Preprint
UCRL-200016-CONF

Large Mesh Simplification using Processing Sequences

M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink

To appear in Proceedings of IEEE Visualization 2003,
Seattle, Washington, October 19–24, 2003

August 12, 2003

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Large Mesh Simplification using Processing Sequences

Martin Isenburg
University of North Carolina
at Chapel Hill
isenburg@cs.unc.edu

Peter Lindstrom
Lawrence Livermore
National Laboratory
pl@llnl.gov

Stefan Gumhold
WSI/GRIS
University of Tübingen
gumhold@gris.uni-tuebingen.de

Jack Snoeyink
University of North Carolina
at Chapel Hill
snoeyink@cs.unc.edu

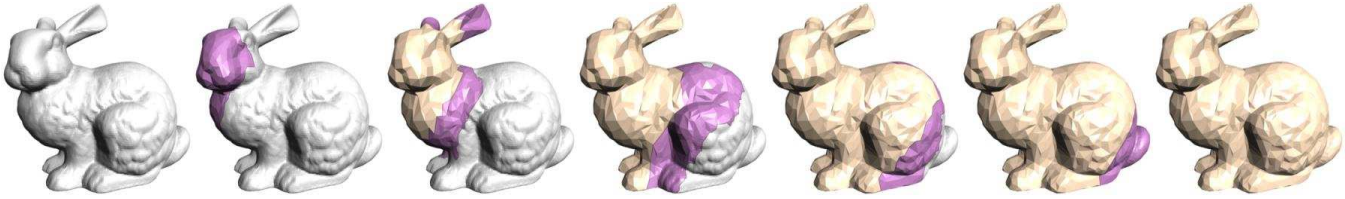


Figure 1: Simplification with a fixed-size in-core buffer (pink). Via processing sequences, original triangles (gray) stream into the buffer and simplified triangles (gold) stream out.

Abstract

In this paper we show how out-of-core mesh processing techniques can be adapted to perform their computations based on the new *processing sequence* paradigm [Isenburg and Gumhold 2003; Isenburg et al. 2003], using mesh simplification as an example. We believe that this processing concept will also prove useful for other tasks, such as parameterization, remeshing, or smoothing, for which currently only in-core solutions exist.

A processing sequence represents a mesh as a particular interleaved ordering of indexed triangles and vertices. This representation allows streaming very large meshes through main memory while maintaining information about the visitation status of edges and vertices. At any time, only a small portion of the mesh is kept in-core, with the bulk of the mesh data residing on disk. Mesh access is restricted to a fixed traversal order, but full connectivity and geometry information is available for the active elements of the traversal. This provides seamless and highly efficient out-of-core access to very large meshes for algorithms that can adapt their computations to this fixed ordering.

The two abstractions that are naturally supported by this representation are *boundary-based* and *buffer-based* processing. We illustrate both abstractions by adapting two different simplification methods to perform their computation using a prototype of our mesh processing sequence API. Both algorithms benefit from using processing sequences in terms of improved quality, more efficient execution, and smaller memory footprints.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—surface, solid, and object representations

Keywords: Out-of-core algorithms, processing sequences, mesh simplification, large meshes.

1 Introduction

Polygonal models acquired with modern 3D scanning technology easily reach sizes of gigabytes—the most prominent examples are the detailed scans of Michelangelo's sculptures generated by teams at IBM [Bernardini et al. 2002] and Stanford [Levoy et al. 2000]. Similarly large polygonal data sets result from extracting dense isosurfaces from volumetric data. A polygon mesh with hundreds of millions of vertices requires gigabytes of raw data, making subsequent processing difficult. The sheer amount of data not only exhausts the main memory resources of common desktop PCs, but also exceeds the 4 gigabyte address space of 32-bit machines.

A straightforward approach for processing meshes that are too large to fit in main memory is to cut them into pieces small enough to be processed in-core. The disadvantages of mesh cutting are the processing artifacts that tend to be introduced along the cut boundaries. Another approach is to design computations to work in increments of single triangles. This allows efficient batch processing, as the CPU can be kept busy by loading and processing triangles as fast as possible. However, the absence of explicit mesh connectivity information typically results in a lower quality output. Finally, there are approaches that use external memory data structures that provide transparent access for online processing of arbitrarily large meshes. However, building and using such complex data structures is typically inefficient.

We recently proposed a new processing paradigm for out-of-core computations on large meshes [Isenburg and Gumhold 2003; Isenburg et al. 2003] that combines the efficiency of batch processing with the advantage of explicit mesh connectivity that is available in online processing. The idea of a *processing sequence* is to restrict access to the mesh to a fixed traversal order, but to support access to full connectivity and geometry information for the active elements of this traversal. In this representation only a small fraction of the mesh is kept in main memory at any time with the bulk of the mesh data residing on disk. While the mesh streams through memory, we provide seamless mesh access for algorithms that can respect a fixed traversal order.

Processing sequences support two computational abstractions: *boundary-based* processing and *buffer-based* processing. Several operations that are useful in dealing with large meshes are naturally supported by these abstractions, including loading, decompression, rendering, and connectivity reconstruction. In this paper we show how they can be used for more complex tasks, which we demonstrate using out-of-core mesh simplification as an example.

The remainder of this paper is organized as follows: The next section summarizes current approaches to out-of-core mesh processing. In Section 3 we describe how processing sequences pro-

vide access to large meshes. In Section 4 we detail current techniques for the simplification of large meshes. Then we adapt two different mesh simplification schemes to sequenced processing: In Section 5 we adapt Lindstrom's non-adaptive OoCS simplification algorithm [2000] to boundary-based processing. Similarly, in Section 6, we map Wu and Kobbelt's adaptive stream simplification algorithm [2003] to buffer-based processing. Both algorithms benefit from using processing sequences in terms of improved quality, more efficient execution, and smaller memory footprints. The last section concludes with a summary and an outlook on other types of mesh processing.

2 Out-of-Core Processing

There are three main approaches for processing meshes that are too large to fit in main memory [Silva et al. 2002]: cutting the mesh into pieces, batch processing of polygon soups, and online processing using external memory data structures.

Mesh cutting is a straightforward approach for processing large meshes: cut the mesh into pieces small enough to fit in main memory and then process each piece separately while giving special treatment to the cut boundaries. This strategy has successfully been used to, for example, simplify [Hoppe 1998; Prince 2000; Bernardini et al. 2002] and compress [Ho et al. 2001] very large polygon models. Despite the apparent simplicity of this approach, the initial cutting step can be expensive when the input mesh is given in an indexed representation, as we will see later. Because mesh cutting typically lowers the quality of the output, many out-of-core algorithms try instead to process the data as a whole.

