STREAMING COMPRESSION OF TRIANGLE MESHES

additional review material for "Streaming Meshes" (papers_0371)

Abstract

Standard schemes for compressing polygonal meshes first construct an explicit representation of the mesh connectivity and then traverse it with some deterministic strategy. This implicitly assumes that the compressor is allowed to reorder the mesh triangles as it sees fit. Hence, current mesh compressors take the entire mesh as input, construct temporary data structures in the size of the mesh, and produce a completely reordered compressed mesh.

In this report we describe a compression scheme that can preserve the original triangle order of a mesh. It incrementally constructs mesh connectivity as triangles are input and immediately compresses them. This compression scheme is mainly intended for meshes in streaming formats, *which interleave vertices and triangles and finalize vertices that are no longer used. The availability of this scheme enables us to compress such meshes* on-the-fly *without ever having the entire mesh in memory and without ever having to reload parts of the mesh as required by current alternatives.*

We can often significantly improve compression rates by employing a small delay buffer *within which the compressor is allowed to locally reorder triangles. The memory requirements of the compressor are proportional to the width of the streaming mesh and the size of the delay buffer.*

1 Introduction

Traditional indexed formats for storing 3D triangle meshes use an array of floats triplets to specify the vertex positions (i.e. the geometry) and an array of integers triplets that index into the vertex array to specify the triangles (i.e. the connectivity). The cost for storing mesh connectivity in this format increases super-linearly with the number of vertices v as the number of bits required for each index is at least $\log_2(v)$. Because vertices are indexed an average of six times, the overall storage costs for indexed connectivity is around $6\log_2(v)$ bits per vertex. In contrast, current connectivity coders [10, 5, 9] can encode mesh connectivity with a constant 1 to 4 bits per vertex.

An indexed format not only specifies the connectivity but also the particular order in which vertices and triangle appear in their respective array. For a mesh with t triangles and v vertices there are $t!$ possible ways in which the triangles can be arranged and $v!$ possible ways to permute the vertex array. Any triangle arrangement can be combined with any vertex permutation leading to $t! \cdot v!$ different descriptions for the same mesh. The $\log$-factor in the storage costs comes from specifying one of these descriptions.

Connectivity compression schemes [10, 5, 9] completely disregard the original triangle and vertex order of the mesh they compress. They encode the triangles in an order that is derived from systematically traversing the connectivity of the mesh and they encode the 3D positions that are associated with the vertices in the order they are first encountered during this traversal. This means that the element ordering of a compressed mesh is dictated by the particular compression scheme used. This is usually acceptable as the order with which triangles and vertices are listed makes no difference to the geometric shape that the mesh describes. However, in some applications such as rendering and streaming it is necessary to keep the triangles in their original order.

Modern graphics cards keep a few recently processed vertices in a local cache to reduce the number of times that a vertex needs to be fetched from memory. A mesh can be rendered more efficient when triangles frequently use vertices from the cache. Several heuristics for reordering the triangles to maximize the efficiency of the cache have been proposed [4, 7]. To compress meshes without affecting rendering efficiency we need a compressor that can compress the mesh triangles in their particular cache-efficient order

Streaming formats [2] provide simultaneous access to vertices and triangles and "finalize" vertices that are no longer used by subsequent triangles. This information allows efficient parsing and processing of large meshes that can not be entirely loaded into memory as one can safely complete operations on finalized vertices and deallocate their associated data structure. Because we want to compress streaming meshes *on-the-fly* we need a compression scheme that can process the mesh triangle in stream order.

In this report we describe a streaming compression scheme that is fundamentally different from all previously proposed schemes. On one hand it can encode the triangles of a mesh in whatever order they happen to be in. But more importantly, it can do this without any preprocessing and while using only minimal memory resources if the input mesh is either stored in a streaming format or produced in a streaming fashion. We have implemented a *stream-*

ing mesh writer and a corresponding *reader* through which compressed streaming meshes can be written and read in increments of single vertices and triangles.

The main advantage of our streaming compression scheme over previous approaches is the elimination of the need of first having to create a mesh representation that provides full access to the mesh. This is especially beneficial for compression large meshes. Previous approaches spend significant amounts of main memory, temporary disk space, CPU time, and file IO on either cutting the mesh in smaller pieces, as suggested by Ho et al. [6] or with constructing complex external memory data structures, as proposed by Isenburg and Gumhold [8] before the compression process can even start. Our streaming mesh writers and reader make the compression and decompression of even the largest meshes basically transparent to the user.

The main disadvantage is that the achieved connectivity compression is no longer guaranteed to be linear in the number of vertices. The $\log$-factor of the indexed format gets introduced by the fact that our streaming compressor can encode triangles in any order. The achieved compression rates vary drastically with different triangle orders and "random" orderings give result poor result. To strictly follow the original triangle order prevents our compressor from using any kind of determinism to improve compression rates.

However, since all previous compression schemes are allowed to completely reorder the triangles of the mesh, it seems only fair if we allow our compressor at least to do some local reordering. We give our compressor a small *delay buffer* within which it can perform triangle reorderings that do not affect the overall stream quality. This can significantly improves the compression rates—bringing them close to those reported by other coders.

2 Efficient Indexing

Even without compression, streaming mesh formats allow more efficient storage of indexed connectivity than standard indexed formats. Finalization of vertices tells us that they are no longer referenced by subsequent triangles, which effectively frees up their indices to be used again. Instead of using a unique absolute index for every vertex of the mesh we only need to assure that every *active* vertex has a unique index. This lowers the number of indices between which we need to distinguish and thereby reduces the number of bits needed to represent them.

