A Memory Insensitive Technique for Large Model Simplification

Peter Lindstrom* LLNL

Cláudio T. Silva† AT&T

Abstract

In this paper we propose three simple, but significant improvements to the OoCS (Out-of-Core Simplification) algorithm of Lindstrom [20] which increase the quality of approximations and extend the applicability of the algorithm to an even larger class of compute systems.

The original OoCS algorithm has memory complexity that depends on the size of the output mesh, but no dependency on the size of the input mesh. That is, it can be used to simplify meshes of arbitrarily large size, but the complexity of the output mesh is limited by the amount of memory available. Our first contribution is a version of OoCS that removes the dependency of having enough memory to hold (even) the simplified mesh. With our new algorithm, the whole process is made essentially independent of the available memory on the host computer. Our new technique uses disk instead of main memory, but it is carefully designed to avoid costly random accesses.

Our two other contributions improve the quality of the approximations generated by OoCS. We propose a scheme for preserving surface boundaries which does not use connectivity information, and a scheme for constraining the position of the "representative vertex" of a grid cell to an optimal position inside the cell.

CR Categories: E.5 [Files]: Sorting; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Surface and object representations.

Keywords: polygonal surface simplification, large data, out-of-core algorithms, external sorting, quadric error metrics.

1 INTRODUCTION

In recent years there has been a rapid increase in the raw size of polygonal datasets. Several technological trends are contributing to this effect, such as the development of high-resolution 3D scanners, and the need to visualize ASCI-size (Accelerated Strategic Computing Initiative) datasets. A useful paradigm for visualizing large datasets is to generate levels of detail. Over the last decade, there has been substantial research in designing algorithms for generating level-of-detail approximations of triangle meshes. In this paper, our focus is on algorithms which have low memory complexity.

A simplification algorithm receives an input mesh of complexity n, and outputs a mesh of complexity m (where $m < n$). Often, the user sets the target size of the output, and the algorithm attempts to minimize the overall error of the approximation. One important aspect of the design of a surface simplification algorithm is its memory usage. In general, different algorithms have different main memory dependencies on n and m. For different applications, it is useful to have algorithms which are memory efficient with respect to n or m (but ideally both). The memory dependency on n affects the usefulness of a given algorithm in the sense that it limits the size of models that can be simplified.

In general the memory requirement of a given algorithm grows with both m and n (for exceptions, see e.g. [29, 30]). The dependency on m has direct implications on the maximum accuracy of the approximation. As an example, an efficient terrain simplification algorithm is presented in [13], whose memory complexity is analyzed to be $3n + 192m$ bytes, where n and m are the number of vertices in the input and output, respectively. In order to generate a high-quality approximation with one eighth of the input points, i.e., $m = \frac{1}{8}n$, one would need to have $27n$ bytes of memory, or nine times as much as the size of the input. Often, the memory complexity is much higher on both n and m (e.g., [21] uses $160n$ bytes for general surface simplification), and generating approximations of large datasets is usually quite hard.

The OoCS algorithm proposed by Lindstrom [20] is a big step forward in that it has no dependency on n, thus allowing for simplification of extremely large datasets. One contribution of our work is to remove the main memory dependency on m from OoCS, thus allowing for an arbitrarily accurate approximation of an arbitrarily large dataset. Our new algorithm, OoCSx, uses *constant memory*, no matter how large the dataset or approximation error.

One might argue that the ability to produce simplified models that are still too large to represent in-core is of little practical value, since the main reason for simplifying the model in the first place is to reduce its complexity to something more manageable. However, we see several important uses of our new algorithm. First, in many situations it is not known beforehand how much RAM will be available on the client machine on which the simplified mesh is to be used, as is generally the case with multi-level-of-detail datasets provided through data repositories. Second, OoCS does not provide a mechanism for specifying the exact size m of the simplified model, and trial and error may be necessary to find a grid resolution that leads to a detailed simplification that, along with the auxiliary data structures used in OoCS, fits in-core. Our memory insensitive algorithm, on the other hand, is able to finish and output a simplified model regardless of the grid resolution. Third, many applications demand a strict error bound, in which case trading memory for mesh accuracy is not a practical option. As we shall see, even when an explicit error bound is not given, the mesh may be so geometrically complex that the most detailed simplification to fit in-core is of unacceptable visual quality. Finally, our work nicely complements the recent trend of developing efficient out-of-core scientific visualization techniques (see, e.g., [7, 11, 32]). With tools like these in hand, further out-of-core processing of a simplified mesh becomes practical.

Our new technique uses disk instead of main memory. In fact, OoCSx generally needs more disk space than OoCS needs main memory. On the other hand, disk is often much cheaper and more readily available than random access memory. The naive use of disk has the potential for considerable slowdown (as in the case of operating system paging). Our algorithm is carefully designed to avoid

*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 7000 East Avenue, L-560, Livermore, CA 94551, USA; pl@llnl.gov.

†AT&T Labs-Research, 180 Park Avenue, Room D265, PO Box 971, Florham Park, NJ 07932, USA; csilva@research.att.com.

random accesses, thus achieving simplification speeds which, although slower than OoCS, are still quite practical. Our experiments show that OoCSx is typically between two to five times slower than OoCS, while using constant main memory. However, when insufficient main memory is available for OoCS to store the simplified model, OoCSx runs faster. Of course, for large enough models, OoCS is not able to finish at all.

