Out-of-Core Compression for Gigantic Polygon Meshes

Martin Isenburg*
University of North Carolina
at Chapel Hill

Stefan Gumhold†
WSI/GRIS
University of Tübingen

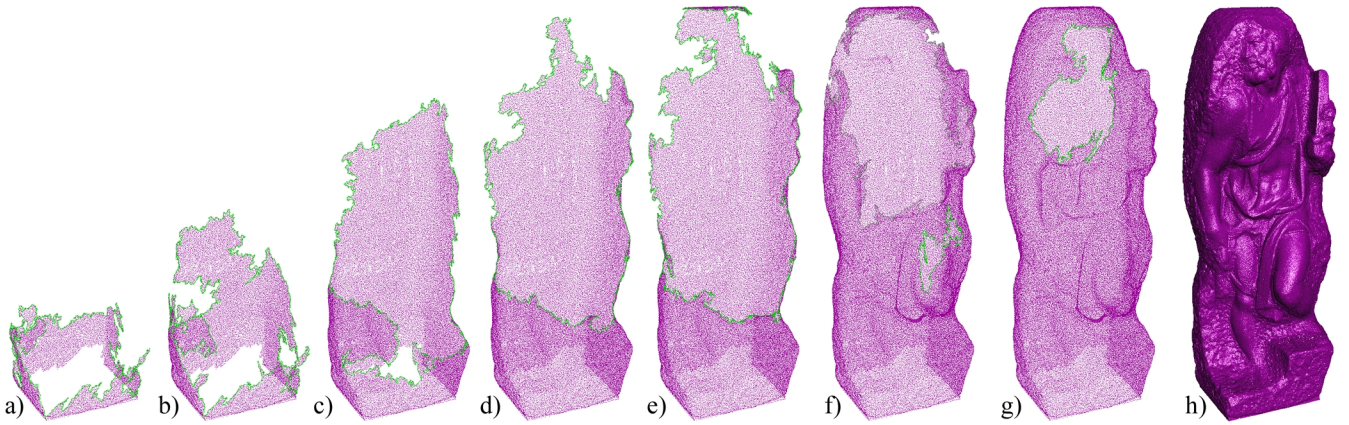


Figure 1: **(a) - (g)** Visualization of the decompression process for the St. Matthew statue. **(h)** Example Out-of-Core Rendering.

Abstract

Polygonal models acquired with emerging 3D scanning technology or from large scale CAD applications easily reach sizes of several gigabytes and do not fit in the address space of common 32-bit desktop PCs. In this paper we propose an out-of-core mesh compression technique that converts such gigantic meshes into a streamable, highly compressed representation. During decompression only a small portion of the mesh needs to be kept in memory at any time. As full connectivity information is available along the decompression boundaries, this provides seamless mesh access for incremental in-core processing on gigantic meshes. Decompression speeds are CPU-limited and exceed one million vertices and two million triangles per second on a 1.8 GHz Athlon processor.

A novel external memory data structure provides our compression engine with transparent access to arbitrary large meshes. This out-of-core mesh was designed to accommodate the access pattern of our region-growing based compressor, which - in return - performs mesh queries as seldom and as local as possible by remembering previous queries as long as needed and by adapting its traversal slightly. The achieved compression rates are state-of-the-art.

CR Categories: I.3.5 [Computational Geometry and Object Modeling]: Boundary representations

Keywords: external memory data structures, mesh compression, out-of-core algorithms, streaming meshes, processing sequences.

*isenburg@cs.unc.edu http://www.cs.unc.edu/~isenburg/oocc
†gumhold@gris.uni-tuebingen.de

1 Introduction

The standard representation of a polygon mesh uses an array of floats to specify the vertex positions and an array of integers containing indices into the vertex array to specify the polygons. For large and detailed models this representation results in files of gigantic size that consume large amount of storage space. The St. Matthew model from Stanford's Digital Michelangelo Project [Levoy et al. 2000], for example, has over 186 million vertices resulting in more than six gigabytes of data. Transmission of such gigantic models over the Internet consumes hours and even loading them from the hard drive takes tens of minutes.

The need for more compact representations has motivated research on mesh compression and a number of efficient schemes have been proposed [Deering 1995; Taubin and Rossignac 1998; Touma and Gotsman 1998; Gumhold and Strasser 1998; Li and Kuo 1998; Rossignac 1999; Bajaj et al. 1999; Isenburg and Snoeyink 2000; Lee et al. 2002]. Ironically, none of these schemes is capable—at least not on common desktop PCs—to deal with meshes of the giga-byte size that would benefit from compression the most. Current compression algorithms and some of the corresponding decompression algorithms can only be used when connectivity and geometry of the mesh are small enough to reside in main memory. Realizing this limitation, Ho et. al [2001] propose to cut gigantic meshes into manageable pieces and encode each separately using existing techniques. However, partitioning the mesh introduces artificial discontinuities. The special treatment required to deal with these cuts not only lowers compression rates but also significantly reduces decompression speeds.

Up to a certain mesh size, the memory requirements of the compression process could be satisfied using a 64-bit super-computer with vast amounts of main memory. Research labs and industries that create giga-byte sized meshes often have access to such equipment. But to decompress on common desktop PCs, at least the memory foot-print of the decompression process needs to be small. In particular, it must not have memory requirements in the size of the decompressed mesh. This eliminates a number of popular *multi-pass* schemes that either need to store the entire mesh for connectivity decompression [Taubin and Rossignac 1998; Rossignac 1999; Bajaj et al. 1999] or that decompress connectivity and geometry in

separate passes [Isenburg and Snoeyink 2000; Karni and Gotsman 2000; Szymczak et al. 2002].

This leaves us with all *one-pass* coders that can perform decompression in a single, memory-limited pass over the mesh. Such schemes (de-)compress connectivity and geometry information in an interwoven fashion. This allows *streaming* decompression that can start producing mesh triangles as soon as the first few bytes have been read. There are several schemes that could be implemented as one-pass coders [Touma and Gotsman 1998; Gumhold and Strasser 1998; Li and Kuo 1998; Lee et al. 2002].

In this paper we show how to compress meshes of giga-byte size in one piece on a standard PC using an external memory data structure that provides transparent access to arbitrary large meshes. Our *out-of-core mesh* uses a caching strategy that accommodates the access pattern of the compression engine to reduce costly loads of data from disk. Our compressor is based on the scheme of Touma and Gotsman [1998] and uses degree coding and linear prediction coding to achieve state-of-the-art compression rates. It borrows ideas from Isenburg [2002] to adapt the traversal, from Gumhold and Strasser [1998] to handle mesh borders, and from Guéziec et al. [1998; 1999] to deal with non-manifold meshes. Our compressed format allows streaming, small memory foot-print decompression at speeds of more than 2 million triangles a second.

