Streaming Compression of Tetrahedral Volume Meshes

Martin Isenburg
UC Berkeley
isenburg@cs.berkeley.edu

Peter Lindstrom
LLNL
pl@llnl.gov

Stefan Gumhold
TU Dresden
stefan@gumhold.de

Jonathan Shewchuk
UC Berkeley
jrs@cs.berkeley.edu

ABSTRACT

Geometry processing algorithms have traditionally assumed that the input data is entirely in main memory and available for random access. This assumption does not scale to large data sets, as exhausting the physical memory typically leads to IO-inefficient thrashing. Recent works advocate processing geometry in a "streaming" manner, where computation and output begin as soon as possible. Streaming is suitable for tasks that require only local neighbor information and batch process an entire data set.

We describe a streaming compression scheme for tetrahedral volume meshes that encodes vertices and tetrahedra in the order they are written. To keep the memory footprint low, the compressor is informed when vertices are referenced for the last time (i.e. are *finalized*). The compression achieved depends on how coherent the input order is and how many tetrahedra are buffered for local reordering. For reasonably coherent orderings and a buffer of 10,000 tetrahedra, we achieve compression rates that are only 25 to 40 percent above the state-of-the-art, while requiring *drastically* less memory resources and less than half the processing time.

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

Keywords: mesh compression, geometry streaming, connectivity coding, stream-processing, tetrahedral meshes, volume data.

1 INTRODUCTION

Compression techniques are widely used for compact storage and faster transmission of digital data. While generic compression algorithms such as "gzip" can be applied to any type of data, compression schemes dedicated to a particular type of data typically achieve far superior rates of compression. Efficient compression algorithms for geometric data sets have been pursued since Deering's groundbreaking work [3] from 1995. Over the course of the last ten years, mesh compression has become a mature area of research and numerous algorithms for coding polygonal surface meshes and polyhedral volume meshes have been proposed.

For audio and video, the increasing size of content has led to the design of "streaming" codecs that can compress and decompress a file while keeping only a small portion of it in memory. This allows compression and decompression of audio or video files with arbitrary duration using a relatively small amount of memory. In contrast, until recently mesh compression schemes were inherently "non-streaming" [23, 5, 21, 7, 19, 17]. All these schemes construct auxiliary data structures to support connectivity queries for deterministically traversing and, consequently, globally reordering the mesh. Compressing a mesh larger than the available memory means either processing the mesh piece by piece [6] or employing complex external memory data structures [10]. This is IO-inefficient and requires lots of preprocessing and temporary disk storage.

Recent works [10, 12, 11, 14] advocate applying the same type of "streaming" that has long been used to compress, resample, or filter audio and video data for batch processing of geometric data sets. Since data that describes geometric models has no similarly "natural" stream order like the temporal succession of sound samples or movie frames, Isenburg and Lindstrom [11] describe a streaming mesh format that allows meshes to stream in *any* order that is reasonably coherent. Mesh elements (vertices, triangles, tetrahedra) appear interleaved in the stream, and vertices are active only as long as there are future references to them. Their last references are documented with explicit *finalization tags* in the format.

Knowledge about finalized vertices gives the necessary guarantees to complete local mesh operations, immediately produce output, and safely deallocate the corresponding data structures to make room for data still streaming in. This enables IO-efficient implementations of simple tasks like dereferencing, counting number of components/handles, or computing normals/gradients, independent of the size of the input mesh. More complex tasks such as simplification [12] or compression [14] of surface meshes, iso-surface extraction [20, 11], and simplification of volume meshes [24] have also been adapted to take advantage of streaming input.

In this paper we describe an algorithm for streaming compression of tetrahedral volume meshes. Given streaming mesh input, our compressor simultaneously reads the mesh, encodes incoming tetrahedra and vertices, and outputs a stream of bits, while dynamically allocating and freeing the data structures that represent mesh elements briefly resident in memory. For reasonably coherent input our connectivity compression rates are nearly independent from the order in which the tetrahedra arrive, so long as we reorder tetrahedra in a small *delay buffer*. With a delay buffer of 10,000 tetrahedra, we achieve connectivity compression rates within a factor of two of the state-of-the-art in-core method. Together with geometry compression that rivals those of others, our total bit rates are only 25 to 40 percent larger than those reported by Gumhold *et al.* [4] but obtained using drastically less memory and less than half the time.

2 PREVIOUS WORK

The standard indexed format for polygonal and polyhedral meshes uses an array of floats that specifies a position for each vertex in 3D (i.e. the geometry), and an array of integers containing indices into that vertex array that specifies the polygons or polyhedra (i.e. the connectivity). Volume meshes often also have simulation data like pressure or temperature values associated with each vertex.

Indexed formats are not the most concise way for storing a mesh; large models occupy files of challenging sizes. The connectivity typically dominates the overall storage costs—especially for tetrahedral volume meshes. While each vertex is referenced about 6 times in a triangular surface mesh, it has an average of 22 references in a tetrahedral volume mesh. Moreover, the cost for storing indexed connectivity increases superlinearly in the number of vertices v, as each index requires at least $\log_2 v$ bits.

Numerous schemes have been proposed for compressing polygonal surface meshes such as [23, 5, 21, 16, 7, 19, 17] and many more. For polyhedral volume meshes there are considerably fewer schemes [22, 4, 9, 1]. This is no surprise as surface meshes are widely used in entertainment and industry while volume meshes are mostly found in specialized scientific and engineering applications.

The challenge to compress the connectivity of tetrahedral volume meshes was first approached by Szymczak and Rossignac [22].

