

# UC Irvine

## ICS Technical Reports

### Title

Fast multiresolution mesh decompression

### Permalink

<https://escholarship.org/uc/item/4gv2t4v9>

### Authors

Pajarola, Renato  
Rossignac, Jarek

### Publication Date

1999

Peer reviewed

Z  
699  
C3  
no. 99-50

# ICS

## TECHNICAL REPORT

### **Fast Multiresolution Mesh Decompression**

*Renato Pajarola, Jarek Rossignac*

UCI-ICS Technical Report No. 99-50  
Department of Information & Computer Science  
University of California, Irvine

October 1999

Notice: This Material  
may be protected  
by Copyright Law  
(Title 17 U.S.C.)

Information and Computer Science  
University of California, Irvine



SLBAR  
Z  
699  
C3  
no. 99-50

# Fast Multiresolution Mesh Decompression

Renato Pajarola

Information & Computer Science  
University of California, Irvine

Jarek Rossignac

Graphics, Visualization & Usability Center  
College of Computing  
Georgia Institute of Technology

## Abstract

*Compression techniques for triangle-mesh representations of 3D models have been the focus of many recent efforts from the graphics, modeling, and theory research community; from developers of computer graphics hardware and software; and from organizations that define international standards. An ideal 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 on the compressed format, previously reported efforts focus primarily on one – sometimes two – of these objectives. The SQUEEZE technique introduced here for Fast and Progressive Decompression of Triangle Meshes addresses all three constraints simultaneously and attempts to provide the best possible compromise for the needs of common internet applications that require frequent access to remote 3D databases. For a typical 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 encoding techniques introduced here lead to an additional 20% improvement in geometry compression over previous schemes. Our initial implementation on a 200 Mhz CPU achieves a decompression rate of about 15000 triangles per second. Finally, in general SQUEEZE downloads a model through 10 successive refinement stages, providing the full benefit of progressivity. After each refinement step, the user may manipulate the current resolution model as SQUEEZE decompresses the next upgrade, or temporarily stop the transmission until a higher level-of-detail is needed.*

**Keywords** progressive triangulation, geometry compression, multiresolution modeling, entropy coding

## 1. Introduction

An increasing number of industrial, business and entertainment applications require that users download large numbers of remotely located 3D models over phone or regular 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 be easily tessellated (i.e. converted to approximating triangle meshes), and because triangles are well supported by most software and hardware graphics subsystems.

Notice: This Material  
may be protected  
by Copyright Law  
(Title 17 U.S.C.)

Loss-less 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 loss-less 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 loss-less 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  $\mathcal{M}_n$  for that particular application. Then, instead of encoding  $\mathcal{M}_n$  as a single-resolution model, one converts it into an equivalent progressive representation [Hop96], which stores a very crude approximation  $\mathcal{M}_0$ , and a series of upgrades  $\mathcal{U}_i$ , for  $i=1$  to  $n$ . Applying upgrade  $\mathcal{U}_1$  to  $\mathcal{M}_0$  produces a slightly more accurate model  $\mathcal{M}_1$ . Applying  $\mathcal{U}_2$  to  $\mathcal{M}_1$  produces an even more accurate model  $\mathcal{M}_2$ , and so on, until the application of  $\mathcal{U}_n$  produces  $\mathcal{M}_n$ . The series of upgrades and the crude model may be produced by a variety of mesh simplification schemes [HRD<sup>+</sup>93, RB93, Hop96, RR96, GH97, LT98, Gue99].

Initially, the user will download  $\mathcal{M}_0$ , and may never need a finer approximation of  $\mathcal{M}_n$ . 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  $\mathcal{M}_i$  exceeds the tolerance imposed by the application, the upgrade  $\mathcal{U}_{i+1}$  is downloaded and used to increase the accuracy of the local representation of the model, as first suggested in [Hop96].

For faster transmission,  $\mathcal{M}_0$  is usually compressed using compression techniques developed for single-precision models. However, the storage of  $\mathcal{M}_0$  is typically very small compared to an encod-

ing of  $\mathcal{M}_n$ . The challenge is thus to compress the upgrades so as to significantly reduce the transmission delays and the decompression costs.

The comparison between two such compressed progressive transmission schemes is difficult, because authors would typically report total bit-per-vertex or bit-per-triangle storage counts and vertex-per-second or triangle-per-second decompression speeds, without any quantitative description of how good the intermediate results are. A methodology for comparing progressive compression systems was first introduced in [PR99] and was based on the curve that plots the relative error between  $\mathcal{M}_j$  and  $\mathcal{M}_n$  as a function of the total number of bits needed to encode  $\mathcal{M}_0, \mathcal{U}_1, \mathcal{U}_2 \dots \mathcal{U}_j$ . The relative error may be defined as the ratio of the absolute error bound to the size of the entire model.