Batch processing aims to keep the memory footprint low and the processor busy by streaming the mesh data through main memory in one or more passes, and by restricting computations to the amount of data that is resident in memory at any time. This makes batch processing computationally very efficient.

Examples include a number of mesh simplification methods [Lindstrom 2000; Lindstrom and Silva 2001; Shaffer and Garland 2001; Garland and Shaffer 2002], which batch-process the input mesh as a sequence of individual triangles. If indexed meshes are used that exhibit no locality in referencing the vertex array (e.g. where vertex indices of subsequent triangles address vertex array entries at random) an initial *de-referencing step* is required [Lindstrom and Silva 2001]. This can be computationally expensive and the resulting immediate mesh (i.e. *polygon soup*) requires at least twice the storage of an indexed mesh, and more if there are additional per-vertex properties such as texture coordinates or surface normals. The output of a batch simplification pass either is small enough to fit in memory, so that the remaining computation can be done in-core [Lindstrom 2000; Shaffer and Garland 2001; Garland and Shaffer 2002], or is directly written to a file, which is then used as input for subsequent passes [Lindstrom and Silva 2001].

Online processing accesses the data through a series of (potentially random) queries. In order to avoid costly disk seeks with each query (resulting in thrashing) the data is usually re-organized to accommodate an anticipated access pattern. Queries can be accelerated by *caching* or *pre-fetching* data that is likely to be accessed.

Some schemes simply use the virtual memory functionality of the operating system and try to organize the data accesses such that the number of page faults is minimized [McMains et al. 2001; Choudhury and Watson 2002]. The performance of such schemes is operating system dependent and their input data is restricted to 4 gigabytes on a 32-bit machine. Going beyond that limit requires dedicated external memory data structures that explicitly manage a virtual address space for the data.

Such external memory data structures enable traditional in-core algorithms to be applied to large data sets. Cignoni et al. [2003], for example, propose an octree-based external memory data structure that makes it possible to simplify a model of Michelangelo's St. Matthew statue [Levoy et al. 2000] from 386 to 94 million triangles using iterative edge contraction [Garland and Heckbert 1997]. Similarly, the out-of-core mesh proposed by Isenburg and Gumhold [2003] allows compressing the St. Matthew statue from over 6.5 GB to 344 MB of data using a compressor based on region growing [Touma and Gotsman 1998].

For comparison, out-of-core algorithms based on batch processing do their work on polygon soups without explicit connectivity information. Thus, they can perform their computations efficiently, but their output tends to be of lower quality than that of algorithms with access to explicit connectivity information. Out-of-core algorithms based on online processing, on the other hand, have explicit connectivity available. However, building these data structures is expensive in time and space, and using them significantly slows down the computations.

Recently we proposed an approach that combines the efficiency of batch processing with the advantages of explicit connectivity information available in online processing. The idea of a *processing sequence* is to restrict the access to the mesh to a fixed traversal order, but to support access to the full connectivity and geometry information for the active elements during this traversal.

Rearranging mesh triangles into a particular order is already used for improving rendering performance on modern graphics cards. The number of times a vertex needs to be fetched from main memory is reduced by caching previously received vertices on the card. The triangles are sent to the card in a *rendering sequence* in an attempt to minimize cache misses [Deering 1995; Evans et al. 1996; Hoppe 1999; Bogomjakov and Gotsman 2001]. Due to the fixed size of a vertex cache, misses cannot be avoided completely [Bar-Yehuda and Gotsman 1996].

Our *processing sequences* exploit a similar strategy for more efficient mesh processing—but at a much larger scale. However, the main memory as a "cache" is much more flexible. The amount of storage necessary to maintain the active elements of a mesh traversal is usually small enough to fit in main memory. Therefore the analogue of a "cache miss" does fortunately not exist.

3 Processing Sequences

A *processing sequence* presents a mesh as a fixed interleaved sequence of indexed vertices and triangles that grow a region. The mesh edges that separate already processed triangles from unprocessed ones form the *processing boundary*. Mesh triangles *generated* by the processing sequence are either edge-adjacent to the processing boundary or start a new region. With each triangle, the processing sequence provides vertex information such as indices, coordinates, first and last time referenced, and non-manifoldness. Similarly, the topological type of edges and their relationship to the processing boundary are made available. Finally, a processing sequence supports storage and retrieval of user data on the evolving processing boundary.

Triangles can change the processing boundary in one of the five ways illustrated in Figure 2. A "start" triangle creates a new component of a processing boundary with three *new* vertices and edges. A new edge may be *entering* the processing boundary, to be paired with an incident triangle later in the sequence, or it may be part of the surface *border*, the topological boundary of the mesh. An "add" triangle completes a boundary edge and connects a new vertex with two new edges. The completed edge *leaves* the processing boundary. A "fill" completes two edges, replacing them and the vertex reference between them with a new edge. A "join" completes one edge, adds two new edges, and either merges two components of the processing boundary into one (usually forming a handle), or splits one component into two. An "end" completes three edges.

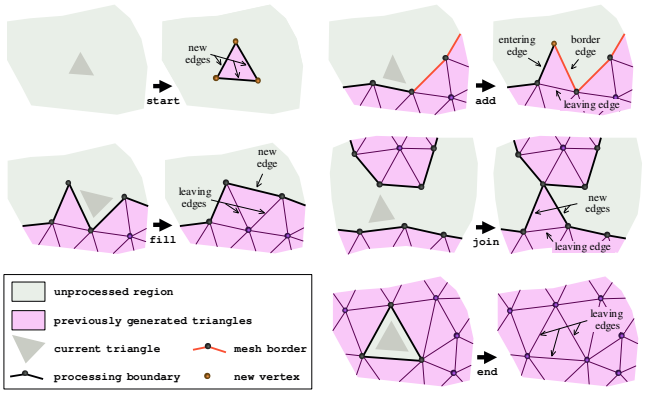


Figure 2: The five different ways generated triangles relate to the processing boundary.

Given a "somewhat" compactly growing processing sequence, this representation allows streaming very large meshes through main memory. At any time only the processing boundary needs to be kept in-core. Yet, as explicit connectivity information can be maintained along the processing boundary, this provides seamless access to large meshes.

Connectivity reconstruction is supported by letting users store their own data with the first appearance of any edge or vertex on the processing boundary. This data is made available when these mesh elements later reappear as part of another triangle, enabling full recovery of mesh connectivity in constant time per element.