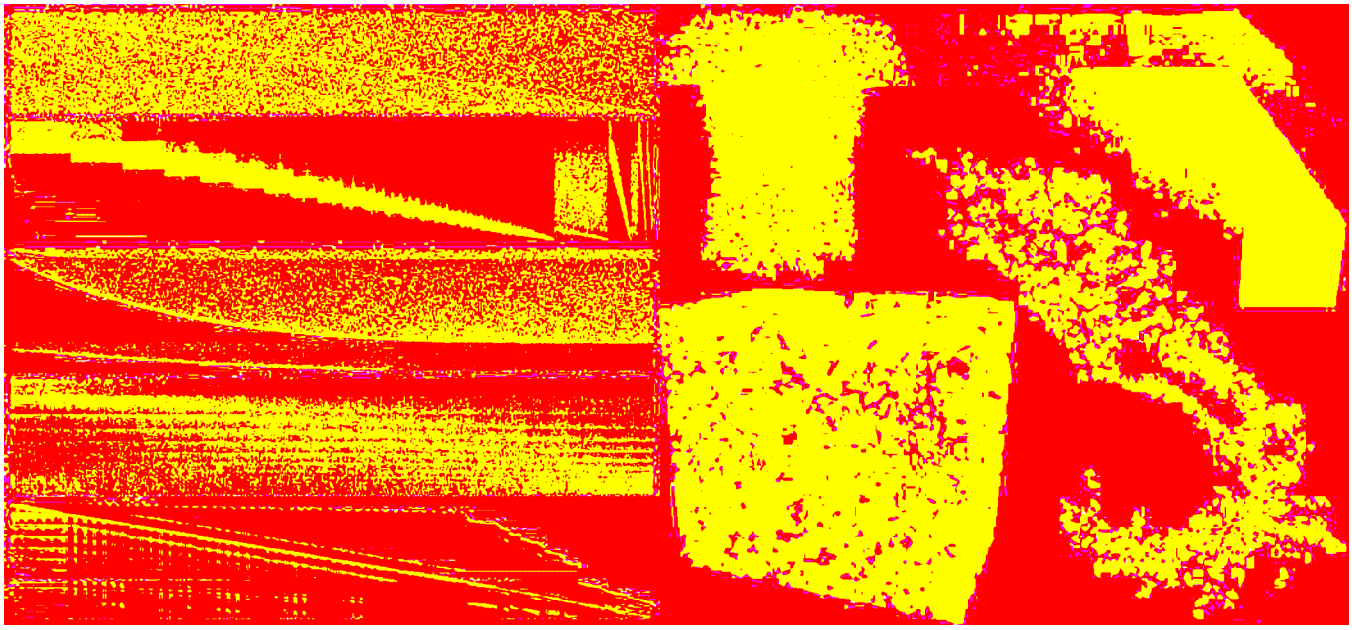


Figure 1: The layout diagrams on the left illustrate the original layout of the "torso," "fighter," "rbl," "f16," and "sf1" meshes (top to bottom). Vertices/tetrahedra are represented along the vertical/horizontal axis in the order they appear in the file. For each tetrahedron there is a point (violet) for each of its four vertices and a vertical line segment (gray) connecting them. For each vertex there is a horizontal line segment (green) connecting all the tetrahedra that reference it. The renderings on the right show the first two percent of tetrahedra in the file for the respective mesh.

Their "Grow&Fold" technique codes tetrahedral connectivity using slightly more than 7 bits per tetrahedron (bpt). The encoding process builds a tetrahedral spanning tree rooted in an arbitrary boundary triangle. This tree is encoded with 3 bpt that indicate for each face whether the spanning tree will continue growing. The boundary of the tetrahedron spanning tree, a triangular surface mesh, has an associated "folding string" that is represented with 4 bpt. This string describes how to "fold" and occasionally "glue" the boundary triangles of the spanning tree to reconstruct the original connectivity. The indices associated with the occasional "glue" operations lift the total bit rate slightly above 7 bpt.

Gumhold *et al.* have extended their connectivity coder for triangular surface meshes [5] to tetrahedral volume meshes [4]. Their algorithm performs a region growing process that maintains a "cut-border," a (possibly non-manifold) triangle surface mesh, that separates processed tetrahedra from unprocessed ones. Each iteration of the algorithm processes a triangle on the cut-border, either declaring it a "border" face or including its adjacent tetrahedron inside the cut-border. The latter operation specifies the fourth vertex of the tetrahedron. If this is not a "new vertex" it is a vertex from the cut-border, which is specified with a "connect" operation using an indexing scheme that is based on a local breadth-first traversal. Because of the order in which cut-border triangles are processed, this fourth vertex is often close to the processed triangle, and thus has a small local index. The bit rates they achieve for connectivity are around 2 bpt, a result that has not been challenged since.

Yang *et al.* propose a compression technique that allows pipelining decompression and rendering of a tetrahedral mesh [26], which can significantly reduce the memory requirements of a ray casting renderer. The contribution of a decompressed tetrahedron to the rays it intersects is incrementally composited and its memory is freed as soon as possible. This allows them to render compressed tetrahedral meshes without keeping the entire uncompressed mesh in memory. First, they encode the surface formed by the boundary triangles using a triangle mesh compression scheme. Then, they grow the boundary surface inwards using a breadth-first traversal. Like Gumhold *et al.* [4], they encode a tetrahedron by specifying its fourth vertex, but if the fourth vertex was already visited, they specify it using three different operations instead of Gumhold's universal "connect". Yang *et al.* use a local index into a list of adjacent faces or edges when the fourth vertex is connected across one of those faces or edges; otherwise, they use a global index into a list of all vertices in memory. Their connectivity compression rates are only slightly worse than those of Gumhold *et al.*

A similar scheme for the special case of back-to-front rendering of compressed Delaunay meshes was proposed by Bischoff and Rossignac [1]. A server reorders the tetrahedra in visibility order by sweeping a triangulated "sheet" through the mesh. Since Delaunay input is convex and without visibility cycles, the sheet of triangles remains a manifold and never changes topology. Only three different configurations arise when including a tetrahedron into the sheet, and only two need to be distinguished: the tetrahedron either shares one triangle with the sheet and "adds" a new vertex, shares two triangles and "flips" an edge, or shares three triangles and "fills" a gap. The server traverses the sheet triangles and includes tetrahedra batch-wise, alternating between only "adding" tetrahedra and only performing "flips" and "fills." The encoding is further compressed by testing edges on the sheet for convexity (which rules out a fill on that edge) and marking some edges as locked (which does the same). The connectivity encodings have entropies as low as 1.6 bpt, but can only represent Delaunay meshes.

