
Processing Sequences: A new Paradigm for Out-of-Core Processing on Large Meshes

Martin Isenburg¹, Stefan Gumhold², and Jack Snoeyink¹

¹ University of North Carolina at Chapel Hill {isenburg|snoeyink}@cs.unc.edu

² WSI/GRIS University of Tübingen gumhold@gris.uni-tuebingen.de

Abstract

In this paper we introduce a new processing paradigm for meshes that are too large to fit entirely into main memory. We define the concept of a *processing sequence* that is essentially just a ordered sequence of triangles. This triangle ordering gives access to a mesh in a particular useful way that may be exploited to design efficient mesh processing algorithms on top of it.

Rearranging mesh triangles in a particular order is already used for fast rendering on modern graphics cards with vertex cache. The triangles are sent to the card in a *rendering sequence* that tries to minimize cache misses. We exploit a similar strategy for more efficient mesh processing—but at a much larger scale. The main memory as a “cache” is much more flexible so that the data necessary for a complete mesh traversal can always be kept in-core. Therefore the analogue of a “cache miss” does fortunately not exist.

In this working draft we define which triangle orderings constitute a processing sequence, discuss which other properties we want a processing sequence to have, sketch out different approaches for generating processing sequences, describe the two computational abstractions, namely *boundary-based* and *buffer-based* processing, provided by processing sequences, and give some simple examples of how to use our prototype API for out-of-core processing.

1 Introduction

Modern scanning technology enables the creation of 3D digital model that represent real-world objects with incredible precision. The resulting polygonal data sets easily reach file sizes of several gigabytes. Recent examples include dense scans of historical statues and massive isosurfaces generated for scientific simulation. To process such large models efficiently, one needs a representation that works well with the memory hierarchy of the computer.

A scenario: To introduce the state of the art in large mesh processing, we begin by describing our work-flow for obtaining the St. Matthew, a statue scanned by the Stanford Michelangelo project [16].

The St. Matthew statue contains over 186 million vertices and more than 372 million triangles and is stored in an indexed mesh format (e.g. binary PLY). In this format each vertex is represented by three 32-bit floating point coordinates and each triangle has a one byte header and three 32-bit integers that specify its three vertices. This adds up to 6.6 GB of raw mesh data.

Stanford’s Digital Michelangelo Webarchive [15] provides the St. Matthew statue in twelve pieces that are stored as twelve gzipped files of about 330 MB each. In addition, eleven *match-files* are available that describe how to stitch these pieces back together. One reason for splitting the statue into pieces is that some filesystems have a file-size limit of 2 GB. Another reason is that common desktop PCs cannot easily operate on 6.6 GB of indexed mesh data.

We spent more than one week just to acquire and to pre-process the three large statues Lucy, David (1mm), and St. Matthew of the Michelangelo Project [16]. Downloading this data from the Webarchive [15] took us more than a day, copying the mesh pieces between local file systems lasted on the order of hours, and loading them off the hard drive into main memory required tens of minutes.

The software that we obtained to stitch the twelve pieces of the St. Matthew back together was written for a supercomputer. It would load all pieces into memory and then create a second copy to merge the data, which cannot be done on a standard 32-bit PC. Therefore we wrote our own software to do the stitching out-of-core. All this pre-processing required additional gigabytes of disk space for intermediate data files, and simply requesting other lab users to clean out enough scratch disk space delayed our work by a couple of days.

Then we converted these large meshes into a highly compressed mesh format. Its decompression order of triangles and vertices also represents our initial prototype of a *processing sequence*. In this format the entire St. Matthew statue can be stored as a single 456 MB file when using quantization of vertices to 20 bits of precision³. The decompressor can now stream the entire mesh through main memory while providing explicit access to connectivity and mesh geometry along the decompression boundaries (see Figure 1). For the entire St. Matthew statue decompression can be done in only 174 seconds on a 1.8 GHz AMD Athlon processor and faster on higher-clocked processors.

The concept of performing computations in decompression order while the mesh streams through memory is what we call *sequenced processing*. We envision a new breed of highly efficient mesh processing algorithms that adapt their computation to the mesh access provided by processing sequences.

The remainder of this paper is organized as follows: The next section summarizes current approaches to out-of-core mesh processing and how they are put to use in various algorithms. In Section 3 we properly define process-

³ This is sufficient to keep the quantization error well below the error of the scanning system. More detail on uniform quantization of floating point numbers is given in the original reference [13]. If absolutely necessary, we can also preserve the original floating point numbers.

