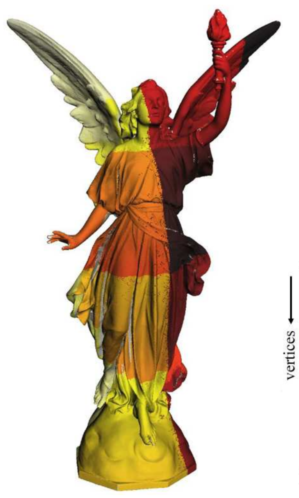
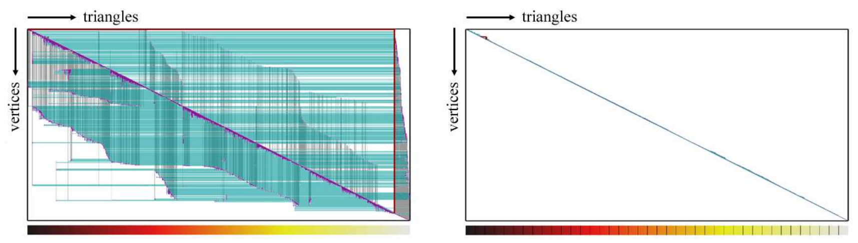
Streaming Meshes

Category: Research





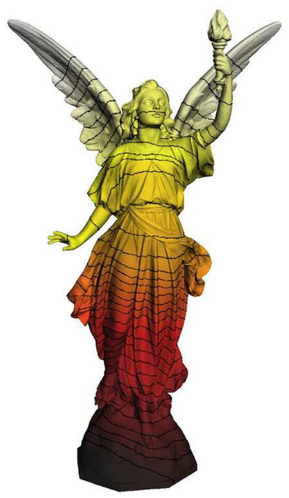


Figure 1: Illustrations of the coherence in the layout of a mesh: On the left the original layout; on the right the layout after reordering the vertex and triangle arrays using spectral sequencing. The large rendering color-codes triangles based on their position in the array. The layout diagram connects triangles that share the same vertex with horizontal line segments (green) and vertices referenced by the same triangle with vertical line segments (gray).

Abstract

Recent years have seen an immense increase in the complexity of geometric data sets. Today's gigabyte-sized polygon models can no longer be completely loaded into the main memory of common desktop PCs. Unfortunately, current mesh formats, which were designed years ago when meshes were orders of magnitudes smaller, do not account for this. Using such formats to store large meshes is inefficient and complicates all subsequent processing.

We describe a *streaming* format for polygon meshes, which is simple enough to replace current offline mesh formats and more suitable for working with large data sets. Furthermore, it is an ideal input and output format for I/O-efficient out-of-core algorithms that process meshes in a streaming, possibly pipelined, fashion. This paper chiefly concerns the underlying theory and the practical aspects of creating and working with this new representation. In particular, we describe desirable qualities for streaming meshes, methods for converting meshes from a traditional to a streaming format, and a novel technique for streaming on-the-fly compression.

The central theme of this paper is the issue of coherent and compatible layouts of the mesh vertices and polygons. We present metrics and diagrams that characterize the coherence of a mesh layout and suggest appropriate strategies for improving its "streamability." To this end, we outline several out-of-core algorithms for reordering meshes with poor coherence, and present results for a menagerie of well known and generally incoherent surface meshes.

1 Introduction

The advances in computer speed and memory size are matched by the growth of data and model sizes. Modern scientific technologies enable the creation of digital 3D models of incredible detail and precision. Recent examples include statues scanned for historical reconstruction and isosurfaces visualized to understand the results of scientific simulation. These polygonal data sets easily reach sizes of several gigabytes, making all sorts of subsequent processing a difficult task. The sheer amount of data may not only exhaust the main memory resources of common desktop PCs, but also exceed the 4 gigabyte address space limit of these 32-bit machines.

In order to process geometric data sets that do not fit in main memory, one resorts to *out-of-core* algorithms. These arrange the mesh so that it does not need to be kept in memory in its entirety, and adapt their computations to operate mainly on the loaded parts. Such algorithms have been studied in several contexts, including visualization, simplification, and compression. A major problem for all these algorithms is dealing with the initial format of the input.

Current mesh formats were designed in the early days of mesh processing when models like the Stanford bunny, with less than 100,000 faces, were considered complex. They use an array of floats to specify the vertex positions followed by an array of indices into the vertex array to specify the polygons. The order in which vertices and polygons are arranged in these arrays is left to the discretion of the person creating the mesh. This was convenient when meshes were relatively small. In the meantime, however, our data sets have grown in size by four orders of magnitude. Storing such large meshes in the same format means that a gigabyte-sized array of vertex data is indexed by a gigabyte-sized block of triangle data. This unduly complicates all subsequent processing.

Most processing tasks need to *dereference* the input mesh (i.e. resolve all triangle-to-vertex references). Memory mapping the vertex array and having the operating system swap in the relevant sections is only practical given a coherent *mesh layout*. The lack of coherence in the layout of the Lucy model is illustrated on the left in Figure 1. Loosely speaking, the farther the green and gray line segments are from the diagonal, the less coherent the layout is. In order to operate robustly on large indexed meshes an algorithm either needs to be prepared to handle the worst possible inputs or make assumptions that are bound to fail on some models.

In this paper we propose a *streaming* format for large polygon meshes that solves the problem of dereferencing. In addition, it enables the design of new I/O-efficient algorithms for out-of-core stream-processing. The basic idea is to interleave indexed vertices and triangles and to provide information when vertices are last referenced. We call such a mesh representation a *streaming mesh*.

The terms "progressive" and "streaming" are often used synonymously in computer graphics. Our streaming meshes are fundamentally different from the multi-resolution representations used for progressive geometry transmission, in which detail is added to a coarse base mesh stored in-core, possibly until exhausting available memory [Hoppe 1996]. In our windowed streaming model triangles and vertices are added to, or removed from, a partial but seamless reconstruction of the mesh that is kept in a finite, fixed-size memory buffer—a "sliding window" over the full resolution mesh.

The advantage of a streaming representation for meshes was first identified by Isenburg and Gumhold [2003], who propose a compressed mesh format that allows streaming decompression. During compression a set of boundaries sweep once over the entire mesh, which is accessed through a complex external memory data structure. During decompression, however, only those boundaries need to be maintained in memory. Later, Isenburg et al. [2003] showed that the streaming access provided by the decompressor can be used for I/O-efficient out-of-core simplification. But these works pay little attention to what makes good stream orders. In fact, the streaming meshes produced by their out-of-core compressor are not particularly well-suited for stream-processing.

In this paper we characterize good stream orders and describe techniques for creating them. Here we provide the theory of streaming meshes and a set of tools that operate on them. Furthermore, we describe a scheme for streaming compression. In contrast to previous schemes that dictate the order in which a mesh is compressed, we encode meshes in their stream order. This allows compression *on-the-fly* without cutting the mesh in pieces, as suggested by Ho et al. [2001], and without resorting to complex external memory data structures, as proposed by Isenburg and Gumhold [2003].