We propose here to improve on that methodology in two ways. We first consider a curve that expresses the error between  $\mathcal{M}_j$  and  $\mathcal{M}_n$  as the function of the total delay, which combines the time it takes to transmit  $\mathcal{M}_0$  and the  $\mathcal{U}_i$ , for  $i$  from 1 to  $j$ , and the time it takes to decompress them and to apply the upgrades. Furthermore, to facilitate the quantitative comparison between compression schemes, we advocate here to use the integral of the area under that curve. This may provide a good single measure of the efficiency of progressive compression schemes. The smaller the integral, the smaller the expected error after any given waiting time.

To best reduce that error, one must strike the optimal balance between the compactness of the compressed representations of  $\mathcal{M}_0$  and of the  $\mathcal{U}_i$  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 tradeoff 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 [Hop96]), require several bits per triangle to identify which vertex must be split. Grouping vertex splits into larger batches, as first proposed in [PR99] and [Hoppe SIG course 99], 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, and therefore increases the error integral, defined above.

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 140000 vertex split records per second, and progressively reconstruct the triangle mesh at a rate of about 15000 triangles per second on a 200 Mhz CPU.

## 2. Related work

Many efficient compression methods for triangulated single-resolution models have been proposed in the last few years [Dee95, TR98, TG98, Ros99]. 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 [Dee95]. An excellent compromise between the performance of software decompression and the file size was presented in [GS98]. Even more impressive compression ratios for almost regular triangle meshes were reported in [TG98]. The mesh compression technique of [TR98] has subsequently been optimized for fast decompression and included in the MPEG-4 standard [Taubin's talk in the SIG99 course].

The progressive transmission of multiresolution mesh models was introduced in [Hop96] 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 [Hop98]. Variations of encoding the incremental mesh updates have been proposed in [FMP98].

The idea of grouping the vertex-splits into batches was introduced by the authors in [PR99] aiming at the reduction of the average storage cost of a vertex split refinement (i.e. [PR99] achieves an average of 3.6 bits per triangle). Batches of vertex splits or vertex insertions have also been advocated in [Hoppe Sigg99 course] and in [CLR99] (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 [TGHL98]. 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 [BPZ99] 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 [PR99] and in [CLR99]. 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, also called the topology. 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. If the mesh is manifold, homeomorphic to a sphere and has no boundary this also means adding two triangles and three edges to the existing mesh. This can also be derived from the Euler relation between vertices, edges and faces of planar

graphs where  $|V| - |E| + |F| = 2$ , and from the fact that in a triangular mesh one face has 3 edges and every edge is shared by 2 faces, thus  $2 \cdot |E| = 3 \cdot |F|$ .

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 the incidence changes between vertices, which can be split into two parts. 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, see also Figure 1. The geometry information includes the 3D coordinates of the new vertex, and possibly also coordinates of previously existing vertices that changed their positions.

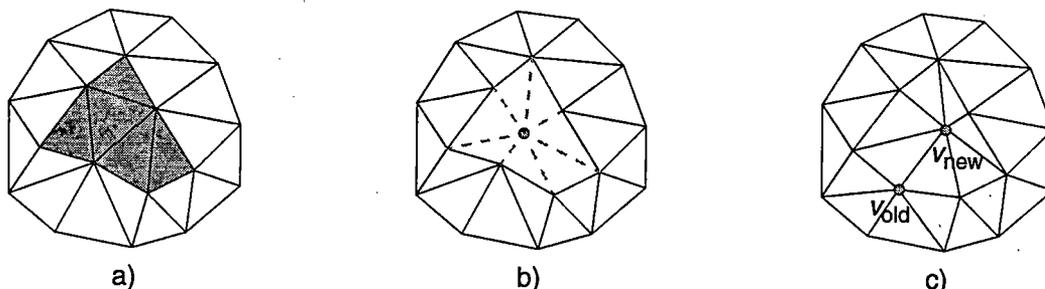


FIGURE 1. Components of incremental updates: a) topological location where the connectivity will change, b) new mesh elements and incidence relations, and c) coordinates for new vertex and updated old vertices.

Applying a sequence of incremental mesh updates to an initial crude mesh  $\mathcal{M}_0$ , generates a progressive series of increasingly complex meshes  $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_n$ , where  $\mathcal{M}_n$  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  $\mathcal{M}_0$  embodies only a crude approximation, the incremental updates  $\mathcal{U}_i = \mathcal{M}_{i-1} \rightarrow \mathcal{M}_i$  increase the mesh complexity and reduce the approximation error with every step, and  $\mathcal{M}_n$  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 [HG97] and [LT99].

