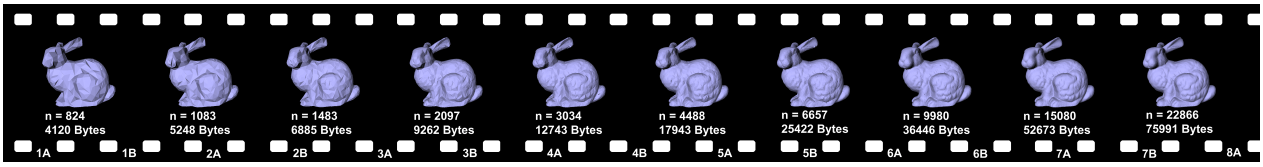
SQUEEZE: Fast and Progressive Decompression of Triangle Meshes

Renato Pajarola
Information & Computer Science
University of California, Irvine
pajarola@acm.org

Jarek Rossignac
Graphics, Visualization & Usability Center
College of Computing
Georgia Institute of Technology
jarek@gvu.gatech.edu



Abstract

An ideal triangle mesh compression technology would simultaneously support the following three objectives: (1) progressive refinements of the received mesh during decompression, (2) nearly optimal compression ratios for both geometry and connectivity, and (3) in-line, real-time decompression algorithms for hardware or software implementations. Because these three objectives impose contradictory constraints, previously reported efforts focus primarily on one – sometimes two – of these objectives. The SQUEEZE technique introduced here addresses all three constraints simultaneously, and attempts to provide the best possible compromise. For a mesh of T triangles, SQUEEZE compresses the connectivity to 3.7T bits, which is competitive with the best progressive compression techniques reported so far. The geometry prediction error encoding technique introduced here leads to 20% improved geometry compression over previous schemes. Our initial implementation on a 300 Mhz CPU achieves a decompression rate of up to 46'000 triangles per second. SQUEEZE downloads a model through a number of successive refinement stages, providing the benefit of progressivity.

1. Introduction

An increasing number of industrial, business and entertainment applications require that users download large numbers of remotely located 3D models over internet connections. It is essential to develop techniques that reduce the waiting time in these applications. When 3D models are required, as opposite to 2D images, a combination of lossy and lossless compression techniques may be invoked.

Most of the popular 3D compression techniques are focused on triangle meshes, because most other representations of 3D shapes may easily be tessellated (i.e. converted to approximating triangle meshes), and because triangles are well supported by most software and hardware graphics subsystems.

Lossless compression techniques strive to significantly reduce the number of bits required for encoding any given 3D model. Compression schemes developed specifically for the most common representations of 3D models perform significantly better than general-purpose compression techniques. The focus of current research in lossless 3D compression is aimed at striking the optimal balance between file size and decompression speed for the wide spread of operating conditions defined by the bandwidth, the local memory, and compute power available for decompression. These conditions vary from decompression hardware, with high bandwidth to main memory, very limited local storage, but extremely fast execution; to mobile devices with limited bandwidth and compute power. The techniques used for lossless compression include connectivity coding schemes for planar and non-planar triangle graphs, prediction of vertex locations from previously decoded vertices, and entropy encoding and transmission of the corrective vectors, which capture the difference between the predicted and the actual locations of the vertices. Many of these techniques have been developed and optimized for the compression of single-precision meshes. Different – and often less effective – solutions are required for progressive meshes, discussed below.

Lossy compression approaches capitalize on two observations. First, most models are represented with more accuracy than demanded by the application. For example, many CAD models represent vertices with double precision floating point numbers when in fact the relative round-off errors resulting from the geometric calculations that were executed to compute the models are greater than one in a million. More surprisingly, many electronic mock-up

or design review applications use tessellated approximations of curved shapes that carry a much larger relative tessellation error than one in a million. Finally, many graphic applications (entertainment, walkthrough) produce images of these models through a series of hardware supported calculations which produce an even larger error in the position of vertices on the screen and in the associated depth. Second, when complicated scenes or assemblies are viewed under perspective, many of the features or details are either out of the viewing frustum, or hidden, or sufficiently far from the virtual viewpoint to project on very small areas of the screen. It is therefore unnecessary to download a full-resolution, precise representation of these models, until they become visible and sufficiently close to the viewpoint for the approximation errors to be noticeable.

The overall strategy for compression is thus to first simplify the models so that they are not over-specified. This is usually done by selecting the appropriate resolution for the approximation (through an adaptive tessellation or simplification process), and by truncating the least significant bits of the vertex coordinates (through a coordinate normalization and quantization process). We refer to the result of this initial accuracy adaptation phase as the full accuracy, or full resolution model M_n for that particular application. Then, instead of encoding M_n as a single-resolution model, one converts it into an equivalent progressive representation [12], which stores a very crude approximation M_0, and a series of upgrades U_i, for i=1 to n. Applying upgrade U_1 to M_0 produces a slightly more accurate model M_1. Applying U_2 to M_1 produces an even more accurate model M_2, and so on, until the application of U_n produces M_n. The series of upgrades and the crude model may be produced by a variety of mesh simplification schemes [11, 25, 12, 23, 6, 17, 7].