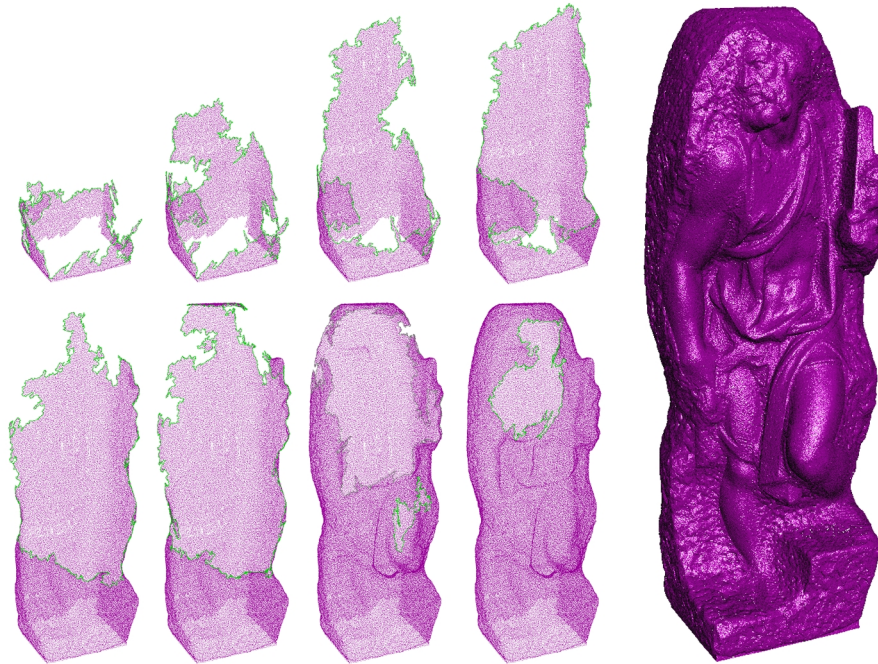


Fig. 1. Visualization of a processing sequence for the St. Matthew statue that was generated with the out-of-core compressor [13]. At any time only the green boundaries need to be kept in memory. On the right an out-of-core rendering that can be thought of as one of the simplest types of out-of-core processing supported by processing sequences.

ing sequences and describe how our API prototype provides access to large meshes. In Section 4 we illustrate two possible ways of generating processing sequences from a standard indexed mesh representation. In Section 5 we show how the processing sequence API can be used in practice for simple tasks such as loading, rendering, and connectivity reconstruction by giving pseudo code. We also investigate how the two computational abstractions may be used for various mesh processing tasks, such as simplification, smoothing, or parametrization. The last section concludes with a summary and an outlook on other types of mesh processing.

2 Out-of-Core Mesh Processing

There are currently three main approaches for processing meshes that are too large to fit into main memory [22]: cutting the mesh into pieces, out-of-core batch-processing of polygon soup, and out-of-core online-processing using external memory data structures.

Mesh cutting is the straight-forward approach for processing large meshes: split them into pieces small enough to fit into main memory and then processes each piece separately while giving special treatment to the split boundaries. This strategy has successfully been used to, for example, simplify [11, 20, 2] and compress [10] very large polygon models. Despite the apparent simplicity of this approach, the initial splitting step can be fairly 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, out-of-core algorithm try instead to process the data as a whole.

Batch-processing aims to keep the memory foot-print low and the processor busy by streaming the data through the main memory in one or more passes, and restricting computation 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 out-of-core mesh simplification methods [17, 18, 21, 9], which batch-process the input mesh in increments of single 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 [18]. This can not only be computationally expensive but the resulting immediate mesh (also known as *polygon soup*) requires double the storage of the already large indexed mesh, and more if there are additional per-vertex properties such as pressure/confidence values 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 [17, 21, 9] or is directly written to a file, which is then used as input for subsequent passes [18].

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

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 [19, 4]. The performance of these schemes is operation 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. [5], for example, propose an octree-based external memory data structure that makes it possible to simplify the St. Matthew statue from 386 to 94 million triangles using iterative edge contraction [8]. Similarly, our out-of-core mesh data structure [13] makes it possible to compress the St. Matthew statue from over 6 GB to 456 MB of data using a compressor based on region-growing [23].

