) completely in-core on a P4 2.2GHz with 1GB of RAM. Further efficiency improvements relating to quadric error metrics will be discussed in the following section.

3.4 Numerical Issues

Great care has to be taken when working with quadric metrics to ensure numerical stability while retaining efficiency. To minimize quadric errors, a positive semidefinite system of linear equations must be solved, for which numerically accurate but heavy-duty techniques such as singular value decomposition (SVD) [15], [26] and QR factorization [27] have been proposed. However, even constructing, representing, and evaluating quadric errors require that special care be taken. Here, we outline an efficient representation of quadric error functions that leads to numerically stable operations, improved speed, and less storage.

The standard representation [14] of quadric errors is parameterized by $(\mathbf{A}, \mathbf{b}, c)$ and is evaluated as

$$Q(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - 2\mathbf{b}^{\mathrm{T}}\mathbf{x} + c. \quad (1)$$

```
SOLVE(A, x, b, n, κ_max)
 1  r = b − Ax                          negative gradient of Q
 2  p = 0
 3  for k = 1,...,n                     iterate up to n times
 4    s = r^T r
 5    if s = 0 then exit                solution found?
 6    p = p + r/s                       update search direction
 7    q = Ap
 8    t = p^T q
 9    if st ≤ tr(A)/(nκ_max) then exit  insignificant direction?
10    r = r − q/t                       update gradient
11    x = x + p/t                       update solution
```

Fig. 5. Conjugate gradient solver for positive semidefinite systems $\mathbf{Ax} = \mathbf{b}$. On input $\mathbf{x}$ is an estimate of the solution, $n = 4$ is the number of linear equations, and κ_{max} is a tolerance on the condition number. $\mathrm{tr}(\mathbf{A})$ is the trace of $\mathbf{A}$.

Typically, the three terms in this equation are "large," but sum to a "small" value, resulting in a loss of precision. One can show that the roundoff error is proportional to $\|\mathbf{A}\|\,\|\mathbf{x}\|^2$. Furthermore, in addition to this quadric information, it is common to store the vertex position (and field value) $\mathbf{p}$ that minimizes Q separately. Lindstrom [28] suggested an alternative representation that removes this redundancy:

$$Q(\mathbf{x}) = (\mathbf{x}-\mathbf{p})^{\mathrm{T}}\mathbf{A}(\mathbf{x}-\mathbf{p}) + \varepsilon, \tag{2}$$

where $\mathbf{A}$ is the same as in the standard representation and

$$\mathbf{Ap} = \mathbf{b},$$
$$\mathbf{p}^{\mathrm{T}}\mathbf{Ap} + \varepsilon = c.$$

This parameterization $(\mathbf{A}, \mathbf{p}, \varepsilon)$ provides direct access to the minimum quadric error ε and the minimizer $\mathbf{p}$. This not only saves memory, but also results in a more stable evaluation of Q as the roundoff error is now proportional to $\|\,\mathbf{A}\,\|\,\|\,\mathbf{x}-\mathbf{p}\,\|^2$ and we are generally interested in evaluating $Q(\mathbf{x})$ near its minimum $\mathbf{p}$ as opposed to near the origin. Another significant benefit of this representation is that it provides a lower bound $\varepsilon_i + \varepsilon_j$ on $Q_i + Q_j$ when collapsing two vertices v_i and v_j. Using randomized edge collapse [18], we can thus often avoid minimizing $Q_i + Q_j$ if the lower bound already exceeds the smallest quadric error found so far. In this paper, this representation is used explicitly to speed up the algorithm, reduce in-core storage, and improve numerical robustness rather than as a means of compressing quadric information for out-of-core storage.