2 Previous Work

While models from 3D scanning or iso-surface extraction have become too large to fit in the main memory of commodity PCs, storing the models on hard disk is always possible. Out-of-core algorithms are designed to efficiently operate on large data sets that mostly reside on disk. To avoid constant reloading of data from slow external memory, the order in which they access the mesh must be consistent with the arrangement of the mesh on disk. Currently the main approaches are: cutting the mesh into pieces, using external memory data structures, working on dereferenced triangle soup, and operating on a streaming representation. All these approaches have to go through great efforts to create their initial on-disk arrangement when the input mesh comes in a standard indexed format.

Mesh cutting methods partition large meshes into pieces that are small enough to fit into main memory and then process each piece separately. This strategy has been successful for distribution [Levoy et al. 2000], simplification [Hoppe 1998; Bernardini et al. 2002], and compression [Ho et al. 2001]. The initial cutting step requires dereferencing, which is expensive for standard indexed input.

Approaches that use **external memory data structures** also partition the mesh, but into a much larger number of smaller pieces often called *clusters*. At run-time only a small number of clusters are kept in memory with the majority residing on disk from where they are paged in as needed. Cignoni et al. [2003], for example, use such an external memory mesh to simplify large models using iterative edge contraction. Similarly, Isenburg and Gumhold [2003] use an out-of-core mesh to compress large models via region growing. Building these data structures from a standard indexed mesh involves additional dereferencing passes over the data.

One approach to overcome the problems associated with indexed data is to not use indices. Abandoning indexed meshes as input, such techniques work on **dereferenced triangle soup**, which streams from disk to memory in increments of single triangles. Lindstrom [2000] showed how to implement clustering based simplification this way. Although his algorithm does not use indices, his input meshes usually come in an indexed format. Ironically, in this case an initial dereferencing step [Chiang and Silva 1997] becomes necessary for doing exactly what the algorithm later avoids: resolving all triangle-to-vertex references. To take full advantage of this type of processing, the input must already be streamable.

While the entire mesh may not fit in main memory, one can easily store a working set of several million triangles. Wu and Kobbelt [2003] simplify large models by streaming coherent triangle soup into a fixed-sized memory buffer, on which they perform randomized edge collapses. Connectivity between triangles is reconstructed through geometric hashing on vertex positions. Only vertices surrounded by a closed ring of triangles are deemed eligible for simplification. Thus mesh borders cannot be simplified until the entire mesh has been read, and adjacent vertices and triangles must remain in the buffer until the end. Their output is therefore guaranteed to be incoherent. Isenburg et al. [2003] show that their compressed **streaming representation** provides exactly the information that Wu and Kobbelt's algorithm needs: "finalization" of vertices. Instead of the algorithm having to guess when a vertex is final, their compressed format informs when this is indeed the case.

Coherence in reference has also been investigated in the context of efficient rendering. Modern graphics cards use a vertex cache to buffer a small number of vertices. In order to make good use of the cache it is imperative for subsequent triangles to re-reference the same vertices. Deering [1995] stores triangles together with explicit instructions that tell the cache which vertices to replace. Hoppe [1999] produces coherent triangle orderings optimized for a particular cache size, while Bogomjakov and Gotsman [2001] create orderings that work well for all cache sizes.

An on-disk layout that is good for streaming is similar to an in-memory layout that is good for rendering—at a much larger scale. But there are differences: For the graphics card cache it is expected that at least some vertices are loaded multiple times. In our case, each vertex is loaded only once as main memory can hold all required vertices for any reasonable traversal. Once a vertex is expelled from the cache of a graphics card, it makes no difference how long it takes until it is loaded again. In our case, the duration between first and last use of a vertex does matter. We seek not just local but global coherence in the triangle layout. While, as we will see, a breadth-first traversal produces good stream layouts, it would constitute a poor rendering sequence.

3 Mesh Layouts

Indexed mesh formats impose no constraints on the order of either vertices or triangles. The three vertices of a triangle can be located *anywhere* in the vertex array and need not be close to each other. And while subsequent triangles may reference vertices at opposite ends of the array, the first and the last triangle can use the same vertex. This flexibility was convenient for small meshes, but has become a major headache with the arrival of gigabyte-sized data sets. Today's mesh formats have originated from a smorgasboard of legacy formats (e.g. PLY, OBJ, IV) that were designed when polygon models were of the size of the Stanford bunny. This model, which has helped popularize the PLY format, abuses this flexibility like no other. Its layout is incoherent in every respect, as illustrated in the form of a *layout diagram* in Figure 2a.

A layout diagram intuitively visualizes the coherency in reference between vertices, which are indexed along the vertical axis, and triangles, which are indexed along the horizontal axis. Both are numbered in the order they appear in the file. We draw for each triangle a point (violet) for all its vertices and a vertical line segment (gray) connecting them. Similarly, we draw for each vertex a horizontal line segment (green) connecting the first and last triangle that reference it. Intuitively speaking, the closer points and lines group around the diagonal the more coherent the layout is.

Nowadays the PLY format is used to archive the scanned statues created by Stanford's Digital Michelangelo Project [2000]. For the Atlas statue of 507 million triangles, a six gigabyte array of triangles would reference into a three gigabyte array of vertex positions. Its layout diagram (see Figure 2b) reveals that vertices are used over spans of up to 61 million triangles, equaling 700 MB of the triangle array. Since such an indexed mesh can not be dealt with on commodity PCs, the statue is provided in twelve pieces.

Definitions

The layout of a mesh is the ordering of its vertices *and* the ordering of its triangles. We characterize it with several measures: The *triangle span* of a vertex is the number of triangles between and including its first and last use. It corresponds to the green horizontal segments in a layout diagram. The triangle span of a layout is the longest triangle span of any vertex. The *vertex span* of a triangle is the maximal index difference (plus one) of its vertices. It corresponds to the gray vertical segments in a layout diagram. The vertex