None of the schemes just described is suited to compress gigabyte-sized input meshes. They expect an entire mesh as input; compression cannot start until the mesh is completely generated and stored at least once in uncompressed form. They construct temporary data structures as large as the mesh to support topological adjacency queries, requiring prohibitively large amounts of memory. They also completely reorder the input mesh, so gigabytes of data have to be globally reorganized. Hence, these schemes cannot easily compress large meshes with standard computing equipment.

Ho *et al.* [6] suggest cutting large triangle meshes into manageable pieces, encoding each separately using previous techniques, and recording how to stitch the pieces back together. Avoiding the cutting step, Isenburg and Gumhold [10] instead make use of a dedicated external memory data structure that supports the topological adjacency queries of their compressor. However, both these

approaches are highly I/O-inefficient and require large amounts of temporary disk space. Isenburg *et al.* [14] propose a radically different approach to mesh compression that incrementally encodes a triangle mesh as it is given to the compressor. This makes the compression process transparent to the user and almost independent of the mesh size. In this paper we extend their approach to similarly compress tetrahedral meshes.

3 Streaming Tetrahedral Volume Meshes

In a streaming mesh format [11], tetrahedra and the vertices they reference are stored in an interleaved fashion. This makes it possible to start operating on the data immediately without having to first load *all* the vertices, as is common practice with standard indexed formats. Furthermore, streaming formats provide explicit information about when vertices are referenced for the last time. This makes it possible to complete operations on these vertices and free the corresponding data structures for immediate reuse.

The *width* of a streaming mesh is the maximal number of vertices that need to be in memory simultaneously. Those are vertices that have already streamed in but have not been finalized yet. The width is the lower bound for the amount of memory needed for processing a streaming mesh since any mesh processing application has to store at least that many vertices simply to dereference the mesh.

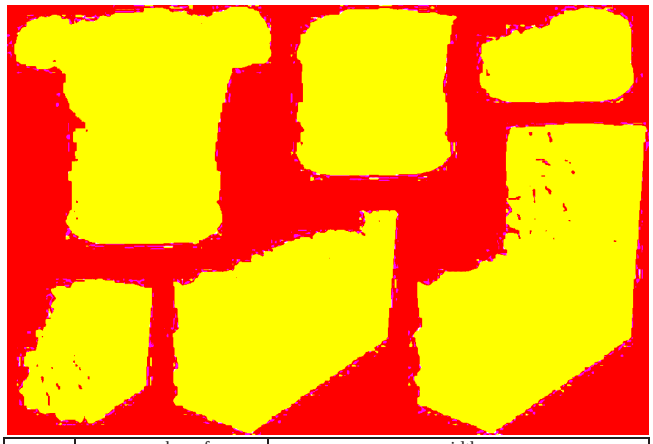
Our streaming approach to compression relies on the input meshes either being stored or produced in a streaming manner. The set of example volume meshes that we use to test our compressor, however, does not fulfill these expectations at all. Not only are these tetrahedral meshes distributed in conventional, non-streaming formats, they also come with absolutely "un-streamable" element orders, as illustrated by the *layout diagrams* [11] in Figure 1. The horizontal axis represents the tetrahedra (in the order they occur in the file), and the vertical axis represents the vertices (also echoing their order in the file). A layout diagram connects all the tetrahedra that reference a common vertex with a horizontal green line segment, and the four vertices of a tetrahedron with a vertical gray line segment. On top of this, the diagram represents each individual reference from a tetrahedron to its four vertices by a violet dot.

The few unclassified data sets that are currently used by the visualization community for performance measurements were created several years ago. Back then, the difficulty of using random access in-core algorithms for producing larger and larger meshes were overcome simply by employing sufficiently powerful computer equipment. But only when there is enough main memory to hold the entire mesh is it possible to output meshes whose vertices and tetrahedra are ordered so "randomly" in the file.

In the near future we anticipate a new generation of meshing algorithms that produces and outputs volume mesh data in a more coherent fashion. This is a necessity if algorithms are to scale to increasingly large data sets. An algorithm for tetrahedral mesh refinement, for example, could be designed to sweep over the data set and restrict its operation at any time to the currently active set until it achieves the desired element quality. For a mesh generation algorithm operating in this manner, it is natural to output reasonably coherent meshes in a streaming manner. To stream legacy data stored in non-streaming formats or with highly incoherent layouts, Isenburg and Lindstrom [11] describe several conversion strategies.

As streaming input for our compressor[1], we use reordered versions of the originally incoherent tetrahedral meshes that we obtained from Vo *et al.* [24] and that are shown in Figure 1. These meshes constitute a good representation of orderings that future layout-aware mesh generators may produce: two geometric orderings—a sweep along one axis and a space-filling curve—and a topological breadth-first ordering. For comparison we include

[1] for demo and software: http://www.cs.unc.edu/∼isenburg/sctvm/



mesh	number of		width			
	vertices	tetrahedra	original	axis	z-order	breadth
torso	168,930	1,082,723	168,909	3,118	7,253	5,528
fighter	256,614	1,403,504	208,338	3,894	9,197	16,629
rbl	730,273	3,886,728	594,107	2,814	10,075	3,206
f16	1,124,648	6,345,709	1,066,871	8,239	37,281	44,200
sf1	2,461,694	13,980,162	484,735	16,898	48,532	30,258

Table 1: Shown are "torso" and "fighter" in breadth-first and z-order. Listed are vertex and tetrahedron counts and the width of streaming tetrahedral meshes in four different orderings. Green bars show the width as a percentage.

results for streaming with the original tetrahedron order by *vertex-compacting* [11] the original mesh. This reorders the vertices (but not the tetrahedra) so they appear in the interleaved stream right before the first tetrahedron that references them. Due to their extremely high widths of up to 99.99 percent (see Table 1), some of these "streaming meshes" do not actually stream.