Our quadric representation also lends itself to an efficient and numerically stable iterative linear solver. To handle ill-conditioned matrices $\mathbf{A}$, we have adapted the well-known conjugate gradient (CG) method [29] to work on semidefinite matrices (see Fig. 5). As in SVD, we provide a tolerance κ_{max} on the condition number $\kappa(\mathbf{A})$ and preempt the iterative solver when all remaining conjugate directions are deemed "insignificant" for reducing Q. The effect of this is similar to zeroing small singular values in SVD. Using our quadric representation, we conveniently initialize the CG solver with the guess $\mathbf{x} = (\mathbf{p}_i + \mathbf{p}_j)/2$. Whereas CG methods are typically used to quickly approximate solutions to very large systems using only a few iterations, our method can be considered "direct" in the sense that we solve for each of the n (i.e., four) components, although in the Krylov basis rather than in the Euclidean basis as done by the Cholesky method. We never require more than n iterations and only in the rank-deficient case do we perform fewer than n iterations.

A final word of caution: The computation of generalized quadrics presented in [14] computes $\mathbf{A} = \mathbf{I} - \mathbf{N}$, whose null space $\mathrm{null}(\mathbf{A}) = \mathrm{range}(\mathbf{N})$ is spanned by the tetrahedron, via subtraction, which, due to roundoff error, can leave $\mathbf{A}$ indefinite, i.e., with one or more negative eigenvalues. This causes Q to have a "saddle" shape with no defined minimum and can cause numerical instability. Instead, we compute a 4D "volume normal" using a generalization of the 3D cross product to 4D.

$$\mathbf{n} = det\begin{pmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \mathbf{e}_4 \\ \nu 1_x & \nu 1_y & \nu 1_z & \nu 1_s \\ \nu 2_x & \nu 2_y & \nu 2_z & \nu 2_s \\ \nu 3_x & \nu 3_y & \nu 3_z & \nu 3_s \end{pmatrix},$$

where $\mathbf{e_i}$ is the i-column of the 4×4 identity matrix and $\mathbf{v1}$, $\mathbf{v2}$, and $\mathbf{v3}$ are three vectors from one of the tetrahedron's vertices to the others. The outer product of this normal with itself gives a positive semidefinite $\mathbf{A}$ for a tetrahedron.

Because of our attention to numerical stability, with $\kappa_{max} = 10^4$ we are able to use single precision floating-point throughout our simplifier, even for the largest meshes. Since the 4D quadric information requires 15 scalars per vertex, this saves considerable memory and improves the speed.

TABLE 4
Performance of Linear Solvers

Dataset	QR	Cholesky	CG
SPX	0.57s	0.30s	0.35s
Blunt	16.47s	9.88s	7.72s
Fighter	46.11s	30.57s	29.89s

4 Results

4.1 Stability and Error Analysis

We have described a CG method for solving the linear equations that arise when minimizing the quadric error. The choice of solver is important because degenerate tetrahedra and regions of near-constant field value can cause singularity. For testing purposes, we constructed a data set by subdividing a tetrahedron into hundreds of smaller tetrahedra by linear interpolating the vertices and field data. Obviously, these small tetrahedra all lie on the hyperplane spanned by the original tetrahedron, thus they are solutions to the linear equations. We then picked a solution as a target for each collapsed edge. We experimented with several linear solvers, as shown in Table 4. The results are for simplifying the Fighter data set (1.4 M tetrahedra) completely in-core. We experimented with Cholesky factorization, the least square QR factorization [29], and CG.

Cholesky with pivoting provides stable solutions for solving $\mathbf{Ax} = \mathbf{b}$ if $\mathbf{A}$ is positive definite. However, in order

TABLE 5
In-Core and Streaming Simplification Results

Dataset	Number of Tets Input	Number of Tets Output	Time (sec)	Max RAM (MB)	Max Field Error (%)	RMS Field Error (%)	Max Surface Error (%)	RMS Surface Error (%)
In-core								
Torso	1,082,723	108,271	14.88	57	0.012	0.000884	0.120	0.013360
Fighter	1,403,504	140,348	15.46	78	4.845	0.280266	0.038	0.000352
Rbl	3,886,728	388,668	59.10	212	0.020	0.002574	0.025	0.000055
Mito	5,537,168	553,711	47.13	285	0.045	0.007355	0.001	0.000008
SF1	13,980,162	1,398,013	191.69	709	5.626	0.262335	0.036	0.000811
Streaming								
Torso	1,082,723	108,270	19.07	20	0.019	0.000879	0.161	0.001226
Fighter	1,403,504	140,345	20.87	20	4.549	0.299081	0.102	0.000470
Rbl	3,886,728	388,671	95.54	20	0.025	0.002833	0.036	0.000089
Mito	5,537,168	553,716	73.58	20	0.045	0.007614	0.044	0.000009
SF1	13,980,162	1,398,012	246.15	20	23.287	0.472869	0.169	0.004150
SF1	13,980,162	1,398,017	244.42	50	5.834	0.315583	0.050	0.001394