If processing sequences are read and written at the same time there are two processing boundaries: one is the *input* boundary, along which triangles are added, and one is the *output* boundary, where triangles are removed. The region between the two boundaries is the called the *triangle buffer*, which contains those triangles that are currently in memory. The triangle order of the input and the output sequence does not need to be identical. In particular, the two sequences can contain a completely different set of triangles and vertices, for example, if remeshing or simplification is performed on the triangle buffer. When the order in which an application outputs triangles and vertices does not immediately correspond to a processing sequence, we use a *processing sequence converter* that temporarily accumulates triangles and vertices in a small *waiting area* and reorders them appropriately, as illustrated in Figure 3.

Processing sequences provide two useful computational abstractions: boundary-based and buffer-based processing.

Boundary-based processing performs its computations directly on the input boundary. It immediately processes the triangles generated at the input boundary and stores intermediate results only along these boundaries. Example applications are simplification methods using vertex clustering, non-iterative smoothing methods, gradient or surface normal computations, etc.

Buffer-based processing performs its computations on the triangle buffer between input and output boundary (Figure 1). It generates triangles at the input boundary to fill the buffer and at the output boundary to empty the buffer. Example applications are simplification methods that use edge contraction, iterative smoothing methods, remeshing methods, etc. We can think of buffer-based processing as bridging the conceptual gap between boundary-based processing and in-core processing. Restricting the buffer size to a single triangle is equivalent to boundary-processing. A buffer size that is large enough to contain the entire mesh is equivalent to in-core processing. Any buffer size in between these extremes provides a compromise that "adapts" to the available resources.

Implementations of either abstraction can perform their computation in a single pass or in multiple passes over the data. For multiple passes, the output sequence of a previous pass becomes the input sequence of the next. Instead of sequentially performing multiple passes, a multi-stage approach streams the results of one pass directly to the next by making the output boundary of one the input boundary of the other. Immediate compression of the output of a simplification algorithm, for example, could be implemented using such a multi-stage approach.

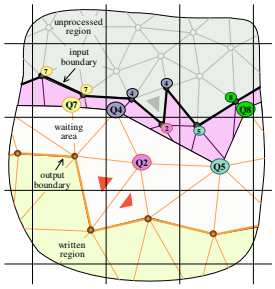


Figure 3: An illustration of how a *waiting area* is used to *on-the-fly* convert the triangle and vertex ordering produced by the simplification method described in Section 5 into a processing sequence. After processing the triangle marked in gray, the simplifier turns the quadric Q2 into a vertex and places it into the waiting area. In this moment the vertex becomes eligible for output, as do the two triangles in the waiting area marked in red that are connected to it. Furthermore, this also *finalizes* the vertex, in the sense that no triangles other than those already in the waiting area reference it.

Generating processing sequences can be done in a number of different ways, as the definition neither imposes a specific traversal order, nor a data format. The input sequences used in this paper were generated in a pre-processing step using an out-of-core compression method [Isenburg and Gumhold 2003]. Most *one-pass* compression schemes naturally generate triangle and vertex orderings that conform to the definition of a mesh processing sequence. In fact, it was the memory-efficient decompression order of our decoder that originally inspired processing sequences. Although these particular processing sequences are compact and very fast to load, their generation is not trivial and they are currently created offline.

The processing sequence converter, mentioned earlier, is one efficient method for *on-the-fly* creation of processing sequences. It accepts indexed vertices and triangles ordered in some loosely localized form, temporarily accumulates them in a *waiting area*, where they are re-ordered into a proper processing sequence. A vertex from the waiting area becomes eligible for output when its first triangle is to be output. A triangle from the waiting area becomes eligible for output when all its vertices are already output *and* it conforms to one of the five configurations shown in Figure 2.

The sole requirement, besides some locality in the input, is that the converter is told when a vertex is *finalized*, i.e., used for the last time. This information is needed to correctly recover connectivity around vertices, as well as to safely deallocate the memory of mesh elements that are no longer used. For our output sequences, the simplification process tells the converter when a vertex is finalized.

The converter automatically buffers as many triangles as needed to produce a valid processing sequence. Increasing the size of the waiting area beyond the minimum gives the converter freedom to choose among several potential output triangles. This allows, for example, sequences with fewer "start" or "join" configurations to be generated. Sequences generated this way are currently stored in a verbose format, but a compressed format for *on-the-fly* compression of arbitrary output sequences is in the works.

This converter also provides an alternative to the out-of-core compressor [Isenburg and Gumhold 2003] for generating processing sequences "from scratch": First we create two spatially ordered sequences, one of vertices and one of triangles. Vertices are sorted together with their index i using one coordinate, for example x, as the sort key k. Triangles are sorted in indexed form using the minimal key k of their three vertices as the sort key. This can be implemented using a few external sorts [Lindstrom and Silva 2001].

In a final pass over the two sorted sequences we load vertices and triangles into the waiting area. We read from the triangle sequence as long as the next triangle key is less than or equal to the next vertex key. Eventually the key of the next triangle is larger than that of the next vertex and we read from the vertex sequence. This

vertex can now be finalized as all its triangles are already in the waiting area. The vertices and triangles leave the waiting area in processing sequence order, as described earlier.

Non-manifold meshes are turned into manifold meshes simply by cutting along non-manifold vertices and edges. However, vertices and edges are not replicated, but re-appear multiple times as a *new* mesh element. This allows representing non-manifold meshes while using only the five operations allowed for generating triangles. An additional flag per vertex and per edge provided by the processing sequence API informs whether an element is non-manifold and whether there are still future non-manifold occurrences of the element remaining.

4 Large Mesh Simplification

Early methods for simplifying large meshes were based on mesh cutting [Hoppe 1998; Prince 2000; Bernardini et al. 2002]. In mesh cutting, the input mesh is partitioned into pieces small enough to be processed in-core, which are then simplified individually. The partition boundaries are left untouched such that the simplified pieces can be stitched back together seamlessly. While the hierarchical approaches of Hoppe [1998] and Prince [2000] automatically simplify these boundaries at the next level, Bernardini et al. [2002] process the mesh more than once—each time using a different partitioning.

Later, out-of-core simplification methods based on batch processing became popular. Lindstrom [2000] performs vertex clustering [Rossignac and Borrel 1993] on a uniform grid and stores one quadric error matrix [Garland and Heckbert 1997] per occupied grid cell in memory. Indexed input meshes are first dereferenced into polygon soups and then batch-processed one triangle at a time, adding each triangle's quadric matrix to the cells in which the triangle has a vertex. The output triangles are those that connect three different grid cells. Each cell is represented by a vertex whose position minimizes the quadric error accumulated in the cell. In more recent work Lindstrom and Silva [2001] show that the limitation of the output mesh having to fit in main memory can be overcome using a series of external sorts.