4 Connectivity Compression

Our connectivity compressor and decompressor use several strategies First, like Gumhold *et al.*, we maintain a cut-border, whose triangular structure is stored in a half-edge data structure. Second, we use the finalization information in the stream to discard vertices and half-edges from memory as soon as the demands of compression and decompression permit it, thereby keeping the memory footprint small. Third, we use a *dynamic indexing* method to reference the vertices in memory, so that the number of bits we use to index a vertex depends on the current width of the stream, not the total number of vertices in the mesh. Fourth, the compressor uses a small *delay buffer* to reorder tetrahedra, and whenever possible it compresses a tetrahedron that adjoins the previously compressed tetrahedron, which reduces the number of vertex indices we must write to the compressed stream. Fifth, we use short *local indices* to reference vertices when we can, resorting to the global dynamic indices only when necessary. If the vertices of the next tetrahedron to be compressed are already connected by edges of the cut-border, the tetrahedron's remaining vertices might be specified by local indices that index the cut-border's connectivity.

The compressor and decompressor maintain a set of *active vertices* that are connected by *active half-edges*. A vertex becomes active the moment it is (de)compressed, and remains active until it is finalized. Up to twelve half-edges become active in the moment a tetrahedron is (de)compressed. They are linked into units of three that describe oriented faces of a tetrahedron. They remain active either until an incident face of opposite orientation is (de)compressed or until one of their vertices is finalized. Each active half-edge stores a pointer to its origin vertex and a pointer to the next half-edge of the tetrahedral face it lives in. Each active vertex stores a list of pointers to the active half-edges for which the vertex is the origin. For some forms of predictive coding of vertex positions (see

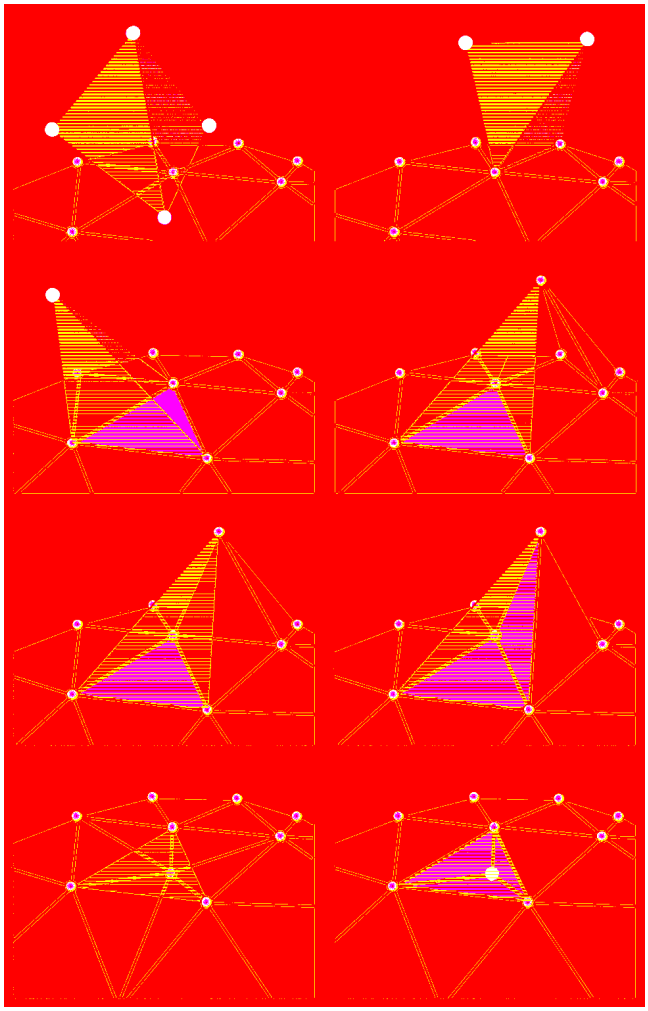


Figure 2: Possible configurations between a tetrahedron and active elements of the cut-border: (a,b,g) START tetrahedra are not adjacent to any face, but may be adjacent to up to four vertices; (c) ADD tetrahedra are adjacent to one face and introduce a new vertex; (d,e) JOIN tetrahedra are also adjacent to one face but do not introduce a new vertex; (f) FLIP tetrahedra are adjacent to two faces; (h) FILL tetrahedra are adjacent to three (or four) faces.

Section 5) each active half-edge also stores a copy of the position of the *tip* vertex—the non-incident fourth vertex of the tetrahedron that "created" the face containing this half-edge.

When we compress a tetrahedron, we first determine how many of its faces and vertices are already active. An arithmetic coder distinguishes the configurations illustrated in Figure 2 and encodes whether the tetrahedron is labeled START, ADD, JOIN, FLIP, or FILL (see the figure caption for the definitions of these labels). As there usually is correlation between successive configurations, we switch contexts based on the previous configuration.

Next, we reference some active vertex of the tetrahedron. If w is the current number of active vertices, this can be done with dynamic indexing [14] using $\log_2 w$ bits per vertex. As consecutive tetrahedra often share faces, edges, or vertices—especially after local reordering (see Section 4.1)—we first check whether a tetrahedron shares a mesh element with its predecessor and if so, we encode which one. This check often avoids the need to specify a global index, saving the $\log_2 w$ bits that are the most expensive part of our connectivity encoding. After specifying a vertex of the tetrahedron (or an edge or face shared with the previous tetrahedron), we can usually reach all other active vertices via an active half-edge. Only for some START and JOIN configurations we do have to specify several vertices. If a vertex is reachable via an active half-edge, we can reference it locally (using a shorter index) within the list of half-edges maintained by a known vertex or (even better) within the intersection of the lists of two already known vertices. Whenever such an intersection results in only one correctly oriented face we do not have to store any further information at all.

START. The tetrahedron may have up to four active vertices. We topologically rotate the tetrahedron so that active vertices come before new ones, and encode how many active vertices there are. Unless they are shared with the previous tetrahedron, we address them through dynamic indexing. New vertices are compressed with predictive coding (see Section 5) and added to the dynamic vector.

ADD. Ideally, the active face is shared with the previous tetrahedron and addressed as such. If not, the active face might share an edge or a vertex, which can be addressed as such. Otherwise we address one vertex through dynamic indexing. To address the remaining vertices we use local indices into the list of active half-edges of the known vertex or—if two vertices are already known—into an intersection of lists. The new vertex is compressed and added as before.

JOIN. Like an ADD, but we also need to address the fourth vertex. Sometimes we can do this locally, through one of the lists of half-edges maintained with the vertices of the active face. Otherwise we use dynamic indexing.