The snap-shots in Figure 1 visualize the decompression process on the St. Matthew statue. For steps (a) to (g) we stored every 1000th decompressed vertex in memory, while (h) is an example out-of-core rendering. Using less than 10 MB of memory, this 386 million triangle model loads and decompresses from a 456 MB file off the hard-drive in only 174 seconds. At any time only the green boundaries need to be kept in memory. Decompressed vertices and triangles can be processed immediately, for example, by sending them to the graphics hardware. The out-of-core rendering took 248 seconds to complete with most of the additional time being spent on computing triangle normals. These measurements were taken on a 1.8 Ghz AMD Athlon processor with an Nvidia Geforce 4200 card.

Our compressed format has benefits beyond efficient storage and fast loading. It is a *better* representation of the raw data for performing out-of-core computations on large meshes. Indexed mesh formats are inefficient to work with and often need to be *de-referenced* in a costly pre-processing step. The resulting polygon soups are at least twice as big and, although they can be efficiently batch-processed, provide no connectivity information. Our compressed format streams gigantic meshes through limited memory and provides seamless mesh access along the decompression boundaries, thereby allowing incremental in-core processing on the entire mesh.

The remainder of this paper is organized as follows: The next section summarizes related work on out-of-core processing, out-of-core data structures, and mesh compression. In Section 3 we introduce our out-of-core mesh and describe how to build it from an indexed mesh. Then, in Section 4, we describe our compression algorithm and report resulting compression rates and decompression speeds on the largest models that were available to us. The last section summarizes our contributions and evaluates their benefits for other algorithms that process gigantic polygon meshes.

2 Related Work

Out-of-core or external memory algorithms that allow to process vast amounts of data with limited main memory are an active research area in visualization and computer graphics. Recently proposed out-of-core methods include isosurface extraction, surface reconstruction, volume visualization, massive model rendering, and—most relevant to our work—simplification of large meshes. Except for the work by Ho et al. [2001], out-of-core approaches to mesh compression have so far received little attention.

The two main computation paradigms of external memory techniques are *batched* and *online* processing: For the first, the data is streamed in one or more passes though the main memory and computations are restricted to the data in memory. For the other, the data is processed through a series of (potentially random) queries. In order to avoid costly disk access with each query (e.g. thrashing) the data is usually re-organized to accommodate the anticipated access pattern. Online processing can be accelerated further by *caching* or *pre-fetching* of data that is likely to be queried [Silva et al. 2002].

Out-Of-Core Simplification methods typically make heavy use of batch-processing. Lindstrom [2000] first creates a vertex clustering [Rossignac and Borrel 1993] in the resolution of the output mesh and stores one quadric error matrix per occupied grid cell in memory. Indexed input meshes are first dereferenced into a polygon-soup and then batch-processed one a triangle at a time by adding its quadric to all cells in which it has a vertex. Later, Lindstrom showed together with Silva [2001] that the limitation of the output mesh having to fit in main memory can be overcome using a series of external sorts.

A different approach for simplifying huge meshes was suggested by Hoppe [1998] and Bernardini et al. [2002]: The input mesh is partitioned into pieces that are 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 approach of Hoppe automatically simplifies these boundaries at the next level, Bernardini et al. simply process the mesh more than once—each time using a different partitioning.

The methods discussed so far treat large meshes different 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. Addressing this issue, 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.

Albeit substantial differences, our out-of-core mesh is motivated by the same idea: it provides the mesh compressor transparent access to the connectivity and geometry of gigantic meshes. Therefore our compressor will produce the same result, no matter if used with our out-of-core mesh or with the entire mesh stored in-core.

Out-Of-Core Data Structures for Meshes have also been investigated by McMains et. al [2001]. They reconstruct complete topology information (e.g. including non-manifoldness) from polygon soups by making efficient use of virtual memory. Their data structure provides much more functionality than our compressor needs and so its storage requirements are high. Also, using virtual memory as a caching strategy would restrict us to 4 GB of data on a PC and we will need more than 11 GB for the St. Matthew statue.

The octree-based external memory mesh of Cignoni et. al [2003] could be adapted to work with our mesh compressor. It has roughly the same build times and stores only slightly more data on disk. However, their octree nodes do not store explicit connectivity information, which has to be built on the fly. While this is acceptable for a small number of loads per node, the query order of our compressor might require to load some nodes more often—especially if we used their octree-based mesh: its nodes are created through regular space partitioning, which is insensitive to the underlying connectivity, while our clusters are compact along the surface.

Mesh Compression techniques have always overlooked the memory requirements of the decompression process. So far meshes were moderately sized and memory usage was at most linear in mesh size. However, today's meshes most in need of compression are those above the 10 million vertex barrier. The memory limitation on common desktop PCs allows the decompression process

only a single, memory-limited pass over such meshes. This eliminates all schemes that need to store the entire mesh for connectivity decompression [Taubin and Rossignac 1998; Rossignac 1999; Bajaj et al. 1999] or that decompress connectivity and geometry in separate passes [Isenburg and Snoeyink 2000; Karni and Gotsman 2000; Szymczak et al. 2002]. Naturally, this constraint also prohibits the use of progressive approaches that require random mesh access for refinement operations during decompression [Taubin et al. 1998; Cohen-Or et al. 1999; Pajarola and Rossignac 2000; Alliez and Desbrun 2001a]. And finally, the sheer size of the data prohibits computation-heavy techniques such as traversal optimizations [Kronrod and Gotsman 2002], vector quantization [Lee and Ko 2000], or expensive per-vertex computations [Lee et al. 2002].

This leaves all those methods whose decompressor can be restricted to a single, memory-limited, computation-efficient pass over the mesh. This coincides with all those methods whose compressor can be implemented as a fast one-pass coder [Touma and Gotsman 1998; Gumhold and Strasser 1998; Li and Kuo 1998].

All these compression algorithms require access to explicit connectivity information, which is usually constructed in a preprocessing step. However, if the mesh does not fit into main memory already this is not possible. Therefore, Ho et. al [2001] suggest to cut large meshes into smaller pieces that can be dealt with in-core. They process each piece separately by first constructing explicit connectivity, which is then compressed with the two-pass coder of Rossignac [1999], before compressing the vertex positions with the parallelogram predictor of Touma and Gotsman [1998] in a third pass. They record additional information that specifies how to stitch the pieces back together after decoding.

The compression scheme we propose here has several advantages over that of Ho et. al [2001]. As we do not break up the model, our compression rates are 20 to 30 percent better. As we can decode the entire model in a single pass, our decompression speeds are about 100 times faster. Finally, as our decompressor streams the entire mesh through main memory with a small memory footprint, our compressed representation is useful beyond reduced file sizes and shortened download times. It supports efficient batch-processing for performing computation on large meshes while at the same time providing seamless access to mesh connectivity.