Although the vertex clustering approach to simplification allows efficient out-of-core implementations, it delivers lower quality results than a typical in-core algorithm. Vertex clustering can not retain details smaller than a grid cell and lacks the adaptivity of an implementation based on, for example, iterative edge contraction. Addressing this issue, Shaffer and Garland [2001; 2002] suggest using batch processing to accumulate error quadrics with a vertex cluster resolution that is higher than that of the output mesh, but still fits in-core. From there a simplified mesh can be created in-core either top-down, using a variation of R-simp [Brodsky and Watson 2000], or bottom-up, using QSlim [Garland and Heckbert 1997]. The accumulated quadrics pass information about the original surface to the in-core algorithm. This allows higher quality simplifications with an exact vertex budget, provided that the available memory is a constant factor larger than the output mesh.

As we will see in Section 5, processing sequences allow efficient implementations of simplification algorithms based on vertex clustering. As the processing boundary sweeps over the entire mesh, visiting every triangle exactly once, we can store, update, and propagate quadric error matrices along these boundaries only. This will significantly reduce the memory footprint, improve the quality of the simplified mesh, and enable pipelined processing by immediately feeding the output to another application.

The simplification methods discussed so far treat large meshes differently from small meshes as they try to avoid performing costly online processing on the entire mesh. Therefore the output produced by an out-of-core algorithm is usually of lower quality than that of an in-core algorithm. Cignoni et al. [2003] propose an octree-based external memory data structure that provides algorithms with transparent online access to huge meshes. This makes it possible to, for example, simplify the St. Matthew statue from 386 to 94 million triangles using iterative edge contraction [Garland and Heckbert 1997]. However, the run times for both constructing an external memory data structure and using it during simplification are orders of magnitude above those of simplification methods based on batch processing.

mesh name	number of						
	vertices	triangles	comp.	holes	handles	n.-m. v.	p. b. v.
buddha	544 K	1.1 M	1	0	104	0	2.3 K
blade	883 K	1.8 M	295	0	165	0	7.3 K
david (2mm)	4.1 M	8.3 M	2	1	19	4	21 K
lucy	14 M	28 M	18	29	0	64	23 K
david (1mm)	28 M	56 M	2.3 K	4.2 K	137	1.1 K	59 K
st. matthew	187 M	373 M	2.9 K	26 K	483	3.8 K	223 K
isosurface	235 M	469 M	168 K	6.2 K	168 K	0	1.6 M

Table 1: Vertex, triangle, component, hole, handle, and non-manifold vertex counts, as well as maximum length of the processing boundary in thousands (K) and millions (M) for all meshes used in our experiments.

Wu and Kobbelt [2003] propose an out-of-core simplification technique that is similar to the buffer-based abstraction of processing sequences. Starting with polygon soup as input, they keep a large in-core buffer of triangles on which they perform edge collapses. Since the input mesh is not indexed, connectivity between triangles must be reconstructed by matching up the coordinates of their vertices. Their method assumes that the polygon soup is spatially ordered so that the triangles in the in-core buffer form connected regions. Thus, an input mesh may need to be pre-sorted using external sorting [Lindstrom and Silva 2001].

One drawback of Wu and Kobbelt's method is that it can not distinguish actual mesh borders from the input boundary of the buffer. As borders cannot be recognized and simplified until the entire mesh has been read, they must keep all triangles along the mesh borders in the buffer. For a mesh with many small holes, which is common in large range scans, this can considerably inflate the memory requirements and may reduce the quality of the output.

Processing sequences provide an ideal input to Wu and Kobbelt's stream-based method: The incoming triangles that populate the buffer are maximally connected. The mesh borders are known, which allows immediate simplification of holes. The connectivity reconstruction is either already provided by the API or can be done more efficiently as triangles are in an indexed format. Finally, their stream-based algorithm maps exactly to the abstraction of buffer-based processing, which is discussed further in Section 6.

5 Boundary-Based Processing

In this section we show how Lindstrom's out-of-core simplification method OoCS [2000] can be adapted to mesh processing sequences using boundary-based processing. Capitalizing on the coherent geometric or topological ordering provided by processing sequences, as well as the connectivity information made available, we improve upon OoCS in a number of ways.

First, we make use of explicit mesh connectivity to detect and preserve surface boundaries. This is trivially accomplished using processing sequences, although it is an important improvement. Second, we avoid the common "pinching" problem that results when two or more (possibly unconnected) layers of the surface pass through the same grid cell and are pinched. This problem is particularly noticeable when simplifying "dense" meshes with many thin structures, such as CAD models and complex isosurfaces (see, for example, Figure 6(a)). Finally, because of spatial coherence, we do not need to maintain the entire simplified mesh in memory, but output vertices and triangles whenever possible as the processing boundary advances through space. As a result, we require in-core storage only on the order of the length of the processing boundary.