to solve this linear system when $\mathbf{A}$ has rank-deficiency (the semidefinite case), we must solve the underconstrained least square problem. Unfortunately, solving this using normal equations requires us to be able to perform Cholesky factorizations on matrices with arbitrary dimensions less than 4×4. This defeats the purpose of optimizing our simplification for working only with 4×4 matrices. Thus, our implementation uses SVD to handle rank-deficiency matrices detected by the Cholesky method. As a result, this method yields the fastest solution when $\mathbf{A}$ is positive-definite, but it becomes slower compared to CG when handling rank-deficient matrices.

Like Cholesky, QR does not handle the problem of rank-deficiency. Nevertheless, the implementation for solving the least square problem using QR factorization is much simpler than using Cholesky with normal equations since it does not explicitly require a general representation of matrices with arbitrary dimensions. We use the least square version of QR as suggested in [29].

Using all of our solvers, we were able to simplify our subdivided tetrahedron to its original shape with small error in both field and geometry. To compare the correctness of their solutions, we recorded *norm-2* residuals of computed solutions to 15,000 linear systems using all three methods while simplifying the fighter data set. Denote by e_{qr}, e_{ch}, and e_{cg} the sum of all norm-2 residuals of computed solutions $\hat{\mathbf{x}}$ (i.e., $\| \mathbf{Ab} - \hat{\mathbf{x}} \|_2$) using QR, Cholesky, and CG, respectively. Given $e_{avg} = \frac{1}{3}(e_{qr} + e_{ch} + e_{cg})$, relative errors of solutions computed by QR, Cholesky, and CG can be computed as $re_{\{qr,ch,cg\}} = e_{\{qr,ch,cg\}}/e_{avg}$. Our experiment found re_{qr} to have the smallest error with *96 percent*, followed by re_{ch} with *99 percent*. Our CG approach obtained a comparable result with re_{ch} of *105 percent*.

Overall, QR gives the most optimal solution in terms of error, but it is approximately twice as slow as the others. On the other hand, while Cholesky is a good choice for both efficiency and accuracy, its implementation is quite complicated. Therefore, we chose CG over the others for its simplicity and performance while still maintaining comparatively optimal solutions.

To estimate the error in the simplified mesh we use two different methods. The first method is to measure the error on the surface boundary of the mesh using the tool *Metro* [30]. The second method is to measure the error in the field data using a similar approach to Cignoni et al. [11]. We sample the domain of the simplified data set at points inside the domain of the original one. These points are not only vertices of the input, but also interpolated ones inside each tetrahedron. The error is then computed by the differences between their scalar values. Our implementation differs because it ignores points outside the domain of the simplified mesh since these points become part of the surface boundary error. Table 5 shows these measured error estimations. Field error percentages are in relation to the range of the field and surface error percentages are in relation to the bounding box diagonal. Fig. 6 shows an example of the quality of the resulting field and Fig. 7 shows an example of the quality of the resulting surface.

4.2 Performance

All timing results were generated on a 3.2 GHz Pentium 4 machine with 2.0 GB RAM. For the streaming experiments, we limit the operating system to only 64 MB RAM by using the Linux bootloader. Table 5 shows the results of simplifying a collection of data sets to 10 percent of their original size using our streaming algorithm and the same

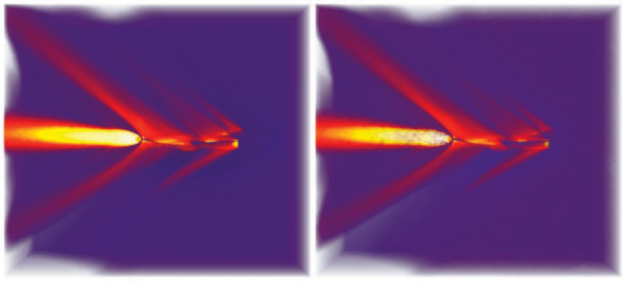

Fig. 6. Volume Rendered images of the Fighter data set show the preservation of scalar values. The original data set is shown on the left (1,403,504 tetrahedra) and the simplified version is shown on the right (140,348 tetrahedra).