We should also mention *shape compression* methods [Khodakovsky et al. 2000; Gu et al. 2002; Szymczak et al. 2002] as they are especially well suited for converting detailed scanned datasets into highly compressed representations. These approaches remesh prior to compression under the assumption that not a particular mesh but rather the geometric shape that it represents is of interest. While these schemes were not designed to handle gigantic meshes, they might benefit from the concepts to out-of-core processing presented here. However, remeshing methods are not applicable to CAD data such as the Double Eagle model (shown in Figure 6).

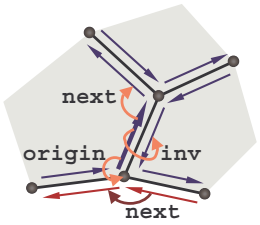
3 Out-of-Core Mesh

We use a half-edge data structure [Mantyla 1988] as foundation for our out-of-core data structure, because it gives us the functionality needed by the compression algorithm at minimal storage space consumption. A static data structure with an array V of vertices and an array of half-edges H is basically sufficient. We provide the compression algorithm with the following (see also Figure 2):

1. enumeration of all half-edges and ability to mark them as visited
2. access to the next and the inverse half-edge, and to the origin vertex
3. access to the position of a vertex and whether it is non-manifold
4. knowledge of border edges

3.1 Half-Edge Data-Structure

In order to efficiently support pure triangular meshes but also accommodate general polygonal meshes we have two modes for the out-of-core data structure: The *implicit mode* is designed for pure triangular meshes. Each internal half-edge consists of an index of its inverse half-edge and an index of its origin vertex. The three half-edges of a triangle are stored in successive order in the half-edge array H, such that the index of the next half-edge can be computed from the half-edge index `i` via $\texttt{next}(\texttt{i}) = 3*(\texttt{i}/3)+(\texttt{i}+1)\%3$. The *explicit mode* is used for polygonal meshes. A `next` index is explicitly stored with each half-edge, which means they can be arranged in any order in H.




```
struct IndexPair {
      int ci : 15;
      int li : 17;
};
struct HalfEdge {
      Index origin;
      Index inv;
      Index next;
      Index myself;
};
```

Figure 2: At each edge (black lines) of the mesh two directed half-edges (blue arrows) are incident, one for each incident face (light grey background). From each half-edge the `next` and `inverse` half-edges and the `origin` vertex are accessible. At the border additional border edges (red arrows) are created. Following their `next` pointers (dark red) cycles around the border loop. On the right is the syntax of an index-pair and of a half-edge. The `next` index-pair is used in explicit mode and for all border edges. The `myself` index-pair is only used for crossing half-edges.

The vertex array V contains the three coordinates x, y and z of each vertex in floating point or as a pre-quantized integer. In addition to V and H, a bit array for each vertex and each half-edge are necessary to maintain the status of manifoldness and visitation respectively. Border edges are also kept in a separate array. They always store an explicit index to the next border edge.

3.2 Clustering

The maximally used in-core storage space of the out-of-core mesh is limited to a user defined number of S_{incore} bytes. For efficient access to the mesh data during compression, a flexible caching strategy is necessary. For this we partition the mesh into a set of clusters. The total number of clusters c_{total} is

$$c_{\text{total}} = \frac{c_{\text{cache}}}{S_{\text{incore}}} \cdot S_{\text{vtx}} \cdot v, \tag{1}$$

where c_{cache} is the maximal number of simultaneously cached clusters, v is the number of mesh vertices, and S_{vtx} the per vertex size of our data structure. There are about six times as many half-edges as vertices, so S_{vtx} sums up to 60 bytes per vertex in implicit mode. For the St. Matthew model compressed with $S_{\text{incore}} = 384\text{MB}$ and $c_{\text{cache}} = 768$ this results in $c_{\text{total}} = 21381$ clusters.

Index-Pairs After clustering vertices and half-edges they are reindexed into so-called *index-pairs* (c_i, l_i) consisting of a cluster index c_i and a local index l_i. If possible, the index-pair (c_i, l_i) is packed into one 32-bit index to reduce the required storage space for the half-edge data structure. The number of bits needed for the cluster index is simply $\lceil \log_2 c_{\text{total}} \rceil$. For the St. Matthew example this is 15 bits, which leaves 17 bits for the local indices. A perfectly balanced clustering needs about $6 \cdot v / c_{\text{total}}$ different local indices for the half-edges. For the St. Matthew model this would be $52,482$. As we would like to use no more than $2^{17} = 131,072$ local indices, a sophisticated clustering approach is inevitable.

Caching Strategy For efficient access to the out-of-core mesh we cache the clusters with a simple LRU strategy. The vertex data, the half-edge data, and the binary flag data of a cluster are kept in separate files because they are accessed differently: The vertex data—once created—is only read. The half-edge data is both read and written when the out-of-core mesh is built, but only read when later queried by the compressor. The only data that needs to be read and written at compression time are the binary flags that maintain

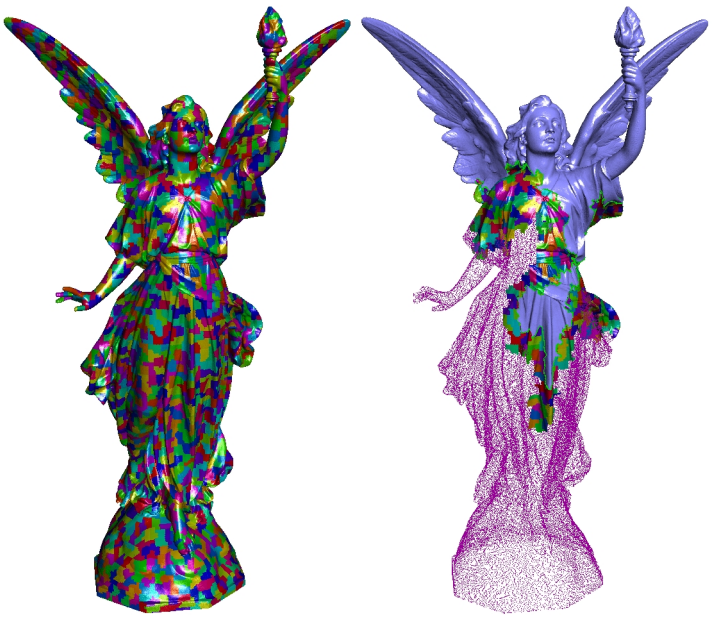


Figure 3: Visualization of the clustering and its usage during compression on the Lucy statue with $S_{\text{incore}} = 64MB$ and $c_{\text{cache}} = 128$. Already encoded regions are rendered with points while the rest is rendered with triangles. Cached clusters are colored.

the visitation status of half-edges. We maintain separate caches for this data, each having c_{cache} entries. For a given mesh traversal the quality of the caching strategy can be measured as the quotient $Q_{\text{read}}/Q_{\text{write}}$ of read/written clusters over the total number of clusters. For a full traversal the minimum quotient is one.