In SQUEEZE, the simplification and reconstruction of the triangular mesh is based on the *edge collapse* (ecol) and *vertex split* (vsplit) operations introduced in [HRD<sup>+</sup>93], see also Figure 2. A coarse mesh  $\mathcal{M}_0$  and a sequence of vsplits define a progressive mesh of increasing approximation quality as presented in [Hop96].

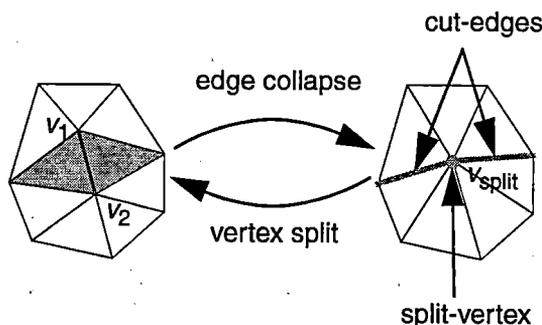


FIGURE 2. Edge collapse (ecol) and vertex split (vsplit) operations for triangle mesh simplification and reconstruction.

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.  $v_{\text{split}} = v_1$ . Thus based on the displacement vector  $v_{\text{disp}} = v_2 - v_1$  the original vertices can be recovered as  $v_1 = v_{\text{split}}$  and  $v_2 = v_{\text{split}} + v_{\text{disp}}$ . Most other placement variants, such as the midpoint placement of the split-vertex  $v_{\text{split}} = 0.5 \cdot (v_1 + v_2)$ , do not guarantee that the coordinates of  $v_{\text{split}}$  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 [LT99] for a discussion of different edge collapse simplification and vertex placement methods.

Figure 3 illustrates the combination of data compression and progressive reconstruction. The difference signals of successive simplification operations are compressed and transmitted. First, the most simplified mesh  $\mathcal{M}_0$  is sent using a suitable single-precision mesh compression method, then the compressed mesh differences are sent in the inverse order of the simplification process. Decompression starts with decoding the initial mesh  $\mathcal{M}_0$ , and consecutively incorporates the decompressed mesh differences  $\mathcal{U}_i$ . Note that the reconstruction of  $\mathcal{M}_{i+1}$  and the decompression of  $\mathcal{U}_{i+1}$  can only use information available from previously reconstructed mesh  $\mathcal{M}_i$ . This has also to be taken into consideration when compressing the difference signals in the first place.

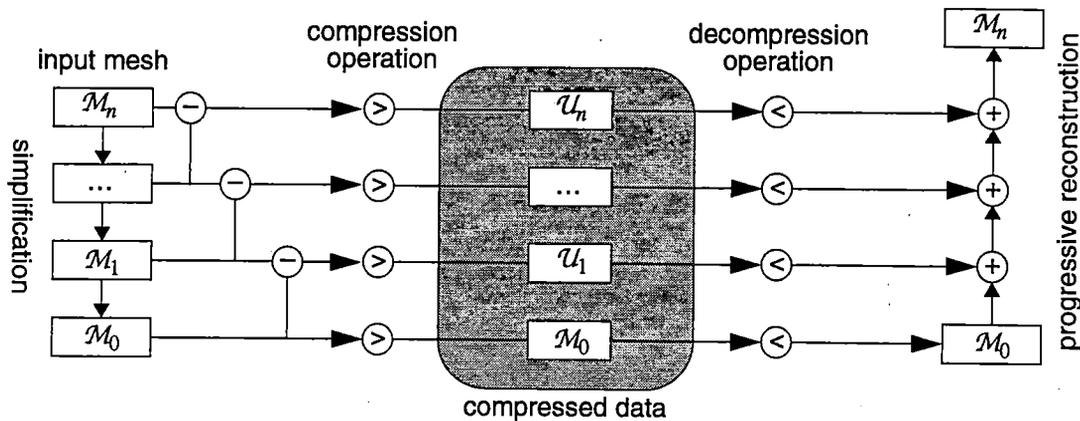


FIGURE 3. Compression and progressive reconstruction. The differences of successively simplified meshes are compressed and transmitted in inverse order of the simplification process. Reconstruction starts with an initial crude mesh and progressively refines it according to the decompressed incremental updates.

#### 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 [PR99] and successfully extended to progressive tetrahedral meshes in [PRS99]. SQUEEZE creates a series of meshes  $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_n$  where each update  $\mathcal{U}_i = \mathcal{M}_{i-1} \rightarrow \mathcal{M}_i$  between consecutive LODs  $\mathcal{M}_{i-1}$  and  $\mathcal{M}_i$  consists of multiple vertex split operations. In the course of this paper  $\mathcal{U}_i$  will denote a batch of vertex split refinement operations, and  $\mathcal{U}_i^{-1}$  a set of edge collapse simplification operations, see also Figure 4 for graphical examples.

