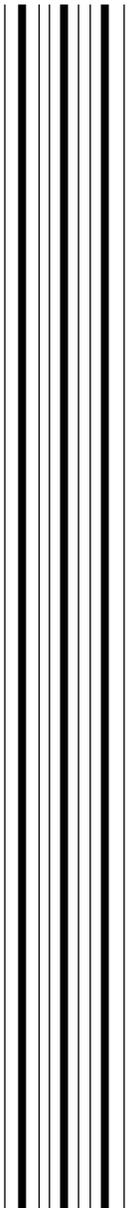


Streaming Meshes

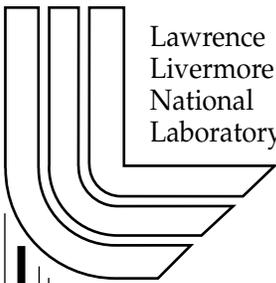
M. Isenburg and P. Lindstrom

To appear at IEEE Visualization 2005

August 1, 2005



U.S. Department of Energy



Lawrence
Livermore
National
Laboratory

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

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

Streaming Meshes

Martin Isenburg
University of North Carolina
at Chapel Hill

Peter Lindstrom
Lawrence Livermore
National Laboratory

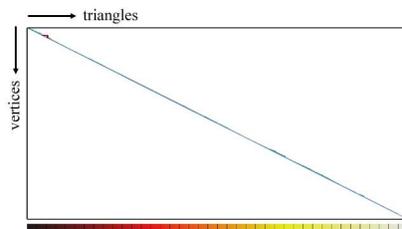
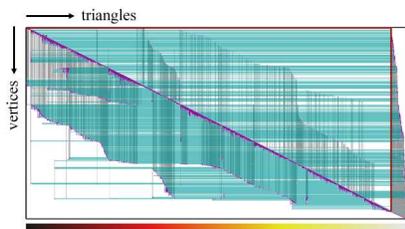


Figure 1: Illustrations of the coherence in the layout of a mesh: On the left the original layout; on the right the layout after reordering the vertex and triangle arrays using spectral sequencing. The large rendering color-codes triangles based on their position in the array. The layout diagram connects 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 that is simple enough to replace current offline mesh formats and is more suitable for representing 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 and methods for converting meshes from a traditional to a streaming format.

A 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 subsequent processing a difficult task. The sheer amount of data may not only exhaust the main memory resources of common desktop PCs, but also exceeds 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,

e-mail: isenburg@cs.unc.edu, pl@llnl.gov.

For demos and source code, visit <http://www.cs.unc.edu/~isenburg/sm/>.

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, however today’s data sets are up to four orders of magnitude larger. 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 [9]. 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 [10], who proposed a compressed mesh format that allows streaming decompression—but not streaming compression. During compression a front sweeps over the entire mesh once, which is accessed through a complex external memory data structure. During decompression, however, only the

front needs to be maintained in memory. Later, Isenburg et al. [11] showed that the streaming access provided by the decompressor can be used for I/O-efficient out-of-core simplification. However, these works pay little attention to what makes good stream orders. In fact, we show that the streaming meshes produced by these prior methods are not particularly well-suited for stream processing.

In this paper we extract the essence of streaming to define a simple input *and* output format. We propose definitions and metrics that give us a language for talking about streaming meshes. We identify two fundamental characteristics, the *width* and the *span* of streaming meshes, and describe the practical impact of these metrics on stream processing. We report these metrics for a number of different mesh layouts and describe out-of-core techniques for creating such layouts using limited memory. We briefly discuss a novel scheme for *streaming compression* that, while being the topic of a separate paper [12], is another example of the type of I/O-efficient algorithm design enabled by our streaming mesh format.