Relative Indexing references a vertex as the difference of the currently highest index to the absolute index of that vertex. Each index can then be stored with $\log_2(span)$ bits since the maximal index difference equals the span of the streaming mesh. But we can also use the current instead of the maximal span, which can easily be kept track of. Then the total storage costs for all indices is an integral of the span's logarithm over the stream. Especially for low-span meshes this type of indexing is beneficial as it allows implementing efficient vertex access through a ring buffer.

mesh	bits per index			largest index range	
(ordering)	abs	rel	opt	rel (span)	opt (width)
armadillo					
(vcompact)	17.4	17.4	16.1	172,715	51,951
(spectral)	17.4	10.6	9.3	4,405	638
(geometric)	17.4	11.0	10.0	3,796	1,042
(breadth)	17.4	10.2	10.1	1,197	1,129
(depth)	17.4	16.9	10.3	171,845	1,457
dragon					
(vcompact)	18.7	15.4	12.5	54,825	4,586
(spectral)	18.7	11.7	9.5	11,617	668
(geometric)	18.7	12.9	10.7	9,243	1,274
(breadth)	18.7	11.1	10.8	1,994	1,671
(depth)	18.7	18.4	12.8	436,834	8,587
lucy					
(vcompact)	23.7	23.3	18.4	13,500,197	255,446
(spectral)	23.7	16.1	12.8	200,237	5,841
(geometric)	23.7	13.6	12.8	20,362	4,985
(breadth)	23.7	13.0	12.9	6,572	5,921
(depth)	23.7	23.2	13.6	12,428,107	12,914
david$_{1mm}$					
(vcompact)	24.7	23.8	14.8	15,821,388	26,383
(spectral)	24.7	17.2	13.2	752,345	7,862
(geometric)	24.7	13.5	13.1	36,421	8,919
st. matthew					
(vcompact)	27.5	25.2	15.2	29,189,836	31,931
(spectral)	27.5	20.7	15.6	3,850,470	33,029
(geometric)	27.5	15.2	14.8	157,920	33,207
ppm					
(vcompact)	27.8	19.1	18.6	462,634	311,280
(spectral)	27.8	21.4	14.4	27,019,863	56,179
(geometric)	27.8	17.7	17.2	290,591	114,593

Table 1. Average number of bits per index required for absolute, relative, and optimal indexing in differently ordered streaming meshes. Absolute indexing always uses a fixed $\log_2(v)$ bits for meshes with v vertices. Relative and optimal indexing use a varying $\log_2(s(t_i))$ and $\log_2(w(t_i))$ bits for the indices of triangle t_i with $s(t_i)$ and $w(t_i)$ being the span and width at moment i. Also reported are the largest index ranges for relative and optimal indexing, which correspond to span and width.

Optimal Indexing references each of the w active vertices with an index that is within the range 0 to $w-1$. Each index can then be stored with $\log_2(width)$ bits as the maximal number of active vertices equals the width of the streaming mesh. Again, using the current width makes the total storage costs an integral of the width's logarithm over the stream. This type of indexing requires a more involved data structure that re-assigns indices as vertices get finalized.

Obviously relative indexing will always be at least as expensive as optimal indexing with equality being reached only if vertices are introduced and finalized in the same or-

```
class SMwriter_smc {
  // specifies optional quantization
  bool open(FILE* file, int bits);

  // may optionally be set if known in advance
  void set_bounding_box(float* min, float* max);
  void set_num_verts(int nverts);
  void set_num_faces(int nfaces);

  bool write_vertex(float* v_pos);
  // finalize indices used for the last time
  bool write_triangle(int* t_idx, bool* t_final);

  bool close();
}

typedef Type enum {SM_VERTEX, SM_TRIANGLE};
```

```
class SMreader_smc {
  int bits;
  // only optionally known
  float *bb_min, *bb_max;
  int nverts, nfaces;

  bool open(FILE* file);
  Type read_element();
  bool close();

  // position of read vertex
  float* v_pos;
  // indices of read triangle ...
  int* t_idx;
  // ...  and their finalization
  bool* t_final;
}
```

Figure 1. Our current API for reading and writing compressed streaming meshes.

der. When comparing relative and optimal indexing with standard indexing we must add one bit per index, which is needed to specify finalization. These indexing costs are listed in Table 1 for streaming meshes in different orders.

The storage requirements for optimal indexing are in general still super-linear. Even if we reorder the mesh to minimize its width, we are subject to the worst-case bound of $O(\sqrt{v})$ for a mesh with v vertices [3]. Hence, the best that could theoretically be guaranteed is $O(v \log_2(\sqrt{v}))$ bits per vertex. In the next section we describe a compression scheme that often requires only one optimal index per triangle by keeping track of how already encoded triangles connect active vertices. Furthermore it avoids this explicit index whenever subsequent triangles reference the same vertex. While the coding costs are still $O(v \log_2(width))$ in the worst case, it does much better in practice.

3 Streaming Compression

We depart from the traditional approach to mesh compression, which traverse meshes in some deterministic manner, and use a scheme that encodes the triangles of the mesh in the particular order they are given to the compressor. Since such a scheme not only encodes the connectivity but also a particular triangle ordering, it can not achieve the same compression rates as traditional schemes. However, being able to immediately compress a mesh as it is written to disk or piped across a network makes this scheme much more usable in a typical mesh processing pipeline.

Previous schemes require loading the complete mesh and constructing a representation that allows traversing its connectivity graph—before the compression process can even begin. To do this for large meshes they need to either cut the mesh into smaller pieces as suggested by Ho et al. [6] or build complex external memory data structures as proposed by Isenburg and Gumhold [8]. We now have a streaming mesh writers and corresponding readers through which on-the-fly compressed meshes can be written and read in increments of single vertices and triangles (see Figure 1).