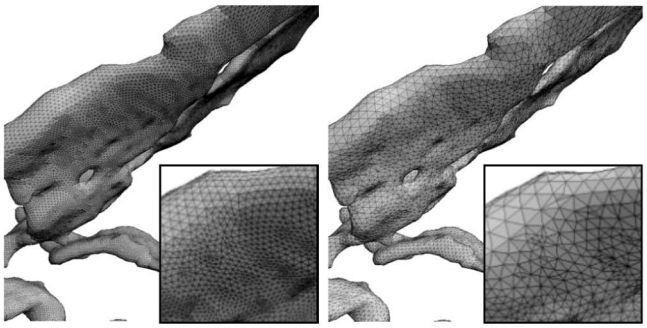

Fig. 7. Views of the mesh quality on the surface of the Rbl data set. The original data set is shown on the left (3,886,728 tetrahedra) and the simplified version is shown on the right (388,637 tetrahedra).

implementation optimized for in-core execution. Laying out the meshes in a stream efficient manner is a one-time operation and can be performed in-core for all the data sets we tested. Even the largest data set (14 million tetrahedra) required only about 40 minutes to lay out using our Breadth-First approach.

We were able to achieve streaming simplification with only a slight increase in time and error compared to an in-core implementation. The streaming technique has the advantage of a smaller memory footprint. With our algorithm, we were able to simplify 14 million tetrahedra while only using 20 MB RAM. Due to the large size of the SF1 data set, certain parts of the stream were not able to be simplified accurately, resulting in a larger error. By increasing the memory slightly, the quality of the simplification is greatly improved and approaches the in-core quality. This behavior is not due to the randomization algorithm since the large buffer size always produces better quality outputs even with random seeds. Instead, the quality is improved because each set of candidates has a wider range to select their targets. Consider a set of expensive edges that are larger than the buffer size; any edge collapse will result in a large error no matter how random the target is. However, if we increase the buffer size such that the buffer is larger than the expensive edges, randomized edge collapses will take those edges that are not so expensive into account, thus improving the quality of the output.

4.3 Large-Scale Experiment

Although extremely large meshes exist, it is difficult to obtain unclassified access to them. To stress our algorithm on current PC hardware and to demonstrate the scalability of the technique, we performed streaming simplification on a huge fluid dynamics data set on a Xeon 3.0GHz machine. The data set was created from sampling slices of a $2,048^3$ simulation and consists of over one billion tetrahedra that use 18 GB of disk space when stored in the binary format. The tetrahedra were laid out in the order in which they were sliced, which is comparable to sorting by axis. An in-core approach to simplifying this data set would require a machine with 64 GB RAM. We were able to simplify the data using only 829 MB RAM to 12 million tetrahedra (1.2 percent) in 10 hours on our test machine. Because error estimations were not possible with the entire data set in-core, we computed the field error on subregions of the mesh separately to verify the results. In the regions that were measured, there was 10.85 percent maximum and 1.57 percent RMS field error. Fig. 8 shows an isosurface extracted from the original and simplified fluid dynamics data set.

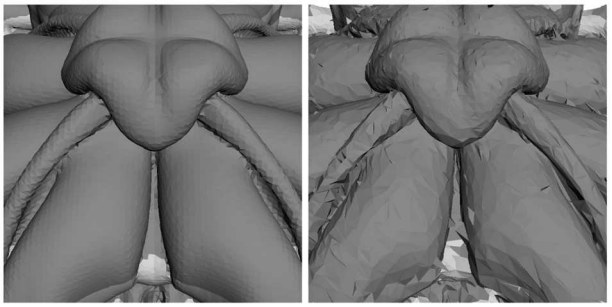

Fig. 8. Isosurfaces of the fluid dynamics data set. A very small portion of the isosurfaces is shown for the original data set of over a billion tetrahedra (left) and the simplified data set of only 12 million tetrahedra (right). The isosurfaces are shown up close using flat shading to enhance the details of the resulting surface. Our algorithm allows extensive simplification (almost 1 percent) with negligible numerical error (1.57 percent RMS) for the fluid dynamics data set, which is too large to simplify with conventional approaches.