2 Previous Work

While models from 3D scanning and isosurface extraction have become too large to fit in the main memory of commodity PCs, storing the models on disk is generally 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 small enough to fit into main memory and then process each piece separately. This strategy has been used for distribution [15], simplification [1], and compression [7]. 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. [4], for example, use such an external memory mesh to simplify large models using iterative edge contraction. Similarly, Isenburg and Gumhold [10] 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 overcoming 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, and either attempt to reconstruct or entirely disregard connectivity. Lindstrom [16] implements clustering-based simplification this way. Although he does not use indices, his input meshes usually come in an indexed format, in which case an initial dereferencing step [3] is needed that does 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 [20] simplify large models by streaming coherent triangle soup through 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

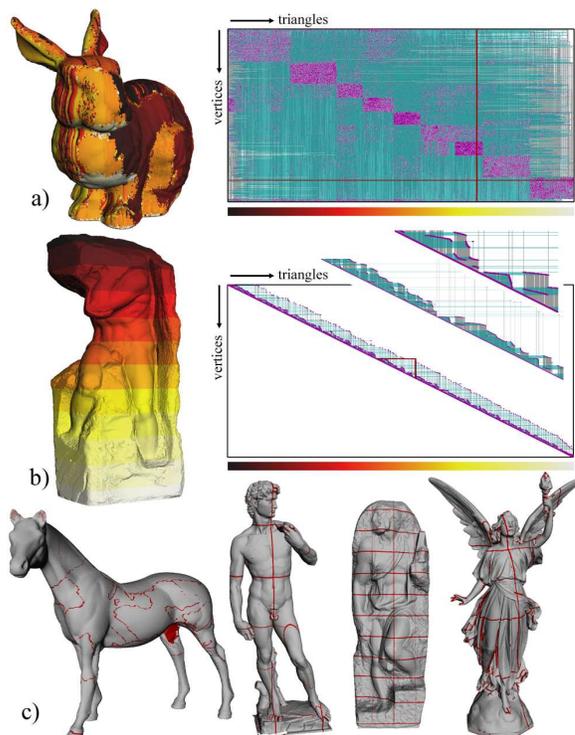


Figure 2: Visual illustrations of mesh layouts: (a) The bunny and (b) the 10,000 times more complex Atlas model. Successive triangles are rendered with smoothly changing colors. Layout diagrams intuitively illustrate incoherence in the meshes. (c) Highlighting triangles with high vertex span often reveals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

must remain in the buffer until the end. Their output is therefore guaranteed to be incoherent. Isenburg et al. [11] 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 [5] stores triangles together with explicit instructions that tell the cache which vertices to replace. Hoppe [8] produces coherent triangle orderings optimized for a particular cache size, while Bogomjakov and Gotsman [2] and Yoon et al. [21] create orderings that work well for all cache sizes.

An on-disk layout that is good for streaming seems similar to an in-memory layout that is good for rendering. 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. While local coherence is of crucial importance for a rendering sequence, it has little significance for streaming. What is of big practical difference here is whether the layout has global coherence or not.

3 Layouts of Indexed Meshes

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

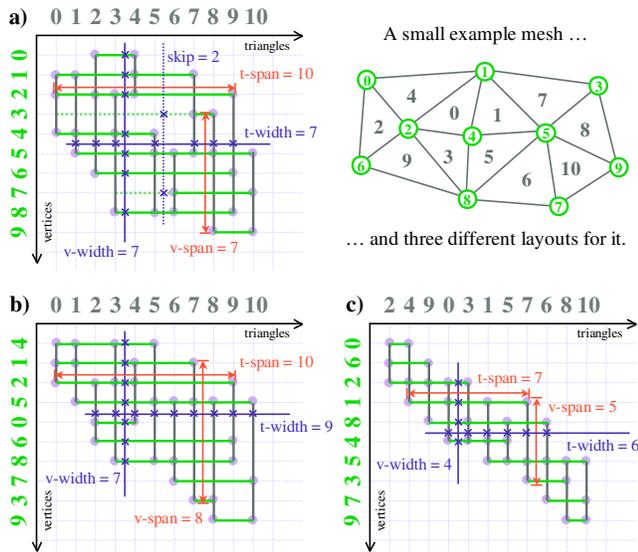


Figure 3: Three layouts of a mesh: (a) An incompatible vertex order results in a skip. (b) Reordering the vertices eliminates the skip but does not affect triangle span or vertex width. (c) Reordering also the triangles can reduce those.

ends of the array, the first and the last triangle may reference 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 coherence 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 [15]. 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 cannot be dealt with on commodity PCs, the statue is provided in twelve pieces.