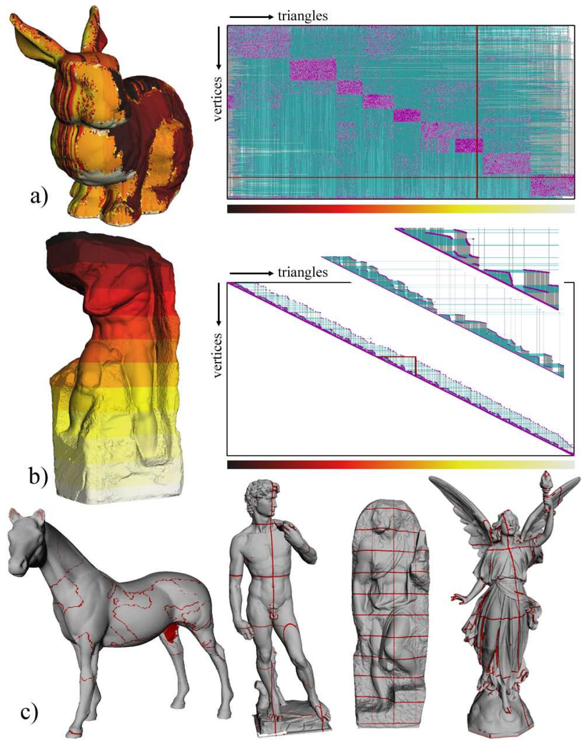


Figure 2: Visual illustrations of poor mesh layouts: (a) The bunny and (b) the 10,000 times more complex Atlas model. Successive triangles are rendered with smoothly changing colors. Layout diagrams intuitively illustrate incoherence in the meshes. (c) Highlighting triangles with high vertex span often reveals something about how the mesh was created or modified.

span of a layout is the longest vertex span of any triangle. Anticipating sequential access to the triangle array, we define the *width* of a layout as the maximal number of vertices used at the same time. It corresponds to the maximal number of horizontal segments cut by a vertical line. The *skip* of a layout is the maximal number of concurrently skipped vertices. A vertex is skipped when another vertex with higher index is first referenced before it.

Three layouts for a small mesh are shown in Figure 3. The first layout has a skip of 2 because vertex #8 is already used by triangle #3, while vertices #3 and #7 are still unused. Reordering the vertices corresponds to vertically rearranging the green line segments, as shown in the second layout. This affects skip and vertex span but not width or triangle span. Reducing those requires reordering the triangles, as illustrated by the third layout.

Incoherent Layouts

The incoherency in a mesh layout can often be explained by how the mesh was produced. The horse, for example, is zipped together from multiple pieces that are the result of scanning from different viewpoints. While the zipping algorithm sorted the triangles spatially along one axis, it simply concatenated the vertex arrays—thereby creating triangles with high vertex spans along the zips. The dinosaur has its triangles ordered along one axis and its vertices along another axis. This projects the model along the third axis into vertex and triangle indices such that they capture a distorted 2D view of the shape. This layout is low in width and span, but has a high skip. For the most part, the dragon has vertices and triangles loosely ordered along the *z*-axis. But there are a few vertices at the very end of the vertex array that are used all across the triangle array, leading to high vertex span. This is due to a post-processing operation for topological cleanup of holes in the mesh.

The large Stanford statues were extracted block by block from a large volumetric representation. The resulting surfaces were then stitched together on a supercomputer by concatenating triangle and

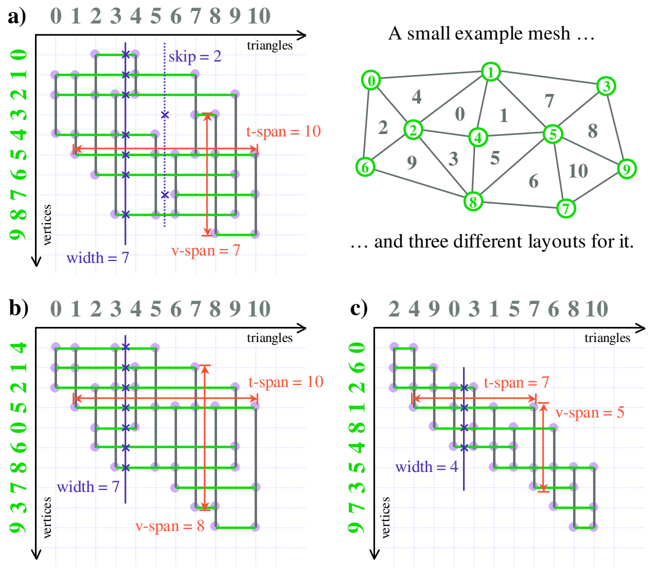


Figure 3: Three layouts for a mesh: (a) An incompatible vertex order results in a skip. (b) Reordering the vertices eliminates the skip but does not affect width or triangle span. (c) Reordering the triangles also can reduce them.

vertex arrays and identifying vertices between neighboring blocks, which is evidenced by high vertex spans in Figure 2. For the two largest statues this incoherence was somewhat reduced when their "blocky" layouts were multiplexed into several files by spatially cutting the statues into twelve horizontal slices.

4 Streaming Meshes

A streaming mesh format interleaves the vertices and the triangles that reference them and provides explicit information about the last time a vertex is referenced. Such a format can be as simple as the examples in Figure 4. Despite its simplicity, a streaming mesh format has tremendous advantages over standard formats. Because vertices and triangles are read and written together, the problem of dereferencing does not exist. Furthermore, the ability to encode "finalization" of vertices enables memory-efficient stream processing.

Envision a scenario where one algorithm extracts an isosurface and pipes it as a streaming mesh to a simplification process, which in turn streams the simplified mesh to a compression engine that encodes it and immediately transmits the resulting bit-stream to a remote location where triangles are rendered as they decompress. In fact, we now have all components of this pipeline—and it is the streaming format that makes it possible to pipe them all together.

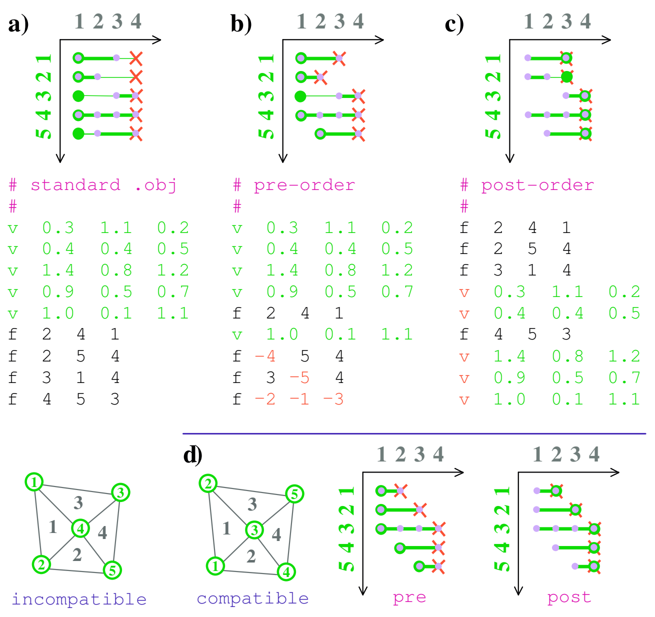
Simply put, a streaming mesh format makes operating on the largest of data sets a feasible task. For example, the images in this paper are rendered *out-of-core* from full resolution input on a laptop with 512 MB of memory. Read vertices are stored in a hash where they are looked up by incoming triangles, which are immediately rendered. The fact that a hash entry can be removed as soon as the vertex is finalized keeps the memory requirements low.

This refers to the "sm_viewer" program that is included on the CD-ROM. Use hot-key 'r' for full resolution out-of-core rendering.