5 Discussion

The use of streaming meshes for simplification reduces the memory footprint of a large mesh considerably. We improved on the algorithm of Wu and Kobbelt [18] by preserving the stream order of the mesh between input and output. A direct comparison with their algorithm shows that our method consistently achieves a lower width, e.g., 9 percent versus 59 percent on the Fighter data set, and span, e.g., 45 percent versus 98 percent, without reducing the approximation quality. In addition, with the optimizations that we employ to our data structures, we have been able to simplify up to 14 million tetrahedra while using only 20 MB RAM. Only the smallest data sets (Torso and Fighter) could be simplified using our implementation of Wu and Kobbelt's algorithm.

Apart from providing a streaming algorithm that operates on meshes of arbitrary size, we also described speed and stability optimizations that improve the performance of tetrahedral simplification. Our quadric representation improves linear solver performance. In addition, our adapted conjugate gradient method and the use of "volume normals" for tetrahedra reduce the numerical errors and allow the use of single precision floating-point numbers. By using a binary format, we improve on storage and speed up I/O. Finally, through the use of a breadth-first mesh layout, we have improved the width and the span, which enables the use of a fixed-size circular array instead of a hash table. Collisions can occur, but it only happens when the buffer size is smaller than the span size. Even in the case of collisions, only a simple primary hash function, e.g., modulo, is needed. Thus, it can also avoid linked-lists with dynamic memory allocation by using an auxiliary hash table. With these changes, we have improved the speed of our simplification method by an order of magnitude over our initial implementation of Garland and Zhou's quadric-based simplification [14].

Due to the efficiency of our algorithm, we easily handle the largest data sets we obtained. In our experimental results, our streaming algorithm simplifies about 60K tetrahedra per second and achieves high quality results. Given a machine with 2.0 GB of RAM, the in-core implementation of our algorithm could handle approximately 35 million tetrahedra. Using our streaming method, we were able to simplify a data set consisting of almost one billion tetrahedra to one percent using only about 800 MB of RAM with very small error. The maximum bound of the streaming algorithm is only constrained by the front width of the mesh, which we have shown to be very small when using a good mesh layout.

6 Conclusions and Future Work

We have presented a streaming technique for simplifying tetrahedral meshes of arbitrary size. We describe several methods for laying out a tetrahedral mesh on disk in a coherent, I/O-efficient format. We also show an analysis of these layouts and the effects they have on the final simplified mesh. We provide a technique for simplifying small portions of the mesh in memory while obtaining a smooth simplification over the entire mesh. The simplification occurs in one pass, preserves mesh topology and scalar information, requires little memory, and runs quickly. We also provide optimizations to traditional simplification data structures that improve speed and efficiency. We present a linear solver that improves stability and speed of quadric-based simplification. Finally, we provide stability and error analysis of our algorithm with results for specific examples that show the time and memory required for processing.

Because the generalized quadric error works on a variety of data types, an interesting extension would be to attempt to handle meshes consisting of other types, e.g., hexahedra. Finally, it would be interesting to take advantage of the coherent streaming tetrahedral format to perform fast, out-of-core isosurface extraction.

Acknowledgments

This work was performed under the auspices of the US Department of Energy (DOE) by Lawrence Livermore National Laboratory (LLNL) under contract no. W-7405-Eng-48. The authors would like to thank Jason Shepherd for the Rbl and Mito data sets and Louis Bavoil for the error analysis code. They also thank NASA for the Langley Fighter data set and Rob MacLeod at University of Utah for the Torso data set. Huy T. Vo is funded by the US National Science Foundation (NSF) Research Experiences for Undergraduates (REU) program. Steven P. Callahan is supported by the US DOE under the ASC VIEWS program. Cláudio T. Silva is partially supported by the DOE under the ASC VIEWS program, the NSF (grants CCF-0401498, EIA-0323604, OISE-0405402, IIS-0513692, CCF-0528201), an IBM Faculty Award, and a University of Utah Seed Grant.