We characterize the *layout* of an indexed mesh with several measures that tell us how “streamable” the current vertex and triangle orderings are: The *triangle span* of a vertex, shown as a green horizontal line in the layout diagram, is the number of triangles between and including the first and last reference to the vertex. Conversely, the *vertex span* of a triangle, shown as a gray vertical line, is the maximal index difference (plus one) of its vertices. The vertex (triangle) span of a layout is the maximal span of all triangles (vertices). The *vertex width* of a layout is the maximal number of green segments that can be cut by a vertical line; the *triangle width* is the maximal number of gray segments cut by a horizontal line. Finally, anticipating sequential access to vertices and triangles, we say a vertex v is skipped from the point that a vertex with higher index is referenced until v is first referenced. The *skip* of a layout is the maximal number of “concurrently” skipped vertices. See Figure 3 for example layouts of a simple mesh.

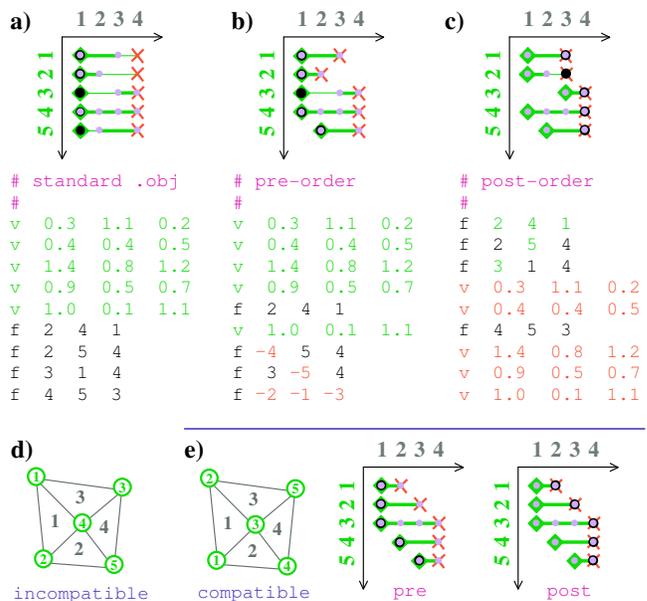


Figure 4: Examples of a streaming ASCII format. (a) Standard OBJ format. (b) Streaming pre-order format: finalization is coded through negative relative indices, introduction coincides with appearance of vertex in the stream. (c) Streaming post-order format: finalization coincides with appearance of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

4 Streaming Meshes

A streaming mesh format provides concurrent access to indexed vertices and triangles that reference them, plus explicit information about the last time a vertex is referenced. While such formats require only simple changes over standard formats (Figure 4), they have tremendous advantages. Because the format tells us which vertices to keep in memory, the problem of repeated, possibly incoherent look-up of vertex data in a gigantic array does not exist. Furthermore, the ability to process and deallocate vertices after they are last referenced keeps the memory footprint small. To formally discuss streaming meshes, we first provide some definitions.

Definitions

A *streaming mesh* is a logically interleaved sequence of indexed vertices and triangles with 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 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 $\max_i \{\max F_i - \min F_i + 1\}$ 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.

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, and 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 would be referenced by the previous triangle. We say that a stream is *vertex-compact* if each vertex is referenced by the previous or the next triangle in the stream. Vertices can always be made compact with respect to the triangles by rearranging them, which causes front width to equal vertex width. Similarly, we say that a stream is *triangle-compact* if each triangle references the previous or the next vertex in the stream. For a pre-order mesh this means that each triangle appears immediately after its last vertex has appeared; for a post-order mesh each triangle appears just before its first vertex is finalized. (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, which causes front span to equal vertex span (since the oldest active vertex could be finalized if it were not waiting on a neighbor). Finally, a streaming mesh is *compact* if it is both vertex- and triangle-compact. At least one of three indices per triangle in a compact mesh is redundant, and may be omitted.

Working with Streaming Meshes

A streaming format allows reading and writing meshes of practically arbitrary size in an I/O-efficient manner without forcing any particular ordering upon the process or the person creating the mesh. Whether a mesh is extracted layer by layer (like the “ppm” isosurface) or block by block (like the “atlas” statue), simply outputting vertices and triangles in the order they are created while finalizing vertices that are no longer in use makes operating on the largest of data sets a feasible task. For example, all 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.

A streaming mesh format is also the ideal input and output for stream processing. In this model, the mesh streams through an in-core *stream buffer* 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, consisting only of the front, is needed. For more elaborate processing tasks, a larger stream buffer may hold as many additional mesh elements as there are memory resources, allowing random access to a localized but continuously changing subset of the mesh.