Initially, the user will download M_0, and may never need a finer approximation of M_n. But if the model moves closer to the view, a finer approximation may be required. As soon as the display error that results from using M_i exceeds the tolerance imposed by the application, the upgrade U_{i+1} is downloaded and used to increase the accuracy of the local representation of the model, as first suggested in [12].

For faster transmission, M_0 is usually compressed using compression techniques developed for single-precision models. However, the storage of M_0 is typically very small compared to an encoding of M_n. The challenge is thus to compress the upgrades so as to significantly reduce the transmission delays and the decompression costs.

To best reduce the approximation error after any given waiting time, one must strike the optimal balance between the compactness of the compressed representations of M_0 and of the U_i and the time it takes to decode and apply the upgrades. This is not an easy compromise, since more compact representations usually require more complex decompression algorithms. Furthermore, the balance must take into account transmission and computing speed factors, which vary with the hardware used and connectivity bandwidth.

A second trade-off must be made between the n number of upgrades and the effectiveness of their compression. In general, having fewer upgrades leads to economy of scale, and thus better compression ratios per triangle. Individual upgrades, which each insert a single vertex (such as the approach in [12]), require several bits per triangle to identify which vertex must be split. Grouping vertex splits into larger batches, as first proposed in [21] and [14], helps to reduce the vertex identification cost. Unfortunately, limiting the number of upgrades implies that the client will have to wait longer at each level of resolution.

The SQUEEZE technology introduced in this paper provides a novel compromise between compression ratios, number of upgrades, and performance of decompression and upgrade application.

For a typical mesh of t triangles, SQUEEZE compresses the connectivity information down to $3.7 \cdot t$ bits. Although this storage cost is more than twice the storage cost for the best non-progressive compression schemes, it compares advantageously with all previously proposed progressive compression techniques, especially given that SQUEEZE produces about 10 different levels of detail (LODs) for a typical mesh, ensuring a continuous improvement of the quality of the received mesh. After each refinement, the user may manipulate the current resolution model as SQUEEZE decompresses the next upgrade, or temporarily stop the transmission until a higher LOD is needed. Furthermore, as an upgrade is being decoded and applied, the mesh resulting from the early refinement steps of the upgrade are immediately available for rendering, before the upgrade is completed.

Our new geometry prediction techniques leads to an additional 20% improvement in geometry compression over previous progressive methods and yet allows a very fast geometry decompression. Our initial implementation of SQUEEZE can decode up to 270’000 vertex split records per second, and progressively apply the updates at up to 70’000 processed vertex splits per second. Overall, decoding and mesh updates combined, SQUEEZE achieves a mesh decompression speed of about 25’000 vertices (or 50’000 triangles) per second on a 300 Mhz CPU.

2. Related work

Many efficient compression methods for triangulated single-resolution models have been proposed in the last few years [4, 29, 30, 24]. In practice, these approaches can compress the connectivity information (i.e., the triangle/

vertex incidence table) down to less than 2 bits per triangle. Furthermore, after a preprocessing normalization step which quantizes the vertex coordinates to a specified resolution (typically 8 to 16 bit integers), these methods use geometric prediction and entropy coding to compress the geometry.

Performance issues for in-line hardware decompression of single-resolution models, as opposite to optimal compression ratios, were addressed in [4]. An excellent compromise between the performance of software decompression and the file size was presented in [9]. Even more impressive compression ratios for almost regular triangle meshes were reported in [30]. The mesh compression technique of [29] has subsequently been optimized for fast decompression and included in the MPEG-4 standard [8].

The progressive transmission of multiresolution mesh models was introduced in [12] as a technique for graphics acceleration, not focusing on compression. Mesh refinements are based on vertex splits, which each require an encoding of more than 7 bits per triangle for the connectivity information if a progressive ordering according to an error measure is used [13]. Variations of encoding the incremental mesh updates have been proposed in [5].

The idea of grouping the vertex-splits into batches was introduced by the authors in [21] aiming at the reduction of the average storage cost of a vertex split refinement (i.e. [21] achieves an average of 3.6 bits per triangle). Batches of vertex splits or vertex insertions have also been advocated in [14] and in [3] (roughly 3 bits per triangle). A different approach to the encoding of upgrades that refine the mesh by the introduction of a significant fraction of new triangles was proposed in [28]. They encode a forest of edges which, when cut, create holes in the mesh. The internal triangulation of each hole is encoded using a variation of the mesh compression method presented in [TR98]. The connectivity for the entire mesh may be compressed to an average of between 4 and 5 bits per triangle. The method proposed in [1] encodes contours of edges, defined in terms of edge-based distance on the triangular graph computed from a seed vertex. This approach requires between 4.5 and 8.7 bits per triangle for the connectivity information.

The compression of the vertex location is generally based on vertex predictors and entropy coding. The best geometry compression ratios for progressive meshes are reported by the authors in [21] and in [3]. A comparison to both methods is presented in Section 6.