Because OoCS does not make use of connectivity information, it has no way of detecting whether an edge is a boundary edge or not. As a consequence, boundaries are generally poorly preserved by OoCS. We propose a technique for preserving boundaries that does not use any connectivity information. Finally, we sketch a technique for enforcing maximum errors, which constrains the optimal cluster representative to lie inside its grid cell while minimizing the approximation error.

2 RELATED WORK

Polygonal simplification has been a hot topic of research over the last decade, with a vast number of published algorithms. Many of the early simplification algorithms were designed to handle modest size datasets of a few tens of thousands of triangles. Recent improvements in scanning and storage technology, however, have lead to datasets as large as billions of triangles [19, 23]. As a result, a number of methods, particularly for out-of-core visualization, have been proposed for coping with models that are too large to fit in main memory, e.g. [3, 5–8, 17, 24, 28, 31, 32].

Rossignac and Borrel proposed one of the earliest simplification algorithms [26]. Their algorithm partitions space into cube-like cells from a uniform rectilinear grid, and replaces all mesh vertices within a grid cell by a single representative vertex. While simple and fast, their method produces rather low quality meshes, in part due to the simple vertex positioning scheme used in their original algorithm. Lindstrom's OoCS algorithm [20] is also based on vertex clustering on a uniform grid, but has a lower time and memory complexity, and uses a quadric error metric to improve the mesh quality. This method was recently extended by Shaffer and Garland [27], who make two passes over the input mesh. During the first pass, the surface is analyzed and an adaptive (instead of uniform) partitioning of space is made. Using this approach, a larger number of irregular grid cells (and thus samples) can be allocated to the more detailed portions of the surface. However, their algorithm requires more RAM than OoCS in order to maintain a BSP-tree and additional quadric information in-core.

Bernardini et al. describe a radically different approach to out-of-core simplification [4]. Their method splits the model up into separate patches that are small enough to be simplified individually in-core using a conventional simplification algorithm. Special care has to be taken along the patch boundaries. A similar technique was proposed by Hoppe for creating hierarchical levels of detail for height fields [15], which was later generalized by Prince to arbitrary meshes [25]. While conceptually simple, the time and space overhead of partitioning the model and later stitching it together adds to an already expensive in-core simplification process, rendering such a method less suitable for simplifying very large meshes.

El-Sana and Chiang [10] propose an external-memory algorithm to support view-dependent simplification of datasets that do not fit in main memory. Similar to [4, 15, 25], they segment the model into sub-meshes that can be simplified independently and later merged in a preprocessing phase. The segmentation and stitching are made simple by ensuring that edges are collapsed in edge-length order, and guaranteeing that sub-mesh boundary edges are longer than interior edges. During run-time, only the portions of the view-dependence tree that are necessary to render the given level of detail are kept in main memory.

3 OUT-OF-CORE SIMPLIFICATION

In order to describe our new simplification algorithm, we will first provide a brief review of OoCS. For full details see [20]. The input to the algorithm is a set of triangles, stored as triplets of vertex coordinates in a file, and the resolution of a three-dimensional grid. (See the appendix for a disk-based technique on how to transform from indexed meshes to this *dereferenced* format.) The algorithm, which is loosely based on the clustering algorithm by Rossignac and Borrel [26], computes for each cluster grid location a representative vertex. (A set of vertices constitute a "cluster" if they all lie inside the same grid cell.) The position of the representative vertex is chosen so as to minimize the *quadric error* [14] measured with respect to the triangles in the cluster. For each triangle $t = (\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$, an associated *quadric matrix* $\mathbf{Q}_t$ is computed:

$$\mathbf{Q}_t = \bar{\mathbf{n}}_t \bar{\mathbf{n}}_t^{\mathsf{T}} \tag{1}$$

$$\bar{\mathbf{n}}_t = \begin{pmatrix} \mathbf{x}_1^t \times \mathbf{x}_2^t + \mathbf{x}_2^t \times \mathbf{x}_3^t + \mathbf{x}_3^t \times \mathbf{x}_1^t \\ -[\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t] \end{pmatrix} \tag{2}$$

where $\bar{\mathbf{n}}_t$ is a 4-vector made up of the area-weighted normal of t and the scalar triple product of its three vertices.[1] $\mathbf{Q}_t$ is then distributed to the clusters associated with each of t's three vertices by adding $\mathbf{Q}_t$ to their quadric matrices.

After summing up all per-triangle quadric matrices in a cluster, we obtain a quadric matrix $\mathbf{Q}_S$ that contains shape information for the piece of surface passing through the grid cell:

$$\mathbf{Q}_S = \sum_t \mathbf{Q}_t = \begin{pmatrix} \mathbf{A} & -\mathbf{b} \\ -\mathbf{b}^{\mathsf{T}} & c \end{pmatrix} \tag{3}$$

Using this decomposition of $\mathbf{Q}_S$, the 3×3 linear system $\mathbf{Ax} = \mathbf{b}$ is solved for the "optimal" representative vertex position $\mathbf{x}$ that minimizes the quadric error. That is, $\mathbf{x}$ is the position that minimizes the sum of squared volumes of the tetrahedra formed by $\mathbf{x}$ and the triangles in the cell. When clustering vertices together, the majority of triangles degenerate into edges or points and can be discarded, thereby reducing the complexity of the model.