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. Envision a scenario where one module 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 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.

Any stream process must map external indices to whatever internal data structures are used for the active vertices. This is conveniently done using a hash table, however a more efficient solution is possible for low-span meshes. If the front span does not exceed the size of the internal vertex buffer, then a fixed-size circular array can be used in place of a hash as “collisions” due to overflow

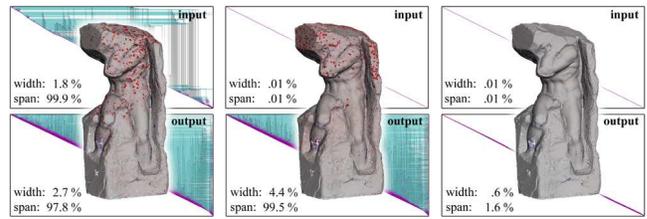


Figure 5: Error-driven [11] (left, middle) vs. order-preserving (right) simplified output for depth-first [10] (left) and breadth-first (middle, right) input.

are not possible. To handle arbitrary streams, we advocate a hybrid approach in which stagnant high-span vertices are moved from the circular array to an auxiliary hash whenever collisions occur.

While the width of a streaming mesh is a lower bound on the amount of memory required to process it, some processing tasks are inherently span-limited. Any process that needs to access adjacency *and* maintain the same element order between input and output must buffer on the order of span elements. Conversion between pre-order and post-order meshes as well as on-the-fly vertex compaction are span-limited operations if the triangle order needs to be maintained. 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 depth-first compression order of [10] and the error-driven simplification order of [11] are poor design choices as they unnecessarily result in near-maximal span (Figure 5). 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 that do not provide information such as manifoldness, valence, incidence, and other useful topological attributes with each read triangle. *Processing sequences* [11] 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 view processing sequences simply as a richer interface for accessing streaming meshes. Implementing a processing sequence API only involves buffering $O(\text{width})$ 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.

Streaming Compression

The sequential nature of streaming I/O invites the opportunity for compression. However, popular mesh compression schemes, while generally supporting streaming decompression, require random access to the entire mesh during compression and globally reorder the mesh. They traverse the mesh using a deterministic strategy agreed upon by the compressor and decompressor and impose this particular order on the mesh. Worse yet, the classic stack-based approaches [13, 17, 19] traverse meshes in depth-first order and thereby generate layouts of maximal span and unnecessarily high width—especially for meshes with many topological handles. Layout artifacts of such compressors are shown in Figures 5 and 6.

We have designed a streaming compressor [12] that accepts and compresses vertices and triangles in whatever order the user provides them in. Optionally, the compressor is allowed to locally reorder triangles within a small buffer to improve compression rates, resulting in a connectivity cost within a factor of two of state of the art. With geometry dominating the overall coding cost, this minor overhead is more than made up for in usability by eliminating hours of pre-processing and gigabytes of temporary disk space [7, 10]—even if the mesh already has a nice layout—and by supporting transparent compressed I/O on-the-fly as the mesh is written.

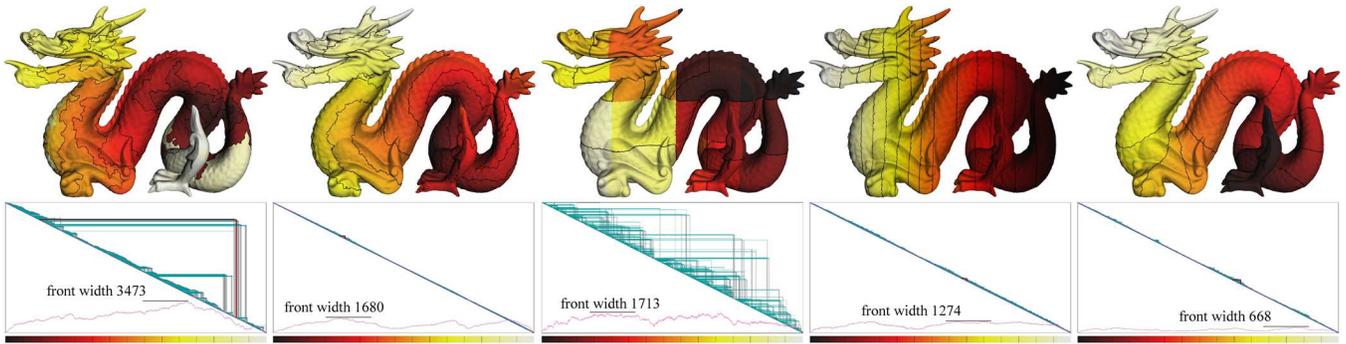


Figure 6: The dragon mesh reordered by (a) a depth-first compressor, (b) a breadth-first compressor, (c) z-order curve, (d) spatial sort, and (e) spectral sequencing.