3.3 Building the Out-of-Core Mesh

Given the input mesh in an indexed format, we build the out-of-core mesh in six stages, all of them restricted to the memory limit S_{incore}:

1. vertex pass: determine the bounding box
2. vertex pass: determine a spatial clustering
3. vertex pass: quantize and sort the vertices into the clusters
4. face pass: create the half-edges and sort them into the clusters
5. matching of incident half-edges
6. linking and shortening of borders, search for non-manifold vertices

First Vertex Pass Each of the three vertex passes reads and processes the vertex array one time sequentially. In the first pass we only determine the number of vertices and the bounding box of the mesh. It can be skipped if this information is given. The required in-core storage for this pass is negligible.

Second Vertex Pass In this pass we compute a balanced, spatial clustering of the vertices into c_{total} clusters similar as Ho et al. [2001]. We subdivide the bounding box into a regular grid of cubical cells and count for each cell the number of vertices falling into it. Only for non-empty cells we allocate counters and keep them in a hash map. This ensures linear storage space consumption in the number of occupied cells. Then we partition the non-empty cells are into c_{total} compact clusters of balanced vertex counts using a graph partitioning package [MeTiS Version 4]. As input we build a $k-$nearest neighbor graph on the centroids of occupied cells using an approximate nearest neighbor package [ANN Version 0.2] (with $k = 6$ and 1% precision) and weigh its vertices using the vertex counts of the associated cells. Ho et al. [2001], on the other hand, derive the graph by connecting cells that are shared by a face. This could potentially give better cluster locality along the surface but would require an additional—especially expensive—face pass.

The second block in Table 1 shows results of cluster balancing. The time to build and cluster the graph is negligible. The standard deviation of the cluster sizes is fairly small and the minimum and maximum are within 10% of the average, which is sufficient for our needs. Figure 3 illustrates an example clustering on the Lucy statue.

Third Vertex Pass In the final pass over the vertices we sort the vertices into clusters and determine their index-pairs (c_i, l_i) using the cell partitioning generated in the last pass. Since the vertices of each cluster are stored in separate files, we use a simple buffering technique to avoid opening too many files at the same time. If a vertex falls into cluster c_i, which already contains k vertices, we assign it index-pair (c_i, k), increment k, and store its position in the respective buffer. If a buffer is full, its contents are written to disk. The mapping from vertex indices to index-pairs is stored in a *map file* that simply contains an array of index-pairs. For the St. Matthew model the map file is 729 MB and cannot be stored in-core.

Face Pass There is only one pass over the faces. We read a face and map its vertex indices to vertex index-pairs according to the map file. Then we create one half-edge for each of its edges, determine a suitable cluster, store them in this cluster, and—if necessary—also store them in some other cluster.

For each cluster c_i we create two files of half-edges. The *primary half-edge file* stores the half-edges sorted into cluster c_i within which they are locally indexed with l_i in the same way as vertices. The *secondary half-edge file* is only temporary. It stores copies of half-edges from other clusters that are needed later to match-up corresponding inverse half-edges. These so called *crossing half-edges* are incident to a half-edge of cluster c_i but reside in a different cluster. They are augmented by their own `myself` index-pair (see Figure 2) that is used later for matching the `inv` index-pairs.

As the map file is too large to be stored in-core, we split it into segments that are cached with a LRU strategy. For our test meshes the vertex indices of the faces were sufficiently localized, such that the read quotients Q_{read} of the map file cache was between 1 and 1.5. Cache thrashing will occur when the indices of the faces randomly address the vertex array. Then the mapping from indices to index-pairs needs to be established differently. One possibility is to perform several face passes, while each time storing a different chunk of the map file in memory and mapping only the stored indices. For the St. Matthew model three face passes would be sufficient when a chunk size of 256 MB is used. Another possibility is to use external sorting as proposed by Lindstrom and Silva [2001].

Before writing the half-edges to file, we store the index-pair of its origin vertex in the `origin` field and the index-pair of its target vertex in its `inv` field. The latter is only temporary and will be used during matching. Depending on the mesh mode we sort the half-edges differently into the primary and secondary half-edge files. In both modes, crossing half-edges receive their `myself` index-pair based on the cluster in which they are stored.

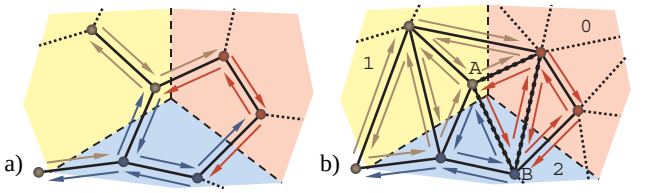


Figure 4: The sorting of the half-edges into the clusters. a) In explicit mode each half-edge is sorted into the cluster of its origin vertex. b) In implicit mode all half-edges of a triangle have to be in the same cluster, which is the cluster in which two or more vertices reside or any of three clusters otherwise.

In *explicite mode* the half-edges can be arranged arbitrarily within a cluster. We sort each half-edge into the cluster of its origin vertex. In this mode a half-edge is crossing when it has its target vertex in a different cluster. As they potentially have a matching inverse half-edge there, we insert them into the secondary file of that cluster. A small example is given in Figure 4a. The colors of the half-edges show, in which of the three clusters they are sorted.

In *implicit mode* all three half-edges of a triangle must be in successive order in the same cluster. They are placed into the cluster in which the triangle has two or three of its vertices. In case all three vertices fall into different clusters we simply select one of them. Figure 4b shows an example, where the dashed triangle spans three clusters. The so called *external half-edges*, like the one from vertex A to vertex B, will require special attention later, because they are stored in a different cluster than either of their incident vertices.

Again, a half-edge is crossing when its origin vertex and its target vertex are in different clusters. However, in implicit mode it is not obvious in which cluster its potential inverse match will be located. Therefore the secondary files are created in two stages. First we write crossing half-edges into a temporary file based on the smaller cluster index of their end vertices. Then we read these temporary files one by one and sort the contained crossing half-edges using their origin and target index-pairs ordered by increasing cluster index as key. Remember, the index-pair of the target vertex was stored in their `inv` field. Now all potential inverse matches among crossing half-edges are in successive order. Finally, all matching half-edges are entered into the secondary file of the cluster of their inverse, which can be determined from their `myself` index-pairs.