3. Preliminaries

A triangular mesh can also be viewed as a graph G (V, E, F) with vertices V, edges E and (triangular) faces F with an embedding in three-dimensional space. The graph G itself without the vertex coordinates represents the *connectivity* of the mesh. The *geometry* of the mesh consists of the 3D coordinates of the vertices which specify the actual embedding of the graph in space.

The idea of progressively refining a triangular mesh means to increase the number of mesh elements, i.e. triangles, with every step starting from an initial crude mesh. For triangular meshes the smallest incremental update consists of adding one vertex. For typical meshes this also means increasing the number of triangles by two and the number of edges by three.

From the discussion above we can see that an incremental update has two main components: the connectivity changes and the geometry information. The connectivity update consists of increasing the number of elements, and changing the incidence relation between vertices. The first part specifies the topological location, the area of the mesh that will be affected by the update, and the second part determines the local incidence changes at that location. The geometry information includes the 3D coordinates of the new vertex, and possibly also coordinates of previously existing vertices that changed their positions.

Applying a sequence of incremental mesh updates to an initial crude mesh M_0, generates a progressive series of increasingly complex meshes M_0, M_1, …, M_n, where M_n refers to the full resolution triangular mesh including all available vertices. As mentioned in the introduction we are interested in progressivity in the sense of increasing the quality of the object that is represented by the triangular mesh with every update. Therefore, the initial mesh M_0 embodies only a crude approximation, the incremental updates $U_i = M_{i-1} \rightarrow M_i$ increase the mesh complexity and reduce the approximation error with every step, and M_n is the highest quality mesh representation of the object. A sequence of progressive mesh refinements can be obtained from mesh simplification methods which create different *levels of detail* (LODs) from a high resolution input mesh by iteratively simplifying the current mesh. Good overviews of mesh simplification methods can be found in [10] and [18].

In SQUEEZE, the simplification and reconstruction of the triangular mesh is based on the *edge collapse* (ecol) and *vertex split* (vsplit) operations introduced in [11], see also Figure 1. A coarse mesh M_0 and a sequence of vsplits define a progressive mesh of increasing approximation quality as presented in [12].

FIGURE 2. Batches of vertex split refinement operations of three different mesh updates (not consecutive in this example). The two triangles inserted by a vertex split are highlighted in each image.

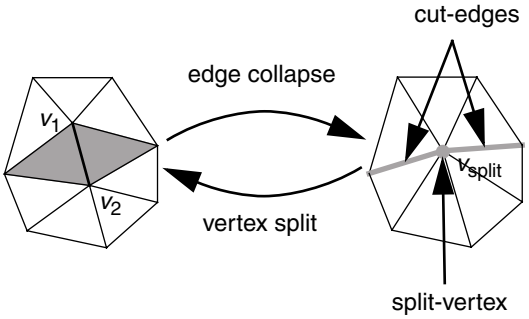


FIGURE 1. Edge collapse (ecol) and vertex split (vsplit) operations for triangle mesh simplification and reconstruction.

Note that we use the *half-edge collapse* simplification that assigns the split-vertex to one of the vertices of the collapsed edge, i.e. $v_{split} = v_1$. Thus based on the displacement vector $v_{disp} = v_2 - v_1$ the original vertices can be recovered as $v_1 = v_{split}$ and $v_2 = v_{split} + v_{disp}$. Most other placement variants, such as the midpoint placement of the split-vertex $v_{split} = 0.5 \cdot (v_1 + v_2)$, do not guarantee that the coordinates of v_{split} stay on the quantized coordinate grid, making geometry encoding more complex. Furthermore, the half-edge collapse has shown to yield better approximation quality than midpoint placement due to better preserving the volume of the given object, see also [18] for a discussion of different edge collapse simplification and vertex placement methods.

4. Progressive mesh encoding

Instead of encoding every vertex split operation individually, SQUEEZE groups simplification and refinement operations into batches to increase connectivity encoding efficiency. This concept was introduced in [21] and successfully extended to progressive tetrahedral meshes in [22]. SQUEEZE creates a series of meshes $M_0, M_1, \ldots, M_n$ where each update $U_i = M_{i-1} \rightarrow M_i$ between consecutive LODs M_{i-1} and M_i consists of multiple vertex split operations. In the course of this paper U_i will denote a batch of vertex split refinement operations, and U_i^{-1} a set of edge collapse simplification operations, see also Figure 2 for graphical examples.

The format of the compressed mesh consists of the initial coarse base mesh M_0, encoded using a single-resolution mesh compression method such as [24], followed by a series of compressed updates U_i. Following we describe the encoding of a batch U_i^{-1} of edge collapse operations. Decompression starts with decoding the coarse base mesh M_0, and then applying the inverse of the encoding steps described below to every compressed update U_i.

The encoding of a mesh update U_i, recovering M_i from the previous mesh M_{i-1}, requires specifying all split-vertices and their cut-edges in M_i, and the coordinates of the newly inserted vertices. The following steps are performed to encode an update batch U_i:

1. Construct and traverse a vertex spanning tree of M_{i-1} and mark all split-vertices – the results of applying the edge collapses U_i^{-1} to mesh M_i.