For reasons of efficiency our compressor only writes vertex-compact pre-order meshes. Vertex positions are encoded in the moment their vertex is referenced for the first time and finalization is encoded in the moment vertices are referenced for the last time. Post-order meshes or meshes that do not immediately finalize vertices are piped through a filter that converts them on the fly. Although the input does not need to be vertex-compact, it is compressed in a vertex-compact manner. When a vertex is written it is simply inserted into a hash using its index as key. When a triangle is written its vertices are looked up in the hash and only then actual compression takes place. This delays out-of-order vertices until they are referenced by a triangle.

The compressor maintains a set of *active vertices* and a set of *active half-edges*. Vertices are active if they have been referenced by previously written triangles but have not yet been finalized. Half-edges are active if they are part of a previously written triangle that connects two still active vertices and if their counterpart of opposite orientation has either not yet appeared or does not exist. Each active vertex has a list of all its incident active half-edges.

When a triangle is written the compressor checks which of the triangle's vertices and which of the triangle's half-edges of opposite orientation are already active. There are eight different configurations that can arise, namely *start*$_0$, *start*$_1$, *start*$_2$, *start*$_3$, *add*, *join*, *fill*, and *end*, which are illustrated in Figure 2. The compressor encodes the configuration of the triangle with an arithmetic using four different symbols: START, ADD, JOIN, FILL_END. For reasons of efficiency it uses only one symbol for all four *start*$_x$ configurations as they are typically of infrequent occurrence. The x is subsequently compressed with a separate context. The *fill* and *end* configurations only need one symbol because they can be distinguished at the decoding end.

Then the active vertices of the written triangle are referenced (unless it is in *start*$_0$ configuration). This can be done with optimal indexing using $\log_2(w)$ bits per vertex, where w is the current number of active vertices. But be-

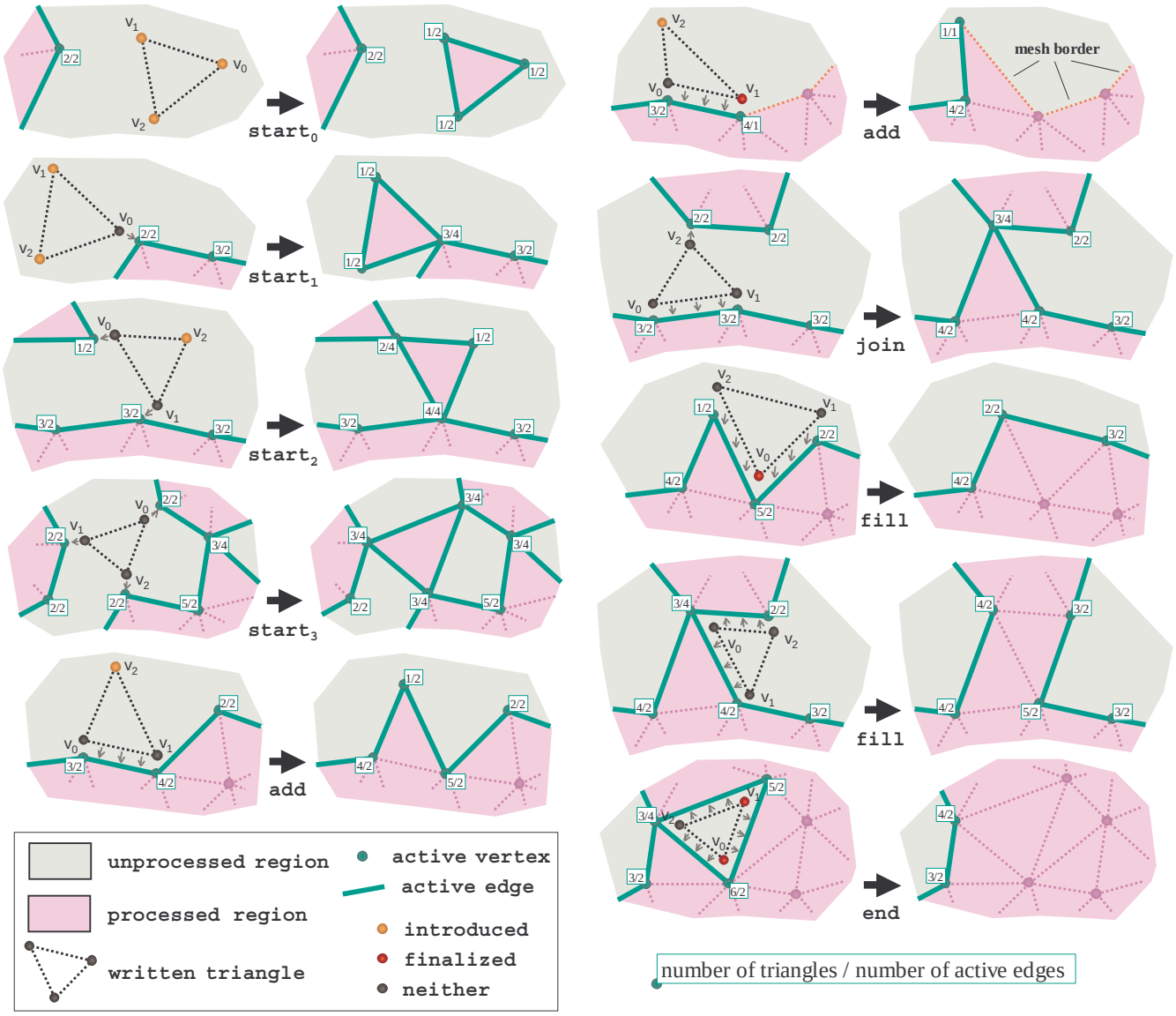

Figure 2. The eight possible adjacency configurations between the written triangle and the active vertices and half-edges maintained by the compressor: a $start_x$ triangle is not adjacent to any active half-edge, but may be adjacent to zero, one, two, or even three active vertices; an *add* triangle is only adjacent to one active half-edge with the third vertex being newly introduced; for the similar *join* configuration this third vertex is already active; a *fill* triangles is adjacent to two half-edges and an *end* triangle is adjacent to three half-edges. The small boxes show the triangle count and number of active half-edges.

cause subsequently written triangles often share vertices we first check whether a vertex was already used by the previous triangle and if so, encode which of the three vertices it was. This often saves us those $\log_2(w)$ bits that are the single most expensive part of our connectivity encoding.