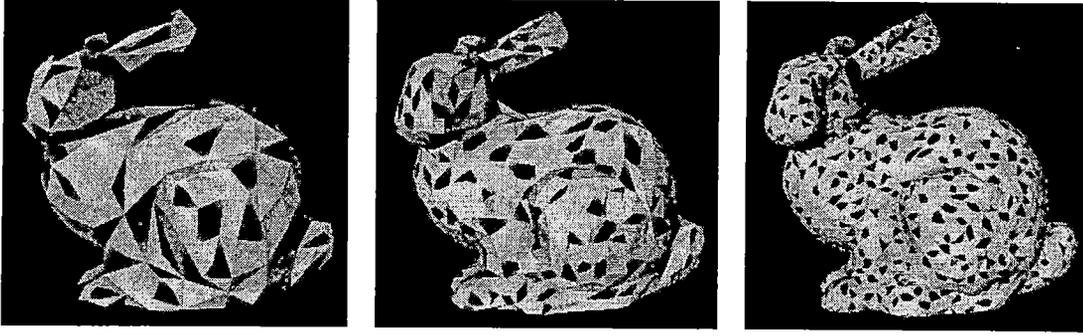


FIGURE 4. Batches of vertex split refinement operations of three different mesh updates (not consecutive in this example). The two triangles inserted by a vertex split are highlighted in each image.

The format of the compressed mesh consists of the initial coarse base mesh  $\mathcal{M}_0$ , encoded using a single-resolution mesh compression method such as [Ros99], followed by a series of compressed updates  $\mathcal{U}_i$ . Following we describe the encoding of a batch  $\mathcal{U}_i^{-1}$  of edge collapse operations, and how such a simplification batch can be constructed during compression of the mesh. Decompression starts with decoding the coarse base mesh  $\mathcal{M}_0$ , and then applying the inverse of the encoding steps described below to every compressed update  $\mathcal{U}_i$ .

The encoding of a mesh update  $\mathcal{U}_i$ , recovering  $\mathcal{M}_i$  from the previous mesh  $\mathcal{M}_{i-1}$ , requires specifying all split-vertices and their cut-edges in  $\mathcal{M}_i$ , and the coordinates of the newly inserted vertices. The following steps are performed to encode an update batch  $\mathcal{U}_i$ :

1. Construct and traverse a vertex spanning tree of  $\mathcal{M}_{i-1}$  and mark all split-vertices – the results of applying the edge collapses  $\mathcal{U}_i^{-1}$  to mesh  $\mathcal{M}_i$ .

For every marked split-vertex  $v_{\text{split}}$ , 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  $v_{\text{split}}$ , clockwise, starting with the edge from  $v_{\text{split}}$  to its parent in the vertex spanning tree.
3. Given the degree  $d$  of the split-vertex in mesh  $\mathcal{M}_i$ , 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  $\lceil \log_2 \binom{d}{2} \rceil$  bits.

Since SQUEEZE uses the half-edge collapse operator the split direction has to be specified:

4. Using one bit we distinguish between  $v_{\text{split}} = v_1$  or  $v_{\text{split}} = v_2$  (see also Figure 2).

Furthermore, the geometry has to be encoded:

5. The displacement vector  $v_{\text{disp}} = v_{\text{new}} - v_{\text{split}}$  is compressed as described in Section 6.

## 5. Simplification process

Coupled with the encoding method mentioned previously is the simplification process that generates the different LODs  $\mathcal{M}_n, \mathcal{M}_{n-1}, \dots, \mathcal{M}_0$  of decreasing accuracy. Each simplification step  $\mathcal{U}_i^{-1} = \mathcal{M}_i \rightarrow \mathcal{M}_{i-1}$  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  $\mathcal{M}_i$  have to be chosen in such a way that the vertex splits can uniquely be identified in  $\mathcal{M}_{i-1}$ . The following three topological requirements for grouping edge collapses in a batch  $\mathcal{U}_i^{-1}$  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  $\mathcal{M}_i$ .
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  $\mathcal{M}_i$ ,  $e_1$  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 [GH97] to order edge collapses according to their introduced approximation error, however, another edge collapse ordering [HRD<sup>+</sup>93, Hop96, LT98] could be used as well. In every simplification batch  $\mathcal{U}_i^{-1}$ , a maximal subset of the least expensive edges in  $\mathcal{M}_i$  that do not violate the topological constraints defined above are greedily selected and collapsed. The following Algorithm 1 provides the outline for the simplification and connectivity encoding process.

```

PROCEDURE Encode (mesh: Triangulation);
VAR v: Vertex; e: Edge; ecol: EdgeSet; list: SortedList
BEGIN
  WHILE mesh not simplified enough DO
    (* sort edges according to approximation error *)
    list := SortedList.EmptyList;
    FOREACH e IN mesh.halfEdges()
      error := mesh.evalError(e);
      list.insert(e, error)
    END;
    (* select maximal group of valid edge collapses *)
    ecol := EdgeSet.EmptySet;
    FOREACH e IN list.ascOrder()
      IF validCollapse(e, mesh, ecol) THEN
        ecol.insert();
      ENDIF
    END;
    (* perform edge collapses, mark split-vertex and
       assign cut-edges to split-vertex *)
    FOREACH e IN ecol.members()
      mesh.collapseAndMark(e)
    END;
    (* traverse mesh, output split-vertex marking bits
       and cut-edges encoding *)
    FOREACH v IN mesh.vertices()
      IF v.marked() THEN
        output(1);
        outputBits(cutEdgesEncoding(v, v.parentVertex, v.cutEdges));
        (* geometry encoding *)
      ELSE
        output(0)
      ENDIF
    END
  END
END Encode;