Finally, a streaming format will make creators of large data sets aware of the mesh layouts they produce and will encourage them to take coherency in the output into consideration when they design large mesh algorithms. Anyone who went through the pain of stitching together the Atlas statue from the twelve pieces that it is provided in will appreciate this as an important contribution.

Definitions

A streaming mesh is a logically interleaved sequence of indexed vertices and triangles *plus* information about when vertices are *introduced* and when they are *finalized*. Vertices become *active* when they are introduced and cease to be active when they are finalized. We call the evolving set of active vertices the *front* F_i, which at time



```
# standard .obj
#
v  0.3  1.1  0.2
v  0.4  0.4  0.5
v  1.4  0.8  1.2
v  0.9  0.5  0.7
v  1.0  0.1  1.1
f  2  4  1
f  2  5  4
f  3  1  4
f  4  5  3
```

```
# pre-order
#
v  0.3  1.1  0.2
v  0.4  0.4  0.5
v  1.4  0.8  1.2
v  0.9  0.5  0.7
f  2  4  1
v  1.0  0.1  1.1
f  -4  5  4
f  3  -5  4
f  -2  -1  -3
```

```
# post-order
#
f  2  4  1
f  2  5  4
f  3  1  4
v  0.3  1.1  0.2
v  0.4  0.4  0.5
f  4  5  3
v  1.4  0.8  1.2
v  0.9  0.5  0.7
v  1.0  0.1  1.1
```

Figure 4: Examples of a streaming ASCII format. (a) Standard OBJ format. (b) Streaming pre-order format: finalization is coded through negative relative indices, introduction coincides with appearance of vertex in stream. (c) Streaming post-order format: finalization coincides with appearance of vertex in stream, introduction occurs at first vertex reference. (d) If the vertex and triangle layouts are compatible, the meshes can be compact.

i partitions the mesh into finalized (i.e. processed) vertices and vertices not yet encountered in the stream. The *front width* (or simply the *width*) is the maximal size $\max_i |F_i|$ of the front, i.e. the maximal number of concurrently active vertices. The width gives a lower bound on the memory footprint as any stream process must maintain the front, e.g. as a hash. The *front span* (or simply the *span*) is the maximal index difference $\max_i \{\max_{v \in F_i} - \min_{v \in F_i} + 1\}$ of vertices on the front, and intuitively measures the longest duration a vertex remains active. Clearly $width \leq span$. Note that at most $\log_2(span)$ bits are needed for relative indexing of the vertices.

Because the triangles and their vertices generally appear close together in the stream, we place no restriction on whether vertices precede triangles (as would normally be the case in a standard indexed mesh) or follow them. Streaming meshes are *pre-order* if each vertex precedes all triangles that reference it, and are *post-order* if each vertex succeeds all triangles that reference it; otherwise they are *in-order*. The introduction of a vertex does not necessarily coincide with its appearance in the stream as triangles can reference and thus introduce vertices before they appear. In this paper we only consider pre- and post-order meshes.

The latest that a vertex can be introduced is just before the first triangle that references it. Similarly, the earliest that a vertex can be finalized is just after the last triangle that references it. We can keep the front small in a pre-order mesh by delaying the appearance (introduction) of a vertex as much as possible, i.e. such that each vertex when introduced is referenced by the next triangle in the stream. Conversely, in a post-order mesh each finalized vertex should be referenced by the previous triangle. More generally, we say that a stream is *vertex-compact* if each vertex is referenced by the previous or the next triangle in the stream (see Figure 4). Note that the vertices can always be made compact with respect to the triangle layout by rearranging them. We should likewise not unnecessarily delay a triangle in a pre-order mesh if all its vertices have appeared, nor should we prematurely specify a triangle in a post-order mesh before any of its vertices can be finalized. We say that a stream is *triangle-compact* if each triangle references the previous or the next vertex in the stream. (Note that vertex-compactness does not imply triangle-compactness, and vice versa.) It is always possible to rearrange the triangles to make them compact with respect to a given vertex layout. Finally, a streaming mesh is *compact* if it is both vertex- and triangle-compact.

Working with Streaming Meshes

Streaming meshes are ideally suited for stream-based processing. In this model, the mesh streams through an in-core *stream buffer*, which is large enough to hold all active mesh elements. For straightforward tasks that simply need to dereference the vertices, such as rendering a flat shaded mesh, a minimal stream buffer is needed. For more elaborate processing tasks, a larger stream buffer may hold as many additional mesh elements as there are memory resources. We call the loops of edges that separate already read triangles from those not yet read an input *stream boundary*. For applications that write meshes, there is an equivalent output boundary.

Streaming meshes allow pipelined processing, where multiple tasks run concurrently on separate pieces of the mesh. One module's output boundary then serves as the down-stream input boundary for another module. Because the mesh is operated on in a single pass and because all data access is sequential, we can create very fast out-of-core stream modules with processing speeds around 0.1–1 million triangles per second. For example, our streaming edge-collapse implementation simplifies the St. Matthew mesh to less than one million triangles in one hour using 72 MB of RAM on a 3.2 GHz PC, compared to 14 hours (plus 8 hours of preprocessing time) on a 0.8 GHz PC for Cignoni et al.'s external memory method [2003]. And contrary to [Isenburg et al. 2003], who rely on a particular compressed streaming input constructed in an elaborate preprocess, we can directly simplify input from any streaming source as it arrives (e.g. a mesh generator, a network). Also, streaming mesh formats enable us to design compressors (Section 6) that are up to 50 times faster than [Isenburg and Gumhold 2003].

While the width of a streaming mesh is a lower bound for the amount of memory required to process it, some processing tasks are inherently span-limited. Any process that requires maintaining the same element order between input and output while accessing all neighboring elements must buffer on the order of span elements. Consider, for example, vertex normal computation on a pre-order mesh that directly pipes through triangles but delays vertices until their normals have been computed (i.e. the output is post-order). Because vertex indices can not change, vertices cannot be output until all vertices with smaller index have. Conversion between pre-order and post-order as well as on-the-fly vertex compaction are likewise span-limited. These are common operations on streaming meshes as algorithms like to consume vertex-compact pre-order input but often produce non-compact post-order output. Hence keeping both width and span low is useful. As such, the compression order of [Isenburg and Gumhold 2003] and the simplification order of [Isenburg et al. 2003], are not well-suited for streaming as they result in near-maximal span. Preferably such processes not only utilize but also preserve the stream layout they are provided with.

Processing Sequences

Streaming meshes are a lightweight mesh representation and do not provide information such as manifoldness, valence, incidence, and other useful topological attributes. Processing sequences [Isenburg et al. 2003] are a specialization of streaming meshes that provide such information as well as a mechanism for storing user data on the stream boundaries (e.g. for mapping between external and in-core vertex indices). We prefer to view processing sequences as nothing but a richer interface for accessing streaming meshes. Implementing a processing sequence API only involves buffering $O(width)$ mesh elements until they are finalized, at which point complete connectivity information about them is known. As a result we can read and write simple streaming meshes but retain the option to process them through the more powerful processing sequence API.