Matching of Inverse Half-Edges For each cluster we read the half-edges from the primary and the crossing half-edges from the secondary half-edge file. With the target vertex index-pairs in the `inv` fields, we again use the sorting strategy for matching inverse half-edges. We reduce the run time for sorting the half-edges with a single bucket-sort over all edges followed by a number of quick-sorts over the edges of each bucket. This results in a sort time of $O(n \log d_{\max})$, where n is the number of half-edges and $d_{\max}$ is the maximum vertex degree—usually a small constant. If origin and target vertex of an edge are both from the current cluster, the key used in the bucket-sort is the smaller of their local indices. Otherwise it is the local index from whichever vertex is in the current cluster. The key used in the quick-sorts is the index-pair of the vertex not used in the bucket-sort. External edges constitute a special case as they do not have any vertex in the current cluster necessary for the bucket-sort. These very rare external edges are gathered in a separate list and matched in the end using a single quick-sort.

All half-edges with the same vertex index-pairs have subsequent entries in the sorted array of half-edges. Looking at their number and orientation, we can distinguish four different types of edges:

1. border edge: an unmatched half-edge
2. manifold edge: two matched half-edges with opposite orientation
3. not-oriented edge: two matched half-edges with identical orientation
4. non-manifold edge: more than two matched half-edges

In case of a manifold edge we set the inverse index-pairs. In all other cases we pair the half-edges with newly created border edges, thereby guaranteeing manifold connectivity. This is similar to the cutting scheme that was proposed by Guéziec et al. [1998].

Border Loops and Non-Manifold Vertices The final three steps in building the out-of-core mesh consists of linking and shortening border loops and of detecting non-manifold vertices. First we cycle for each border half-edge via `inv` and `next` around the origin vertex until we hit another border half-edge. Its `next` field is set to the half-edge we started from. This links all border loops.

The second step can shorten border loops that are the result of cutting non-manifold edges. We iterate again over all border half-edges, this time checking if a sequence of `next`, `next`, and `origin` addresses the same vertex as `origin`. In this case we can match the `inv` fields of their incident half-edges and discard the border half-edges. This can shorten or even close a border loop.

The third and last step detects and marks non-manifold vertices using two binary flags per vertex and one binary flag per half-edge. Each flag is administered in one LRU-cached file per cluster with a bit container holding as many bits as there are vertices/half-edges in the cluster. The first vertex flag specifies whether a vertex was visited before, the second whether a vertex is non-manifold, while the half-edge flag marks visited half-edges. For each non-manifold vertex we also maintain an occurrence counter. We iterate over all half-edges. If the current edge has not been visited before, we cycle via `inv` and `next` around its origin and mark all out-going edges as visited until we come back to the edge we started. Then we check if the visited flag of the origin vertex has already been set. If yes, we mark this vertex as non-manifold using the second flag and create or increase its occurrence counter. If no, we set its visited flag. This way we mark all types of non-manifold vertices including those, which cannot be found along non-manifold edges.

mesh name	lucy	david (1mm)	st. matthew
vertices	14,027,872	28,184,526	186,836,665
in-core storage limit	96 MB	192 MB	384 MB
cached clusters	192	384	768
clusters	1,605	3,225	21,381
out-of-core size	871 KB	1.7 GB	11.2 GB
counter grid resolution	[283,163,485]	[340,196,856]	[409,1154,373]
ANN nearest neighbor	0:00:02	0:00:05	00:00:18
METIS graph partitioning	0:00:03	0:00:08	00:00:33
min vertices per cluster	8,569	8,550	8,360
max vertices per cluster	8,922	8,907	9,223
std over all clusters	0.00583	0.00529	0.01232
first vertex pass	0:00:14	0:00:34	0:03:24
second vertex pass	0:00:20	0:00:49	0:04:34
third vertex pass	0:00:51	0:02:04	0:53:56
face pass	0:05:22	0:11:01	2:09:20
matching	0:08:39	0:14:06	2:31:46
border link & shorten	0:00:01	0:00:39	0:09:06
non-manifold marking	0:03:26	0:06:36	1:02:06
total build time	0:18:57	0:35:52	6:54:17
compression time	0:48:46	0:13:42	3:36:24
Q_{read} half-edges	11.0	1.3	2.1
precision	20 bits	20 bits	20 bits
compressed size	47 MB	77 MB	456 MB

Table 1: Four blocks of measurements that characterize the out-of-core mesh: global parameters, performance of clustering stage, timings for different building steps, compression statistics. Times are in h:mm:ss taken on a Windows PC with 1 GB of memory and a 2.8 GHz Pentium IV processor. The system cache was software disabled.

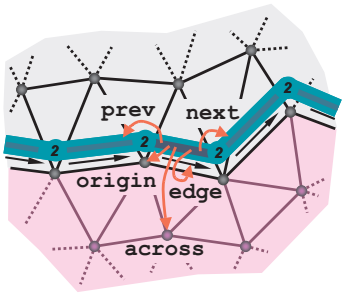
3.4 Results

Performance results of the out-of-core mesh are gathered in Table 1 for Lucy, David (1mm), and St. Matthew. The in-core memory was restricted to 96/192/384 MB and we allowed 192/384/768 clusters to be cached simultaneously. The resulting out-of-core meshes consumed 0.8/1.7/11.2 GB on disk with the build times being dominated by the face pass and the inverse half-edge matching.

The best compression times are achieved when enough clusters are cached to cover the entire compression boundary. But since its maximal length is not known in advance, this cannot be guaranteed. If too few clusters are cached, the compression process becomes heavily IO-limited. However, even then compression times are acceptable given the small in-core memory usage. Lucy, for example, has a poor cache quality factor Q_{read} of 11.0. Although Q_{read} for Lucy is much better with $c_{\text{cache}} = 384$, handling the larger number of files results in an overall longer running time. Twice the number of clusters as MB of in-core storage seemed a good trade-off between the two. When increasing S_{incore} to 128 MB and caching 256 clusters, then Q_{read} is 2.1 and Lucy compresses in about 5 minutes.