3.1 OoCSx

The main idea of our modification of the OoCS algorithm is to eliminate the list of occupied clusters which OoCS allocates in main memory and uses for keeping track of their quadrics. Instead of directly computing quadrics, OoCSx computes $\bar{\mathbf{n}}_t$ for the current triangle t and outputs this information to a file, three times for each of the three vertices, together with the information for what grid cell the vertex belongs to. At the same time, we also output another file which contains the non-degenerate triangles (those triangles that have vertices in three different clusters) represented as indices to the grid cells. Then, we externally sort the file containing the vectors $\bar{\mathbf{n}}_t$ using the grid locations as the primary key. After this, all the information related to one grid cell is placed contiguously in the file. By scanning it, it is possible to compute the quadrics and the optimum location of the representative vertex for a particular grid cell. After the vertices in the simplified mesh have been computed, we are left with the task of associating the grid cell references in the triangle file with vertex representatives. This step is described in more detail below.

Here are the steps of OoCSx in detail:

(1) **Read triangles, compute quadric information for later use**. For each triangle $t = (\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$ in the input mesh, we compute $\bar{\mathbf{n}}_t$ (Equation 2). Note that we do not compute any

[1] In this paper, $\bar{\mathbf{a}}$ denotes a 4-vector, and $\hat{\mathbf{a}}$ is a unit-length 3-vector.

quadric matrices at this point. For each vertex $\mathbf{x}_i^t$ of t, we also determine the grid location $G(\mathbf{x}_i^t)$ (as an integer ID) that the vertex will be mapped to. As triangles are read, we output this information to two files:

- A "plane equation" file, which contains 3 entries for each triangle, one for each vertex. Each entry is of the form: $\langle G(\mathbf{x}_i^t), \bar{\mathbf{n}}_t \rangle$. Using 32-bit integers to represent G and 32-bit floats for $\bar{\mathbf{n}}_t$, this file takes 20 bytes of disk per entry.
- A "triangle cluster" file, which contains records of the form $\langle G(\mathbf{x}_1^t), G(\mathbf{x}_2^t), G(\mathbf{x}_3^t) \rangle$. Each record takes 12 bytes, and is written only when $G(\mathbf{x}_1^t) \neq G(\mathbf{x}_2^t) \neq G(\mathbf{x}_3^t)$.

(2) **Sort "plane equation" file using G as the sort key**. This step is performed using an external sort algorithm, which is discussed below.

(3) **Compute quadrics and output optimal vertices**. In order to find the representative vertex for a given cluster, we need to sum up all the quadrics that contribute to its position. Because the "plane equation" file has been sorted on cluster IDs (i.e., G), all the vectors $\bar{\mathbf{n}}_t$ that contribute to a given grid cell are together in the file. That is, in a single scan, we can sum all the $\bar{\mathbf{n}}_t \bar{\mathbf{n}}_t^\mathsf{T}$ into a quadric matrix $\mathbf{Q}_S$, which is used to compute the representative vertex position.[2]

As we find the representative vertex $\mathbf{x}$ for a given grid cell G, we output 16-byte records $\langle G(\mathbf{x}), \mathbf{x} \rangle$. Note that we get this file already in "sorted" order for free.

(4) **Replace cluster IDs in triangle file with corresponding vertices**. At this point, the file with the representative vertices and the "triangle cluster" file hold the complete simplified mesh. A more useful format for this data is to "dereference" the triangle cluster file and create a file which lists the vertices of each triangle. This can be done in three passes, one for each of the three fields $G(\mathbf{x}_i^t)$. In each pass, the triangle file is sorted on the current vertex field. After each sort, the cluster IDs are scanned and replaced with entries from the representative vertex file, which is read sequentially, in tandem. Many applications prefer an indexed mesh representation, for which one would replace the cluster IDs with vertex indices.

Time and Space Complexity

The memory usage of the OoCSx algorithm we have described does not depend on the size of the input dataset. The algorithm just needs to have enough memory to hold the data structures for one triangle and perform the other calculations for computing the quadrics and optimal vertices. In fact, we use slightly more memory in our external sort implementation, which by default uses four megabytes of memory. Overall, on a PC running Linux, the code never uses more than five megabytes of memory (eight megabytes on IRIX due to larger executables) regardless of the size of the input dataset or the level of approximation desired.

The time complexity of OoCS is $O(n)$, since it only performs a single scan over the mesh file and keeps all the information regarding the quadrics in main memory. Because of the need to sort several files, OoCSx has time complexity $O(n \log n)$.

[2] Although our input and output files use single-precision floating point numbers, we perform the in-memory computations in double precision. 32-bit floats do not provide enough precision for the computations done for very large models like the St. Matthew statue and fluid isosurface.

External Sorting

At the center of OoCSx are a series of external sorts. External sort algorithms are very important for the design and implementation of I/O-efficient algorithms (see [1, 16]). There are several issues in implementing external memory algorithms, and these issues can greatly affect the overall performance of a system. In general trying to mimic the interface of the C **qsort** routine, although often pursued, does not seem the most efficient implementation technique. In our experience with different external sorts [2, 12, 18], the most efficient implementation uses a combination of radix and merge sort, for which the keys are compared lexicographically. A particularly efficient external sort is **rsort** written by John Linderman at AT&T Research [18]. We use **rsort** for the results presented in this paper. Luckily, it is relatively easy to generate keys which can be compared lexicographically (see the man page for **fixcut**, also from Linderman). In OoCSx, we only need integer keys. For these, we simply have to write them in big-endian format.