In comparison, out-of-core algorithms based on batch-processing work without explicit connectivity information. This enables them to efficiently do their computations, but their output tends to be of lower quality than that of algorithms that have access to explicit connectivity. Out-of-core algorithms based on online-processing, on the other hand, have explicit connectivity available. However, building the required external memory data structures is expensive in time and space and using them significantly slows down the computations—even for algorithms that have some locality of reference.

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

Traversing mesh triangles in a particular order is already used for fast rendering on modern graphics cards with vertex cache. 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 [3, 6, 7, 12]. Misses cannot be avoided altogether due to the fixed-size caches that are used by typical graphics hardware [1].

Our *processing sequences* exploit a similar strategy for more efficient mesh processing—but at a much larger scale. While the main memory as a cache is much more flexible, a cache miss would be much more expensive as the data would have to be fetched from disk. Fortunately, the amount of memory necessary for a complete mesh traversal is typically so small that we can always keep all the data in main memory and completely avoid cache misses.

While this work was in progress, we learned about the stream-based simplification method of Wu and Kobbelt [24]. They stream polygon soup through main memory and reconstruct connectivity among all triangles they have in memory. They do not explicitly define an ordering on mesh triangles that is as strict as ours, but require only a loosely defined geometric ordering. Therefore they allow triangles that would be considered out-of-order in a processing sequence. However, these triangles do not contribute to any computation, but merely end up waiting in the buffer until all neighboring triangles have arrived. These out-of-order triangles can potentially make things worse, as they reduce the available memory and thereby limit the effective size of the triangle buffer. In fact, the method of Wu and Kobbelt works best if the triangles were provided in processing sequence order. Finally, the fact that processing sequences provide *indexed* rather than *immediate* mesh access is also beneficial for various reasons. In [14] we show that Wu and Kobbelt’s algorithm is indeed perfectly suited to work with our processing sequence API.

3 Processing Sequences

A *processing sequence* is a particular interleaved ordering of the indexed triangles and vertices of a mesh that simulates a region growing process. Triangles are either edge-adjacent to the *processing boundary* or start a new region. The possible configurations are shown in Figure 2. Vertices always precede the first triangle that uses them. In addition, a processing sequence provides with each triangle information on border edges, last vertex use, and eventual non-manifoldnesses. We first define processing sequences for manifold meshes and describe the extension for the non-manifold case later.

A processing sequence *generates* triangles that relate in five different ways to the processing boundary (see Figure 2):

- start:** The triangle is neither not adjacent to a vertex nor an edge of the processing boundary. It introduces three *new* vertices and its three edges are either classified as *border* edges or as *entering* edges.
- add:** The triangle is edge-adjacent to the processing boundary with one edge. It introduces one *new* vertex and two *border* or *entering* edges. The edge at which the triangle is adjacent to the processing boundary is called *leaving*.
- fill:** The triangle is edge-adjacent to the processing boundary with two edges. It introduces one edge, which is either classified as *border* or *entering*.
- join:** The triangle is edge-adjacent to the processing boundary with one edge and vertex-adjacent with one vertex. It introduces two *border* or *entering* edges.
- end:** The triangle is edge-adjacent to the processing boundary with all its edges, which are therefore all classified as *leaving* edges.

The edges of a generated triangle have three types: *leaving*, *entering*, or *border*. Leaving edges are those that were previously on the processing boundary. The others are either entering edges or border edges. Entering edges are adjacent to unprocessed triangles. Border edges correspond to topological borders (surface boundaries) in the mesh. The vertices of a generated triangle have two types: *new* and *old*. Vertices are new if the triangle is the one that uses them for the first time, otherwise they are old. The *last* use of a vertex coincides with the moment the number of its incident entering edges drops to zero. The processing sequence API provides this type information for each generated triangle (see Figure 3).

Connectivity reconstruction is supported by allowing the user to store his own data pointers with every entering edge and every new vertex that are then made available later whenever the corresponding element reappears as a leaving edge or an old vertex.

Given a “somewhat” compactly growing processing sequence, this representation allows streaming very large meshes through main memory. At any time only a small portion of the mesh, namely the processing boundary, needs to be kept in-core, while the bulk of the mesh data can reside on disk. Yet, as explicit connectivity information can be maintained along the processing boundary, this provides seamless access to very large meshes.