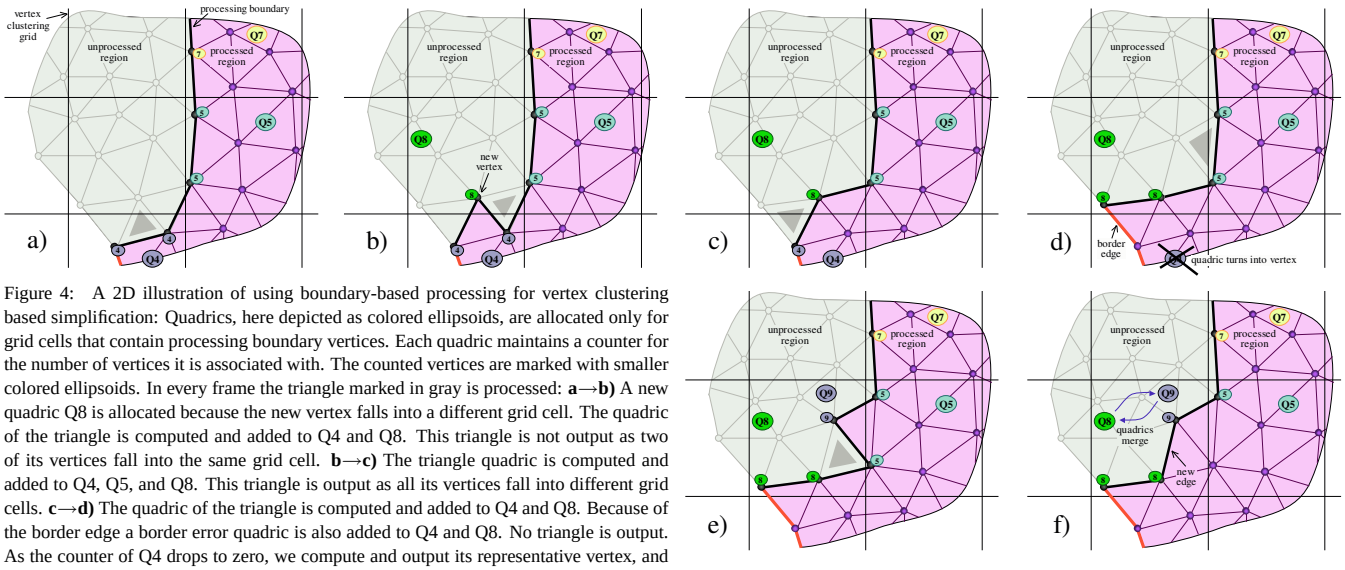


Figure 4: A 2D illustration of using boundary-based processing for vertex clustering based simplification: Quadrics, here depicted as colored ellipsoids, are allocated only for grid cells that contain processing boundary vertices. Each quadric maintains a counter for the number of vertices it is associated with. The counted vertices are marked with smaller colored ellipsoids. In every frame the triangle marked in gray is processed: **a→b)** A new quadric Q8 is allocated because the new vertex falls into a different grid cell. The quadric of the triangle is computed and added to Q4 and Q8. This triangle is not output as two of its vertices fall into the same grid cell. **b→c)** The triangle quadric is computed and added to Q4, Q5, and Q8. This triangle is output as all its vertices fall into different grid cells. **c→d)** The quadric of the triangle is computed and added to Q4 and Q8. Because of the border edge a border error quadric is also added to Q4 and Q8. No triangle is output. As the counter of Q4 drops to zero, we compute and output its representative vertex, and deallocate the quadric. **d→e)** A new quadric Q9 is allocated because the new vertex falls into a different grid cell from those it is connected to. The quadric of the triangle is added to Q5 and Q9. No triangle is output. **e→f)** The new edge connects two vertices of the same grid cell that have different quadrics. Therefore, quadrics Q8 and Q9 are merged. The triangle quadric is added to Q5 and Q8/Q9.

We will describe two extensions to the original clustering method—one simple and one somewhat more involved—and begin by explaining the general idea behind the two new techniques.

In both of our extensions, quadric error matrices are allocated, updated, and evaluated only along the processing boundaries, which sweep over the entire mesh, visiting every triangle exactly once. As in [Lindstrom 2000], triangles add their quadric error to the respective matrices the moment they are processed. However, the life-time of each matrix is limited to the duration that a processing boundary pierces the grid cell associated with the matrix. More precisely, a grid cell is *active* whenever it contains vertices from the processing boundary. Quadric matrices are stored only with currently active grid cells, thus obviating the need to explicitly store the entire grid. Similar to the original method, but more efficient since only the active subset of the cells intersected by the surface are stored, this sparse grid representation is implemented using a hash table.

With each active cell, we also store a counter that is incremented whenever a new vertex falls into this cell and decremented whenever a vertex from this cell is used for the last time (Figure 4). Thus, the active cells are those with non-zero vertex counters. When the value of the counter drops to zero, we compute the cell's representative vertex from the accumulated quadric matrix and place it on the output. The grid cell, including the counter and the quadric matrix, is then deallocated (i.e. removed from the hash table).

Notice that the processing boundary may enter and leave any given cell several times when multiple layers of the mesh pass through the cell. Therefore we will often generate one representative vertex for each layer. This is in contrast to the original approach that represents all mesh layers passing through a grid cell with a single vertex. This difference becomes especially noticeable for aggressive simplification, as illustrated by the simplified blade model in Figure 6. The original approach collapses many layers into one vertex, which modifies the underlying topology and leads to poor positioning of the representative vertex.

Occasionally we generate more than one vertex per layer for a single cell, e.g. when an edge with no endpoint in the cell divides the layer passing through the cell in two (see Figure 5). However, such additional vertices are generally beneficial since they serve to unfold what would otherwise become non-manifold mesh pieces. Indeed, a single additional vertex can sometimes untangle multiple non-manifold vertices, as evidenced by Table 2.

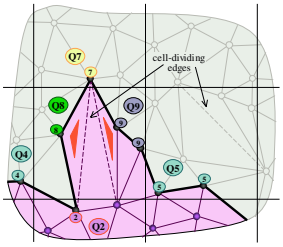
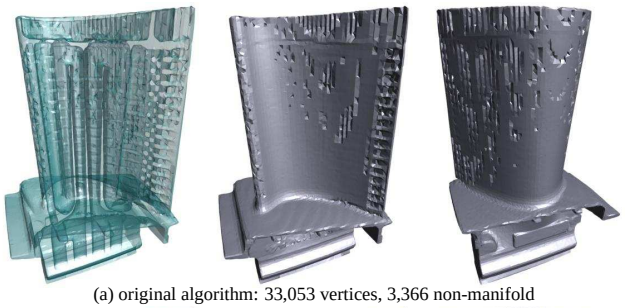


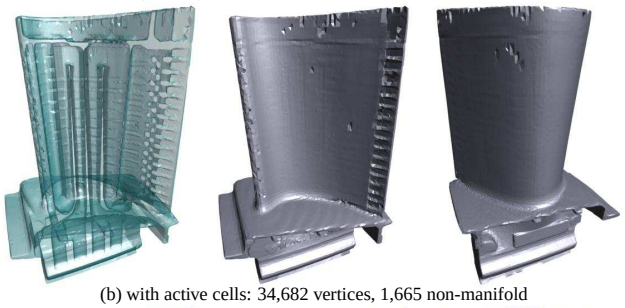
Figure 5: The presence of *cell-dividing* edges (shown stippled) results in more than one representative vertex per grid cell for a single mesh layer. Here they prevent Q8 and Q9 from merging. This is beneficial as it prevents the output triangles {Q7,Q8,Q2} and {Q2,Q9,Q7} (shown in red) from collapsing into a pair of oppositely oriented triangles with non-manifold edges. The grid cell on the right illustrates another example of a cell-dividing edge.