3.2 Quality Improvements

Surface Boundary Preservation

Because OoCS does not make use of connectivity information, it has no way of detecting whether an edge is a boundary edge or not. Consequently, surface boundaries are not well preserved by the method. We propose a variation on the technique used by Garland and Heckbert [14], which makes use of planes parallel to the boundary edges and orthogonal to their incident triangles.

Building on this idea, we can create an edge quadric. For each half-edge e of each triangle, we compute a plane $\bar{\mathbf{m}}_e$ that passes through the two vertices of e. The normal vector $\mathbf{m}_e$ of this plane is orthogonal to both e and the normal of the face that e belongs to (Figure 1). The distance of a point to this plane provides a measure of how close the point is to the associated edge. We are here only concerned with distances parallel to the plane of the incident face—the per-triangle quadrics from Equation 2 already penalize deviations orthogonal to the face. Using these definitions, we distribute for each half-edge $e = (\mathbf{x}_1^e, \mathbf{x}_2^e)$ its plane equation $\bar{\mathbf{m}}_e$ to the clusters corresponding to its two vertices. After adding up all the plane equations (4-vectors) in a cluster, we compute a quadric matrix $\mathbf{Q}_B$ for the boundary as:

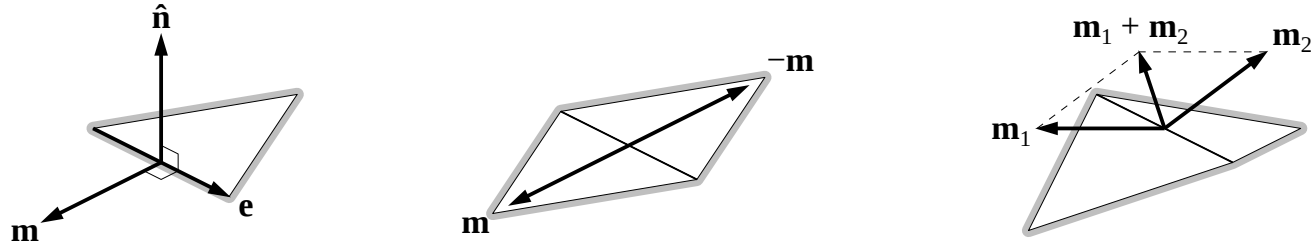
$$\mathbf{Q}_B = \left(\sum_e \bar{\mathbf{m}}_e \right) \left(\sum_e \bar{\mathbf{m}}_e \right)^\mathsf{T} \tag{4}$$

$$\bar{\mathbf{m}}_e = \begin{pmatrix} \mathbf{m}_e \\ -\frac{1}{2}(\mathbf{x}_1^e + \mathbf{x}_2^e)^\mathsf{T} \mathbf{m}_e \end{pmatrix} \tag{5}$$

$$\mathbf{m}_e = \|\mathbf{e}_e\| (\mathbf{e}_e \times \hat{\mathbf{n}}_e) \tag{6}$$

$$\mathbf{e}_e = \mathbf{x}_2^e - \mathbf{x}_1^e \tag{7}$$

Note that all edges, whether manifold or on the boundary, are treated equally. What makes the algorithm sensitive to boundary edges is that, when adding the implicit plane equations $\bar{\mathbf{m}}_e$, there is no opposing half-edge from the neighboring triangle to cancel $\bar{\mathbf{m}}_e$. This is illustrated in Figure 1(b), where the plane equations for two adjacent coplanar faces exactly cancel each other. For non-coplanar faces, the plane equations will not totally cancel, but a residual vector (the normal vector of a new plane) remains that penalizes positions away from the edge in the plane that bisects the dihedral angle formed by the two triangles. The sharper an edge is, the larger this penalty becomes. When used as part of an error measure, this would tend to preserve sharp edges, which is often desired. Based on this argument, non-manifold edges would also tend to be preserved, which is likely desirable since they typically form sharp creases in the mesh. Note that this scheme makes no use



(a) vector **m** orthogonal to boundary edge (b) coplanar faces (c) non-coplanar faces

Figure 1: Illustration of the vectors used for surface boundary preservation. The boundary normal **m** is orthogonal to the face normal $\hat{\mathbf{n}}$ and the vector **e** along the edge *e*. For manifold edges that share two coplanar faces, the boundary normals cancel. In the case of non-coplanar faces, the residual vector $\mathbf{m}_1 + \mathbf{m}_2$ lies in the plane that bisects *e*'s dihedral angle.

of connectivity information, yet implicitly accounts for the feature edges in the mesh.

The final quadric for the cluster is computed as a linear combination $\lambda\mathbf{Q}_S + (1-\lambda)\mathbf{Q}_B$ of the surface quadric and the new boundary quadric. Note that we have been careful to weight the boundary quadric so as to ensure scale invariance and compatibility with the area-squared weighted triangle quadrics. We have found that weighting the quadrics equally ($\lambda = \frac{1}{2}$) tends to give good results.