In case of an *add*, *join*, *fill*, or *end* configuration the current triangle is also adjacent to one or more active half-edges. After having referenced the first active vertex (either with $\log_2(w)$ bits or as a vertex from the previous triangle) we can now reference other active vertices using the list of half-edges maintained with this already referenced vertex. Since this list contains usually only one half-edge with the correct orientation, we often avoid storing any further information. Only vertex v_2 of a *join* configuration can not be referenced this way, which makes a *join* configuration the most expensive configuration to encode.

In case of an *add* configuration we predict the position of the newly introduced vertex with the parallelogram rule [10] For this we store the position of the *across* vertex with each active half-edge. This is the vertex that is neither origin nor target of the half-edge but the third vertex of the triangle that created this half-edge. In case of a $start_x$ configuration we

predict the positions of newly introduced vertices as that of a known neighboring vertex. For the first vertex of a *start*$_0$ configuration there is no known neighbor. Here we simply use the most recent vertex that was compressed as the prediction. We compress the corrective vectors with different arithmetic contexts depending on whether a parallelogram, a neighbor, or a most recent prediction was used.

Finalization information is encoded by specifying for all three vertices whether the current triangle finalizes them or not. These binary flags can be efficiently compressed with context-sensitive arithmetic coding. The context is chosen based on the current triangle count and the number of active half-edges around this vertex. As most vertices are finalized when they are surrounded by a closed ring of triangles there is a strong correlation between the moment a vertex no longer has active half-edges and its finalization. Border vertices, which will still have one or two half-edges tend to be surrounded by a smaller number of triangles.

Reading the pseudo-code of Figure 3 may further clarify the compression algorithm that we have just described.

The vertices are maintained in two data structures: a hash table and a dynamic vector. The hash table is used to look up vertices by their index. A vertex is added to the hash when it is written, it is looked up when a triangle is written that references it, and it is removed from the hash when it is finalized. The dynamic vector is used to address previously encoded vertices with an index between 0 and $w-1$. A vertex is added to the dynamic vector when the triangle that references it for the first time is written. Subsequently the encoder looks up the index for a vertex in the dynamic vector whenever it needs to encode an explicit reference to it. These indices are then compressed with $\log_2(w)$ bits. The dynamic vector implements constant time insertion and removal of vertices and constant time lookup for vertex indices simply by moving the last entry to a deleted position. This means that the indices with which vertices are addressed in the dynamic vector change over time, but they do so in a consistent manner at both encoder and decoder.

3.1 Bounding-box less quantization

To support quantization of floating-point geometry for streaming meshes whose bounding box is not known in advance, we use a scheme that quantizes conservatively using a bounding box that is learned as the mesh streams by. The first two vertex positions are compressed without quantization and their distance gives us the initial conservative guess on the number of mantissa bits that need to be preserved to guarantee the user requested precision. This maximum distance is updated with every compressed vertex position and will eventually match the extent of the actual bounding box. Hence, how long we quantize overly conservative depends on the order in which the vertex positions arrive.

We perform predictions in floating-point and encode separate correctors for sign, exponent, and mantissa. For compressing the mantissa, we switch between multiple arithmetic contexts as the maximal range of the correctors varies with the exponent. The scheme is part of our current prototype and works well in practice. However, we still need to analyze its average performance and optimize its compres-

```
bool write_vertex(float* v_pos) {
  Vertex* v = allocVertex(v_pos);
  hash->insert(v, v_count);
  v_count++;
}

bool write_triangle(int* t_idx, bool* t_final) {
  Vertex* v[3];
  for (i = 0; i < 3; i++) {
    v[i] = hash->get(t_idx[i]);
    if (v[i] == 0) return false; // must be pre-order
  }
  determine and compress configuration;
  rotate triangle so that vertex v0 of Figure 2 is in v[0];
  if (START_x configuration) {
    compress which START_x it is;
    for (i = 0; i < 3; i++) {
      if (x--) {            // v[i] is an old vertex
        if (v[i] is used by previous triangle) {
          compress which of its vertices is v[i];
        } else {
          index = dvector->get_index(v[i]);
          compress explicit index of v[i];
        }
      } else {              // v[i] is a new vertex
        dvector->add(v[i]);
        compress position of v[i];
      }
    }
    create three new half-edges;
  } else if (ADD or JOIN configuration) {
    if (v[0] or v[1] is used by previous triangle) {
      compress which of its vertices is v[0] or v[1];
    } else {
      index = dvector->get_index(v[0]);
      compress explicit index of v[0];
    }
    compress which half-edge of v[0] leads to v[1];
    if (ADD configuration) {
      dvector->add(v[2]);
      compress position of v[2];
    } else {
      index = dvector->get_index(v[2]);
      compress explicit index of v[2];
    }
    create two new and delete one old half-edges;
  } else if (FILL or END configuration) {
    if (v[0] or v[1] or v[2] is used by previous triangle) {
      compress which of its vertices is v[0] or v[1] or v[2];
    } else {
      index = dvector->get_index(v[0]);
      compress explicit index of v[0];
    }
    compress which half-edge of v[0] leads to v[1];
    compress which half-edge of v[0] leads to v[2];
    create new and delete old half-edges;
  }
  for (i = 0; i < 3; i++) {
    compress whether v[i] is finalized
    if (t_final[i]) {
      dvector->remove(v[i]);
      hash->erase(v[i]);
      delete all half-edges of v[i];
      deallocVertex(v[i]);
    }
  }
}
```