FLIP. One of the two faces is addressed like an ADD. The respective other face is specified locally within the intersection of half-edge lists maintained with the two vertices that are shared by both faces.

FILL. One of the three faces is addressed like an ADD. The other two faces are then addressed using half-edge lists of the known vertices. A FILL also corresponds to the case when all four faces are active. This does not need to be explicitly distinguished; it can be detected by the decoder.

Finalization is encoded by specifying for all four vertices of the currently processed tetrahedron whether they have now been referenced for the last time. These binary choices can be efficiently compressed with context-sensitive arithmetic coding. The context is chosen based on the total number of tetrahedra that have referenced this vertex so far, and on the current number of active half-edges around this vertex. As most vertices are finalized when they are surrounded by a closed star of tetrahedra, there is a strong correlation between the moment a vertex no longer has active half-edges and its finalization. Boundary vertices, which still have on average six active half-edges, tend to be surrounded by fewer tetrahedra.

4.1 Reordering in a Delay Buffer

When we compress the tetrahedra exactly in the order they stream in, we generally have to store at least $\log_2 w$ bits per tetrahedron as we need to reference at least one of its active vertices with dynamic indexing. With the same strategy as Isenburg *et al.* [14], we significantly improve compression rates by employing a small *delay buffer* from which the compressor can pick the next tetrahedron to encode. The compressor always looks for tetrahedra that share a face, an edge, or at least a vertex with the previous tetrahedron so that global indices having $\log_2 w$ bits can be avoided.

The compressor follows a greedy strategy that favors tetrahedra that are as connected as possible with the currently active elements, but also share as many vertices as possible with the previous tetrahedron. It picks the next tetrahedron from the delay buffer that

- finalizes a shared vertex;
- shares a face but does not introduce a new vertex;
- shares a face and introduces a new vertex;
- shares an edge but does not introduce new vertices;
- shares an edge and introduces one new vertex;
- shares a vertex but does not introduce new vertices;
- shares a vertex and introduces one new vertex;
- shares an edge and introduces two new vertices;
- shares a vertex and introduces two new vertices;
- shares a vertex and introduces three new vertices;
- is the oldest tetrahedron in the delay buffer.

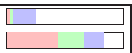
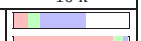
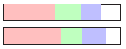
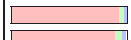
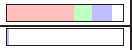
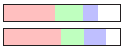
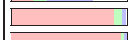
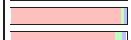
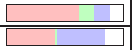
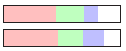
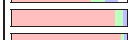
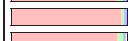
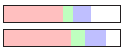
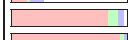
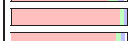
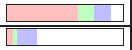
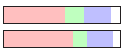
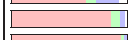
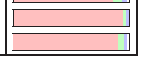
mesh	ordering	bit rate [bpt] 0	100	1 k	10 k	element sharing [%] 0	10 k
torso	original	27.9	25.1	24.8	22.3		
	axis	6.0	5.0	4.6	3.9		
	z-order	6.6	4.3	4.1	4.1		
	breadth	6.9	4.3	4.1	4.0		
fighter	original	28.2	27.7	25.3	18.3		
	axis	5.5	4.7	4.1	3.4		
	z-order	6.3	3.9	3.7	3.6		
	breadth	6.4	3.8	3.6	3.5		
rbl	original	13.2	9.2	9.1	8.9		
	axis	5.5	4.8	4.2	3.4		
	z-order	6.2	3.9	3.7	3.6		
	breadth	6.2	3.8	3.5	3.5		
f16	original	31.2	30.0	29.9	28.7		
	axis	8.6	5.0	4.7	3.7		
	z-order	7.9	4.1	3.8	3.7		
	breadth	6.8	4.0	3.8	3.7		
sf1	original	28.3	19.6	15.3	11.3		
	axis	5.9	3.7	3.7	3.4		
	z-order	5.7	3.7	3.5	3.5		
	breadth	6.7	3.9	3.6	3.5		

Table 2: Bit rates for streaming connectivity compression with different delay buffer sizes and the effect of a 10 k buffer on the percentage of subsequent tetrahedra that share a face, an edge, or at least a vertex.

4.2 Connectivity Compression Results

We measured the performance of our streaming tetrahedral connectivity compressor on five example meshes. We made measurements when streaming tetrahedra in their original incoherent order as well as in three different coherent orderings. In Table 2 we report bit rates for compressing tetrahedra exactly in stream order (with no delay buffer) and when using delay buffers of 100, 1,000, and 10,000 tetrahedra. The table also illustrates the increase in the percentage of tetrahedra that share a face, an edge, or at least a vertex when we use a delay buffer of 10,000 tetrahedra for reordering.

As one would expect, a delay buffer of size 10,000 is of little use for completely incoherent input orderings. However, on more coherent input the percentage of tetrahedra that share some mesh element approaches 100 percent and gives average connectivity rates of around 3 to 4 bpt for all three coherent orderings.

4.3 Streaming Compression without Finalization

Our compression scheme does not necessarily need streaming mesh input. We can also compress reasonably coherent indexed meshes (i.e. a block of vertices followed by a block of tetrahedra) or reasonably coherent interleaved meshes (i.e. streaming meshes without vertex finalization). This will obviously affect compression rates, as the costs for dynamic indexing increase when unfinalized vertices accumulate. More severe, however, is the impact on the memory, as all unfinalized vertices will eventually be in memory.

One approach to deal with unfinalized streaming input is to "guess" finalization, as proposed by Wu and Kobbelt [25]. They assume a vertex in a surface mesh is finalized when it is surrounded by a closed star of triangles. This technique can fail in the presence of a nonmanifold triangulation (e.g. where a dangling triangle shares a vertex that also has a complete ring of triangles). Moreover, a vertex does not get finalized when it is on the mesh boundary or when flipped triangles prevent the star of triangles from being closed.