Processing sequence are both read and written in such an order. In addition, a processing sequence may be read and written at the same time. In

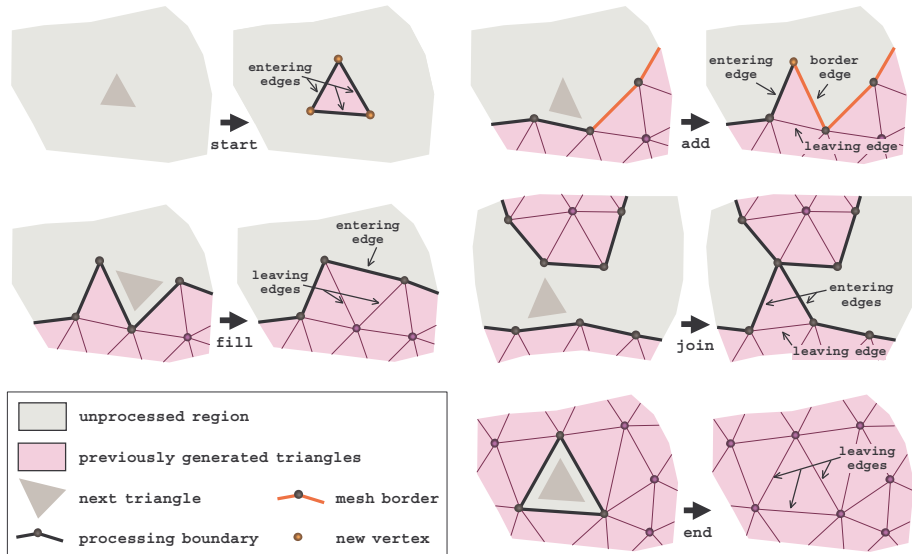


Fig. 2. Generated triangles relate in five different ways to the processing boundary. Their edges are either *leaving*, *entering*, or *border*. Their vertices are either *new* or *old*.

this case there are two processing boundaries: one is the *input* boundary along which triangles are added and one is the *output* boundary along which triangles are removed. The zone between the two boundaries is called the triangle buffer. Those are the triangles in memory. The triangle order of the input and the output sequence can be different. In particular, the output processing sequence can be a completely different set of triangles. After all, remeshing and simplification methods are potential users of processing sequences.

The processing sequence interface for input and output is basically identical. In both cases we can think of triangles getting generated on the processing boundary. Generating a triangle on the input boundary means to read the next triangle from the processing sequence used as input. Generating a triangle on the output boundary means to write the next triangle to the processing sequence used as output. As the output sequence might be used as input sequence of subsequent processing it makes sense to use the term “generate” for both boundaries. Note, however, that what was defined so far neither determines an absolute ordering on the triangles, nor a particular file format in which processing sequence is stored. Only the allowable triangle sequences were defined. Whatever the underlying representation may be, the processing sequence API should roughly look as sketched in Figure 3.

Often, the order in which an application produces triangles and vertices does not immediately correspond to a processing sequence. In this case we use a processing sequence *converter* that temporarily accumulates triangles and

```

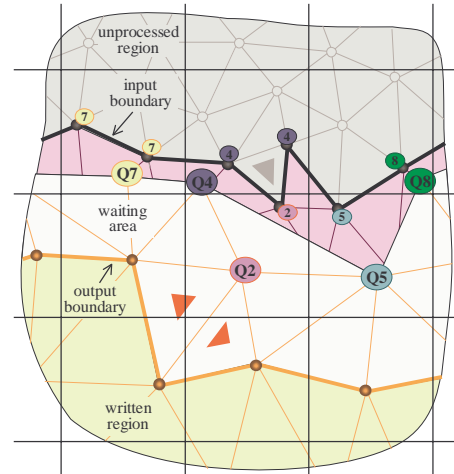
class ProcessingSequence {
    int    open(char* file_name, char* rw);
    int    get_num_vertices();
    int    get_num_triangles();
    bool   has_triangles();
    int    generate_triangle();
    int    close();
    int    get_vertex_index(int i);
    void   set_vertex_index(int i, int index);
    float* get_vertex_position(int i);
    void   set_vertex_position(int i, float* position);
    vtype  get_vertex_type(int i);
    void   set_vertex_type(int i, vtype type);
    etype  get_edge_type(int i);
    void   set_edge_type(int i, etype type);
    void*  get_vertex_data(int i);
    void   set_vertex_data(int i, void* data);
    void*  get_edge_data(int i);
    void   set_edge_data(int i, void* data);
}

```

Fig. 3. A rough sketch of the processing API.

vertices in a *waiting area* and reorders them appropriately, as illustrated in Figure 4.