Figure 3. An implementation of the SMC compression algorithm (without the delay buffer) in C-like pseudo code.

mesh (ordering)	configurations [%] s	a	j	f	e	use [%]	details for conn [bpv] con	pre	exp	adj	fin	totals [bpv] conn	geom	time [sec]	mem [MB]
armadillo															
(vcompact)	20	15	15	30	20	**10**	4.2	1.3	32.7	1.1	.00	**39.35**	22.24	1.4	9.3
(spectral)	.0	50	.9	48	.5	**49**	1.3	1.1	8.5	.04	.01	**11.04**	19.15	.8	.8
(geometric)	.7	49	4.3	42	4.3	**51**	2.0	2.0	9.2	.14	.01	**13.26**	19.16	.8	.8
(breadth)	.0	50	1.4	47	1.4	**97**	1.8	0.9	0.6	.01	.01	**3.27**	19.14	.7	.9
(depth)	.0	50	2.5	45	2.5	**97**	2.0	0.1	0.5	.03	.01	**2.66**	19.17	.7	1.0
(rendering)	.5	49	4.7	41	4.6	**91**	2.5	3.1	1.7	.14	.01	**7.45**	19.35	.7	.8
dragon															
(vcompact)	15	24	9.6	37	14	**50**	3.9	3.9	13.0	.83	.04	**21.62**	21.49	2.6	1.6
(spectral)	.4	50	1.5	47	1.5	**49**	1.8	1.2	8.8	.07	.04	**12.02**	22.04	1.8	.8
(geometric)	1.1	48	3.7	43	3.8	**57**	2.2	2.4	8.7	.14	.04	**13.45**	21.86	1.8	.9
(breadth)	.0	50	3.1	44	3.1	**94**	2.1	1.8	1.3	.05	.04	**5.23**	21.98	1.8	.9
(depth)	.0	50	5.1	40	5.1	**94**	2.3	0.5	1.4	.09	.04	**4.35**	21.96	1.9	1.8
(rendering)	.6	49	5.1	40	4.9	**90**	2.5	3.1	1.8	.15	.04	**7.62**	22.13	1.8	.8
lucy															
(vcompact)	1.6	47	8.1	34	8.7	**77**	2.5	2.2	8.6	.35	.00	**13.60**	14.70	77	37
(spectral)	7.2	35	6.9	45	6.3	**70**	3.1	3.1	7.4	.44	.00	**14.09**	15.47	65	1.5
(geometric)	.5	49	2.1	46	2.1	**53**	1.9	1.9	11.3	.07	.00	**15.18**	14.58	65	1.6
(breadth)	.0	50	2.3	45	2.3	**96**	2.0	1.3	1.1	.06	.00	**3.51**	14.54	59	1.6
(depth)	.0	50	3.8	42	3.8	**96**	2.1	0.2	1.0	.05	.00	**3.32**	14.55	61	2.7
david$_{1mm}$															
(vcompact)	12	28	5.8	44	9.4	**66**	2.8	2.6	9.9	.60	.02	**15.94**	10.95	126	4.8
(spectral)	7.4	35	7.1	45	6.3	**71**	3.1	3.1	7.6	.45	.02	**14.30**	11.94	126	1.8
(geometric)	.8	49	2.0	47	2.0	**67**	1.9	2.8	8.0	.07	.02	**12.71**	11.63	131	2.4
st. matthew															
(vcompact)	11	31	6.2	44	8.5	**67**	2.7	2.4	10.0	.53	.02	**15.62**	8.22	865	5.2
(spectral)	7.5	34	7.2	45	6.2	**71**	3.2	3.1	8.9	.46	.02	**15.60**	9.44	837	7.4
(geometric)	.9	48	2.2	46	2.3	**69**	1.9	2.9	8.6	.08	.02	**13.60**	8.97	907	4.0
ppm															
(vcompact)	9.4	31	8.4	43	8.1	**65**	3.2	3.2	12.7	.53	.01	**19.64**	14.91	1200	57
(spectral)	7.4	34	7.2	45	6.3	**71**	3.2	3.1	8.1	.46	.01	**14.88**	15.76	1030	9.1
(geometric)	1.2	47	2.7	46	2.6	**63**	2.1	2.5	12.2	.11	.01	**16.84**	16.05	1310	18

ooc compressor [8]		
[bpv]	time	mem
conn	prepro.	disk
geom	compr.	main
1.88	19 min	0.9 GB
14.60	5 min	128 MB
1.79	36 min	1.7 GB
11.32	14 min	192 MB
1.84	7 hrs	11 GB
8.83	4 hrs	384 MB

Table 2. For compressing in stream order we report the percentages of *s*tart, *a*dd, *j*oin, *f*ill, and *e*nd configurations and of subsequent triangles that re-*use* vertices. We give itemized coding costs for triangle *con*figuration, *pre*vious and *exp*licit references, edge *adj*acency, and vertex *fin*alization. Total bit-rates for *conn*ectivity and *geom*etry (quantized at 16 bits) and both *time* and *mem*ory footprint for reading, compressing, and writing the meshes on a 1.1 GHz Dell Inspiron laptop are listed.

sion speed. Our streaming mesh writer also supports absolutely lossless floating-point compression [1]. This is less efficient since the low-order bits of the mantissa typically contain noise that is hard to compress. But providing this functionality makes it possible to use compression when quantization—for whichever reason—is not an option.