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 isosurface 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 2. 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 large incoherent meshes as the mesh generating application can only hold and work on small pieces of the data at any time. However, most current meshes are stored in legacy formats. We now outline various out-of-core algorithms for converting from a standard indexed format to a streaming format, and for improving the layout of 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 $k_t = \min_{v \in t} v$. These keys are, of course, based on the indices in the reordered layout. 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 [3]. 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 and sorting it on the vertex field v . We then compute the degree $d = |\{t : v \in t\}|$ for each vertex, which will later be used as a reference count. For streaming input, degrees are computed on-the-fly.

Output We now have a file with vertex records $\langle k_v, v, d, x, y, z \rangle$ and a file with triangle records $\langle k_t, v_1, v_2, v_3 \rangle$ that we externally sort on the k fields. We then output a streaming mesh by scanning these files in parallel. Pre-order output is driven by triangles: for each triangle we read and output vertices one at a time until all three vertices of 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.

5.1.1 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.

5.1.2 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 $O(\text{span})$ 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.

Because of inter-dependencies creating streams that are fully compact is more challenging than ensuring vertex- or triangle-compactness alone. Given a (not necessarily compact) triangle sequence we output a compact pre-order mesh as follows. For each remaining triangle t in the sequence, we output its vertices one at a time if they have not yet been output (thus ensuring vertex-compactness). For each newly output vertex v , we also output any triangle (other than t) incident on v whose other vertices have also been output (thus ensuring triangle-compactness). We then output t and continue with the next triangle. This (possibly rearranged) triangle order is therefore induced by the vertex-compact vertex order given by the original triangle order, i.e. $k_t = \max_{v \in t} \min_{s \ni v} s$.

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.

5.1.3 Spatial Sorting

Perhaps the simplest method for constructing a streaming mesh is to linearly order its elements along a spatial direction such as its maximal x , y , or z extent. For nonstreaming input we first “rank” the vertices by sorting them in geometric order and use the rank both as the new (unique) index and sort key. Once vertices and triangles have been sorted, we drive the output by triangles (vertices) to produce a vertex-compact (triangle-compact) mesh.

We also examine layouts based on space-filling curves. For simplicity, we chose the (Morton order) z -curve, for which we can compute sort keys by quantizing the vertex coordinates and interleaving their bits, e.g. as $x_n y_n z_n x_{n-1} y_{n-1} z_{n-1} \dots x_1 y_1 z_1$.

5.1.4 Topological Sorting

An alternative to spatial sorting is topological traversal of the mesh. In Table 2 we report results for breadth-first vertex sorts 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. [4], or a variation on the clustering scheme below could also be employed.

5.1.5 Layouts of Minimal Width and Span

With width and span defining the streamability of a layout, it is natural to ask how to create layouts that minimize these measures. We first note that in a triangle-compact mesh the triangle order is “induced” by the vertex order, and we can therefore without loss of generality focus only on ordering vertices and treat this as a graph layout problem since triangle compaction can only further reduce the width and span. In a triangle-compact mesh front span equals vertex span, which in turn is equivalent to graph *bandwidth* [6], while front width, also known in the finite element literature as *wavefront* [18], is equivalent to *vertex separation* [6].

Both bandwidth and vertex separation are known to be NP-hard, and hence heuristics are needed. Intuitively, breadth-first sorting is a good heuristic for bandwidth, and is often used as an initial guess in iterative bandwidth minimization methods. One popular heuristic for vertex separation is *spectral sequencing*, which minimizes the sum of squared edge lengths in a linear graph arrangement. Spectral sequencing amounts to finding a particular eigenvector (the *Fiedler vector*) of the graph’s Laplacian matrix. To solve this problem efficiently, we use the ACE multiscale method [14].