Fig. 4. An illustration of how a *waiting area* is used to *on-the-fly* convert the triangle and vertex ordering produced by some application into a proper processing sequence. The example application is a vertex-clustering based simplification method [14]. 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.



Processing sequences naturally support 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. 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, parametrization 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 those extremes provides a compromise that so to speak adapts to the available resources.

Implementations of either abstraction can perform their computation in a *single-pass* or in multiple passes over the data. The latter can be organized either as a sequential *multi-pass* approach or as a streaming *multi-stage* approach, which hands over the results of one pass directly to the next by making the output boundary of one the input boundary of the other. Finally, parallel processing can be supported by splitting and merging of processing sequences. Let's discuss these possibilities in more detail:

Single-pass processing performs its computations in a single pass over the processing sequence while generating all desired output. Examples for boundary-based single-pass processing are vertex-clustering based simplification, area or curvature computations, or rendering in immediate mode. Examples for buffer-based single-pass processing are edge contraction based simplification, iterative smoothing computations, or rendering with vertex buffers.

Multi-pass processing performs its computations in multiple passes over the processing sequence. As it may require to store intermediate results other than the output sequence, the next generation of our API should also allow storage of such data. In order to make such data available in the next pass it would be stored in an order determined by the output sequence (e.g. per output vertex, per output edge, per every tenth output triangle, ...). Both, boundary-based and buffer-based processing can be multi-pass. An example for boundary-based multi-pass computation are reaction-diffusion simulations. An example for buffer-based multi-pass computations are simplification followed by a pass that compresses the generated output sequence.

Multi-stage processing performs its computations in multiple stages, basically by making the output boundary of the first processing the input boundary of the next. Immediate compression of the output of a simplification algorithm, for example, can be implemented as a multi-stage algorithm.

Sequence splitting splits an input sequences into several output sequences. This leads to replication of all those vertices that appear in more than one output sequence and creates artificial border edges. A mechanism should be in place for marking these vertices and edges such that they (a) can potentially receive special treatment and (b) can be identified at merging time. Mesh partitioning is an example where the output might already be the

final result. Parallel processing is an example where sequence partitioning is used for speeding up computations on a multi-processor machine.

Sequence merging merges several input sequences into one output sequences. The input sequences should either have been generated by sequence partitioning or should be conforming output on processing of such. Usually this will merge a previously split surface back into one piece. However, it may be also use to, for example, merge unconnected mesh components into one geometrically ordered processing sequence.

For a high-performance parallel computing application envision a multi-stage scenario in which one process splits the incoming processing sequence into several output sequences that are then processed independently while feeding their output boundaries to a process that merges them back into one output sequence. While the idea of parallel computing sounds cool and would surely be useful for CPU-intensive computations on large surfaces, for now this remains envisioned future.

Non-manifold meshes are turned into manifold meshes simply by cutting along non-manifold vertices and edges. However, the vertices are not replicated, but appear multiple times as a *new* vertex with the same index. This allows representing non-manifold meshes while using only the five configurations 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 usages remaining.

4 Generating Processing Sequences

Generating processing sequences can be done in a number of different ways, as the definition neither imposes a specific traversal order, nor an internal file format. The sequences that we currently use were generated in a pre-processing step using an out-of-core compression method [13]. Most *one-pass* compression schemes naturally generate triangle and vertex orderings that conform with the definition of a mesh processing sequence. In fact, it was the memory-efficient decompression order of our decoder that originally inspired processing sequences.

The processing sequences that are result of our out-of-core mesh compression scheme [13] can be efficiently read from a highly compressed file. However, generating them is very expensive as we performs in-core processing using an external memory structure to accomodate the region growing traversal of the compressor. Furthermore the compressor (as presented) has no mechanism to explicitly direct the traversal. While this allows very high compression, it is not compatible with out ultimate goal of immediately re-compressing a processing sequence along the output boundary.