3.2 Results

Detailed performance measurements of our SMC compressor for streaming meshes with different triangle orderings are listed in Table 2. The *compact* meshes have the original triangle order of the mesh. The *geometric* meshes have spatially sorted triangles and the *spectral* meshes are sorted in an attempt to minimize the width [2]. The *breadth* meshes are a triangle-compacted breadth-first ordering of the vertices, the *depth* meshes have a depth-first triangle ordering, and the *rendering* meshes are generated with the recursive cut algorithm of Bogomjakov and Gotsman [4].

As anticipated, our connectivity compression rates are much higher than those of previous compressors [10, 5, 9], whereas the achieved geometry compression rates are similar. The most expensive item for connectivity is the encoding of explicit indices. We need to encode many of those when the re-use of vertices among subsequent triangles is low, which results in poor compression rates. Re-use is especially low if triangles appear somewhat "randomly" such as in the compact orderings but also if they "hop around" along the advancing front such as in the spectral and geometric orderings. The high percentage of *start* and *join* configurations in the spectral orderings is due to the clustering and the fact that triangles within a cluster are left in a "random" order. The topological breadth- or depth-first sorts derive the triangle order from the local adjacency of

mesh elements, which gives a high percentage of vertex re-use and therefore the best compression rates.

The biggest selling points of the SMC compressor are the dazzling speed and the tiny memory footprint with which it can compress even the largest models. The only competing approach that can compress models as large as, for example, the "St. Matthew" statue is the ooc-compressor by Isenburg and Gumhold [8], for which we report compression rates and time and memory consumption in Table 2. Running on a 2.8 GHz Pentium IV processor they first spent 7 hours creating an 11 gigabyte data structure on disk before the actual compression starts that then takes another 4 hours while using 384 megabytes of main memory. In contrast, running on a 1.1 GHz mobile Pentium III we complete compression after 15 minutes while using only 6 megabyte of main memory and no temporary disk space. However, their 11 hour and 11 gigabyte effort pays off with a state-of-the-art connectivity compression rate that leave ours in the dust. But so far we have not reordered a single triangle.

4 Reordering in a Delay Buffer

When the compressor is forced to follow the exact triangle order in which the streaming mesh is written, it can not avoid storing those $\log_2(w)$ bits whenever a triangle does not share a vertex with the previous. We can easily construct a triangle sequence where no subsequent pair of triangle uses the same vertex. However, if we allow the compressor to locally reorder triangles with a simple strategy for increasing the vertex re-use among subsequent triangle we can significantly improve the compression rates.

We give the SMC compressor the option to store a user-specified number of triangles in a *delay buffer* from which it can choose any triangle for output. To globally preserve the stream order we place a strict constraint on the maximum delay (and the maximum rush) with which a triangle can leave the buffer. Setting this delay to be the size of the buffer allows us to efficiently implement both the buffering as well as the delay control using a simple ring buffer.

The SMC compressor uses a greedy strategy that picks the next triangle with the following priority order: First, are triangles that share two vertices with the previous triangle. Ties between candidates are broken by using the triangle whose shared vertex has the lowest index. Second, are triangles that share only one vertex but also finalize some vertex. Third, are triangles that that share one vertex and have only one active vertex. Ties between candidates are broken by using the triangle whose active vertex has the highest index. Fourth, are triangles that that share one vertex and have two active vertices (neither of which is finalized, or it would be second). Having this as the fourth instead of the third priority avoids a bias towards expensive *join* operations. Fifth, are triangles that share one vertex but have no other active vertex. And sixth, is the oldest triangle in the delay buffer.

4.1 Results

We have implemented this greedy reordering scheme and report in Table 3 what effect different delay buffer sizes ranging from 25 to 50,000 triangles have on the compression rates. Probably the biggest surprise is the enormous improvement in compression rates that we can get with a delay buffer that is as small as 25 triangles. For the compacted "David" and "St. Matthew" models, which contain 56 and 372 million triangles, a maximal delay of 25 triangles from the original order improves the bit-rates from around 16 bpv down to around 7 bpv while having no affect whatsoever on the width or the span. For a model of several hundred million triangles even a delay buffer of 50,000 triangles does not have a significant impact on the overall stream quality, while the compression rates come within a factor of two of the ooc-compressor [8].

Only for the *breadth* and the *depth* orderings, the use of the delay buffer can worsen the bit-rates. This is because we destroy the overall regularity of the ordering that such global orderings possess but which a scheme that makes greedy local decisions can not create. Overall the bit-rates of all mesh orderings seem to converge towards 4 to 5 bpv as the buffer size is increased.

The drastic increase of width and span for the smaller models, "armadillo" and "dragon", when using a large delay buffer should not be a surprise. The greedy triangle picking strategy does not attempt to remove triangle as early as possible from the delay buffer. As long as it can find triangles that share vertices with the previous triangle it neglects all other triangles in the delay buffer. To preserve the overall stream quality of the input mesh the size of the delay buffer needs to be set in accordance to width and span. A delay buffer of 10,000 or larger for meshes whose span is as low as 1,197 (see Table 1) is clearly not the right choice.