4 Compression

In order to enable fast out-of-core decompression with small memory foot-print, our mesh compressor performs a single pass over the mesh during which both connectivity and geometry are com-




```
struct Boundary {
   BoundaryEdge* gate;
   int length;
   int one_slots;
};
struct BoundaryEdge {
   BoundaryEdge* prev;
   BoundaryEdge* next;
   Index edge;
   bool border;
   Index origin;
   Index across;
   int slots;
};
```

Figure 5: The minimal data structures required for out-of-core (de-)compression: The `prev` and `next` pointers organize the boundary edges into double-linked loops. The `edge` index refers to the mesh edge that a boundary edge coincides with. The `origin` index refers to the vertex at the origin of this edge. The `across` index is used for (de-)compressing vertex positions with the parallelogram rule. It refers to the third vertex of an adjacent and already (de-)compressed triangle. For an out-of-core version of the decompressor, the indices `origin` and `across` are replaced with integer vectors containing the actual positions. The `slots` counter and the `border` flag are required for (de-)compressing the mesh connectivity with degree coding. The `Boundary` struct is used to store information about boundaries on the stack.

pressed in an interleaved fashion. It grows a region on the connectivity graph by including faces adjacent to its boundaries one by one. Whenever a previously unseen vertex is encountered, its position is compressed with a linear prediction. The decompressor can be implemented such that at any time it only needs to have access to the boundaries of this region. In order to decompress out-of-core these boundaries are equipped with some additional information. Maintaining such extra information also at compression time reduces the number of required queries to the out-of-core mesh.

4.1 Connectivity Coding

Our connectivity coder is based on the degree coder that was pioneered by Touma and Gotsman [1998] for triangle meshes and extended to polygon meshes by Isenburg [2002] and Khodakovsky et al. [2002]. Starting from an arbitrary edge it iteratively grows a region by always including the face adjacent to the *gate* of the *active boundary*. This boundary is maintained as loop of *boundary edges* that are doubly-linked through a *previous* and a *next* pointer. Each boundary edge maintains a *slot* count that specifies the number of unprocessed edges incident to its *origin*. The boundary edges also store an index to their corresponding half-edge in the mesh, which is used for queries. If the compressor is used in conjunction with the out-of-core mesh we want to make as few queries as possible. Therefore each boundary edge keeps a copy of the index to the *origin* and the *across* vertex as illustrated in Figure 5.

The face to be included can share one, two, or three edges with the active boundary. If it shares three edges, then the boundary ends and a new boundary is popped from the stack. If the stack is empty we iterate over the half-edges of the mesh to find any remaining components. If there are none, compression is completed. If the face shares two edges with the active boundary, no explicit encoding is needed. Otherwise it shares only one edge and has a *free vertex*. This can lead to three different cases: *add*, *split*, or *merge*.

In the most common case the free vertex is not on any boundary. Here we *add* the vertex to the boundary, record its degree, and update the slot counts. A more complex case arises if the free vertex is already on some boundary. If it is on the *active* boundary it *splits* this boundary into two loops that will be processed one after the other. A stack is used to temporarily buffer boundaries. We record the shorter direction and the distance in vertices along the boundary to reach the free vertex. If, however, the free vertex is on a boundary from the stack it *merges* two boundaries. This happens exactly once for every topological handle in the mesh. In addition to how the free vertex can be reached starting from that boundary's gate, we record the index of this boundary in the stack. The compressor does not query the out-of-core mesh to search for a free vertex along the boundaries. It uses the *origin* indices, which are stored (mainly for this purpose) with each boundary edge, to find this vertex.

The resulting code sequence contains the degree of every vertex plus information associated with the occurrence of split and merge operations. While it is not possible to avoid splits altogether, their number can be significantly reduced using an adaptive region growing strategy [Alliez and Desbrun 2001b]. Instead of continuing the mesh traversal at the current gate, one selects a gate along the boundary that is less likely to produce a split. We implemented the simple heuristic proposed by Isenburg [2002], which picks a boundary edge with a slot count of 1 if one exists, or stays where it is otherwise. However, we restrict this adaptive conquest to +/- 10 edges along the boundary, as moving the gate to a more distant location could cause a cache-miss on the next query to the out-of-core mesh. By keeping track on the number of these *one-slots* currently on the boundary we avoid searching for them unnecessarily.

Continuing compression on the smaller of the two boundary loops resulting from a split operation keeps the boundary stack shallow and the number of allocated boundary edges low. This helps further lowering the memory foot-print of the decoding process.

Holes in the mesh require special treatment. Isenburg [2002] suggests to include a hole into the active boundary in the same way it is done for faces. However, this requires immediate processing of all vertices around the hole. Since holes can be as large as, for example, the hole at the base the St. Matthew statue shown in Figure 1, this would result in an bad access pattern to the out-of-core mesh—potentially causing many cache-misses. Furthermore, it would lead to poor geometric predictions for all vertices around the hole since the parallelogram rule could not be applied.

Instead, similar to Gumhold and Strasser [1998], we record for every edge whether it is a border edge or not in the moment it joins the active boundary using a binary arithmetic context. If an edge has a slot count of zero on either end, we do not need to record this information explicitly. In this case the edge will be of the same type as the boundary edge it connects to via this *zero-slot*.

Non-manifold vertices are present when the neighborhood of a vertex is not homeomorphic to a disk or a half-disk. The out-of-core mesh provides our compressor with manifold connectivity and marks multiple occurrence of originally non-manifold vertices. We encode how to *stitch* these vertices back together using a simplified version of Guéziec et al.'s [1999] stack-based approach. Whenever a vertex is processed with an *add* operation we record whether it is manifold or not with a binary arithmetic context. For non-manifold vertices we specify whether this is its first appearance using a second arithmetic context. Only the first time a non-manifold vertex is encountered its position is compressed. These first-timers are then inserted into an indexable data structure. Each subsequent time this vertex makes a non-manifold appearance it is addressed with $\log_2 k$ bits among the k entries of that data structure. Then a third binary context is used to specify whether this was its last appearance or not. If yes, it is deleted from the data structure.

4.2 Geometry Coding

Quantization of the vertex positions into integer values is needed before they can be efficiently compressed with predictive coding. Especially for large datasets any loss in precision is likely to be considered unacceptable. Vertex positions are usually stored as 32-bit IEEE floating point numbers. In this format the least precise (e.g. the widest spaced) samples are those with the highest exponent. Within the range of this exponent all points have 24 bit of precision: 23 bit in the mantissa and 1 bit in the sign. Hence, once the bounding box (e.g. the exponent) is fixed we can capture the precision of the floating point samples by uniform quantization with 24 bits. Unless, of course, the data was specifically aligned with the origin to provide higher precision in some areas. But in general we can assume that the sampling within the bounding box is uniform.

Figure 6: Less than 1 MB of memory is used by the out-of-core process that loads, decompresses, and renders this 82 million triangle "double eagle" model in 78 seconds from a 180 MB file. This example application does not store triangles in memory, but immediately renders them with one call to glNormal3fv() and three calls to glVertex3iv() as the model is too large to be decompressed into the main memory of common desktop PCs. Loading and decompression alone takes only 63 seconds. Most of the additional 15 seconds are spent on computing triangle normals. The above frame was captured in anti-aliased 2048x768 dual-monitor mode on a 1.8 Ghz AMD Athlon processor with an Nvidia Geforce 4200.