```

**ALGORITHM 1.** Simplification and connectivity encoding process.

The decoding process performs the inverse steps. It traverses the mesh  $\mathcal{M}_{i-1}$  and reads the marking bits for the vertices. Whenever a split-vertex is detected, the cut-edges are decoded and assigned to that vertex. Furthermore, the displacement coordinates for recovering the other vertex resulting

from the split are decompressed and stored with the split-vertex too. In a second traversal all vertex splits can safely be performed and the next higher resolution mesh  $\mathcal{M}_i$  with better approximation quality can be fully recovered.

## 6. Fast geometry decomposition

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.  $v_{\text{split}} = v_1$ , and the deleted vertex  $v_{\text{del}} = v_2$  must be encoded for recovery. The deleted vertices of one simplification batch  $\mathcal{M}_i \rightarrow \mathcal{M}_{i-1}$  are all geometrically close to the surface of the simplified mesh  $\mathcal{M}_{i-1}$ , thus their values change smoothly over the surface  $\mathcal{M}_{i-1}$ , but the frequency distribution of individual coordinate values is uniform with some noise. The coordinates of local displacement vectors  $v_{\text{disp}} = v_{\text{del}} - v_{\text{split}}$  have a much more skewed distribution. The displacement vector represents a simple vertex prediction error using the old vertex  $v_{\text{split}}$  as the estimate for the new vertex  $v_{\text{del}}$ . In [PR99] and [CLR99] 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 [PR99], we simplified the prediction method to computing the average of direct neighbors of  $v_{\text{del}}$  in the presented approach, similar to [CLR99]. Based on the connectivity decoding the correct mesh topology of  $\mathcal{M}_i$  can be reconstructed without actually knowing the geometric coordinates of the new vertex  $v_{\text{del}}$ , see also Figure 5. Therefore, the decoder can use the same neighbors  $a_1, \dots, a_k$  of  $v_{\text{del}}$  that have been used for compressing the geometry. The estimated position of the deleted vertex is calculated as:

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

The geometry information that is encoded with each vertex split is therefore the difference  $v_{\text{err}} = v_{\text{del}} - v_{\text{est}}$  between actual and estimated vertex positions. During decompression the correct vertex position can be recomputed again by using Equation 1 and decoding  $v_{\text{err}}$ . Decompression speed is mainly determined by the vertex position prediction function, which is greatly simplified by Equation 1 compared to the approach presented in [PR99], and by decoding  $v_{\text{err}}$  for which an efficient solution is presented below.

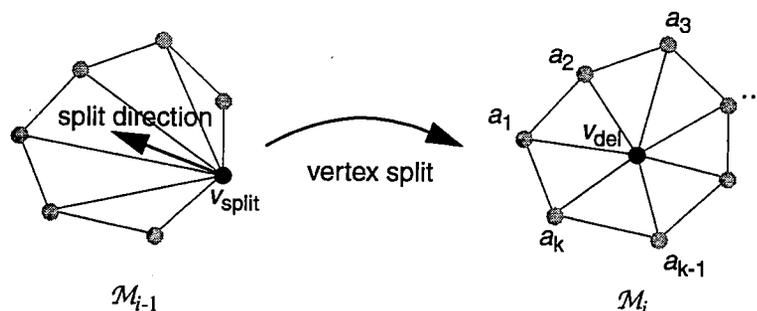


FIGURE 5. Estimating displacement vectors. The new vertex  $v_{\text{del}}$  is estimated as the average of its immediate neighbors  $a_1, \dots, a_k$ , which are known from connectivity decoding.

The frequency distribution of prediction errors 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* [CNW87, Huf52]. The efficiency prob-

lem 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 decoded symbols can be returned. To speed up decoding performance we implemented the high-performance Huffman decoder proposed in [Paj99] that allows to read and process multiple-bit words, i.e. bytes, instead of single bits from the compressed data stream. The approach is similar to the method presented in [Sie88], 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 leaf, and it also provides a list of decoded symbols that have been encountered in the leaves 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 6.