5 Generating Streaming Meshes

Many applications that generate large meshes can easily produce streaming meshes. They only need to interleave the output of vertices and triangles and provide information when vertices are no longer referenced. Even if this information is not exact, some conservative bounds often exist. For example, a marching cubes iso-surface implementation could output all vertices of one volume layer, followed by a set of triangles, and then finalize the vertices before moving on to the next layer. This is the technique we used to produce the coherent ppm mesh from Table 3. Here, even *implicit* finalization in the form of a bound on the maximum number of vertices per layer would be sufficient to finalize vertices.

In this sense, streaming meshes are often the natural output of existing applications. Given limited memory resources, it is quite *difficult* to produce incoherent meshes as the mesh generating application can only hold and work on small pieces of the data at any time. However, there are a large number of meshes that are stored in legacy formats. We now outline various out-of-core algorithms that convert from a standard indexed format to a streaming format. They may also be used to improve the layout of existing streaming meshes that either introduce/finalize vertices too early/late or that have an overly incoherent layout.

5.1 Out-of-Core Mesh Layout

For all of our streaming mesh conversion tools, we rely on a few basic steps. To create a streaming mesh in pre- or post-order, we need: a vertex layout, a triangle layout, and finalization information.

Layout In an initial pass over the input mesh we write vertices and triangles to separate temporary files. We store with each vertex its original index so that after reordering it can be identified by its triangles. If we do not wish to keep both vertex and triangle layouts fixed, we specify only one layout explicitly and ensure the other layout is made compatible. Each explicit layout is specified as an array of unique sort keys, one for each input vertex or triangle, which we merge with the input elements into their temporary files and on which we perform an external sort (on increasing key value) to bring the elements into their desired order.

For a specified triangle layout, we assign (not necessarily unique) sort keys k to vertices v based on their new incident triangle indices t: for pre-order meshes we use $k_v = \min_{v \in t} t$; for post-order $k_v = \max_{v \in t} t$. Conversely, if a vertex layout is specified, we compute pre-order triangle keys $k_t = \max_{v \in t} v$ and post-order keys $k_t = \max_{v \in t} v$. These keys are, of course, based on the indices in the reordered mesh. Thus, when an explicit vertex order is specified we must first dereference triangles and update their vertex indices. For a conventional indexed mesh, we accomplish this dereferencing step via external sorts on each vertex field [Chiang and Silva 1997]. If on the other hand the input is already a streaming mesh, we can accomplish this step much faster by dereferencing the (active) vertices, whose keys are maintained in-core, on-the-fly while the input is first processed. Whether we wish to create a streaming mesh or simply reorder an indexed mesh, this is yet another benefit of having streaming input.

Finalization For nonstreaming input we compute implicit finalization information by first writing all corners $\langle v, t \rangle$ to a temporary file. We sort this file on the vertex field v and then compute the degree d (number of incident triangles) for each vertex, which will later be used as a reference count. For streaming input we compute the degrees on-the-fly.

Output Once vertex and triangle files have been laid out, we output a streaming mesh by scanning these files in parallel. For a pre-order mesh the output is driven by triangles: for each triangle we read and output vertices one at a time until all three vertices referenced by the triangle have been output (possibly also outputting skipped vertices not referenced by this triangle). Conversely, for a post-order mesh we drive the output by vertices: for each vertex we tap the triangle file until all incident triangles have been output. We maintain for each active vertex a degree counter that is decremented each time the vertex is referenced. Once the counter reaches zero the vertex is finalized and removed from the front.

Interleaving Indexed Meshes

If the indexed mesh layout is reasonably coherent, we can construct a streaming mesh simply by *interleaving* vertices and triangles, i.e. without reordering them. As outlined above, we first separate vertices and triangles and compute vertex degrees. Since the mesh elements are already in their desired order, no further sorting is needed and we simply interleave them using the output procedure above. We arbitrarily chose pre-order output for our interleaved meshes.

Compaction

It makes little sense to apply pre-order interleaving to meshes with high skip, such as the dinosaur or the lucy model. To stream these meshes we must change at least one of the vertex and triangle layouts. We can always eliminate the skip via pre-order *vertex compaction* by fixing the triangles and reordering the vertices using the pre-order vertex sort keys defined above. Hence during output each triangle's vertices have either already been output or appear next in the vertex file. Post-order vertex compaction is more difficult, and either requires $O(span)$ memory or additional external sorts.

If the vertex layout is already coherent but the triangle layout is not, *triangle compaction* is worthwhile. For each vertex, in pre-order triangle compaction we immediately output all triangles that can be formed with this and previous vertices; in post-order compaction we output all triangles formed with this and later vertices.

Spatial and Topological Sorting

Perhaps the simplest method for constructing a streaming mesh is to order its elements along a spatial direction. We can compute a sort key for each vertex by projecting its coordinates onto an axis. To simplify laying out the triangles from nonstreaming input, however, we prefer unique integer vertex keys, which can be used for both sorting and indexing. Thus we first sort the geometric keys and replace them with consecutive integers. Once vertices and triangles have been sorted, we drive the output by triangles (vertices) to produce a vertex-compact (triangle-compact) mesh. We sorted the mesh along its maximal x, y, or z extent in our experiments.

An alternative to spatial sorting is topological traversal of the mesh. In Table 3 we report results for breadth-first vertex sorts and and depth-first triangle sorts. These meshes were laid out in-core on a computer with large memory, although techniques based on external memory data structures, e.g. [Cignoni et al. 2003], or a variation on the clustering scheme below could also be employed.

Spectral Sequencing

For many meshes, spatial or topological sorting will produce sufficiently coherent layouts. However, if the mesh is "curvy" (like the dragon), with changing principal direction, or "spongy" (like the ppm surface), with complex topology and space-filling geometry, these strategies produce layouts that are far from optimal. The traversal shown in Figure 5(d), for example, follows the winding body of the dragon and achieves a much lower front width. We here describe a method particularly aimed at generating low-width streams. Because the stream boundaries have to turn corners to be as short as possible, and therefore do not advance uniformly, the span will suffer in favor of the width.