Constrained Optimization over Cell Boundaries

As discussed in [20], the minimum quadric error sometimes falls outside the cluster's grid cell. While rare, the minimum may be arbitrarily far from the grid cell given the right conditions. Our previous approach to handling these degeneracies was to use one of a number of ad hoc methods for clamping the vertex coordinates, such as projecting the vertex onto the grid cell boundary. To ensure that the vertex is contained in the grid cell, but also results in the smallest possible quadric error, we perform a linearly constrained optimization over the grid cell boundary whenever the global optimum is outside it. Because the quadric functional is quadratic and the grid cell constraints are linear, the solution to this optimization problem can be found by solving a set of linear equations (cf. [22]). This optimization problem is made particularly easy by the fact that the linear constraints are all perpendicular to each other and parallel to the coordinate axes, and can therefore generally be solved as a 2D or 1D problem.

4 EXPERIMENTAL RESULTS

Table 1 summarizes our experimental results. We used two machines for our experiments, most of which were performed on a Linux PC with 512 MB of main memory and two 800 MHz Pentium III processors. The simplification of the statue and fluid isosurface was performed on one processor of a SGI Onyx2 with forty-eight 250 MHz R10000 processors and 15.5 GB of main memory. On the SGI, we used one of its one-terabyte striped disks. Overall, OoCSx was between two to five times slower than OoCS, but sometimes the speed difference was even smaller. In one case, for a high-resolution simplification of the blade, OoCSx was faster than OoCS. The reason for this is that OoCS ran out of memory, and numerous page faults occurred. This happened while trying to simplify the blade to one quarter of its initial size. The ratio in memory usage of OoCS and disk usage of OoCSx varied widely, going from a low of 6 to a high of 245! These variations are due to the dependency on *n*, the size of the input model, in OoCSx, whereas the memory usage of OoCS is proportional to *m*, the size of the output model. For the external sort code **rsort** used in our implementation, we empirically determined the maximum disk usage of OoCSx to be $172T_{in} + 12T_{out}$ bytes.[3] These results indicate that **rsort** requires roughly twice the input size of additional storage. If necessary, there are techniques for lowering the disk overhead of OoCSx. For instance, it would be possible to perform multiple sorts, instead of a single one, and accumulate phases if disk space is at a premium.

model	T_{in}	T_{out}	*RAM:disk (MB)*		*time (h:m:s)*	
			OoCS	*OoCSx*	*OoCS*	*OoCSx*
dragon	871,306	47,236	4:0	5: 150	6	13
		113,058	9:0	5: 152	7	14
		244,568	21:0	5: 153	9	17
buddha	1,087,716	62,346	5:0	5: 187	7	16
		204,766	20:0	5: 191	10	19
blade	28,246,208	507,104	49:0	5: 4,850	2:46	13:14
		1,968,172	160:0	5: 4,899	3:11	14:30
		7,327,888	859:0	5: 4,993	19:14	17:04
statue	372,963,401	3,012,996	261:0	8:64,004	44:22	2:37:24
		21,506,180	3,407:0	8:64,256	51:23	2:49:30
fluid	467,614,855	6,823,739	588:0	8:80,334	55:56	3:11:48
		26,086,125	3,427:0	8:80,510	1:08:48	3:23:42
		94,054,242	-	8:81,345	-	4:19:09

Table 1: Run-time performance of OoCS and OoCSx. The results reported for the dragon, buddha, and blade were computed on a Linux PC. The statue and fluid models were simplified on a SGI Onyx2. Even on the 15.5 GB SGI, not enough RAM was available for OoCS to produce the finest level of detail of the fluid dataset.

Figure 4 shows the effect of using edge quadrics in the simplification of the boundary (shown in red) of the bunny. From this figure, it is evident that the boundaries have been preserved with better visual accuracy. This subjective result is also supported numerically by Figure 2, which shows the maximum (Hausdorff) and root mean square (RMS) distances between closest points on the boundaries for several levels of detail of the bunny. These error measures were evaluated symmetrically by considering all points on the boundaries of both the original and the simplified model. Clearly, the use of boundary quadrics greatly reduced the boundary errors. In addition, we found that the use of boundary quadrics did not negatively impact the errors measured over the surface interiors. Instead, using boundary quadrics reduced both boundary and surface errors for models with boundaries, and did not result in a measurable increase in surface error for models without boundaries.

Figure 3 is an isosurface of a time slice from a large-scale turbulent-mixing fluid dynamics simulation, consisting of $2,048 \times 2,048 \times 1,920$ voxels at 27,000 time steps [23]. This surface is challenging to simplify due to its highly complex topology and wispy geometry. Table 1 lists the performance data for simplifying the entire isosurface. To avoid too much clutter in the images presented here, we also extracted a small piece (one third of a percent) of the volume and simplified it independently (Figure 6). As can be

[3] This usage is for the intermediate files only, and does not include the space needed for the input and output files.