   For every marked split-vertex v_{split}, we encode its cut-edges as follows:
2. We compute the indices of the two cut-edges in the sorted list of the incident edges on v_{split}, clockwise, starting with the edge from v_{split} to its parent in the vertex spanning tree.
3. Given the degree d of the split-vertex in mesh M_i, the two edge indices are identified as one possible choice out of $\binom{d}{2}$ for selecting the cut-edges, we encode this choice using exactly $\left\lfloor \log_2 \binom{d}{2} \right\rfloor$ bits.

   Since SQUEEZE uses the half-edge collapse operator the split direction has to be specified:
4. Using one bit we distinguish between $v_{split} = v_1$ or $v_{split} = v_2$ (see also Figure 1).

   Furthermore, the geometry has to be encoded:
5. The displacement vector $v_{disp} = v_{new} - v_{split}$ is entropy encoded as described in Section 6.

5. Simplification and compression

Coupled with the encoding method mentioned previously is the simplification process that generates the different LODs M_n, M_{n-1}, …, M_0 of decreasing accuracy. Each simplification step $U_i^{-1} = M_i \rightarrow M_{i-1}$ has to select as many edge collapses as possible to reduce the number of wasted 0-bits during the split-vertex marking of Step 1 above. However, the edge collapses in M_i have to be chosen in such a way that the vertex splits can uniquely be identified in M_{i-1}. The following three topological requirements for grouping edge collapses in a batch U_i^{-1} are sufficient:

1. At most two vertices may be collapsed into one.
2. For each edge $e = (v_1, v_2)$ that will be collapsed and any other vertex w that is incident to both v_1 and v_2, the triple (v_1, v_2, w) must define a valid triangle in the mesh M_i.
3. For each edge $e_1 = (v_1, v_2)$ that will be collapsed and any edge $e_2 = (w_1,w_2)$ forming a quadrilateral (v_1, v_2, w_1,w_2) with e_1 in M_i, e_1 and e_2 cannot be collapsed in the same batch.

To achieve a good approximation at each stage, the simplification process must also use an error metric to evaluate and order edge collapses in addition to satisfying the topological constraints mentioned above. The current implementation uses a variant of the quadric error metric introduced in [6] to order edge collapses according to their introduced approximation error, however, another edge collapse ordering [11, 12, 17] could be used as well. In every simplification batch U_i^{-1}, a maximal subset of the least expensive edges in M_i that do not violate the topological constraints defined above are greedily selected and collapsed.

6. Fast geometry decompression

Geometry compression, encoding the 3D coordinates of vertex split operations, is a problem of variable length coding of scalar values. If these values change smoothly over time or space, but their frequency distribution is uniformly noisy, then *prediction error coding* can efficiently be applied. The half-edge collapse used in the simplification process shrinks an edge $e = (v_1, v_2)$ to one vertex, i.e. $v_{split} = v_1$, and the deleted vertex $v_{del} = v_2$ must be encoded for recovery. The deleted vertices of one simplification batch $U_i^{-1} = M_i \rightarrow M_{i-1}$ are all geometrically close to the surface of the simplified mesh M_{i-1}, thus their values change smoothly over the surface M_{i-1}, but the frequency distribution of individual coordinate values is noisy. The coordinates of local displacement vectors $v_{disp} = v_{del} - v_{split}$ have a much more skewed distribution. The displacement vector represents a simple vertex prediction error that uses the old vertex v_{split} as the estimate for the new vertex v_{del}. In [21] and [3] more sophisticated prediction schemes have been proposed based on the local neighbors of the split-vertex and the deleted vertex.

To speed up decompression time compared to [21], we simplified the prediction method to computing the average of direct neighbors of v_{del} in the presented approach, similar to [3]. Based on the connectivity decoding the correct mesh connectivity of M_i can be reconstructed without actually knowing the geometric coordinates of the new vertex v_{del}, see also Figure 3. Therefore, the decoder can use the same immediate neighbors a_1, …, a_k of v_{del} that have been used for compressing the geometry. The estimated position of the deleted vertex is calculated as:

$$v_{est} = \frac{1}{k} \cdot \sum_{i=1}^{k} a_i \qquad \textbf{(EQ 1)}$$

The geometry information that is encoded with each vertex split is therefore the difference $v_{err} = v_{del} - v_{est}$ between actual and estimated vertex positions. During decompression the correct vertex position v_{del} can be recomputed again by using Equation 1 and adding the decoded v_{err}. Decompression speed is mainly determined by the vertex position prediction function, which is greatly simplified by Equation 1 compared to the approach presented in [21], and by decoding v_{err} for which an efficient solution is presented below.