For large meshes, we presimplify the input using a variation of the streaming edge collapse technique from [11], and contract vertices into clusters based purely on topological criteria aimed at creating uniform and well-shaped clusters. Clusters are maintained in a memory-mapped array as circular linked lists of vertices, using one index per vertex. We then apply ACE to order 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.

6 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. Interleaving takes gzipped PLY as input and writes a binary streaming mesh. We chose to use this as input to compaction, which outputs a compressed streaming mesh for input to spatial sorting, and so on. Here spatial sorting assumes that the input layout is unstreamable, while spectral sequencing takes advantage of coherent streaming input; hence the large speedup in the reordering phase.

Streamability of Layouts

We now turn our attention to Table 2, which lists layout and stream measures for several meshes and layout strategies. We immediately notice the high vertex and triangle spans in the layout diagrams of the original meshes. Often this can be explained by how the mesh was produced. The horse, for example, is zipped together from multiple range scans. 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 vertex arrays and identifying vertices between neighboring blocks, which is evidenced by high vertex spans in Figure 2. For the two largest statues the vertex and triangle spans were somewhat reduced when their “blocky” layouts were multiplexed into several files by spatially cutting the statues into twelve horizontal slices.

Shifting our attention from layouts to streams, interleaving does not work well on high-skip layouts (e.g. lucy), but otherwise produces results similar to vertex compaction. The width and span of a vertex-compacted stream are proportional to the vertex width and triangle span of the layout. Vertex compaction can create low-span streams when only the vertex span is high (e.g. horse and dragon). Triangle compaction produces streams whose width and span are proportional to the triangle width and vertex span, and can produce low-span streams for layouts where only the triangle span is high (e.g. the buddha). To improve layouts where both spans (or both widths) are high, we need to reorder both vertices and triangles.

Breadth-first traversals naturally are low in span—and hence width—since the “oldest” vertex introduced tends to be finalized first. Indeed, even for the 500+ million triangle atlas mesh, we can reference vertices using only 15-bit relative indices. 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 circular array.

Depth-first traversals, on the other hand, leave the oldest vertices hanging and therefore guarantee high-span layouts. 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 [10, 11], is not well-suited for streaming. The z -order layouts also consistently exhibit high spans, although of bounded length and frequency, which results in a lower width.

While spatial or topological sorting produce good layouts for many meshes, they can be far from optimal if the mesh is “curvy” (e.g. the dragon), with changing principal direction, or “spongy” (e.g. ppm), with complex topology and dense geometry. The spectral order in Figure 6e, for example, follows the winding body of the dragon and achieves a much lower width. This low width often comes at the expense of a high span, as the front in spectral layouts does not always advance uniformly in order to be as short as possible. For the large statues, even the width suffers due to coarse granularity clustering (we used at most one million clusters), which leaves the front increasingly ragged as it winds around the clusters.

Stream Processing

Because streaming operates on the mesh in a single pass and because all data access is sequential, disk latency is amortized and I/O throughput is maximized. Our mesh simplification and compression algorithms, for example, achieve throughputs of 0.1–1 million triangles per second. Using edge collapse and quadric error metrics we simplify the St. Matthew mesh to one million triangles in 45 minutes using 9 MB of RAM and 300 MB of disk (for the compressed input) on a 3.2 GHz PC, compared to 14 hours (plus 11 hours of preprocessing time), 80 MB of RAM, and 12 GB of disk on a 0.8 GHz PC for Cignoni et al.’s external memory method [4]. And contrary to [11], 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), and typically using much less memory (e.g. 40 times less for the ppm surface) by using more streamable layouts. Finally, our streaming mesh compressor writes the St. Matthew mesh in 15 minutes while requiring no disk and only 4 MB of RAM, compared to 10.5 hours, 384 MB of RAM, and 11 GB of disk for [10].

7 Discussion

Many of the ideas presented here are quite simple, however we believe they will have significant impact on how large meshes are stored, organized, and processed. And while most of our theory was developed with efficient stream processing in mind, some ideas such as order-preserving compression and maximally compatible, “compact” layouts may find utility in non-streaming applications.