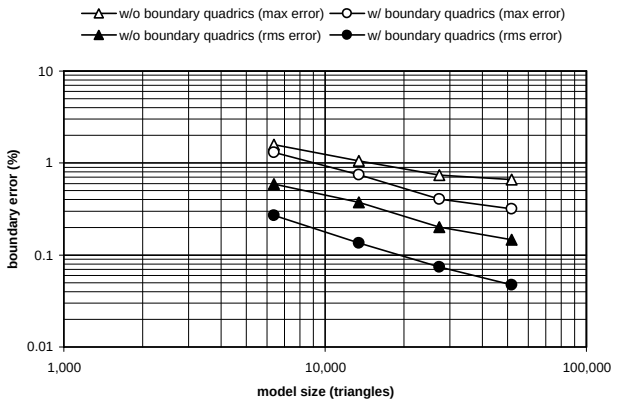


Figure 2: Maximum and root mean square boundary error for bunny model, simplified with and without boundary quadrics.

seen in Figure 6(e), there is significant loss in topological structure and geometric detail as the triangle count drops to a few million. A simplification of a complex dataset like this requires more triangles than can be stored in RAM on most computers, and must be simplified using a memory insensitive method such as OoCSx. Notice also the improved boundaries in Figure 6(d) over the model simplified without boundary quadrics (Figure 6(b)).

Finally, we evaluated the effect of performing constrained optimization over the cell boundary in those cases where the optimal vertex position lies outside the cell. We compared this approach to (1) leaving the vertex outside the cell, and (2) projecting it onto the cell boundary. In all cases, the constrained optimization performed as well or better than the other two approaches, both in terms of maximum and RMS error. Figure 5 shows an example where constrained optimization resulted in nearly a factor of six reduction in the maximum error over leaving the vertices unclamped. Notice how the artifacts near the lower jaw, ears, and hind leg are eliminated by clamping and optimizing the vertices, leaving a more visually pleasing model.

5 CONCLUSIONS

In this paper, we proposed improvements to the out-of-core simplification (OoCS) technique [20]. First, we described OoCSx, a memory insensitive variation of OoCS. The key feature of OoCSx is its ability to efficiently simplify arbitrarily large datasets using a constant amount of main memory. OoCSx uses a disk-based technique for storing information about the simplified mesh and arranging it in a cache-coherent manner. We also discussed an efficient implementation of OoCSx and compared its performance with OoCS. Second, we proposed a technique for preserving surface boundaries without making use of connectivity information. Our approach is to compute and minimize an edge-based quadric error for all edges of the mesh, regardless of their topological type. We showed that this technique can dramatically improve the shape of boundary curves, with little or no loss in geometric quality over the remaining surface. Finally, we proposed using a linearly constrained optimization over grid cell boundaries to compute vertex positions whenever the global optimum is outside the grid cell.

One shortcoming of the current approach is that the overall simplification has constant feature size. Similar to [27], it would be interesting to extend OoCSx to simplify the mesh adaptively. Taking this one step further, we will investigate how to adapt our out-of-core algorithms to perform dynamic view-dependent refinement of the mesh for interactive visualization. Another drawback of OoCSx is that it requires significant amounts of disk space. The per-triangle quadric information stored on disk constitutes a large portion of the overall space requirements. We believe that careful encoding of these 4-vectors, using normal quantization [9] and per-grid-cell coordinate representations, will allow this information to be represented using as little as 32 bits per vector. Finally, many datasets come with surface attributes such as scalar field values, normal and curvature information, and color. We hope to extend our simplification code to take into account and preserve such information.

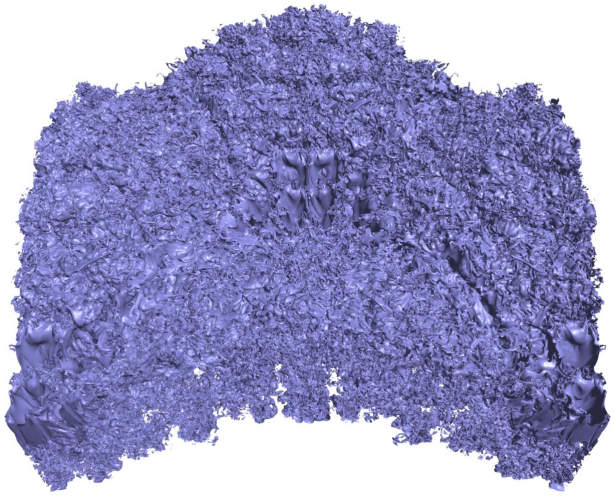


Figure 3: 470 million triangle isosurface of entire fluid dynamics dataset.

Acknowledgements

This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48. We would like to thank the reviewers for useful comments. Many thanks to Glenn Fowler and John Linderman for several discussions and access to their external sorting code. We wish to thank Stanford University and the Digital Michelangelo Project for providing the bunny, dragon, Buddha, and St. Matthew datasets, and Kitware for the turbine blade model. Thanks to David Bremer, Mark Duchaineau, and Randy Frank for preparing the fluid dynamics dataset.

Appendix: Dereferencing Indexed Meshes

The file format we assume in our algorithm is different from the indexed mesh formats commonly used for main memory techniques. In main memory, it is common to store a list of vertex coordinates (x,y,z), and a list of triangles, represented by three integers that refer to the vertices of the given triangle. Before such datasets can be used in our algorithm, they need to be "normalized", a process that dereferences the pointers to vertices. This process is thoroughly explained in [7]. For completeness, we briefly explain how to normalize such a file with V vertices and T triangles. In an initial pass, we create two (binary) files, one with the list of vertices, and another with the list of triangles. Next, in three passes, we dereference each index in the triangle file, and replace it with the actual position for the vertex. In order to do this efficiently, we first (externally) sort the triangle file on the index we intend to dereference. This takes time $O(T \log T)$ using an (external memory) mergesort. Then, we perform a synchronous scan of both the vertex and the (sorted) triangle file, reading one record at a time, and appropriately outputting the deferenced value for the vertex. Note that this can be done efficiently in time $O(V+T)$ because all the vertex references are sorted. When we are done with all three passes, the triangle file will contain T records with the "value" (not reference) of each of its three vertices.