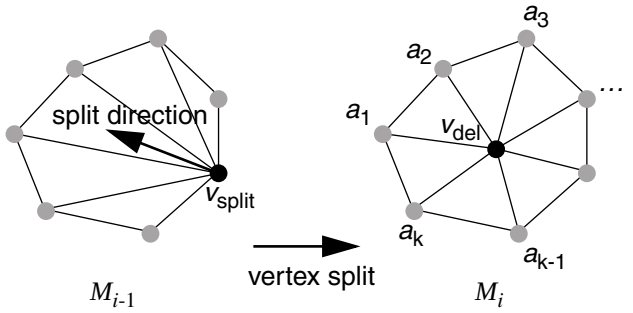
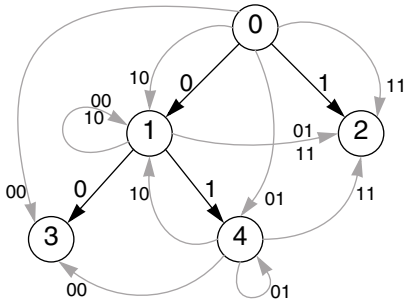


FIGURE 3. Estimating displacement vectors. The new vertex v_{del} is estimated as the average of its immediate neighbors a_1, …, a_k, which are known from connectivity decoding.

The frequency distribution of prediction errors v_{err} is very skewed towards small absolute values, and centered around 0 for most shapes. Also the frequencies tend to decrease exponentially for larger absolute values which makes it suitable for *entropy coding* [2, 16]. The efficiency problem with such variable length coding methods is that a straight-forward implementation of the decoder has to examine every single bit to proceed in the decision or coding tree, until reaching a state in which a decoded symbol can be returned. To speed up decoding performance we implemented the high-performance Huffman decoder proposed in [20] that allows to read and process multiple-bit words, i.e. bytes, instead of single bits from the com-



a)

words

start-nodes	00	01	10	11
0	3	4	1	2
1	1	2	1	2
2	*3*	*4*	*1*	*2*
3	*3*	*4*	*1*	*2*
4	*3*	*4*	*1*	*2*

b)

from node	word	symbols
0	00	A
0	01	B
0	10	C
0	11	C, C
1	00	A
1	01	A, C
1	10	B
1	11	B, C

c)

FIGURE 4. a) Huffman code tree with some indicated node transitions using a word size of 2 bits for the codes A=00, B=01 and C=1. b) The complete node transitions table for all nodes indicating the end-node for any given combination of start-node and data word. Note that all leaves have the same *jump* table as the root node and can

pressed data stream. The approach is similar to the method presented in [27], every node in the binary Huffman code tree has a *jump table* associated with it. This jump table captures the necessary information to decode any 8-bit sequence starting at the current node: it yields the next node resulting from a tree traversal according to the 8-bit sequence, restarting at the root whenever reaching a leave, and it also provides a list of decoded symbols that have been encountered passing any leave of the tree while processing the 8 bits. Therefore, the decoder can read the compressed data stream in bytes, update its current node accordingly and output the decoded symbols for every node transition. An example Huffman code tree and the corresponding jump table for 2-bit words are illustrated in Figure 4.

The Huffman code for a set of symbols – quantized prediction error values in our case – is based on their frequency distribution. The decoder has to use exactly the same code as the encoder, thus either the Huffman code itself or the symbol frequencies have to be transmitted before decompression is possible. Note that this has to be done for every refinement batch since the prediction error distribution changes with every LOD. To avoid this overhead of sending decoding information and constructing a code tree on-the-fly for every refinement batch, we model the actual prediction error distribution by a probability distribution, and precompute Huffman codes for a fixed set of different distributions.

One can observe that prediction errors for good estimators have a probability distribution that decreases exponentially with the absolute value of the prediction error. The *Laplace* distribution of Equation 2 is widely used for statistical coding of differential signals in image compression [19, 26]. We use this Laplace distribution to model the prediction error histogram of our geometric vertex position estimator of Equation 1. For symmetric error distributions, the mean μ is 0, and the variance υ uniquely defines the shape of the Laplace distribution. The variance is adjusted for every batch by the average of the squared prediction errors, $\upsilon = (1/|\text{batch}|)\sum_{v_{err} \in \text{batch}}(v_{err}-\mu)^2$. Thus the only information that has to be sent at the beginning of every batch is the current variance for the Laplace distribution.

$$L_\upsilon(x) = \frac{1}{\sqrt{2\upsilon}} e^{-\sqrt{\frac{2}{\upsilon}}|x-\mu|} \qquad \textbf{(EQ 2)}$$

Given the frequency distribution defined by the variance υ and the Laplace function $L_\upsilon(x)$, a Huffman code can be constructed for every batch based on its variance. However, this process can be very time consuming. We avoid this problem by precomputing a set of Huffman codes for a set of 37 pre-specified variances that guarantee an unnoticeable loss in coding efficiency as shown in [15, 26]. At the beginning of every batch, the Huffman code for a fixed variance closest to the given variance is chosen and used to decode the compressed geometry data of the entire batch.