The results in Table 3 show that the lack of explicit finalization has little impact on the bit rate. Local reordering in a 10,000 tetrahedron delay buffer eliminates most of the dynamic indices, which is the only coding item suffering from unfinalized vertices. The memory footprint, however, grows to the size of the input if vertices are not finalized. By guessing finalization, we limit the accumulating unfinalized data structures to boundary elements (and perhaps tetrahedra with the wrong topological orientation, and their neighbors). For meshes with proportionally large boundaries such as "rbl" and "f16," this entails a significant increase in memory footprint.

mesh	bit rate explicit	none	guessed	accumulating data structures vertices	%	edges	[MB]
torso	3.88	3.97	3.88	3,056	5.5	18,354	1
fighter	3.41	3.51	3.46	41,754	6.1	250,512	10
rbl	3.44	3.54	3.50	162,472	22.2	979,086	38
f16	3.71	3.92	3.83	154,976	13.8	929,808	36
sf1	3.44	3.60	3.52	242,259	9.8	1,453,542	56

Table 3: Bit rates for streaming connectivity compression of the axis-ordered meshes with a delay buffer size of 10,000 tetrahedra using explicit, none, and guessed finalization. For the latter we also report the total amount of unfinalized data structure that has accumulated on reaching the end of the stream.

5 Geometry Compression

When a vertex is encountered for the first time (i.e. the first tetrahedron to reference this vertex is encoded), we compress its position with a predictive coding scheme. Floating-point positions are quantized to a user-defined precision—for example, 16 bits per coordinate. This is reasonable as the 3D positions of a geometric data set usually have uniform precision within the bounding box and do not require the variable precision provided by the IEEE floating-point format. Furthermore, real-world data is generally much less precise than the 24 signed bits of uniform precision that are supported by single-precision floats. However, our coder also supports lossless floating-point compression [13] if needed.

Predictive coding methods use the positions of already known neighbor vertices to predict the position of the newly encountered vertex, and store only an offset vector that corrects this prediction. The absolute value of the correction vector tends to be less than that of the original position, so it can be stored with fewer bits.

A popular predictor for triangle meshes predicts vertices to complete a parallelogram spanned by three already known vertices of an adjacent triangle [23]. One approach for improving the success of the *parallelogram rule* first locates triangle pairs of parallelogram-like shape in the mesh and then directs the mesh traversal to include a maximal number of them [18]. However, this method significantly increases the overall complexity of the compression algorithm and does not easily extend to a streaming implementation.

For tetrahedral meshes, Gumhold *et al.* [4] use the midpoint (i.e. centroid) M of the known base triangle of a tetrahedron to predict the position of its tip vertex. They report that compression with their midpoint rule further improves when the correction vector is expressed in a local coordinate system that has one axis pointing along the base triangle's normal and whose origin is M. Chen *et al.* [2] suggest mirroring the tip vertex A of the already known tetrahedron from the other side of the base triangle through the midpoint M; so $2M - A$ is the prediction. They view it as a natural extension of the parallelogram rule [23], and find pairs of tetrahedra that give good predictions similar to those of Kronrod and Gotsman [18].

We found that the midpoint rule of Gumhold *et al.* and the mirror rule of Chen *et al.* both have shortcomings. By expressing the correction vectors in a local coordinate system, we separate their prediction error into a normal and a tangential component. By measuring these errors for both rules across our set of example meshes, we quickly see how they can be improved (see Table 4). The midpoint rule gives lower errors in the tangential direction, but has higher errors in the normal component. This is no surprise, since the midpoint rule does not predict anything in the normal direction. The mirror rule does a good job in predicting the normal component, but performs surprisingly poorly in the tangential direction.

This is because the tangential part of the parallelogram rule is based on a two-dimensional geometric regularity that does not extend to three dimensions. A regular triangular tiling of the plane turns every triangle pair into an equilateral parallelogram. An analogous regular tetrahedral tiling of three-dimensional space does not exist. Therefore, we cannot expect much correlation in the tangential positions of the tip vertices of two tetrahedra that share a base triangle. To mirror the tip vertex through the midpoint adds a tangential bias that has no geometric justification. Mirroring the tip

mesh	rule	prediction error			bit rate	
		total	normal	tangent	global	local
torso	midpoint	580	466	303	30.7	29.8
	mirror	508	151	467	31.2	30.9
	heightmirror	358	151	303	29.6	29.7
	baseheight	340	123	303	**29.4**	**29.3**
fighter	midpoint	256	233	96	26.1	24.2
	mirror	164	43	152	25.0	24.6
	heightmirror	110	43	96	23.4	23.3
	baseheight	107	36	96	**23.2**	**23.1**
rbl	midpoint	144	125	66	24.4	22.4
	mirror	99	16	95	23.9	21.3
	heightmirror	72	16	66	**22.6**	**20.8**
	baseheight	74	26	66	22.7	22.6
f16	midpoint	22	19	10	14.7	12.8
	mirror	16	4	15	14.0	13.6
	heightmirror	11	4	10	12.5	12.4
	baseheight	11	4	10	**12.3**	**12.1**
sf1	midpoint	269	220	145	**10.3**	17.5
	mirror	183	11	178	11.3	14.9
	heightmirror	151	11	145	13.5	**14.6**
	baseheight	168	69	145	14.7	17.7

Table 4: Prediction errors and bit rates for different prediction rules, with positions quantized to 16 bits and correctors in global and local coordinate systems. Values are averages over different orderings and delay buffer sizes.

vertex perpendicularly through the triangle would add a similar tangential bias. Either bias leads to a higher average tangential error than predicting without a bias by using the midpoint.

We combine the lower tangential error of the midpoint rule with the lower normal error of the mirror rule by mirroring only the normal component. In other words, we add to the midpoint a normal whose length is the height of the tetrahedron on the other side of the base triangle. We call this the heightmirror rule. We can also predict in normal direction without involving the tip vertex, which sometimes is not known. We estimate the normal component from the average edge length of the base triangle, or from its area. Our baseheight rule adds a normal whose length is 0.8 times the square-root of twice the base triangle's area. Found experimentally, this value is meaningful: it slightly under-predicts the height of an equilateral tetrahedron whose sides have the same area as the base triangle.

When compressing the vertices of a START tetrahedron (see Figure 2), we often do not have enough neighbor information to apply either of these prediction rules; then we fall back on a simpler prediction method like delta coding. As our reordering strategy tries to avoid START tetrahedra, few vertices are predicted this way.