For scanned datasets it is often not necessary to preserve the full floating point precision. The samples acquired by the scanner are typically not that precise, in which case the lowest-order bits contain only noise and not actually measured data. A reasonable quantization level keeps the quantization error just below the scanning error. In Table 2 we lists the resulting sample spacings per millimeter for different levels of quantization. For the 20 cm tall Buddha statue even 16 precision bits seem an overkill, whereas the 195 meter long Double Eagle is more likely to make use of all 24 bits.

The quantized vertices are compressed with the parallelogram rule [Touma and Gotsman 1998] in the moment they are first encountered. A vertex position is predicted to complete the parallelogram formed by the vertices of a neighboring triangle. The resulting corrective vectors are subsequently compressed with arithmetic coding. Vertices are always encountered during an add operation.

The first three vertices of each mesh component cannot be compressed with the parallelogram rule. While the first vertex has no obvious predictor, the second and third can be predicted with delta-coding [Deering 1995]. To maximize compression it is beneficial to encode correctors of less promising predictions with different arithmetic contexts [Isenburg and Alliez 2002]. For meshes with few components this hardly makes a difference, but the Power Plant and the Double Eagle model each consist of millions of components.

Other properties such as colors or confidence values can be treated similarly to vertex positions. However, for models of this size one might consider to store properties in separate files. Not everybody is interested, for example, in the confidence values that are stored for every vertex with all of Stanford's scanned datasets. Despite being in separate files, the decoder can add them on-the-fly during decompression, if the appropriate file is provided.

mesh		number of samples per millimeter				
name	extent [m]	16 bit	18 bit	20 bit	22 bit	24 bit
happy buddha	0.2	327	1311	5243	20972	83886
lucy	1.5	47	187	749	2995	11984
david	5.2	13	50	202	807	3226
double eagle	195	0.3	1.4	5.4	22	86
st. matthew	2.7	24	97	388	1553	6214

Table 2: This table reports the length of the longest bounding box size of a model in meters and the resulting number of samples per millimeter for different quantization.

4.3 Results

The compression gains of our representation over standard binary PLY are listed in Table 3. Depending on the chosen level of precision the compression ratios range from 1 : 10 to 1 : 20. Comparing measurements on the same models to Ho et al. [2001], our bit-rates are about 25% better. Another advantage of our compressed format is the reduced time to load a mesh from disk. Decompression speeds are CPU-limited and exceed one million vertices and two million triangles per second on a 1.8 GHz Athlon processor.

The compression rates for connectivity and for geometry at different precision levels are detailed separately in Table 4. One immediate observation is that an additional precision of 2 bits increases some geometry compression rates by about 6 bits per vertex (bpv) or more. While predictive coding is known not to scale well with increasing precision, here this is likely a sign for the precision of quantization being higher than that of the data samples. In this case the additional two bits only add incompressible noise to the data.

5 Conclusion

We presented a technique that is able to compress gigantic models such as the St. Matthew statue in one piece on standard desktop PCs. Our compression rates are about 25% lower and our decompression speeds about 100 times faster than the technique by Ho et al. [2001] that processes such models by cutting them in pieces.

For this, we introduced an external memory data structure, the out-of-core mesh, that provides our compressor with transparent access to large meshes. We described how to efficiently build this data structure from an indexed mesh representation using only limited memory. Our out-of-core mesh may also be useful to other algorithm that process large meshes. To use it efficiently the order of mesh queries should be localized, but most traversal-based processing is readily accommodated. While our current implementation only allows index-pairs to use a combined maximum of 32 bits, this data structure can theoretically handle arbitrary large meshes. Storing more bits per index-pair, however, will increase in-core and on-disk storage and make its build-up/usage more IO-limited.

Our compressed format has benefits beyond efficient storage and fast loading. It provides *better* access to large meshes than indexed formats or polygon soup by allowing to stream gigantic meshes through limited memory while providing seamless connectivity information along the decompression boundary. This *streaming mesh* representation offers a new approach to out-of-core mesh processing that combines the efficiency of batch-processing with the advantages of explicit connectivity information as it is available in online-processing. The concept of *sequenced processing* restricts the access to the mesh to a fixed traversal order, but at the same time provides full connectivity for the active elements of this traversal.

Traversing mesh triangles in a particular order is already used for fast rendering on modern graphics cards. The number of times a vertex needs to be fetched from the main memory is reduced by caching previously received vertices on the card. The triangles are sent to the card in a *rendering sequence* that tries to minimize cache misses [Hoppe 1999]. Misses cannot be avoided altogether due to the fixed size of the cache [Bar-Yehuda and Gotsman 1996].

In a similar spirit our compressed format provides *processing sequences* for more efficient mesh processing—but at a much larger scale. With the main memory as the "cache" we usually have more than enough storage space for all active elements throughout the traversal of a mesh. Therefore the analogue of a "cache miss" does not exist. Any algorithm that requires a complete mesh traversal

mesh name	number of						size of raw and compressed files on disk [MB]						load time	foot-print
	vertices	triangles	components	holes	handles	non-manifold	ply	16 bit	18 bit	20 bit	22 bit	24 bit	[sec]	[MB]
happy buddha	543,652	1,087,716	1	0	104	0	20	1.6	1.9	2.2	2.5	3.0	0.68	0.7
david (2mm)	4,129,614	8,254,150	2	1	19	4	150	7	10	12	15	17	4.1	1.3
power plant	11,070,509	12,748,510	1,112,199	1,221,511	10	37,702	285	19	23	28	32	35	8.9	0.7
lucy	14,027,872	28,055,742	18	29	0	64	508	28	37	47	58	70	14.6	1.5
david (1mm)	28,184,526	56,230,343	2,322	4,181	137	1,098	1,020	44	61	77	93	108	27	2.8
double eagle	75,240,006	81,806,540	5,100,351	5,291,288	1,534	3,193,243	1,875	116	146	180	216	244	63	0.7
st. matthew	186,836,665	372,767,445	2,897	26,110	483	3,824	6,760	236	344	456	559	672	174	9.4

Table 3: This table lists vertex, triangle, component, hole, handle, and non-manifold vertex counts for all meshes. Furthermore, the size of a binary ply file containing three floats per vertex and three integers per triangle is compared to our compressed representation at 16, 18, 20, 22, and 24 bits of precision. Also the total time in seconds required for loading and decompressing the 20 bit version on a 1.8 GHz AMD Athlon processor is reported. Finally, we give the maximal memory foot-print of the decompression process in MB.