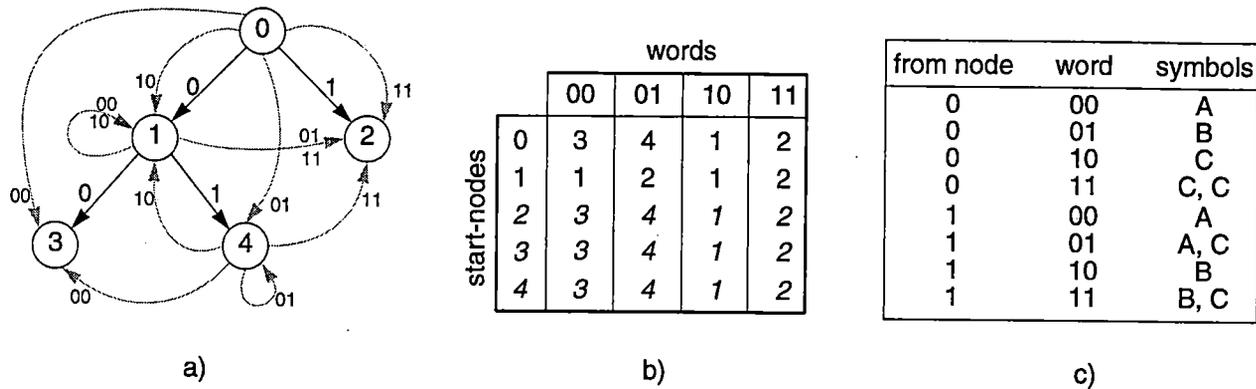


FIGURE 6. a) Huffman code tree with some indicated node transitions using a word size of 2 bits for the codes A=00, B=01 and C=1. b) The complete node transitions table for all nodes indicating the end-node for any given combination of start-node and data word. Note that all leaves have the same *jump table* as the root node and can be left out to save storage. c) Table listing the output symbols for every node transition.

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 [NH95, Sal98]. 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  $\mu$  is 0, and the variance  $\nu$  uniquely defines the shape of the Laplace distribution. The variance is adjusted for every batch by the average of the squared prediction errors,  $\nu = (1/|\text{batch}|) \sum_{v_{\text{err}} \in \text{batch}} (v_{\text{err}} - \mu)^2$ . Thus the only information that has to be sent at the beginning of every batch is the current variance for the Laplace distribution.

$$L_{\nu}(x) = \frac{1}{\sqrt{2\nu}} e^{-\frac{\sqrt{2}}{\nu}|x-\mu|} \quad (\text{EQ 2})$$

Given the frequency distribution defined by the variance  $\upsilon$  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 [How93, Sal98]. 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 performance. Compression efficiency was challenged with one of the best known single-resolution mesh compression techniques [TG98], and compared to various recently developed multiresolution mesh compression methods [PR99, CLR99, BPZ99]. The progressive forest split mesh compression method introduced in [TGHL98] cannot compete in terms of geometry compression to the newer approaches and is not included in the experiments. Also the progressive mesh compression method presented in [Hop98] only reported tests on highly quantized meshes, and it does not provide advanced connectivity compression. The progressive approach presented in [FMP98] needs even more bits than the initial progressive meshes representation of [Hop96] and can thus not compete in terms of compression ratio. Decompression performance of progressive methods is only reported for [Hop98], and thus we also compare our results to single-resolution compression methods [FMP98, GS98].

Tables 1 and 2 evaluate the compression efficiency of our approach compared to other methods mentioned in the literature. The proposed method is competitive with the single-resolution compression approach of [TG98] in terms of geometry compression. The added functionality of progressive multiresolution mesh reconstruction comes at higher cost for connectivity encoding due to the complexity of the multiresolution model. Our approach outperforms other multiresolution compression methods [CLR99, BPZ99] mainly in geometry encoding. Note that our method and the method in [CLR99] use a very similar geometry prediction and error coding mechanism. However, in [CLR99] the simplification process is not driven by an error metric, and their entropy coder depends on symbol statistics. The compressed progressive meshes (CPM) method presented in [PR99] compresses slightly better on average than the presented approach, however, at much higher processing cost for geometry prediction.

				[TG98]		[CLR99]		fast	
quantization	models	vertices	bits / $\Delta$ for connectivity	bits / vertex for coordinates	bits / $\Delta$ for connectivity	bits / vertex for coordinates	bits / $\Delta$ 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 [TG98] provides highest compression ratios for single-resolution meshes, whereas the approach of [CLR99] shows smallest connectivity encoding for progressive meshes and also efficient geometry compression.

quantization	models	vertices	[BPZ99]		CPM [PR99]		fast	
			bits / $\Delta$ for connectivity	bits /vertex for coordinates	bits / $\Delta$ for connectivity	bits /vertex for coordinates	bits / $\Delta$ 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 a previous version of the proposed method. [PR99] has a more sophisticated and more complex vertex prediction function than the proposed solution.