References

[1] J. Abello and J. Vitter. *External Memory Algorithms and Visualization*. American Mathematical Society Press, 1999.

[2] L. Ammeraal. *Algorithms and Data Structures in C++*. Addison Wesley, 1996.

[3] C. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel Accelerated Isocontouring for Out-of-Core Visualization. *Proceedings of IEEE Parallel Visualization and Graphics Symposium 1999*, pages 97–104, October 1999.

[4] F. Bernardini, J. Mittleman, and H. Rushmeier. Case study: Scanning Michaelangelo's Florentine Pietà. In *ACM SIGGRAPH 1999 Course notes, Course #8*, August 1999.

[5] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The Ball-Pivoting Algorithm for Surface Reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, October - December 1999.

[6] Y.-J. Chiang and C. T. Silva. I/O Optimal Isosurface Extraction. *IEEE Visualization '97*, pages 293–300, November 1997.

[7] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive Out-of-Core Isosurface Extraction. *IEEE Visualization '98*, pages 167–174, October 1998.

[8] M. B. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. *IEEE Visualization '97*, pages 235–244, November 1997.

[9] M. F. Deering. Geometry Compression. *Proceedings of SIGGRAPH 95*, pages 13–20, August 1995.

[10] J. El-Sana and Y.-J. Chiang. External Memory View-Dependent Simplification. *Computer Graphics Forum*, 19(3), August 2000.

[11] R. Farias and C. Silva. Out-of-Core Rendering of Large Unstructured Grids. *IEEE Computer Graphics & Applications*, 21(4):42–50, 2001.

[12] G. Fowler. AST **sort**. http://www.research.att.com/sw/download.

[13] M. Garland and P. Heckbert. Fast Polygonal Approximation of Terrains and Height Fields. Technical Report CMU-CS-95-181, Carnegie Mellon University, 1995.

[14] M. Garland and P. Heckbert. Surface Simplification Using Quadric Error Metrics. *Proceedings of SIGGRAPH 97*, pages 209–216, August 1997.

[15] H. H. Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. *IEEE Visualization '98*, pages 35–42, October 1998.

[16] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[17] S. Leutenegger and K.-L. Ma. Fast Retrieval of Disk-Resident Unstructured Volume Data for Visualization. In *External Memory Algorithms and Visualization*, DIMACS Book Series, American Mathematical Society, vol. 50, 1999.

[18] J. Linderman. **rsort** and **fixcut** man pages. April 1996 (revised June 2000).

[19] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. *Proceedings of SIGGRAPH 2000*, pages 131–144, July 2000.

[20] P. Lindstrom. Out-of-Core Simplification of Large Polygonal Models. *Proceedings of SIGGRAPH 2000*, pages 259–262, July 2000.

[21] P. Lindstrom and G. Turk. Fast and Memory Efficient Polygonal Simplification. *IEEE Visualization '98*, pages 279–286, October 1998.

[22] P. Lindstrom and G. Turk. Evaluation of Memoryless Simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April - June 1999.

[23] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimitis, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System. *Proceedings of Supercomputing 99*, November 1999.

[24] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Proceedings of SIGGRAPH 97*, pages 101–108, August 1997.

[25] C. Prince. Progressive Meshes for Large Models of Arbitrary Topology. M.S. thesis, University of Washington, 2000.

[26] J. Rossignac and P. Borrel. Multi-Resolution 3D Approximation for Rendering Complex Scenes. In *Modeling in Computer Graphics*, pages 455–465, 1993.

[27] E. Shaffer and M. Garland. Efficient Adaptive Simplification of Massive Meshes. *IEEE Visualization '01*, October 2001.

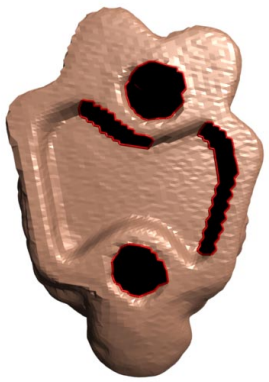
[28] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree. *IEEE Visualization '99*, pages 371–378, October 1999.

[29] C. Silva, J. Mitchell, and A. E. Kaufman. Automatic Generation of Triangular Irregular Networks Using Greedy Cuts. *IEEE Visualization '95*, pages 201–208, November 1995.

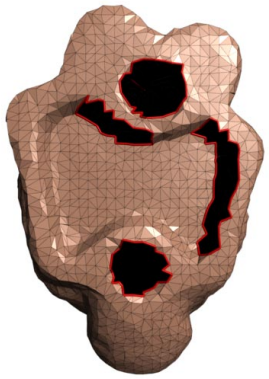
[30] C. Silva and J. Mitchell. Greedy Cuts: An Advancing Front Terrain Triangulation Algorithm. *Proceedings of the 6th ACM Workshop on Advances in GIS*, pages 137–144, November 1998.

[31] P. M. Sutton and C. D. Hansen. Accelerated Isosurface Extraction in Time-Varying Fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107, April - June 2000.