7. Experimental results

We conducted a variety of experiments comparing our fast progressive mesh compression method to other approaches with respect to compression efficiency and decompression speed. Compression efficiency was challenged with one of the best known single-resolution mesh compression techniques [30], and compared to various recently developed multiresolution mesh compression methods [21, 3, 1]. The progressive mesh compression method introduced in [28] cannot provide sufficient geometry compression compared to the newer approaches and is not included in the experiments. Also the progressive mesh compression method presented in [13] only reported tests on highly quantized meshes, and does not provide advanced connectivity compression. The progressive approach presented in [5] needs even more bits than the initial progressive meshes representation of [12], and thus does not improve compression ratio. Decompression performance of progressive methods was only reported in [13]. We also compare speed to single-resolution compression methods [5, 9].

			Touma et al. [30]		Cohen-Or et al. [3]		SQUEEZE	
quantization	models	vertices	bits / Δ for connectivity	bits /vertex for coordinates	bits / Δ for connectivity	bits /vertex for coordinates	bits / Δ for connectivity	bits /vertex for coordinates
8 bit	triceratops	2832	1.1	8.3	N / A		3.7	9.6
	blob	8036	0.9	7.9			3.7	9.2
12 bit	triceratops	2832	1.1	22	2.9	26	3.7	21
	blob	8036	0.9	21	3	26	3.8	21

TABLE 1. Compression efficiency compared to other existing techniques based on published results. The method presented in [30] provides highest compression ratios for single-resolution meshes, whereas the approach of [3] shows smallest connectivity encoding for progressive meshes.

			Bajaj et al. [1]		Pajarola et al. [21]		SQUEEZE	
quantization	models	vertices	bits / Δ for connectivity	bits /vertex for coordinates	bits / Δ for connectivity	bits /vertex for coordinates	bits / Δ for connectivity	bits /vertex for coordinates
10 bit	fandisk	6475	N / A		3.4	15	3.7	15
	fohe	4005			3.7	15	3.8	19
	triceratops	2832			3.5	20	3.8	15
12 bit	bunny	34834	5.0	28	3.6	16	3.8	17
	phone	83044	4.5	31	3.6	14	3.8	13

TABLE 2. Compression efficiency compared to published results and to an initial version of the proposed method. [21] has a more sophisticated but more complex vertex prediction function than SQUEEZE.

Models	Quantization	Model size		Vertex splits per second		
		base mesh	*vertex splits*	*decoding*	*mesh update*	*combined*
triceratops	12 bit	53	2779	277900	55579	46315
	10 bit	53	2779	92633	69975	39862
fandisk	10 bit	89	6386	159650	45614	35477
blob	12 bit	137	7899	112842	41573	30380
	8 bit	126	7910	197750	43944	35954
bunny	12 bit	824	34010	117275	21662	18284
phone	12 bit	1403	81641	114987	14173	12617

TABLE 3. Decompression speed performance. Model size denotes the number of vertices in the base mesh, and the number of inserted ones due to vertex splits. Performance is measured in processed vertex splits per second for decoding only, for applying updates on the triangle mesh, and for combined decompression and mesh update speed.

Tables 1 and 2 evaluate the compression efficiency of our approach compared to other methods mentioned in the literature. SQUEEZE is competitive with the single-resolution compression approach of [30] in terms of geometry compression. The added functionality of progressive multi-resolution mesh reconstruction comes at a higher cost for connectivity encoding due to the complexity of the multi-resolution model. Our approach outperforms other multi-resolution compression methods [3, 1] mainly in geometry encoding. Note that the presented method and the method in [3] use a very similar geometry prediction method but quite different entropy coding techniques. Note that in [3] the simplification process is not driven by an error metric. The method presented in [21] compresses slightly better on average than the presented approach, however, at much higher processing cost for geometry prediction.

Decompression speed performance is reported in Table 3 for various test models, measured on a 300MHz R12000 SGI O2. While decoding speed is limited by geometry prediction and Huffman decoder, updating the triangle mesh depends on the size of the model due to the growing mesh data structures. In [13] timings are reported for decoding vertex split records, but only for a *gzip* compression version and not for the advanced arithmetic coding version which is expected to be slower by order of magnitudes. Nevertheless, our decoding performance of up to 270'000 decoded vertex split records per second compares favorably with the 80'000 reported in [13],[1] consid-

ering the much better compression ratio that is achieved. In [5] decompression speed has been reported for a single-resolution mesh coding method only, achieving roughly 16'000 vertices per second.[1] Much higher connectivity decoding speed was reported in [9] with up to 400'000 vertices per second.[2] Both of these approaches *do not include geometry coding* which dominates decompression speed. Considering that the presented approach and the reported timings in Table 3 include the complete geometry information and provide a much higher complexity in functionality – progressive reconstruction of a multiresolution model – our approach is superior to [5] and is a convincing alternative to [9] when progressivity and speed are equally important.

Figure 5 shows the different test models, and results from SQUEEZE. Indicated are the number of bits needed in SQUEEZE to represent the mesh M_i and the time needed for transmission using a 56Kb/s connection. Note that the bits and time of M_i include all previous LODs M_j for $j = 0$ to i. In roughly one-tenth of the time and size of the full resolution model M_{10} SQUEEZE provides already 6 progressive LODs M_0 to M_5.