Decompression speed performance is reported in Table 3 for various test models, measured on a 200MHz R10000 SGI O2 machine. While decoding speed varies only slightly and 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 [Hop98] timings are not presented for the advanced arithmetic coding version, but only for a *gzip* compression of vertex split records which has a much lower compression ratio compared to other approaches. Nevertheless, our decoding performance of about 90000 decoded vertex split records per second compares favorably with the 80000 reported in [Hop98],<sup>1</sup> considering the much better compression ratio that is achieved. In [FMP98] decompression speed has been reported for a single-resolution mesh coding method, achieving roughly 35000 triangles per second.<sup>2</sup> Much higher connectivity decoding speed was reported in [GS98] with up to 800000 triangles per second.<sup>3</sup> Both of these approaches do not include geometry coding which dominates decompression speed. Considering that the presented approach and the reported timings include the imperative geometry information and provide a much higher complexity in functionality – progressive reconstruction of a multiresolution model, our approach can easily match [FMP98] and is a convincing alternative to [GS98] when progressivity and speed are important.

Models	Quantization	Model size		Vertex splits per second	
		base mesh	vertex splits	decoding	mesh update
triceratops	10 bit	53	2779	92633	21377
	12 bit	53	2779	138950	15438
fohe	10 bit	124	3881	64683	18480
fandisk	10 bit	89	6386	91228	18246
blob	12 bit	137	7899	78990	16806
	8 bit	126	7910	98875	18833
bunny	12 bit	824	34010	62981	11687
phone	12 bit	1403	81641	68034	9438

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

1. 200 MHz Pentium
2. SGI Indigo2
3. 175 MHz R10000 SGI O2

Figure 7 shows the different test models, and results from SQUEEZE. Indicated are the number of bits needed in SQUEEZE to represent the mesh  $\mathcal{M}_i$  and the time needed for transmission using a 56Kb/s connection. Note that the bits and time of  $\mathcal{M}_i$  include all previous LODs  $\mathcal{M}_j$  for  $j = 0$  to  $i$ . In roughly one-tenth of the time and size of the full resolution model  $\mathcal{M}_{10}$  SQUEEZE provides already 6 progressive LODs  $\mathcal{M}_0$  to  $\mathcal{M}_5$ .

With an average speed of 13'000 decoded and applied vertex splits per second, thus reconstructing the mesh at a rate of 26'000 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 (200MHz CPU) for connections with a transmission rate of 286Kb/s.