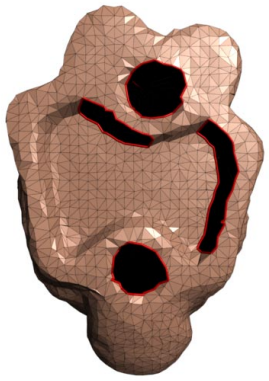
[32] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, October - December 1997.



(a) original; $T = 69,451$



(b) w/o boundary quadrics; $T = 14,761$

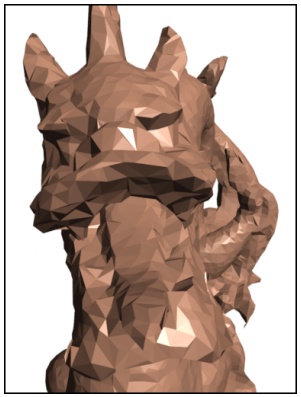


(c) w/ boundary quadrics; $T = 14,761$

Figure 4: Bunny model simplified with and without the surface boundary preservation technique.

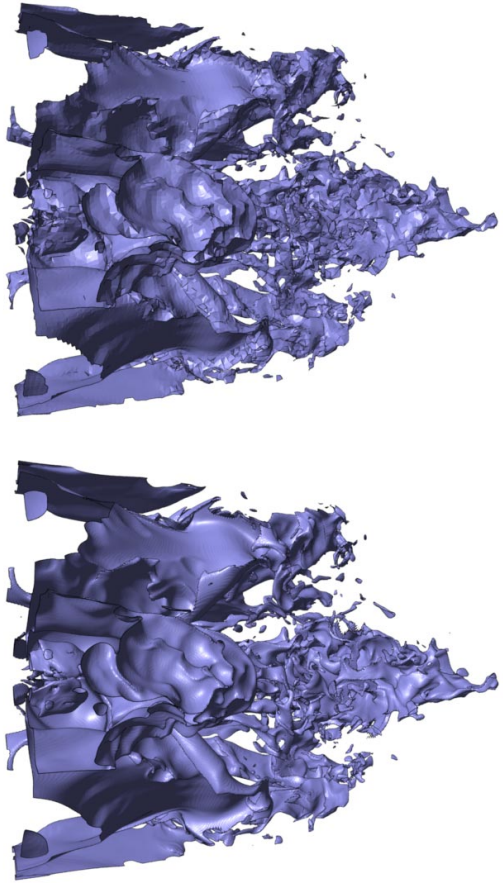
(a) original; $T = 871,306$

(b) without clamping; $T = 20,292$

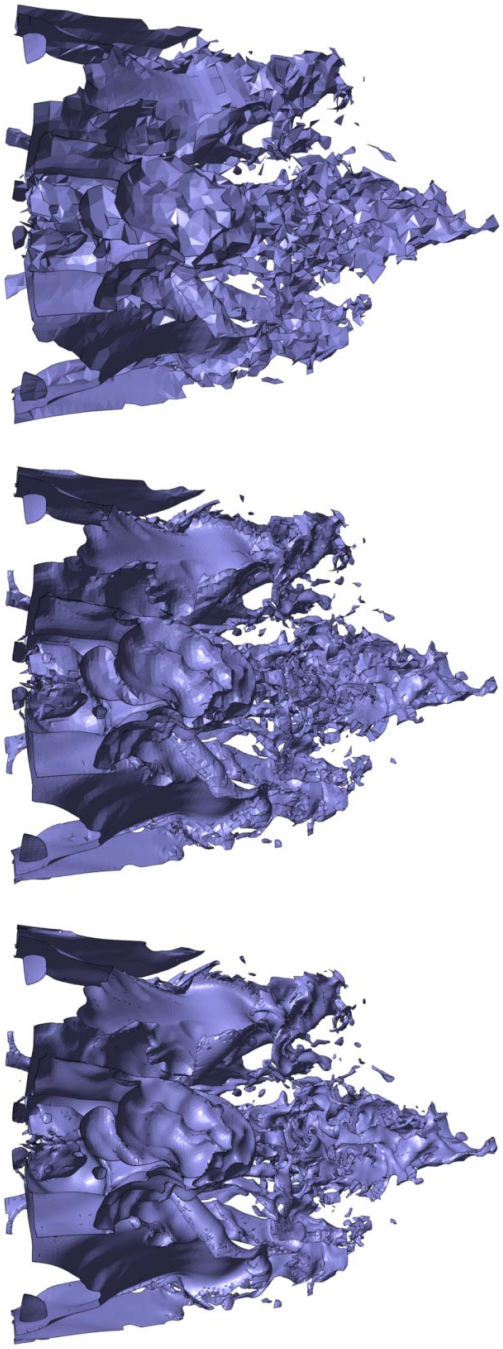


(c) with clamping; $T = 20,292$

Figure 5: Use of constrained optimization over cell boundaries to reposition stray vertices.



(a) original; $T = 467,614,855$



(b) without boundary quadrics; $T = 26,086,125$

(c) with boundary quadrics; $T = 94,054,242$

(d) with boundary quadrics; $T = 26,086,125$

(e) with boundary quadrics; $T = 6,823,739$

Figure 6: Small subset of isosurface of turbulent-mixing fluid dynamics simulation. The triangle counts correspond to simplifications of the entire dataset.