Sometimes the delay buffer also improves width and span. Although its greedy triangle selection strategy is not designed for this purpose, it can lower the width of a "fractal" front by "smoothing" its growth into triangle fans and it can lower the span by delaying or advancing triangles that appear before or after their neighbors. In the future we want to investigate how delay buffers can be used to "repair" streaming meshes that are locally incoherent, such as the spectral and geometric orderings. In fact, we plan to implement various streaming mesh "filters" that re-order with different objectives and that can simply be plugged on top of a reader or a writer.

5 Correcting a Deterministic Traversal

In the last section we greedily reordered incoming triangles in a small delay buffer to increase the vertex re-use among subsequent triangles before feeding them to the SMC compressor. Even for small buffers this gave significant improvements in compression while preserving the

mesh	bit-rates for delay buffers of different size [bpv]										width and span change [%]					
(ordering)	none	25	50	100	250	500	1000	5000	10000	50000	250		10000		50000	
armadillo																
(vcompact)	39.35	34.72	33.03	31.46	30.00	29.26	28.60	23.73	20.52	12.93	-	-	-	-	-	-
(spectral)	11.04	10.37	9.85	8.90	7.01	5.49	4.39	3.65	3.55	3.47	-	-	9	46	41	292
(geometric)	13.26	10.16	9.32	8.26	6.74	5.56	4.52	3.56	3.52	3.49	-	-	-	88	25	478
(breadth)	3.27	2.48	2.51	2.56	2.65	3.09	3.26	3.62	3.61	3.50	-	8	7	430	32	1795
(depth)	2.66	2.71	2.77	2.85	2.99	3.41	3.76	3.64	3.57	3.52	-	-	-	-	9	-
dragon																
(vcompact)	21.62	15.24	13.30	12.05	11.36	11.21	11.14	9.49	7.35	4.83	-	-	-11	-	-9	-11
(spectral)	12.02	11.03	10.48	9.54	7.74	6.31	5.28	4.42	4.33	4.30	-	-	24	15	67	113
(geometric)	13.45	9.71	8.97	8.33	7.50	6.80	6.08	4.69	4.49	4.38	-	-	5	36	58	177
(breadth)	5.23	3.71	3.70	3.69	3.73	3.78	3.97	4.55	4.53	4.40	-	6	6	286	42	1306
(depth)	4.35	4.50	4.49	4.50	4.56	4.60	4.88	5.11	4.85	4.63	-	-	-	-	-	-
lucy																
(vcompact)	13.60	12.69	12.43	12.25	12.08	11.96	11.75	9.68	8.46	5.96	-	-	-	-	-	-
(spectral)	14.09	7.68	7.07	7.04	7.02	6.97	6.88	6.28	5.75	4.22	-	-	-	-	-7	5
(geometric)	15.18	13.56	13.41	13.30	13.19	12.05	10.49	6.22	4.95	3.80	-	-	-	-	5	89
(breadth)	3.51	2.56	2.56	2.56	2.56	2.57	2.59	2.80	3.37	3.80	-	-	-	132	6	423
(depth)	3.32	3.37	3.37	3.36	3.37	3.36	3.40	3.60	4.19	3.88	-	-	-	-	-	-
david$_{1mm}$																
(vcompact)	15.94	7.13	6.16	5.45	4.85	4.62	4.49	4.00	3.80	3.48	-	-	-	-	-	-
(spectral)	14.30	7.94	7.11	6.64	6.61	6.58	6.54	6.23	5.91	4.51	-	-	-	-	-8	-
(geometric)	12.71	7.60	7.11	6.77	6.39	6.07	5.65	4.77	4.20	3.66	-	-	-	6	-9	70
st. matthew																
(vcompact)	15.62	6.83	6.12	5.45	4.77	4.48	4.32	3.99	3.94	3.64	-	-	-	-	-	-
(spectral)	15.60	8.51	7.71	7.06	6.34	5.95	5.87	5.83	5.77	5.48	-	-	-	-	-	-
(geometric)	13.60	7.67	7.30	6.93	6.58	6.43	6.23	5.44	5.20	3.91	-	-	-	-	-	7
ppm																
(vcompact)	19.64	15.11	15.07	14.96	13.92	11.02	8.82	6.41	6.15	6.05	-	-	-	-	-	-
(spectral)	14.88	8.12	7.32	6.67	5.99	5.64	5.48	5.34	5.26	5.02	-	-	-	-	-	-
(geometric)	16.84	14.80	14.75	14.56	13.11	11.32	10.17	8.56	7.97	6.35	-	-	-	-	-	-

Table 3. Bit-rates for streaming connectivity compression when using delay buffers of different size. Respecting the maximal allowed delay, triangles are greedily chosen from the buffer to maximize vertex re-use among subsequent triangles. Also reported is the effect that greedy reordering has on the width and span given that this change is bigger than five percent.

global order. But no matter how we order the triangles, the SMC compressor needs to encode at least one previous or one explicit reference per triangle into the set of active vertices. In contrast to non-streaming schemes it lacks the determinism of choosing "on its own" where (i.e. adjacent to which half-edge or vertex) it will encode the next triangle.

In this section we report initial results on our "experimental" SMD compressor that takes a different approach to streaming compression. This compressor chooses at which active half-edge to encode the next triangle but corrects the choice if necessary with a special command. The compressor's choice needs to be corrected either if the corresponding triangle has not yet appeared in the stream or if other triangles in the buffer have reached their maximal allowed delay. But whenever the choice is acceptable there is no longer the need to specify where to encode the next triangle with a reference into the set of active vertices. For maximal compression we like to have to correct the compressor as seldom as possible.