We should point out that streaming formats are no universal solution for all out-of-core processing purposes, as some interactive tasks inherently require random access. However, as an archival and interchange format, streaming meshes are a considerable improvement over the only current alternatives: standard indexed meshes and polygon soup. Even for coherent layouts, simply memory mapping a binary (uncompressed) indexed mesh is not a viable alternative as mesh adjacency and incidence cannot be resolved reliably without finalization or equivalent information. While implicit finalization can be inferred, e.g., in a spatially sorted mesh [11], imposing such a strict order on all mesh reading and writing applications is quite inflexible. Instead memory mapping would require a heavy-weight external mesh data structure, which seems undesirable as a canonical mesh interchange format. On the other hand, such a data structure can be constructed efficiently from streaming meshes [21].

Documenting coherence 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 known to be low in span—by buffering $O(\text{span})$ vertices and finalizing them conservatively. A streaming format, on the other hand, allows processing even high-span meshes using only $O(\text{width})$ memory.

Whereas prior work [10, 11, 20] demonstrated the potential of stream processing as a framework for offline computations on large meshes, the more principled study of streaming presented here reveals, e.g., that the read-only format of Isenburg and Gumhold, which imposes a depth-first layout of inherently poor streamability, is in fact of marginal utility for streaming. Instead, our more general formats that can additionally be *written* in a streaming manner, coupled with the new knowledge of how to create, preserve, and exploit streamable layouts, open up the possibility of truly pipelined stream processing—a previously envisioned but unattained goal.

8 Conclusion

We have identified a major headache in mesh processing—large meshes stored in standard indexed formats without coherence guarantees—and suggested how to avoid this pain—by keeping the mesh in a streaming format that documents how long each vertex is needed. We have defined the metrics *width* and *span* as the two qualities that affect stream processing. We have presented out-of-core tools for converting a mesh from a standard to a streamable format and for improving the width or span of a streaming mesh. For this, we have reported measures and diagrams that characterize existing mesh layouts. These give us a language to talk about the “streamability” of a given layout and help us decide how much re-ordering work is necessary. We have given simple ASCII examples to show that a streaming format is just as easy to create and parse as other formats. Finally, we have sketched a compression scheme that encodes streaming meshes on-the-fly as they are written.

In the future we plan to investigate concurrent streaming at multiple resolutions, multiplexing streaming meshes for parallel processing, and extensions to volume meshes. We also envision that explicit “space finalization,” in contrast to the implicit finalization possible in spatially ordered meshes, would be useful for algorithms that require a spatially—as opposed to a strictly topologically—coherent traversal, such as vertex clustering algorithms.

Acknowledgements This work was in part supported by NSF grant 0429901 and performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48.

References

- [1] 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.
- [2] A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. *Graphics Interface ’01*, 81–90.
- [3] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. *Visualization ’97*, 293–300.
- [4] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.
- [5] M. Deering. Geometry compression. *SIGGRAPH 95*, 13–20.
- [6] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356, 2002.
- [7] J. Ho, K. Lee, and D. Kriegman. Compressing large polygonal models. *Visualization ’01*, 357–362.
- [8] H. Hoppe. Optimization of mesh locality for transparent vertex caching. *SIGGRAPH 99*, 269–276.
- [9] H. Hoppe. Progressive meshes. *SIGGRAPH 96*, 99–108.
- [10] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. *SIGGRAPH 2003*, 935–942.
- [11] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. *Visualization ’03*, 465–472.
- [12] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. *Symposium on Geometry Processing ’05*, 111–118.
- [13] M. Isenburg and J. Snoeyink. Face Fixer: Compressing polygon meshes with properties. *SIGGRAPH 2000*, 263–270.
- [14] Y. Koren, L. Carmel, and D. Harel. ACE: A fast multiscale eigenvector computation for drawing huge graphs. *InfoVis ’02*, 137–144.
- [15] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project. *SIGGRAPH 2000*, 131–144.
- [16] P. Lindstrom. Out-of-core simplification of large polygonal models. *SIGGRAPH 2000*, 259–262.
- [17] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. on Vis. and Comp. Graphics*, 5(1):47–61, 1999.
- [18] J. A. Scott. On ordering elements for a frontal solver. *Communications in Numerical Methods in Engineering*, 15(5):309–323, 1999.
- [19] C. Touma and C. Gotsman. Triangle mesh compression. *Graphics Interface ’98*, 26–34.
- [20] J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes. *Graphics Interface ’03*, 185–192.
- [21] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *SIGGRAPH 2005*, 886–893.