Minimizing front width has received attention in the graph layout [Díaz et al. 2002] and finite element literature [Scott 1999]. The front width (also called wavefront) of a compact mesh is equivalent to the *vertex separation* of its associated graph (1-skeleton). Minimizing vertex separation is NP-hard [Díaz et al. 2002], however good heuristics exist, such as *spectral sequencing*, which minimizes the sum of squared edge lengths in a linear graph arrangement. Spectral sequencing involves finding a particular eigenvector (the *Fiedler vector*) of the graph's Laplacian matrix. To solve this

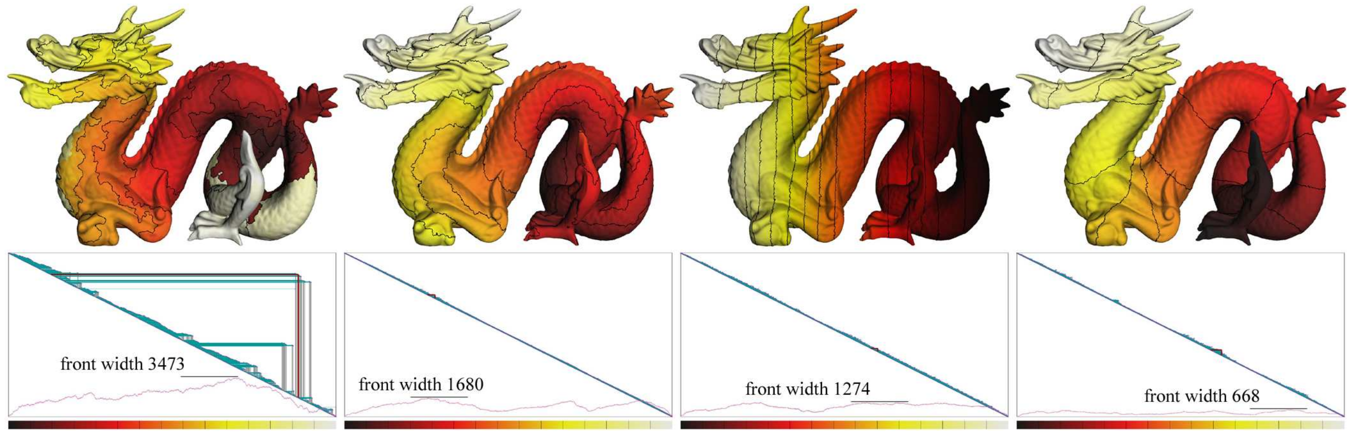


Figure 5: The dragon mesh reordered by (a) a depth-first sort compressor, (b) a breadth-first sort compressor, (c) spatial sort, and (d) spectral sequencing.

mesh name	inter-leaving	com-paction	spatial sorting		spectral sequencing	
			ranking	total	ranking	total
buddha	0:05	0:09	0:02	0:13	0:27	0:33
lucy	4:36	5:11	1:03	11:33	3:41	6:59
st. matthew	1:41:29	2:08:36	21:18	4:09:52	45:42	2:31:47

Table 1: Timings (h:m:s) including compressed I/O on a 3.2 GHz PC.

problem efficiently on large meshes, we have adapted the ACE multiscale method [Koren et al. 2002] to work in an out-of-core setting.

To make the problem more tractable, we presimplify the mesh using a variation of the streaming edge collapse technique from [Isenburg et al. 2003], and contract vertices into clusters based purely on topological criteria aimed at creating uniform and well-shaped clusters. We then apply ACE to lay out the clusters in-core, and finally order the mesh cluster by cluster, with no particular vertex order within each cluster. While the intra-cluster order can be improved, the reduction in width is bounded by the cluster size .

5.2 Results

We have measured the performance of our mesh layout tools on a 3.2 GHz Intel XEON PC running Linux with 2 GB of RAM. Table 1 summarizes the performance on a few meshes. Like spatial sorting, spectral sequencing is broken down into the vertex ranking phase and the reordering phase, with ranking running at up to 140 Ktps for large meshes. Interleaving uses gzipped PLY as input and writes a binary streaming mesh. This is used as input to compaction, which outputs a compressed streaming mesh for input to spatial sorting, and so on. Spatial sorting considers the compact but sometimes high-width input unstreamable, while spectral sequencing takes advantage of streaming input during layout; hence the large speedup in this phase.

We now turn our attention to Table 3, which lists layout and stream measures for several meshes and layout strategies. We immediately notice from the layout diagrams the poor coherence in most of the original meshes. Hence interleaving works well only in a few cases, most notably for the ppm surface. Vertex compaction works well for the horse and dinosaur, whose triangles are well-ordered but whose vertex layouts are either poor or incompatible.

Breadth-first traversals naturally minimize the front span since the “oldest” vertex introduced tends to be finalized first. Thus these spans, and hence widths, are consistently low. Indeed, even for the 500+ million triangle atlas mesh, we can reference vertices using only 15-bit relative indices. For such low-span streams, we may opt to use a circular buffer of size *span* instead of a hash of size *width* to maintain active vertices. Thus using simple and fast sequential I/O we can very quickly dereference (e.g. for previewing) the 8.5 GB atlas using no data structures other than a 380 KB fixed-size array.

Depth-first traversals, on the other hand, leave the oldest vertices hanging and therefore guarantee high-span layouts, as evidenced in Table 3. Long spans also tend to accumulate into high widths—especially for high-genus meshes such as the ppm surface. For each topological handle, the front elements of a traversal eventually split into two unconnected groups. A depth-first traversal leaves one group hanging on the stack until reaching it from the other side. This suggests that standard mesh compression based on depth-first traversals, as used for example in [Isenburg and Gumhold 2003; Isenburg et al. 2003], is not well-suited for streaming.

Finally, spectral sequencing tends to yield the lowest width at the expense of a high span, although with a few notable exceptions. For large meshes, this is a direct result of coarse granularity clustering, which leaves the front increasingly ragged as it winds around the clusters. (We capped the number of clusters at one million for all meshes.) Instead, for meshes with simple geometry like the large statues, spatial sorting works sufficiently well.

6 Streaming Compression

Current mesh compression schemes do not preserve the layout of a mesh. They first construct an explicit representation of the mesh connectivity, which they then traverse using a deterministic strategy, while encoding only an unlabeled connectivity graph. The mapping from graph nodes to vertices is established by compressing vertex positions in the order they are encountered. This implicitly assumes that the compressor is free to reorder the mesh, in which case the particular traversal strategy used dictates its layout.

Although compressing an initially incoherent mesh will usually improve its layout, traversal heuristics really aim at lowering the bit-rates, with good output layouts being coincidental and not part of the design. On the contrary, the classic stack-based approaches [Touma and Gotsman 1998; Rossignac 1999; Isenburg and Snoeyink 2000] traverse meshes in depth-first order and thereby generate triangle orderings of maximal span. The layout artifacts of such compressors are shown in Figure 5. They also produce orderings of unnecessary high width—especially for meshes with many topological handles. So far no attention has been given to what these compressors do to the layout of a mesh—maximum compression and algorithmic elegance have been the sole design criteria.