Obviously there is no deterministic strategy that can avoid corrections altogether for any triangle order, unless, of course, we buffer *all* the triangles (which is exactly how standard compressors operate). Our SMD compressor implements a simple breadth-first strategy of the active half-edges. The three half-edges that are created when the first triangle is compressed are put into a traversal queue. At every step the compressor chooses to compress the triangle adjacent to the first active half-edge in the queue. If this triangle has not yet appeared in the stream a *skip* command is encoded instead of this triangle's configuration and the compressor compresses the oldest triangle in the waiting queue instead (using . If, however, this triangle simply does not exist in the mesh a *border* command is encoded and the compressor continues with the next element in the traversal queue. We can detect borders case when the active half-edge has at least one finalized vertex.

In Table 4 reports the connectivity compression rates that are achieve with this approach together with the percentage

of *skip* commands relative to the total number of triangles. Roughly speaking, when there are less than two percent of *skip* commands the SMD compressor beats the SMC compressor and gives bit-rates that are nearly as good as those of the ooc-compressor of Isenburg and Gumhold [8].

One interesting observation is that the original (vertex-compacted) triangle ordering of "David" and "St. Matthew" statues gives better results than their "spectral" or "geometric" re-orderings because they need fewer *skips* during compression. We will have to re-design the strategy with which the SMD compressor currently picks the next triangle to account for that. Our current strategy seems to benefit from the "blocky" triangle order of these meshes.

Chopping meshes into spatial blocks and storing them block after block with a compact vertex order results in a small *advancing front* and a large *stagnant front*. Although for each block the entire stagnant front will need to be skipped, the entire advancing front can be fully pursued. A smaller advancing front also needs fewer triangles in the delay buffer to avoid skips. In a way those skips are well invested because the skipped triangles are really far down-stream. The geometric ordered meshes do not have a stagnant front and a larger advancing front. When the delay buffer barely stays ahead of the traversal front, each *skip* is a lousy investment because the skipped triangle will arrive very soon, but will then need an explicit index.

6 Summary and Conclusion

In this report have described a streaming compression scheme that allows to encode meshes *on-the-fly* by operating on an partial representation of the connectivity that is created *and* deleted as the mesh is fed in increments of single triangle and vertices to the compressor. In contrast, previous schemes [10, 5, 9] expect to be given the entire mesh up front and first construct a temporary representation that allows them to traverse the mesh connectivity at will before the compression process starts.

The advantage of a streaming compressor grows with the size of the input mesh, as both construction and use of these temporary representation becomes more and more cumbersome. For the 372 million triangle "St. Matthew" statue, the ooc-compressor by Isenburg and Gumhold [8] first spends 7 hours creating an 11 gigabyte data structure on disk before the actual compression starts that then takes another 4 hours and uses 384 megabytes of main memory. In contrast, we can complete compression in 15 minutes using only 6 megabyte of main memory and no temporary disk space.

When we preserve the exact triangle order our connectivity compression rates are a lot worse compared to those of Isenburg and Gumhold [8]. However, employing a small buffer within we delay triangle to put them into an order that is more "compressible" allows us to significantly improve compression. While streaming connectivity compression will in general not be able to rival those achieved by non-streaming approaches, it makes compression a more useful tool in a typical mesh processing pipeline because it remains transparent to the user.

mesh	bit rates for delay buffer size					
(ordering)	50,000		500,000		2,000,000	
armadillo						
(vcompact)	18.18	(29.2)	2.15	(0.0)	2.15	(0.0)
(spectral)	2.51	(1.6)	1.97	(0.0)	1.97	(0.0)
(geometric)	2.36	(1.0)	1.97	(0.0)	1.97	(0.0)
dragon						
(vcompact)	4.89	(4.1)	3.24	(0.2)	3.12	(0.0)
(spectral)	3.25	(0.7)	3.06	(0.0)	3.06	(0.0)
(geometric)	3.81	(2.2)	3.08	(0.0)	3.08	(0.0)
lucy						
(vcompact)	9.74	(15.3)	3.82	(3.0)	3.19	(1.9)
(spectral)	4.52	(5.3)	2.93	(1.7)	2.30	(0.4)
(geometric)	4.61	(5.8)	2.93	(1.8)	2.19	(0.2)
david$_{1mm}$						
(vcompact)	2.78	(1.7)	2.11	(0.3)	2.03	(0.1)
(spectral)	5.60	(8.5)	2.96	(1.9)	2.32	(0.7)
(geometric)	4.84	(6.9)	3.34	(3.2)	2.66	(1.6)
st. matthew						
(vcompact)	3.21	(2.6)	2.22	(0.4)	2.14	(0.2)
(spectral)	6.91	(9.2)	4.40	(4.7)	2.71	(1.2)
(geometric)	9.40	(21.1)	4.03	(4.2)	3.45	(3.0)
ppm						
(vcompact)	17.55	(33.7)	16.02	(29.7)	6.57	(6.8)
(spectral)	5.38	(5.4)	3.93	(2.6)	3.25	(1.2)
(geometric)	20.37	(48.3)	6.87	(8.6)	4.90	(4.3)

Table 4. The connectivity compression rates of our experimental SMD compressor in bits per vertex for different delay buffer sizes. In brackets we report the percentage of *skip* commands compared to the total number of triangles. Keeping 500,000 (2,000,000) triangles in the buffer results in a memory footprint of roughly 50 (195) MB.

References

[1] A. Anonymous. Lossless compression of predicted floating-point geometry. *JCAD*, 2005. to appear.

[2] A. Anonymous. Streaming meshes. *submitted*, 2005.

[3] R. Bar-Yehuda and C. Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Trans. on Graphics*, 15(2):141–152, 1996.

[4] A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Graphics Interface'01 Proceedings*, pages 81–90, 2001.

[5] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Proc.*, 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] H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH 99 Proceedings*, pages 269–276, 1999.

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

[9] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

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