mesh description				original layout			spectral sequencing		geo. sort	topo. sort
name	skip			<i>inter-leaved</i>	<i>v-com-pacted</i>	<i>t-com-pacted</i>		<i>linear</i>	<i>breadth</i>	
genus	v-width	layout diagram		width span	width span	width span	width span	width span	width span	
# comp.	t-width						width span	z-order	depth	
# vertices	v-span							width span	width span	
# triangles	t-span									
bunny	34,569									
0	9,133							413	334	
1	11,135			34,813	9,133	3,924	228	1,502	354	
35,947	35,742			34,834	34,549	34,641	785	583	405	
69,451	69,181							22,999	21,704	
horse	40,646									
0	550							419	443	
1	4,272			40,653	550	2,070	303	2,563	466	
48,485	48,471			48,485	3,167	48,471	3,286	482	440	
96,966	6,204							23,728	47,446	
dinosaur	55,196									
0	496							568	357	
1	2,048			55,331	496	1,028	241	1,825	383	
56,194	4,353			55,680	1,083	4,353	1,382	991	409	
112,384	2,017							42,083	51,315	
armadillo	171K									
0	51,951							1,042	1,115	
1	40,529			172K	51,951	17,873	638	3,796	1,199	
172,974	172K			172K	172K	172K	4,405	1,160	1,457	
345,944	345K							124K	171K	
dragon	434K									
46	4,586							1,274	1,680	
151	7,147			434K	4,586	3,918	668	9,243	2,015	
437,645	434K			434K	54,825	434K	11,617	1,713	8,583	
871,414	109K							252K	435K	
buddha	94,080									
104	5,037							1,556	1,975	
1	6,907			98,121	5,037	3,472	883	12,682	2,335	
543,652	24,889			111K	102K	24,889	6,993	1,688	14,639	
1,087,716	205K							205K	543K	
thai statue	0									
3	53,416							6,003	7,051	
1	54,832			53,416	53,416	29,337	3,761	43,970	7,897	
4,999,996	4.70M			4.70M	4.70M	4.70M	150K	4,989	35,461	
10,000,000	9.41M							1.93M	4.99M	
lucy	11.5M									
0	255K							4,985	5,904	
18	231K			11.6M	255K	113K	5,841	20,362	6,547	
14,027,872	13.5M			13.5M	13.5M	13.5M	200K	11,654	12,904	
28,055,742	26.8M							5.63M	12.4M	
david_{1mm}	1,568									
137	26,383							8,919	8,282	
2,322	52,515			26,405	26,383	26,375	7,862	36,421	8,971	
28,184,526	15.8M			15.8M	15.8M	15.8M	752K	10,705	35,770	
56,230,343	31.5M							6.30M	28.1M	
st. matthew	2,121									
483	31,931							33,207	23,602	
2,897	58,916			31,932	31,931	31,895	33,029	157K	25,554	
186,836,665	29.1M			29.1M	29.1M	29.1M	3.85M	23,858	110K	
372,767,445	58.3M							32.8M	185M	
ppm	306K									
167,636	311K							114K	99,410	
167,584	813K			616K	311K	381K	56,179	290K	112K	
234,901,044	617K			617K	462K	617K	27.0M	148K	3.07M	
469,381,488	924K							125M	206M	
atlas	139									
5,496	28,701							22,638	29,923	
38	58,281			28,705	28,701	28,701	45,998	64,354	32,156	
254,837,027	30.6M			30.6M	30.6M	30.6M	28.5M	37,469	246K	
507,512,682	61.2M							94.8M	254M	

Table 2: Layout and stream measures for the meshes used in our experiments. We report the skip, vertex span, and triangle width of the original vertex order, and the vertex width and triangle span of the original triangle order (which can be quite incoherent). Starting from the original layout, we report the front width and span of pre-order streaming meshes created by interleaving, vertex compaction, and triangle compaction. The rightmost columns highlight the improvements of vertex-compact streams obtained by reordering both triangles and vertices using spectral sequencing, geometric sorting along the axis of maximum extent and along a z-order space-filling curve, and topological breadth- and depth-first traversals. We also list the genus and component, vertex, and triangle counts for each mesh.