Previous schemes break each coordinate of the corrector into smaller chunks and compress them separately. Gumhold *et al.* [4], for example, break their 16-bit correctors into four chunks of four bits. We use a scheme that first encodes the number k of nonzero bits in the corrector, and then only compresses those. Since most of the efficiency in predictive coding of vertex positions comes from eliminating the high-order bits [8], we could store the nonzero bits raw without losing much coding efficiency—unless the same corrector appears over and over again, as it does for one of our test meshes. To compromise between these possibilities, we employ a hybrid approach that compresses the highest eight nonzero bits with an entropy coder, and stores any remaining bits raw.

5.1 Geometry Compression Results

In Table 4 we compare the performance of the four predictive rules on our five test meshes when they code the correctors in either global or local coordinates. We report prediction errors and bit rates averaged over twenty-one compression runs—each of the three coherent orderings "axis", "z-order", and "breadth" compressed with seven different delay buffer sizes, ranging from 0 to 10,000.

The best results are usually obtained with the heightmirror or the baseheight rule. In agreement with Gumhold *et al.* [4], we find that for better compression it is best to express correction vectors in local coordinates. However, the improvement in bit rate when going from the global midpoint rule to the local heightmirror rule or the local baseheight rule is at most 15 percent. If robustness and ease of implementation are important, the global midpoint predictor may be the right choice. Unlike the mirror and heightmirror rules, it does not require access to the tip vertex. And while heightmirror, baseheight, and local coordinates all perform floating-point arithmetic, including square root computations, the global midpoint rule only adds three integers and divides them by three.

Despite having the highest prediction errors, the global midpoint rule achieves by far the best bit rates for the "sf1" data set. The vertex positions of this model are aligned on an adaptive octree grid and their precision is much lower than 16 bits, so that the same coordinate values appear over and over again. It is known that predictive coding is not the best choice for compressing such data sets [13]. Applying more complex prediction rules and expressing correctors in local coordinates destroys the regularity in the grid-aligned positions. The global midpoint rule causes, so to speak, the least damage. Instead, simply using some neighbor vertex as the prediction (i.e. delta coding) brings the average bit rate down to 7.8 bpv.

6 Streaming versus Non-Streaming

The trade-off between compression rates and IO-efficiency offered by streaming compression is best explored in an actual performance comparison with a conventional, non-streaming coder. We obtained an executable that implements the current state of the art in tetrahedral mesh compression from Gumhold *et al.* [4]. On our request, their software was recently improved for faster preprocessing and higher memory efficiency, so that a direct comparison with our software would be as meaningful as possible. We list in Table 5 bit rates, timings, and main memory use for conventional and streaming compressors. The streaming results are averages over the three coherent orderings when using a delay buffer of 10,000 and the heightmirror rule with correctors represented in local coordinates. Measurements were taken on a new Dell Inspiron 6000 laptop with an Intel Centrino 2.13 GHz Processor and 1 GB of main memory. The models were read from an external firewire drive and written to the local hard disk. Timings include reading and writing.

We should mention that the technique of coding with 16 bits as reported in [4] does not actually compress coordinates with 16 bits of precision. The authors quantize signed correctors that have twice the range as the original values into 16 bits. In doing so they effectively quantize their positions with 15 bits of precision. Furthermore they scale these 15 bits of precision along the bounding box diagonal, not along the longest side of the bounding box. For a fair comparison, we lower our level of quantization to match that of the conventional software. For the "torso," for example, setting the quantization to 16 bits with their software means quantizing the longest side of the bounding box with 22,481 different values.

While our compression rates for geometry are similar (or better due to height prediction) to those of Gumhold *et al.* [4], our compressed connectivity information takes almost twice as much space. But as the models get bigger, other merits of streaming compression start to outweigh the weaker compression. For the large "sf1" model, the conventional compressor needed almost the entire 1 GB of main memory and would start thrashing when other programs were running in parallel. After a fresh reboot it was able to compress the model without thrashing in 100 seconds using 645 MB of main memory. In contrast, our streaming compressor completed compression in 44 seconds using only 20 MB of memory.

For meshes too large to fit into main memory a conventional compressor can not be used. Recent techniques for streaming computation of 3D Delaunay triangulations [15], for example, make it possible to generate meshes containing several hundred million tetrahedra with relative ease. This translates into Gigabytes of generated mesh data that—assuming we could compress it with con-

mesh	conventional					streaming				
	conn [bpt]	geom [bpv]	total [KB]	time [sec]	mem [MB]	conn [bpt]	geom [bpv]	total [KB]	time [sec]	mem [MB]
torso	2.14	25.4	**807**	9	50	3.98	25.0	**1,043**	3.4	5
fighter	1.82	20.3	**947**	11	70	3.49	18.5	**1,177**	4.7	8
rbl	1.92	18.0	**2,514**	32	220	3.50	16.8	**3,162**	12	5
f16	1.95	10.5	**2,942**	53	313	3.75	8.8	**4,110**	21	18
sf1	1.72	19.8	**8,888**	100	645	3.48	13.5	**9,989**	44	20

Table 5: Comparing a state-of-the-art non-streaming compressor [4] with our streaming compressor. Processing times are in seconds, memory use is in MB.

ventional methods—would still have to be stored at least once in uncompressed form. Our streaming compressor makes it possible to write the output of a streaming Delaunay triangulator directly in compressed form to disk. For example, triangulating 30 million points generates 200 million tetrahedra resulting in 3.5 GB of streaming mesh output. Using less than 200 MB of memory we directly compress this mesh into a 220 MB file. This avoids the additional I/O and temporary disk space that would be needed if we first had to store this 3.5 GB mesh to disk before compressing it.

7 Conclusion

We have described the first streaming algorithm for compressing tetrahedral volume meshes. Our streaming compressor can create compressed meshes that are only 25 to 40 percent larger than the state of the art. For large meshes, the reduced effectiveness of compression is more than offset by higher processing speeds and drastically reduced memory footprints. Although our compressed "f16" is 4 MB compared to the 3 MB size achieved by Gumhold *et al.* [4], our streaming compressor only needs 21 seconds of processing time and 18 MB of memory, versus 53 seconds and 313 MB. Moreover, we can handle meshes of significantly larger size. A conventional encoder quickly exhausts the main memory and starts thrashing because its memory requirements grow with the size of the input mesh. Our compressor's memory footprint depends only on the width of the input stream and the delay buffer size.