We depart from the traditional approaches and use a scheme that encodes the triangles of the mesh in the order they are given to the compressor. Since our scheme not only encodes the connectivity but also a particular triangle ordering, it can not guarantee the same compression as traditional schemes. But being able to compress a mesh *on-the-fly* makes our scheme more usable in a mesh processing pipeline. The main advantage is the elimination of the pre-processing step that constructs explicit mesh connectivity. This is especially beneficial for compressing large meshes. Previous approaches spend significant amounts of memory, temporary disk space, CPU time, and file I/O on either cutting the mesh in

mesh	ordering	delay = 0			= 250		= 10,000		ooc-compressor [IG2003]				
		rate	time	RAM	rate	time	rate	time	rate	prepr.	compr.	RAM	disk
lucy	vcompact	13.6	1	37	12.1	2	8.5	2	1.9	19	5	128	0.9
	breadth	3.5	1	1.6	2.6	1	3.4	2	bpv	min	min	MB	GB
david_{1mm}	vcompact	15.9	2	4.8	4.9	3	3.8	4	1.8	36	14	192	1.7
	spectral	14.3	2	1.8	6.6	3	5.9	4	bpv	min	min	MB	GB
st.	vcompact	15.6	14	5.2	4.8	20	3.9	28	1.8	7	4	384	11
matthew	geometric	13.6	15	4.0	6.6	23	5.3	29	bpv	hrs	hrs	MB	GB

Table 2: Comparing connectivity rates [bpv], timings [min], and memory use [MB] of our streaming compressor with [Isenburg and Gumhold, 2003].

smaller pieces [Ho et al. 2001] or constructing external memory data structures [Isenburg and Gumhold 2003], before the compression process even starts. Our scheme makes compression nearly user-transparent and is practically independent of the mesh size.

Compressing in Stream Order

We have designed a new streaming compression scheme that can encode the triangles of a mesh in whatever order they happen to be in. It requires no preprocessing and uses only minimal memory resources if the input mesh is stored in a streaming format or produced in a streaming fashion. We have implemented a *streaming mesh writer* and a corresponding *reader* through which compressed meshes can be written and read in increments of single vertices and triangles. Post-order meshes need to be piped through a "post2pre" filter because our compressor only writes pre-order meshes.

We include a fully functional compressor and decompressor on the CD-ROM that accompanies this paper.

In Table 2 we compare our streaming compressor to that of Isenburg and Gumhold [2003]. For the "St. Matthew" they spend 7 hours creating an 11 GB data structure on disk before actual compression begins, which takes another 4 hours and uses 384 MB of RAM on a 2.8 GHz Pentium IV. In contrast, running on a 1.1 GHz mobile Pentium III we compress this model in 15 minutes using only 6 MB of RAM and no temporary disk space. While geometry compression rates (not reported) are similar, their 11 hour/11 GB effort pays off with superior, state-of-the-art connectivity compression rates. But so far we have not reordered a single triangle.

When the compressor follows the exact triangle order in which the mesh is written, it generally needs to store at least $\log_2(width)$ bits per triangle. We can significantly improve compression by employing a small *delay buffer* within which the compressor locally reorders triangles. It greedily brings them into a vertex-connected order that often allows avoiding those $\log_2(width)$ bits without affecting the overall stream quality. A delay buffer of 10,000 triangles, for example, gives average connectivity rates of 4 to 5 bits per vertex [bpv] while slowing compression by a factor of two (not optimized yet) and increasing the memory footprint by only 5 MB.

To support quantization of floating-point geometry for streaming meshes whose bounding box is not known in advance, we use a scheme that quantizes conservatively using a bounding box that is learned as the mesh streams by. Our streaming mesh writer also supports lossless floating-point compression when quantization—for whatever reason—is not an option. A detailed description of our streaming compressor can be found in [Anonymous 2005].

We have included this tech report as additional review material in print and on the CD-ROM.

7 Conclusion

We have identified a major headache in large mesh processing—poor mesh layouts—and suggested how to avoid this pain—keeping the mesh in a streamable layout. We have both established a theoretical framework that characterizes the quality of a layout and presented out-of-core tools for improving poor layouts. Coherent mesh layouts can be streamed by interleaving vertices and triangles in their original order and adding finalization information. Incompatible vertex and triangle layouts can can be made streamable by *compacting* (reordering) one of the layouts. Layouts high in width and in span require reordering both triangles and vertices.

We should point out that streaming formats are no universal solution for all out-of-core processing purposes. But they are much better than the only current alternative, standard indexed formats. Documenting coherency in the file format makes processing large meshes considerably more efficient. It solves the main problem of dereferencing that complicates most out-of-core mesh applications, such as rendering an initial image to get an idea of what data one is dealing with; counting the number of holes, non-manifolds, and components; computing shared normals, the total surface area, or curvature information; segmenting, simplifying, or compressing the mesh; or constructing hierarchical mesh structures. Streaming meshes are not tied to a particular format. One may even read streaming meshes from standard formats such as PLY or OBJ given that the mesh layout is low in span and compatible by buffering $O(span)$ vertices and finalizing them conservatively.

We described a novel streaming compression algorithm that encodes a mesh triangle by triangle, optionally reordering them only locally for better compression. While we do not achieve state-of-the-art connectivity compression, the sacrifice in bit-rate is well spent. We can now compress streaming meshes of unlimited size on-the-fly, which makes compression a more useful tool for mesh processing. Contrast this with prior schemes that first spend hours constructing external data structures that use gigabytes of auxiliary disk space—even if the mesh already has a nice layout. We also point out that traditional, stack-based compressors systematically create meshes of maximal span and become inefficient for high-genus models where the stack can grow very deep. Instead, traditional, non-streaming compressors should traverse meshes breadth-first and give vertices a similar "lifetime" on the front.

In the future we would like to investigate concurrent streaming at multiple resolutions, multiplexing of streaming meshes for parallel processing, and extensions to volume meshes. We also envision that some sort of 'space finalization' would be useful for algorithms that require a spatially—as opposed to a strictly topologically—coherent traversal, such as vertex clustering algorithms.

References

ANONYMOUS. 2005. Streaming compression of triangle meshes. tech report.

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

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

CHIANG, Y.-J., AND SILVA, C. T. 1997. I/O optimal isosurface extraction. In *Visualization'97 Proceedings*, 293–300.

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

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

DÍAZ, J., PETIT, J., AND SERNA, M. 2002. A survey of graph layout problems. *ACM Computing Surveys 34*, 3, 313–356.

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

HOPPE, H. 1996. Progressive meshes. In *SIGGRAPH'96 Proceedings*, 99–108.

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

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

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

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

ISENBURG, M., LINDSTROM, P., GUMHOLD, S., AND SNOEYINK, J. 2003. Large mesh simplification using processing sequences. In *Visualization'03*, 465–472.

KOREN, Y., CARMEL, L., AND HAREL, D. 2002. ACE: A fast multiscale eigenvector computation for drawing huge graphs. In *IEEE Info. Visualization*, 137–144.

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

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

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

SCOTT, J. A. 1999. Ordering elements for a frontal solver. *Communications in Numerical Methods in Engineering 15*, 309–323.