mesh name	compression rates [bpv]						decompression speed [sec]					decompression and rendering speed [sec]				
	conn	16 bits	18 bits	20 bits	22 bits	24 bits	16 bits	18 bits	20 bits	22 bits	24 bits	16 bits	18 bits	20 bits	22 bits	24 bits
happy buddha	2.43	21.79	26.44	32.15	36.92	43.95	.75	.81	.97	1.13	1.38	1.15	1.21	1.37	1.53	1.79
david (2mm)	1.50	12.54	17.81	23.22	28.37	34.13	4.9	5.3	5.7	6.2	7.1	7.7	8.0	8.5	9.0	10.3
power plant	2.50	11.57	15.26	18.54	21.48	24.23	11.1	11.8	12.5	13.4	15.1	14.9	15.7	16.5	17.4	19.2
lucy	1.88	14.60	20.41	26.51	32.87	39.08	17.8	18.9	21.1	22.8	27.3	26.7	28.0	30.2	32.1	36.7
david (1mm)	1.79	11.32	16.50	21.20	25.99	30.43	33	35	38	41	45	51	53	56	60	64
double eagle	3.39	9.58	12.92	16.66	20.67	23.84	77	81	88	94	113	94	105	110	119	134
st. matthew	1.84	8.83	13.71	18.86	23.63	28.61	215	228	242	259	294	327	351	363	384	419

Table 4: This table details compression rates and decompression speeds on our example models. Compression rates are reported in bits per vertex (bpv) separately for connectivity and for geometry at 16, 18, 20, 22, and 24 bits of precision. Decompression times are reported first for loading/decompression alone and then for loading/decompression and out-of-core rendering. All timings are taken on a Dell Inspiron 8100 laptop with an Intel 1.1 Ghz Mobile Pentium III processor and a Nvidia Geforce2go card. Compare to Table 3.

without being particular about its order can perform computations at decompression speed—we can envision an entire breed of mesh processing algorithms adapted to this kind of mesh access.

Acknowledgements The Happy Buddha and Lucy are courtesy of the Stanford Computer Graphics Laboratory. The Power Plant model was provided by the Walkthru Project at the University of North Carolina at Chapel Hill. The Double Eagle model is courtesy of Newport News Shipbuilding. The two Davids and the St. Matthew statue are courtesy of the Digital Michelangelo Project at Stanford University.

References

ALLIEZ, P., AND DESBRUN, M. 2001. Progressive encoding for lossless transmission of 3D meshes. In *SIGGRAPH'01 Conference Proceedings*, 198–205.

ALLIEZ, P., AND DESBRUN, M. 2001. Valence-driven connectivity encoding for 3D meshes. In *Eurographics'01 Conference Proceedings*, 480–489.

ANN. Version 0.2. A library for approximate nearest neighbor searching by D. Mount and S. Arya. *University of Maryland*.

BAJAJ, C., PASCUCCI, V., AND ZHUANG, G. 1999. Single resolution compression of arbitrary triangular meshes with properties. In *Data Compression'99*, 247–256.

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.

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

COHEN-OR, D., LEVIN, D., AND REMEZ, O. 1999. Progressive compression of arbitrary triangular meshes. In *Visualization'99 Conference Proceedings*, 67–72.

DEERING, M. 1995. Geometry compression. In *SIGGRAPH'95 Conf. Proc.*, 13–20.

GU, X., GORTLER, S., AND HOPPE, H. 2002. Geometry images. In *SIGGRAPH'02 Conference Proceedings*, 355–361.

GUÉZIEC, A., TAUBIN, G., LAZARUS, F., AND HORN, W. 1998. Converting sets of polygons to manifolds by cutting and stitching. In *Visualization'98*, 383–390.

GUÉZIEC, A., BOSSEN, F., TAUBIN, G., AND SILVA, C. 1999. Efficient compression of non-manifold polygonal meshes. In *Visualization'99 Conf. Proceedings*, 73–80.

GUMHOLD, S., AND STRASSER, W. 1998. Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Conference Proceedings*, 133–140.

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

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

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

ISENBURG, M., AND ALLIEZ, P. 2002. Compressing polygon mesh geometry with parallelogram prediction. In *Visualization'02 Conference Proceedings*, 141–146.

ISENBURG, M., AND SNOEYINK, J. 2000. Face Fixer: Compressing polygon meshes with properties. In *SIGGRAPH'00 Conference Proceedings*, 263–270.

ISENBURG, M. 2002. Compressing polygon mesh connectivity with degree duality prediction. In *Graphics Interface'02 Conference Proceedings*, 161–170.

KARNI, Z., AND GOTSMAN, C. 2000. Spectral compression of mesh geometry. In *SIGGRAPH'00 Conference Proceedings*, 279–286.

KHODAKOVSKY, A., SCHROEDER, P., AND SWELDENS, W. 2000. Progressive geometry compression. In *SIGGRAPH'00 Conference Proceedings*, 271–278.

KHODAKOVSKY, A., ALLIEZ, P., DESBRUN, M., AND SCHROEDER, P. 2002. Near-optimal connectivity encoding of 2-manifold polygon meshes. *Graphical Models 64*, 3-4, 147–168.

KRONROD, B., AND GOTSMAN, C. 2002. Optimized compression of triangle mesh geometry using prediction trees. In *Proceedings of International Symposium on 3D Data Processing Visualization and Transmission*, 602–608.

LEE, E., AND KO, H. 2000. Vertex data compression for triangular meshes. In *Proceedings of Pacific Graphics*, 225–234.

LEE, H., ALLIEZ, P., AND DESBRUN, M. 2002. Angle-analyzer: A triangle-quad mesh codec. In *Eurographics'02 Conference Proceedings*, 198–205.

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'00*, 131–144.

LI, J., AND KUO, C. C. 1998. A dual graph approach to 3D triangular mesh compression. In *Proceedings of Intern. Conf. on Image Processing '98*, 891–894.

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

LINDSTROM, P. 2000. Out-of-core simplification of large polygonal models. In *SIGGRAPH'00 Conference Proceedings*, 259–262.

MANTYLA, M. 1988. *An Introduction to Solid Modeling*. Computer Science Press.

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.

METIS. Version 4. A software package for partitioning unstructured graphs by G. Karypis and V. Kumar. *University of Minnesota*.

PAJAROLA, R., AND ROSSIGNAC, J. 2000. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics 6*, 1, 79–93.

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

ROSSIGNAC, J. 1999. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 5*, 1, 47–61.

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, Tutorial 4.

SZYMCZAK, A., ROSSIGNAC, J., AND KING, D. 2002. Piecewise regular meshes: Construction and compression. *Graphical Models 64*, 3-4, 183–198.

TAUBIN, G., AND ROSSIGNAC, J. 1998. Geometric compression through topological surgery. *ACM Transactions on Graphics 17*, 2, 84–115.

TAUBIN, G., GUÉZIEC, A., HORN, W., AND LAZARUS, F. 1998. Progressive forest split compression. In *SIGGRAPH'98 Conference Proceedings*, 123–132.

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