We are currently developing a compressed format for processing sequences that allows arbitratry triangle orderings (as long as they constitute proper processing seqences). While these will be less compressed than the processing

sequences generated by [13] it will allow to attach a re-compressor to the output boundary. This will make it possible to both read and write compressed processing sequences.

The processing sequence converter, mentioned earlier, is one possible way for *on-the-fly* creation of processing sequences. It accepts indexed triangles and corresponding vertices ordered in some loosely localized form, temporarily accumulates them in a *waiting area*, where they are re-ordered into a proper processing sequence (see Figure 4). 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 needs to be told when a vertex is *finalized*, i.e., used for the last time. It needs this information to be able to detect border edges and non-manifold vertices. It also needs to know this for being able to safely deallocate the memory of mesh elements that are no longer used. When converting, for example, the triangles and vertices produced by the simplification algorithm from Figure 4 into a proper output sequence, it is the simplifier that 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 immediate compression of arbitrary output sequences is in the works.

This converter also provides an alternative to the out-of-core compressor [13] 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 number of external sorts [18].

In a final pass over the two sorted sequences we load vertices and triangles into the waiting area. We read from the triangle sequence whenever the next triangle key is less than or equal to the next vertex key. After reading a triangle, its vertices are looked up by index in a hash table. In case they are not present a new entry is created. The actual coordinates of the vertex are not known at this point. Eventually the key of the next triangle is larger than that of the next vertex and we read from the vertex sequence. After reading a vertex, we use its index i to look up its entry in the hash and store its coordinates. 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.

5 Using Processing Sequences

Here we want to give some intuition how the processing sequence paradigm can be used in practice. The interface to all processing sequence APIs should support these operations, whereas the underlying on-disk representations of the processing sequence could be anything, ranging from a verbose ASCII format, over strictly ordered triangle soup with on-the-fly reconstruction of connectivity, to highly compressed formats providing explicit connectivity. Let us first look at the input interface, as we already have a fully working prototype implementation of this.

```

ProcessingSequence ps = new ProcessingSequence();
ps.open("stmatthew.compressed.20bit.ply", 'r');
int* indices = new int[ps.get_num_triangles()*3];
float* vertices = new float[ps.get_num_vertices()*3];
int i_count = 0;
int v_count = 0;
while (ps.has_triangles()) {
    ps.generate_triangle();
    for (int i = 0; i < 3; i++) {
        indices[i_count++] = ps.get_vertex_index(i);
        if (ps.get_vertex_type(i) == NEW) {
            float* v = ps.get_vertex_pos(i);
            vertices[v_count++] = v[0];
            vertices[v_count++] = v[1];
            vertices[v_count++] = v[2];
        }
    }
}

```

Fig. 5. Pseudo code that illustrates how simple it is to load a mesh into memory. This example is just meant to give a “feel” for the API. In practice we should not be able to actually load the entire mesh into memory, which is the main reason to use processing sequences in the first place.

6 Summary

We have defined a new paradigm for out-of-core processing that is based on reordering the vertices and the triangles of a mesh into a dedicated interleaved processing order. These processing sequences grow one (or multiple) edge-connected regions until a mesh component is completely processed.

A remaining degree of freedom lies in the traversal order with which the processing boundary is grown. Depending on the objective it may be grown for maximum compression as done in [13] or quickly generated *on-the-fly* using the converter. A third (so far uninvestigated) strategy would try to minimize the maximal length of the processing boundary as this is beneficial to keep the memory footprint as small as possible.

The compressed format in [13] had originally been optimized for maximal compression without considering the potential needs of sequenced processing.

```

ProcessingSequence ps = new ProcessingSequence();
ps.open('stmatthew_compressed_20bit.ply','r');
while (ps.has_triangles()) {
  HalfEdge he[0] = new HalfEdge();
  HalfEdge he[1] = new HalfEdge();
  HalfEdge he[2] = new HalfEdge();
  he[0].next = he[1]; he[0].prev = he[2];
  he[1].next = he[2]; he[1].prev = he[0];
  he[2].next = he[0]; he[2].prev = he[1];
  ps.generate_triangle();
  for (int i = 0; i < 3; i++) {
    if (ps.get_edge_type(i) == ENTER) {
      ps.set_edge_data(i, (void*)he[i]);
    } else if (ps.get_edge_type(i) == LEAVE) {
      HalfEdge tmp = (HalfEdge*)ps.get_edge_data(i);
      tmp.inv = he[i];
      he[i].inv = tmp;
    } else {
      he[i].inv = 0;
    }
    if (ps.get_vertex_type(i) == NEW) {
      Vertex v = new Vertex();
      v.index = ps.get_vertex_index(i);
      v.pos = ps.get_vertex_pos(i);
      ps.set_vertex_data(i, (void*)v);
      he[i].origin = v;
    } else {
      he[i].origin = (Vertex*)ps.get_vertex_data(i);
    }
  }
}
}