References

[1] A. Mascarenhas, M. Isenburg, V. Pascucci, and J. Snoeyink, "Encoding Volumetric Grids for Streaming Isosurface Extraction," *Proc. Int'l Symp. 3D Data Processing, Visualization, and Transmission,* 2004.

[2] M. Isenburg, P. Lindstrom, and J. Snoeyink, "Streaming Compression of Triangle Meshes," *Proc. 2005 Eurographics/ACM SIGGRAPH Symp. Geometry Processing,* pp. 111-118, 2005.

[3] I.J. Trotts, B. Hamann, K.I. Joy, and D.F. Wiley, "Simplification of Tetrahedral Meshes," *Proc. IEEE Visualization Conf. '98,* pp. 287-295, 1998.

[4] I.J. Trotts, B. Hamann, and K.I. Joy, "Simplification of Tetrahedral Meshes with Error Bounds," *IEEE Trans. Visualization and Computer Graphics,* vol. 5, no. 3, pp. 224-237, July-Sept. 1999.

[5] A. Van Gelder, V. Verna, and J. Wilhelms, "Volume Decimation of Irregular Tetrahedral Grids," *Computer Graphics Int'l,* pp. 222-230, 1999.

[6] D. Uesu, L. Bavoil, S. Fleishman, J. Shepherd, and C.T. Silva, "Simplification of Unstructured Tetrahedral Meshes by Point-Sampling," *Volume Graphics,* pp. 157-238, 2005.

[7] H. Hoppe, "Progressive Meshes," *Proc. ACM SIGGRAPH '96,* pp. 99-108, 1996.

[8] J. Popović and H. Hoppe, "Progressive Simplicial Complexes," *Proc. 24th Ann. Conf. Computer Graphics and Interactive Techniques,* pp. 217-224, 1997.

[9] O.G. Staadt and M.H. Gross, "Progressive Tetrahedralizations," *Proc. IEEE Visualization Conf. '98,* pp. 397-402, 1998.

[10] P. Chopra and J. Meyer, "TetFusion: An Algorithm for Rapid Tetrahedral Mesh Simplification," *Proc. IEEE Visualization Conf. '02,* pp. 133-140, 2002.

[11] P. Cignoni, D. Constanza, C. Montani, C. Rocchini, and R. Scopigno, "Simplification of Tetrahedral Meshes with Accurate Error Evaluation," *Proc. IEEE Visualization Conf. '00,* pp. 85-92, 2000.

[12] M. Garland and P.S. Heckbert, "Surface Simplification Using Quadric Error Metrics," *Proc. ACM SIGGRAPH '97,* pp. 209-216, 1997

[13] V. Natarajan and H. Edelsbrunner, "Simplification of Three-Dimensional Density Maps," *IEEE Trans. Visualization and Computer Graphics,* vol. 10, no. 5, pp. 587-597, 2004.

[14] M. Garland and Y. Zhou, "Quadric-Based Simplification in Any Dimension," *ACM Trans. Graphics,* vol. 24, no. 2, Apr. 2005.

[15] P. Lindstrom, "Out-Of-Core Simplification of Large Polygonal Models," *Proc. 27th Ann. Conf. Computer Graphics and Interactive Techniques,* pp. 259-262, 2000.

[16] J. Rossignac and P. Borrel, *Geometric Modeling in Computer Graphics,* chapter on multiresolution 3D approximations for rendering complex scenes, pp. 455-465. Springer-Verlag, 1993.

[17] P. Lindstrom and C.T. Silva, "A Memory Insensitive Technique for Large Model Simplification," *Proc. IEEE Visualization Conf. '01,* pp. 121-126, 2001.

[18] J. Wu and L. Kobbelt, "A Stream Algorithm for the Decimation of Massive Meshes," *Proc. Graphics Interface Conf. '03,* pp. 185-192, 2003.

[19] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink, "Large Mesh Simplification Using Processing Sequences," *Proc. IEEE Visualization Conf. '03,* pp. 465-472, 2003.