Several components of our compressor have yet to be optimized. Timings and memory usage could be improved by integrating the delay buffer into the compressor. This buffer is implemented as a filter through which the mesh is piped before reaching the compressor. This allows modular implementation but requires extra computation time, a second hash table, and other duplicate data structures. Further speed-up is possible by improving our implementation for picking the next tetrahedron from the delay buffer, which currently consumes 30 percent or more of the total computation time.

Furthermore, we have shown that—contrary to what is reported by Gumhold *et al.* [4]—geometry compression rates can be improved further by adding a height component to the midpoint predictor. The extra component consistently lowers the prediction error across our set of test meshes and (usually) improves compression.

Conceptually, our algorithm operates in the same spirit as the streaming triangle mesh compressor of Isenburg *et al.* [14]. However, the extension from triangular surface meshes to tetrahedral volume meshes is not straightforward. When we began this work, it was not clear what connectivity compression rates would be possible, given the irregular nature of tetrahedral connectivity. This paper establishes the first benchmark in streaming compression for future research to measure itself against.

Acknowledgements

This research was supported in part by NSF grant CCF-0429901: "Collaborative Research: Fundamentals and Algorithms for Streaming Meshes," and in part by the Max Planck Center For Visual Computing and Communication in Saarbrücken, Germany. Parts of this work were performed under the auspices of the U.S. DOE by LLNL under contract no. W-7405-Eng-48. The "torso" data set is courtesy of the SCI Institute. The "fighter" data set comes from a wind tunnel model of a fighter jet and is courtesy of Neely and Batina from NASA. The "rbl" data set is a portion of an endoplasmic reticulum in a cell and courtesy of Alex Smith and Bridget Wilson from University of New Mexico and Jason Shepherd and Shawn Means of Sandia National Laboratory. The "f16" data set is courtesy of Udo Tremel from EADS-Military. The "sf1" data set is courtesy of the Quake Project at CMU.

References

[1] U. Bischoff and J. Rossignac. Tetstreamer: Compressed back-to-front transmission of delaunay tetrahedra meshes. In *Proceedings of Data Compression Conference*, pages 93–102, 2005.

[2] D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Geometry compression of tetrahedral meshes using optimized prediction. In *Proceedings of European Conference on Signal Processing*, 2005.

[3] M. Deering. Geometry compression. In *SIGGRAPH 95 Conference Proceesings*, pages 13–20, 1995.

[4] S. Gumhold, S. Guthe, and W. Strasser. Tetrahedral mesh compression with the cut-border machine. In *Visualization'99*, pages 51–58, 1999.

[5] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Proceedings*, pages 133–140, 1998.

[6] J. Ho, K. Lee, and D. Kriegman. Compressing large polygonal models. In *Visualization'01 Proceedings*, pages 357–362, 2001.

[7] M. Isenburg. Compressing polygon mesh connectivity with degree duality prediction. In *Graphics Interface'02 Proc.*, pages 161–170, 2002.

[8] M. Isenburg and P. Alliez. Compressing polygon mesh geometry with parallelogram prediction. In *Visualization'02*, pages 141–146, 2002.

[9] M. Isenburg and P. Alliez. Compressing hexahedral volume meshes. *Graphical Models*, 65(4):239–257, 2003.

[10] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *SIGGRAPH 2003 Proc.*, pages 935–942, 2003.

[11] M. Isenburg and P. Lindstrom. Streaming meshes. In *Visualization'05 Proceedings*, pages 231–238, 2005.

[12] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In *Visualization'03 Proceedings*, pages 465–472, 2003.

[13] M. Isenburg, P. Lindstrom, and J. Snoeyink. Lossless compression of predicted floating-point geometry. *JCAD - Journal for Computer-Aided Design*, 37(8):869–877, 2005.

[14] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. In *Proceedings of 3rd Symposium on Geometry Processing*, pages 111–118, 2005.

[15] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. *manuscript under review*, preprint available at http://www.cs.unc.edu/˜isenburg/sd/.

[16] M. Isenburg and J. Snoeyink. Face Fixer: Compressing polygon meshes with properties. In *SIGGRAPH'00 Proc.*, pages 263–270, 2000.

[17] F. Kälberer, K. Polthier, U. Reitebuch, and M. Wardetzky. Freelence - coding with free valences. In *Eurographics'05*, pages 469–478, 2005.

[18] B. Kronrod and C. Gotsman. Optimized compression of triangle mesh geometry using prediction trees. In *International Symposium on 3D Data Processing Visualization and Transmission*, pages 602–608, 2002.

[19] H. Lee, P. Alliez, and M. Desbrun. Angle-analyzer: A triangle-quad mesh codec. In *Eurographics'02 Proceedings*, pages 198–205, 2002.

[20] A. Mascarenhas, M. Isenburg, V. Pascucci, and J. Snoeyink. Encoding volumetric grids for streaming isosurface extraction. In *Proceeding of 2nd Symposium on 3D Data Processing, Visualization and Transmission*, pages 665–672, 2004.

[21] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. on Vis. and Computer Graph.*, 5(1):47–61, 1999.

[22] A. Szymczak and J. Rossignac. Grow & fold: Compression of tetrahedral meshes. In *ACM Solid Modeling and App.*, pages 54–64, 1999.

[23] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface'98 Proceedings*, pages 26–34, 1998.

[24] H. Vo, S. Callahan, P. Lindstrom, V. Pascucci, and C. Silva. Streaming simplification of tetrahedral meshes. Technical Report UCRL-CONF-208710, LLNL, 2005.

[25] J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes. In *Graphics Interface'03 Proc.*, pages 185–192, 2003.

[26] C. Yang, T. Mitra, and T. Chiueh. On-the-fly rendering of losslessly compressed irregular volume data. In *Visualization'00 Proceedings*, pages 101–108, 2000.