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

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

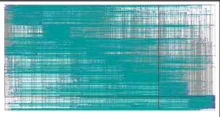
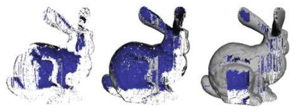
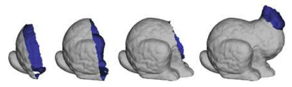
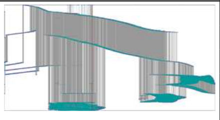
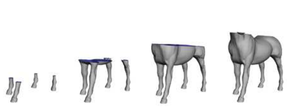
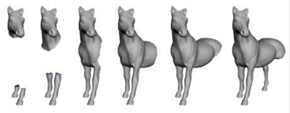
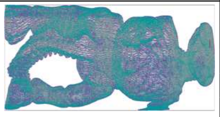
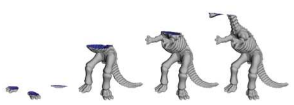
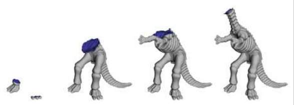
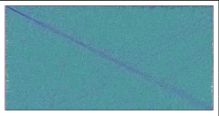
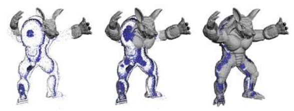
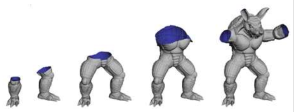
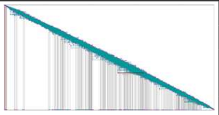
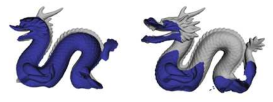
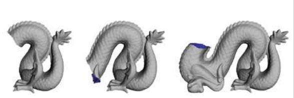
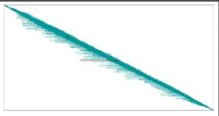
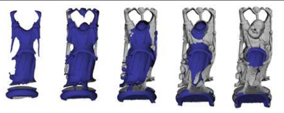
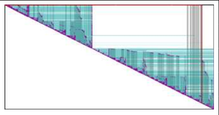
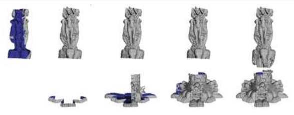
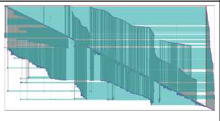
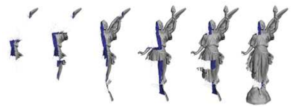
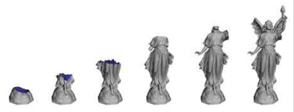
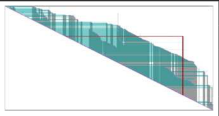
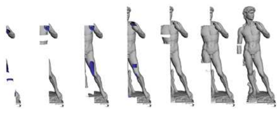
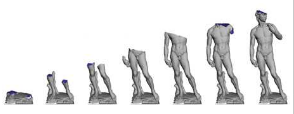
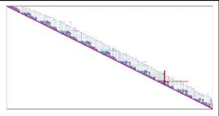
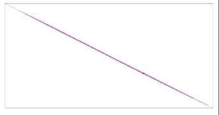
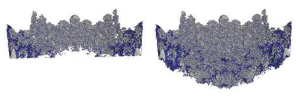
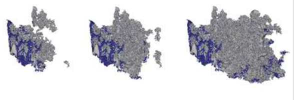
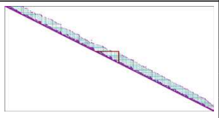
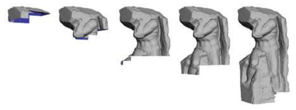
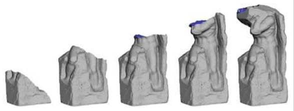
mesh description			original triangle order				spectral sequencing		spa-tial sort	breadth first sort	depth first sort
name genus # comp. # vertices # triangles	width skip v-span t-span	layout diagram	*inter-leaved* width span	*v-com-pacted* width span	snapshots		width span	snapshots	width span	width span	width span
bunny 0 1 35,947 69,451	9,133 34,569 35,742 69,181		34,813 34,834	9,133 34,549			**228** 785		413 1,502	334 **354**	405 21,704
horse 0 1 48,485 96,966	550 40,646 48,471 6,204		40,653 48,485	550 3,167			**303** 3,286		419 2,563	443 **466**	440 47,446
dinosaur 0 1 56,194 112,384	496 55,196 4,353 2,017		55,331 55,680	496 1,083			**241** 1,382		568 1,825	357 **383**	409 51,315
armadillo 0 1 172,974 345,944	51,951 171K 172K 345K		172K 172K	51,951 172K			**638** 4,405		1,042 3,796	1,115 **1,199**	1,457 171K
dragon 46 151 437,645 871,414	4,586 434K 434K 109K		434K 434K	4,586 54,825			**668** 11,617		1,274 9,243	1,680 **2,015**	8,583 435K
buddha 104 1 543,652 1,087,716	5,037 94,080 24,889 205K		98,121 111K	5,037 102K			**883** 6,993		1,556 12,682	1,975 **2,335**	14,639 543K
thai statue 3 1 4,999,996 10,000,000	53,416 0 4.70M 9.41M		53,416 4.70M	53,416 4.70M			**3,761** 150K		6,003 43,970	7,051 **7,897**	35,461 4.99M
lucy 0 18 14,027,872 28,055,742	255K 11.5M 13.5M 26.8M		11.6M 13.5M	255K 13.5M			5,841 200K		**4,985** 20,362	5,904 **6,547**	12,904 12.4M
david$_{1mm}$ 137 2,322 28,184,526 56,230,343	26,383 1,568 15.8M 31.5M		26,405 15.8M	26,383 15.8M			**7,862** 752K		8,919 36,421	8,282 **8,971**	35,770 28.1M
st. matthew 483 2,897 186,836,665 372,767,445	31,931 2,121 29.1M 58.3M		31,932 29.1M	31,931 29.1M			33,029 3.85M		33,207 157K	**23,602** **25,554**	110K 185M
ppm 167,636 167,584 234,901,044 469,381,488	311K 306K 617K 924K		616K 617K	311K 462K			**56,179** 27.0M		114K 290K	99,410 **112K**	3.07M 206M
atlas 5,496 38 254,837,027 507,512,682	28,701 139 30.6M 61.2M		28,705 30.6M	28,701 30.6M			45,998 28.5M		**22,638** 64,354	29,923 **32,156**	246K 254M

Table 3: Layout and stream measures for the meshes used in our experiments. We report the vertex width and triangle span of the original triangle order, as well as the vertex skip and vertex span of the original vertex order (which can be quite incoherent). For the original triangle order, we report the front width and span of streaming meshes created through interleaving and vertex compaction, and include snapshots of these meshes. The rightmost four columns highlight the improvements of vertex-compact streams obtained by reordering both triangles and vertices using spectral sequencing, spatial sorting along the axis of maximum extent, breadth-first vertex sorting, and depth-first triangle sorting. We also list the genus and component, vertex, and triangle counts for each mesh.