[20] M. Isenburg and P. Lindstrom, "Streaming Meshes," *Proc. IEEE Visualization Conf. '05,* pp. 231-238, Oct. 2005.

[21] M. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," *Proc. IEEE Visualization Conf. '97,* pp. 235-244, 1997.

[22] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno, "External Memory Management and Simplification of Huge Meshes," *IEEE Trans. Visualization and Computer Graphics,* vol. 9, no. 4, pp. 525-537, Oct.-Dec. 2003.

[23] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-Oblivious Mesh Layouts," *ACM Trans. Graphics,* vol. 24, no. 3, pp. 886-893, 2005.

[24] J. Rossignac, "3D Compression Made Simple: Edgebreaker with Zip&Wrap on a Corner-Table," *Proc. Int'l Conf. Shape Modeling & Applications,* p. 278, 2001.

[25] B. Cutler, J. Dorsey, and L. McMillan, "Simplification and Improvement of Tetrahedral Models for Simulation," *Proc. 2004 Eurographics/ACM SIGGRAPH Symp. Geometry Processing,* pp. 93-102, 2004.

[26] L.P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel, "Feature-Sensitive Surface Extraction from Volume Data," *Proc. ACM SIGGRAPH '01,* pp. 57-66, 2001.

[27] T. Ju, F. Losasso, S. Schaefer, and J. Warren, "Dual Contouring of Hermite Data," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 339-346, July 2002.

[28] P. Lindstrom, "Out-of-Core Construction and Visualization of Multiresolution Surfaces," *Proc. ACM Symp. Interactive 3D Graphics,* pp. 93-102, 2003.
[29] G.H. Golub and C.F. Van Loan, *Matrix Computations,* third ed. Johns Hopkins Univ. Press, 1996.
[30] P. Cignoni, C. Rocchini, and R. Scopigno, "Metro: Measuring Error on Simplified Surfaces," *Computer Graphics Forum,* vol. 17, no. 2, pp. 167-174, 1998.

Huy T. Vo received the BS degree in computer science from the University of Utah in 2005, where he is currently pursuing the PhD degree in computer science. Currently, he is working as a research assistant in the Scientific Computing and Imaging Institute at the University of Utah under the direction of Dr. Cláudio T. Silva. His research interests include optimization and out-of-core algorithms, scientific visualization, and visualization systems.

Steven P. Callahan received the BS degree in computer science from Utah State University in 2002 and the MS degree in computational engineering and science from the University of Utah in 2005. He is currently pursuing the PhD degree in computer science at the University of Utah, where he works as a research assistant in the Scientific Computing and Imaging Institute. His research interests include graphics, large-scale scientific computing, visualization, and data management for visualization systems.

Peter Lindstrom received the BS degrees in computer science, mathematics, and physics from Elon College in 1994, and the PhD degree in computer science from the Georgia Institute of Technology in 2000. He is currently a computer scientist at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research interests include mesh simplification and compression, multiresolution modeling, geometry processing, and scientific visualization. He is a member of the IEEE.

Valerio Pascucci received the Laurea degree in ingegneria elettronica from the University "La Sapienza," Rome, Italy, in 1993 and the PhD degree in computer science from Purdue University in 2000. He joined the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory in May 2000, where he is currently a research scientist and project leader. His research interests include progressive out-of-core techniques, analysis of scientific data sets, multiresolution geometric modeling, solid modeling, and computational geometry. He is a member of the IEEE.

Cláudio T. Silva received the BS degree in mathematics from the Federal University of Ceara, Brazil, in 1990, and the PhD degree in computer science from the State University of New York at Stony Brook in 1996. He is an associate professor of computer science and a faculty member of the Scientific Computing and Imaging (SCI) Institute at the University of Utah. Before joining Utah in 2003, he worked in industry (IBM and AT&T), government (Sandia and LLNL), and academia (Stony Brook and OGI). He has coauthored more than 80 technical papers and seven patents, primarily in visualization, geometric computing, and related areas. He served as papers cochair for the IEEE Visualization Conference in 2005 and 2006. In 2005, he received an IBM Faculty Award. He is a member of the IEEE.