The framework just described is the basis of our first extension to OoCS. As should be evident, it involves a minor change to the original algorithm—an additional per-cell counter and the ability to remove cells—yet it can have a dramatic impact on the topological quality of the output mesh. For example, disconnected components are guaranteed not to be merged if the processing sequence traverses the mesh one component at a time. Nevertheless, it is still possible for pinching to occur, e.g. if the processing boundary wraps around and re-enters an already active cell, or if multiple boundaries simultaneously pass through a cell. Ideally, we would like to further partition each cluster of vertices within a cell into connected components, which would eliminate pinching altogether. This is accomplished in our second and more elaborate extension.

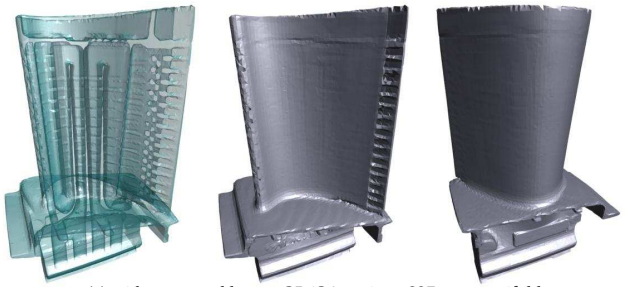
Conceptually, we construct connected components within a cell by initially assigning each new vertex introduced in the processing sequence to a unique cluster. Then, for each triangle processed, we collapse clusters that both share an edge of the triangle and are part of the same grid cell. As a result, vertices from the input mesh are merged only if they share an edge, which in effect renders our vertex clustering algorithm as an edge collapse method. That is, our method is functionally equivalent to collapsing all edges whose vertices are contained in the same grid cell. Indeed, for simplicity, our implementation explicitly makes use of edge collapse and a conventional mesh data structure for the partially simplified mesh near the processing boundary. Contrary to conventional edge collapse methods, however, we do not have access to the entire input mesh. In the context of processing sequences, this implies maintaining which cluster each of the vertices on the processing boundary belongs to, merging clusters (i.e. collapsing edges), and keeping track of when a single cluster (as opposed to all clusters) within a cell becomes inactive. We accomplish the latter by adding the vertex counters of two partial clusters when merging them.



(a) original algorithm: 33,053 vertices, 3,366 non-manifold



(b) with active cells: 34,682 vertices, 1,665 non-manifold



(c) with connected layers: 35,134 vertices, 897 non-manifold

Figure 6: Semitransparent and opaque views of the turbine blade model, simplified using the original OoCS algorithm and our extensions to it. Notice the severe pinching in (a) as interior and exterior layers of the surface pass through single grid cells and are collapsed. The grid dimensions are $57 \times 96 \times 44$ in all three cases.

The order in which triangles and vertices are finalized does not directly result in a proper output sequence. This is because output triangles are usually generated before their vertices are ready for output, i.e. before their clusters become inactive. Therefore, output triangles are first put into a waiting area, as illustrated earlier in Figure 3. Whenever a vertex is output, we check whether waiting triangles that reference the vertex are eligible for output, i.e. whether all three of their vertices have been output, and if so output and deallocate the triangle. Because the generated vertices and triangles can be written (almost) directly to disk, the memory requirements of this approach are independent of the size of the output mesh. Rather, the memory usage depends solely on the maximal length of the processing boundary.

The information on border edges available during sequenced processing further improves the quality of the simplified mesh. Instead of adding tangential error terms for every edge that completely neutralize each other only across coplanar triangles, as suggested in [Lindstrom and Silva 2001], we explicitly penalize deviation from actual border edges using specialized quadric error matrices, similar to [Garland and Heckbert 1997].

5.1 Results

Figure 6 and Table 2 highlight the results of using our boundary-based processing methods to simplify the turbine blade model. Notice the large reduction in non-manifold vertices relative to the small increase in total number of vertices (in all cases a higher than 100% efficiency). As can be seen in Figure 6, many of these non-manifold vertices are the result of pinching. These models were simplified on a 2 GHz Pentium 4 Windows 2000 PC with 1 GB of RAM.

T_{out}	method	V_{out}	V_{nm}	$\frac{\Delta V_{\text{nm}}}{\Delta V_{\text{out}}}$	RAM (MB)	time (s)	speed (T_{in}/s)
70,546	original	33,053	3,366	–	10.7	5.62	314 K
	active cells	34,682	1,665	104%	7.3	5.78	305 K
	connected	35,134	897	119%	3.4	6.18	285 K
122,470	original	59,675	3,103	–	11.0	6.97	253 K
	active cells	60,618	2,008	116%	8.0	7.07	250 K
	connected	61,109	1,172	135%	3.4	7.10	249 K
230,642	original	113,961	3,472	–	21.9	9.02	196 K
	active cells	114,695	2,436	141%	13.8	9.13	193 K
	connected	115,238	1,360	165%	3.5	9.15	193 K

Table 2: Results of simplifying the blade model using the original OoCS algorithm and our "active cells" and "connected layers" extensions based on processing sequences. The fifth column lists the change in number of non-manifold vertices (ΔV_{nm}) over the change in total number of output vertices (ΔV_{out}) relative to the original method. Note that, on average, each added vertex generally makes more than one previously non-manifold vertex manifold. The last column reports the simplification speed as number of input triangles processed per second.

In addition to higher quality meshes, our "connected layers" method is also more memory efficient than the original method, which requires storing the entire simplified mesh in-core. We simplified the St. Matthew model from 373 million triangles to 23 million using these two methods on a 250 MHz SGI Onyx2 with 40.5 GB of RAM. The original OoCS took 67 minutes and used 3,282 MB of RAM, while the boundary-based method took 83 minutes and used only 121 MB of RAM; a reduction in memory usage by a factor of 27.

6 Buffer-Based Processing

In this section we show how an adaptive simplification method based on iterative edge contraction [Garland and Heckbert 1997] can use processing sequences. We modify the algorithm by Wu and Kobbelt [2003], which uses a buffering mechanism based on a geometric triangle ordering that directly maps to the buffer-based computation abstraction of processing sequences.

Their algorithm uses three operations, READ triangle, DECIMATE triangles, and WRITE triangle, to maintain an *active* portion of the mesh that is memory-resident and eligible for simplification. It stores a quadric error matrix with each active vertex.

READ inputs the next triangle in the triangle ordering, hooks it into the active mesh, and adds the quadric error of the triangle to the quadric matrices of its three vertices.

DECIMATE chooses an edge with minimal quadric error that is eligible for collapse, merges its two active vertices and their quadric matrices, and eliminates the triangles that share the edge. Constant-time complexity is achieved by choosing this edge from only a small, fixed-size set of random candidates.