With an average speed of 25'000 decoded and applied vertex splits per second, thus reconstructing the mesh at a rate of 50'000 triangles per second, and a typical encoding of about 11 bits per triangle, SQUEEZE can decompress and reconstruct a progressive mesh in real-time on the test machine (300MHz CPU) for connections with a sustained transmission rate of 550Kb/s.

8. Conclusion

The progressive multiresolution mesh compression method SQUEEZE introduced in this paper provides an effective compromise between optimal compression ratio, flexibility in mesh reconstruction, and decompression speed. SQUEEZE matches or improves the best progressive mesh compression methods in terms of compression efficiency, and additionally provides high-speed decompression. Even for single-resolution mesh compression methods there have not been reported faster methods than SQUEEZE that incorporate both connectivity as well as geometry information.

The improvements achieved by SQUEEZE are based on a unique combination of new and improved techniques for progressive mesh encoding, geometry prediction, entropy coding, and variable-length prefix decoding. Good compression efficiency is realized by grouping mesh refinements and locally encoding updates based on a vertex traversal order, by encoding the coordinates of new vertices based on a local prediction error encoding, and by removing code tables from the data file. High decompression speed is gained by using a computationally simple geometry predictor, a fast prefix-code decoding algorithm and data structure, and by precomputing Huffman codes which can be used for multiple downloads.

Acknowledgements

This work was supported by the Swiss NF grant 81EZ-54524 and US NSF grant 9721358. We would like to thank Peter Lindstrom for discussions on geometric predictors and for providing geometric models.

1. 200MHz Pentium
1. SGI Indigo2
2. 175MHz R10000 SGI O2

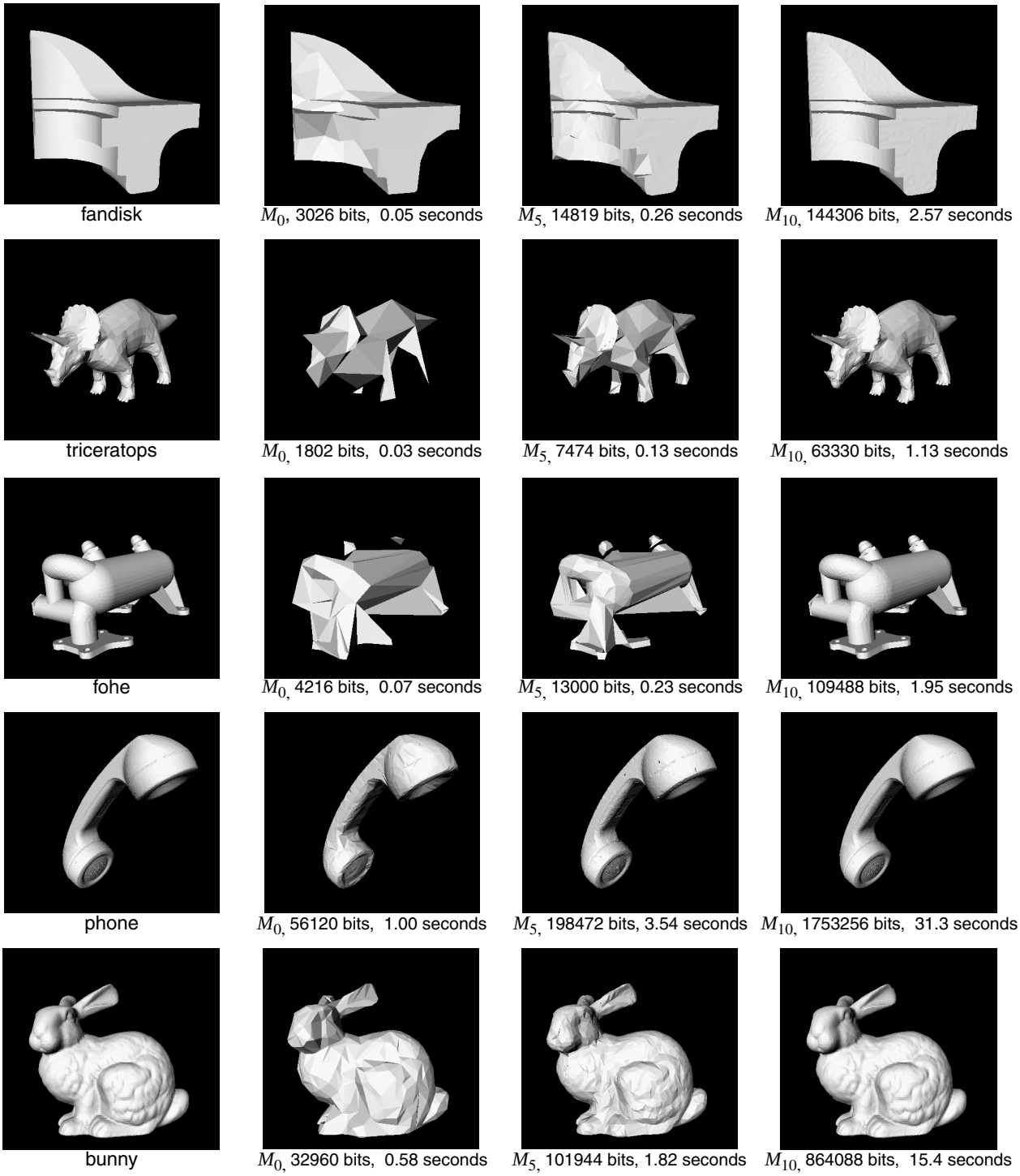


FIGURE 5. Original test models shown in the first column, followed by the base mesh M_0, an intermediate representation M_5 and the full resolution model M_{10} of SQUEEZE. Bits and transmission time of M_i include all prior LODs, and time is estimated for a 56Kb/s communication.