```

Fig. 6. Pseudo code that illustrates how simple it is to reconstruct explicit mesh connectivity when loading a processing sequence, here at the example of a simple half-edge-based structure. This is achieved by using the API's ability to maintain user specific data per-vertex and/or per-edge along the decompression boundaries.

The triangle orderings of the processing sequences it generates are strictly determined by the compression scheme and cannot be changed according to the needs of an application. Furthermore its handling of processing non-manifold vertices might be unsuitable for some processing needs. In which way a processing sequence should support this (e.g. handling of non-manifold vertices and edges) needs some further investigation.

Future work will address compressed storage of *on-the-fly* generated processing sequences that are either the output of an algorithm or created from scratch with the converter and external sorting. While compressing processing sequences in a traversal order that is dictated by an application instead of by the compressor can be expected to deliver lower compression rates, it will allow both inputting and outputting processing sequences in compressed form.

References

1. R. Bar-Yehuda and C. Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15(2):141–152, 1996.
2. F. Bernardini, I. Martin, J. Mittleman, H. Rushmeier, and G. Taubin. Building a digital model of michelangelo’s florentine pieta. *IEEE Computer Graphics and Applications*, 22(1):59–67, 2002.
3. A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Graphics Interface’01 Conference Proceedings*, pages 81–90, 2001.
4. P. Choudhury and B. Watson. Completely adaptive simplification of massive meshes. Technical Report CS-02-09, Northwestern University, 2002.
5. P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *to appear in IEEE Transactions on Visualization and Computer Graphics*, 2003.
6. M. Deering. Geometry compression. In *SIGGRAPH’95 Conference Proceedings*, pages 13–20, 1995.
7. F. Evans, S. S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *Visualization’96 Conference Proceedings*, pages 319–326, 1996.
8. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH’97 Conference Proceedings*, pages 209–216, 1997.
9. M. Garland and E. Shaffer. A multiphase approach to efficient surface simplification. In *Visualization’02 Conference Proceedings*, pages 117–124, 2002.
10. J. Ho, K. Lee, and D. Kriegman. Compressing large polygonal models. In *Visualization’01 Conference Proceedings*, pages 357–362, 2001.
11. H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Visualization’98 Conference Proceedings*, pages 35–42, 1998.
12. H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH’99 Conference Proceedings*, pages 269–276, 1999.
13. M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *SIGGRAPH’03 Conference Proceedings*, 2003. to appear.
14. M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. submitted for publication, 2003.
15. M. Levoy. The Digital Michelangelo Project archive of 3D models. <http://graphics.stanford.edu/data/mich/>.
16. M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Gintzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project. In *SIGGRAPH’00 Conference Proceedings*, pages 131–144, 2000.
17. P. Lindstrom. Out-of-core simplification of large polygonal models. In *SIGGRAPH’00 Conference Proceedings*, pages 259–262, 2000.
18. P. Lindstrom and C. Silva. A memory insensitive technique for large model simplification. In *Visualization’01 Conference Proceedings*, pages 121–126, 2001.
19. S. McMains, J. Hellerstein, and C. Sequin. Out-of-core build of a topological data structure from polygon soup. In *Proceedings of the 6th ACM Symposium on Solid Modeling and Applications*, pages 171–182, 2001.
20. C. Prince. Progressive meshes for large models of arbitrary topology. Technical Report Master Thesis, University of Washington, 2000.

21. E. Shaffer and M. Garland. Efficient adaptive simplification of massive meshes. In *Visualization'01 Conference Proceedings*, pages 127–134, 2001.
22. C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization'02 Course Notes*, 2002.
23. C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface'98 Conference Proceedings*, pages 26–34, 1998.
24. J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes. In *Graphics Interface'03 Conference Proceedings*, pages XX–XX, 2003.