WRITE chooses a triangle with maximal quadric error that has an edge on the output boundary and outputs it. Again, the search is restricted to a random set of potential output triangles for constant-time selection. When all triangles incident on a vertex have been written, the vertex is deleted together with its quadric.

Wu and Kobbelt READ triangles to keep an in-core buffer full, and interleave batches of WRITE and DECIMATE operations to maintain a simplified mesh whose resolution corresponds to a user-specified percentage reduction of the original mesh.

Figure 7 illustrates Wu and Kobbelt's algorithm adapted to the processing sequence paradigm. The unprocessed region is shown at the top. Shown in black is the processing boundary of the input sequence, where new triangles are read and where vertices accumulate the quadric error of incoming triangles in their quadric matrices. Furthermore, the input sequence provides information about connectivity and border edges to the in-core buffer (shown

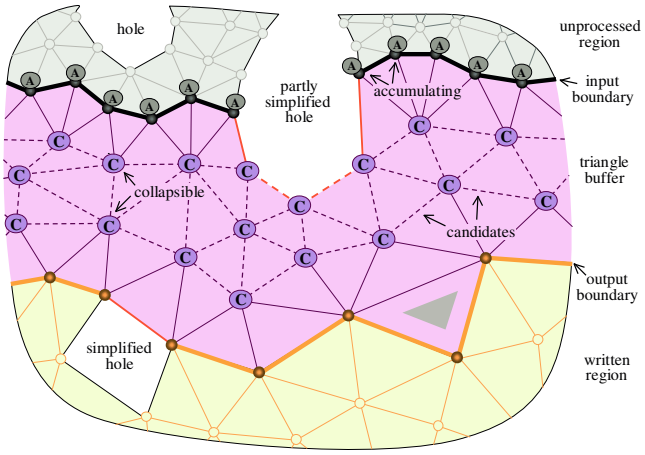


Figure 7: A 2D illustration of buffer-based computation using processing sequences. Such an algorithm, here the simplification algorithm of Wu and Kobbelt [2003], operates on a triangle buffer between an input and an output boundary. Triangles generated at the input boundary are read from disk. They are not immediately processed, but used to (re-)fill the buffer in which the actual processing takes place. Their quadric error is added to the accumulating error quadrics of vertices on the input boundary. Edge collapse operations are restricted to those edges (shown dashed) that are not incident to vertices on either boundary. They merge collapsible quadrics. Triangles adjacent to the output boundary empty the buffer and are written to disk. The next candidate for output is the triangle with all three vertices on the output boundary (shown in gray).

in the middle). Edge collapse operations are disallowed for edges that have vertices on the input or the output boundary. After decimation, the surviving triangles are output in the form of a second processing sequence. Again connectivity and border information is stored along the boundary of the output sequence, allowing for further processing such as on-the-fly compression.

In order to output a processing sequence, we slightly modify Wu and Kobbelt's method to select triangles to output. As in the original method, we try to minimize the number of "start" operations (compare with Figure 2) for output triangles in order to keep the output boundary as short and the triangle buffer as connected as possible. This is achieved by choosing an output triangle only from triangles incident to an edge of the output boundary, and allowing "start" operations only if no such triangle is available. Furthermore, we favor outputting triangles whose three vertices are on the output boundary, i.e. "end," "fill," and "join" operations (in that order), since its vertices can no longer be involved in an edge collapse. When no such triangle exists, we choose (using multiple choice selection) some triangle with one vertex between the input and output boundaries, i.e. we perform an "add" operation. To determine which such triangle to output from a set of multiple choice candidates, we choose the one with the largest quadric error at the non-boundary vertex rather than evaluating the quadric error for the entire triangle, as in Wu and Kobbelt's method. We decided on this approach since, in our method, vertices on the output boundary have no impact on the error involved in future potential edge collapses.

When a new vertex is encountered in the input, a corresponding vertex is allocated in the in-core mesh data structure. The processing sequence API optionally maintains a mapping between the vertices it knows to be on the boundary and corresponding client-side vertices. This eliminates the need for the client to establish this mapping, e.g. via hashing on global vertex indices, for each previously visited vertex in the sequence, which gives us connectivity reconstruction essentially for free. Furthermore, using processing sequences, the mesh border edges are not (miss-)classified as input boundary edges, as in [Wu and Kobbelt 2003]. This allows border edges and nearby incident edges to be directly involved in decimation; we need not set aside precious space in the fixed-size mesh buffer to hold such edges until the entire input mesh has been read.

mesh name	T_{in}	T_{buf}	T_{out}	p (%)	RAM (MB)	time (h:m:s)	speed (T_{in}/s)
happy buddha	1,087,716	400 K	21,754	2	41	27	40,663
		400 K	217,544	20	41	26	42,434
blade	1,765,388	400 K	35,308	2	41	41	43,292
		400 K	353,078	20	42	45	39,396
david (2mm)	8,254,150	400 K	82,541	1	43	3:06	44,491
		400 K	825,415	10	44	3:50	35,915
lucy	28,055,742	400 K	280,557	1	43	10:05	46,408
		400 K	1,402,788	5	43	10:45	43,502
david (1mm)	56,230,343	400 K	562,303	1	48	14:40	63,898
		400 K	2,811,517	5	48	16:07	58,149
st. matthew	372,767,445	800 K	559,152	0.15	104	1:30:32	68,624
		800 K	1,863,837	0.5	105	1:33:00	66,804
isosurface	467,614,855	4 M	2,346,907	0.5	776	2:25:11	53,883

Table 3: Results of buffer-based simplification. T_{buf} specifies the size (in number of triangles) of the in-core buffer, and p is the simplification ratio. For these results, we used 8 multiple choice candidates. The top four models were simplified on an 800 MHz Linux PC, while the bottom three were simplified on a 2 GHz Windows PC.

6.1 Results

Table 3 lists the results of running our adaptive simplification method on several meshes. The majority of these meshes were simplified on an 800 MHz Pentium 3 with 880 MB of RAM, running Red Hat Linux 7.1 (allowing a fair comparison with several other methods, including [Wu and Kobbelt 2003; Lindstrom and Silva 2001; Cignoni et al. 2003]). The larger meshes were simplified on the PC described in Section 5.1. Except for lower memory requirements and higher speed, these results generally agree with those published by Wu and Kobbelt. The performance differences may be attributed in part to our method not requiring hashing, but may also be the result of a more efficient implementation. Finally, Figure 8 shows a simplified mesh produced by our method.