References

[1] Chandrajit L. Bajaj, Valerio Pascucci and Guozhong Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *Proceedings Visualization 99*, 307–316. IEEE, Computer Society Press, Los Alamitos, California, 1999.

[2] John G. Cleary, Radford M. Neal, and Ian H. Witten. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[3] Daniel Cohen-Or, David Levin and Offir Remez. Progressive compression of arbitrary triangular meshes. In *Proceedings Visualization 99*, 67–72. IEEE, Computer Society Press, Los Alamitos, California, 1999.

[4] Michael Deering. Geometry compression. In *Proceedings SIGGRAPH 95*, pages 13–20. ACM SIGGRAPH, 1995.

[5] Leila De Floriani, Paola Magillo, and Enrico Puppo. Compressing TINs. In *Proceedings of the 6th ACM Symposium on Advances in Geographic Information Systems*, pages 145–150, 1998.

[6] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings SIGGRAPH 97*, pages 209–216. ACM SIGGRAPH, 1997.

[7] André Guéziec. Locally toleranced surface simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):168–189, April-June, 1999.

[8] André Guéziec, Frank Bossen, Gabriel Taubin and Claudio Silva. Efficient compression of non-manifold polygonal meshes. Technical Report RC 21453/96815, IBM T.J. Watson Research Center, 1999. (also in SIGGRAPH 99 Course Notes)

[9] Stefan Gumhold and Wolfgang Strasser. Real time compression of triangle mesh connectivity. In *Proceedings SIGGRAPH 98*, pages 133–140. ACM SIGGRAPH, 1998.

[10] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. In *SIGGRAPH 97 Course Notes 25*. ACM SIGGRAPH, 1997.

[11] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings SIGGRAPH 93*, pages 19–26. ACM SIGGRAPH, 1993.

[12] Hugues Hoppe. Progressive meshes. In *Proceedings SIGGRAPH 96*, pages 99–108. ACM SIGGRAPH, 1996.

[13] Hugues Hoppe. Efficient implementation of progressive meshes. Technical Report MSR-TR-98-02, Microsoft Research, 1998. (also in SIGGRAPH 98 Course Notes 21)

[14] Hugues Hoppe. Efficient implementation of progressive meshes. In *SIGGRAPH 99 Course Notes*. ACM SIGGRAPH, 1999.

[15] Paul G. Howard. *The Design and Analysis of Efficient Lossless Data Compression Systems*. Ph.D. thesis, Department of Computer Science at Brown University, 1993.

[16] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. Inst. Electr. Radio Eng.*, pages 1098–1101, 1952.

[17] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In *Proceedings Visualization 98*, pages 279–286. IEEE, Computer Society Press, Los Alamitos, California, 1998.

[18] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April-June, 1999.

[19] Arun N. Netravali and Barry G. Haskell. *Digital Pictures: Representation, Compression and Standards*. Plenum Press, New York and London, second edition, 1995.

[20] Renato Pajarola. Fast Huffman code processing. Technical Report UCI-ICS-99-43, Information and Computer Science, University of California Irvine, 1999.

[21] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. Technical Report GIT-GVU-99-05, GVU Center, Georgia Institute of Technology, 1999.

[22] Renato Pajarola, Jarek Rossignac and Andrzej Szymczak. Implant Sprays: Compression of progressive tetrahedral mesh connectivity. In *Proceedings Visualization 99*, pages 299–305. IEEE, Computer Society Press, Los Alamitos, California, 1999.

[23] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. *IEEE Computer Graphics Forum*, 15(3):C67–C76, August 1996.

[24] Jarek Rossignac. Edgebreaker: Compressing the incidence graph of triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, January-March, 1999.

[25] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In Bianca Falcidieno and Tosiyasu L. Kunii, editors, *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, Berlin, 1993.

[26] David Salomon. *Data compression: the complete reference*. Springer-Verlag, New York, 1998.

[27] Andrzej Sieminski. Fast decoding of the huffman codes. *Information Processing Letters*, 26(5):237–241, January 1988.

[28] Gabriel Taubin, André Guéziec, William Horn and Francis Lazarus. Progressive forest split compression. In *Proceedings SIGGRAPH 98*, pages 123–132. ACM SIGGRAPH, 1998.

[29] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[30] Costa Touma and Craig Gotsman. Triangle Mesh Compression. In *Proceedings Graphics Interface 98*, pages 26–34, 1998.