7 Conclusion

We have demonstrated that the mesh access provided by processing sequences allows highly efficient out-of-core computations on large meshes. We have illustrated this by adapting two different simplification algorithms to access the mesh through a prototype of our processing sequence API [Isenburg et al. 2003]: one using boundary-based, the other using buffer-based processing. In both cases using processing sequences was beneficial.

Boundary-based processing significantly reduces the memory-requirements of the vertex clustering based simplification method of Lindstrom [2000], enabling it to produce very large output meshes in a single pass. Furthermore, the quality of the simplified mesh improves significantly—especially in the case of aggressive simplification—as multiple mesh layers that pierce one grid cell are no longer collapsed into a single vertex. Finally, information about border edges supports dedicated error quadrics that better preserve surface boundaries.

Buffer-based processing readily accommodates the stream-based simplification method of Wu and Kobbelt [2003], providing it with a triangle ordering that keeps the buffer maximally connected. Furthermore, the indexed nature of processing sequences removes the overhead associated with polygon soups. Additional speed-ups are gained through assistance in reconstructing connectivity. Finally, information about border edges solves the issue of uncollapsible triangles clogging the triangle buffer.

We would like to see these two computational abstractions applied to other types of mesh processing, in particular parameterization and remeshing algorithms. For this we will make the processing sequence API available to other researchers. Their needs and experiences may result in improvements to our current API [Isenburg et al. 2003] or in slight changes to its definition.

The maximal length of the processing boundary directly impacts

Figure 8: Adaptive simplification of David (2mm) to 1% of the input mesh with a stream-based simplifier using processing sequences and buffer-based processing.

the memory footprint of the simplification process. For the isosurface data set this length is 1.6 million vertices, far above the $O(\sqrt{n})$ worst-case bound established by Bar-Yehuda and Gotsman [1996]. Our processing sequences are currently generated by a compression scheme that traverses the mesh with a heuristic for lowering the bit rate, and does not attempt to keep the maximal boundary length small. Currently we are investigating how to create processing sequences that have a smaller footprint.

Future work will also address *on-the-fly* compression of processing sequences that are either the output of an algorithm or created from scratch with the converter and geometric sorting. When compressing processing sequences in a traversal order that is dictated by an application rather than chosen by the compressor we can expect lower compression rates. However, this will allow both inputting and outputting processing sequences in compressed form.

Acknowledgements

This work was performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48. We thank Kitware for the Blade model. The Happy Buddha and the Lucy model are courtesy of the Stanford Computer Graphics Laboratory. The two versions of David and the St. Matthew statue are courtesy of the Digital Michelangelo Project at Stanford University. The isosurface is courtesy of the LLNL ASCI Program.

References

BAR-YEHUDA, R., AND GOTSMAN, C. 1996. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics 15*, 2, 141–152.

BERNARDINI, F., MARTIN, I., MITTLEMAN, J., RUSHMEIER, H., AND TAUBIN, G. 2002. Building a digital model of Michelangelo's Florentine Pieta. *IEEE Computer Graphics and Applications 22*, 1, 59–67.

BOGOMJAKOV, A., AND GOTSMAN, C. 2001. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Graphics Interface'01 Proceedings*, 81–90.

BRODSKY, D., AND WATSON, B. 2000. Model simplification through refinement. In *Graphics Interface'00 Proceedings*, 221–228.

CHOUDHURY, P., AND WATSON, B. 2002. Completely adaptive simplification of massive meshes. Tech. Rep. CS–02–09, Northwestern University.

CIGNONI, P., MONTANI, C., ROCCHINI, C., AND SCOPIGNO, R. 2003. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*. To appear.

DEERING, M. 1995. Geometry compression. In *SIGGRAPH 95 Proceedings*, 13–20.

EVANS, F., SKIENA, S. S., AND VARSHNEY, A. 1996. Optimizing triangle strips for fast rendering. In *Visualization'96 Proceedings*, 319–326.

GARLAND, M., AND HECKBERT, P. 1997. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Proceedings*, 209–216.

GARLAND, M., AND SHAFFER, E. 2002. A multiphase approach to efficient surface simplification. In *Visualization'02 Proceedings*, 117–124.

HO, J., LEE, K., AND KRIEGMAN, D. 2001. Compressing large polygonal models. In *Visualization'01 Proceedings*, 357–362.

HOPPE, H. 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Visualization'98 Proceedings*, 35–42.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH 99 Proceedings*, 269–276.

ISENBURG, M., AND GUMHOLD, S. 2003. Out-of-core compression for gigantic polygon meshes. In *SIGGRAPH 2003 Proceedings*, 935–942.

ISENBURG, M., GUMHOLD, S., AND SNOEYINK, J. 2003. Processing sequences: A new paradigm for out-of-core processing on large meshes. Preprint available at http://www.cs.unc.edu/~isenburg/oocc/.

LEVOY, M., PULLI, K., CURLESS, B., RUSINKIEWICZ, S., KOLLER, D., PEREIRA, L., GINZTON, M., ANDERSON, S., DAVIS, J., GINSBERG, J., SHADE, J., AND FULK, D. 2000. The Digital Michelangelo Project. In *SIGGRAPH 2000 Proceedings*, 131–144.

LINDSTROM, P., AND SILVA, C. 2001. A memory insensitive technique for large model simplification. In *Visualization'01 Proceedings*, 121–126.

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. In *SIGGRAPH 2000 Proceedings*, 259–262.

MCMAINS, S., HELLERSTEIN, J., AND SEQUIN, C. 2001. Out-of-core build of a topological data structure from polygon soup. In *Proceedings of the 6th ACM Symposium on Solid Modeling and Applications*, 171–182.

PRINCE, C. 2000. *Progressive Meshes for Large Models of Arbitrary Topology*. Master's thesis, University of Washington.

ROSSIGNAC, J., AND BORREL, P. 1993. Multi-resolution 3d approximation for rendering complex scenes. In *Modeling in Computer Graphics*, 455–465.

SHAFFER, E., AND GARLAND, M. 2001. Efficient adaptive simplification of massive meshes. In *Visualization'01 Proceedings*, 127–134.

SILVA, C., CHIANG, Y., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization'02 Course Notes*.

TOUMA, C., AND GOTSMAN, C. 1998. Triangle mesh compression. In *Graphics Interface'98 Proceedings*, 26–34.

WU, J., AND KOBBELT, L. 2003. A stream algorithm for the decimation of massive meshes. In *Graphics Interface'03 Proceedings*, 185–192.