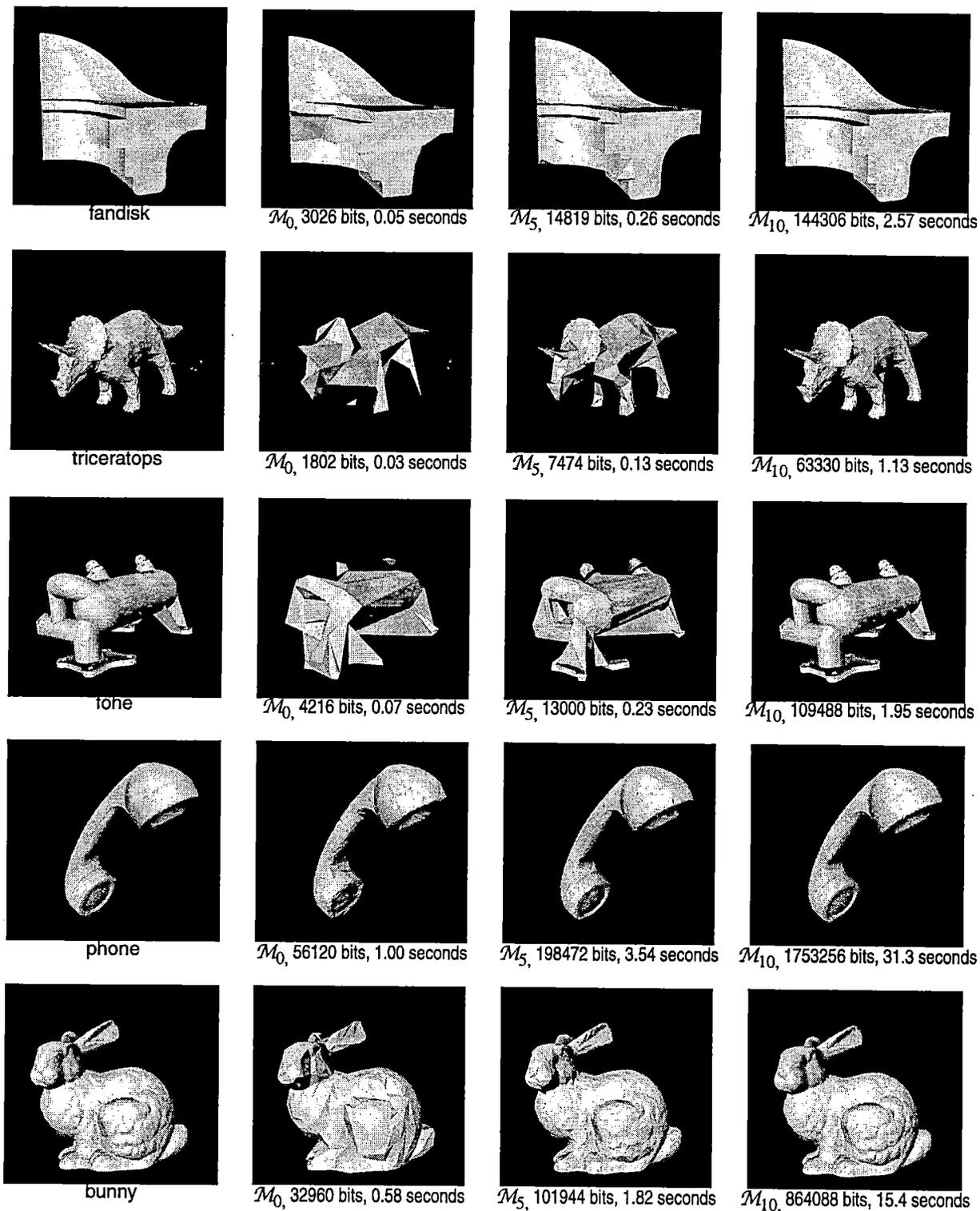


FIGURE 7. Original test models shown in the first column, followed by the base mesh  $\mathcal{M}_0$ , an intermediate representation  $\mathcal{M}_5$  and the full resolution model  $\mathcal{M}_{10}$  of SQUEEZE. Bits and transmission time of  $\mathcal{M}_i$  include all prior LODs, and time is estimated for a 56Kb/s communication.

## 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.

## References

- [BPZ99] Chandrajit L. Bajaj, Valerio Pascucci and Guozhong Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *Proceedings Visualization 99*, pages -. IEEE, Computer Society Press, Los Alamitos, California, 1999.
- [CNW87] 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.
- [CLR99] Daniel Cohen-Or, David Levin and Offir Remez. Progressive compression of arbitrary triangular meshes. In *Proceedings Visualization 99*, pages -. IEEE, Computer Society Press, Los Alamitos, California, 1999.
- [Dee95] Michael Deering. Geometry compression. In *Proceedings SIGGRAPH 95*, pages 13–20. ACM SIGGRAPH, 1995.
- [FMP98] 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.
- [GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings SIGGRAPH 97*, pages 209–216. ACM SIGGRAPH, 1997.
- [Gue99] André Guézic. Locally toleranced surface simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):168–189, April-June, 1999.
- [GS98] Stefan Gumhold and Wolfgang Strasser. Real time compression of triangle mesh connectivity. In *Proceedings SIGGRAPH 98*, pages 133–140. ACM SIGGRAPH, 1998.

- [HG97] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. In *SIGGRAPH 97 Course Notes 25*. ACM SIGGRAPH, 1997.
- [HRD<sup>+</sup>93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *Proceedings SIGGRAPH 93*, pages 19–26. ACM SIGGRAPH, 1993.
- [Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings SIGGRAPH 96*, pages 99–108. ACM SIGGRAPH, 1996.
- [Hop98] Hugues Hoppe. Efficient implementation of progressive meshes. Technical Report MSR-TR-98-02, Microsoft Research, 1998. (also in SIGGRAPH 98 Course Notes 21)
- [How93] Paul G. Howard. *The Design and Analysis of Efficient Lossless Data Compression Systems*. PhD thesis, Department of Computer Science at Brown University, 1993.
- [Huf52] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. Inst. Electr. Radio Eng.*, pages 1098–1101, 1952.
- [LT98] 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.
- [LT99] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April-June, 1999.
- [NH95] Arun N. Netravali and Barry G. Haskell. *Digital Pictures: Representation, Compression and Standards*. Plenum Press, New York and London, second edition, 1995.
- [Paj99] Renato Pajarola. Fast Huffman code processing. submitted for publication, 1999.
- [PR99] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. Technical Report GIT-GVU-99-05, GVU Center, Georgia Institute of Technology, 1999.
- [PRS99] Renato Pajarola, Jarek Rossignac and Andrzej Szymczak. Implant Sprays: Compression of progressive tetrahedral mesh connectivity. In *Proceedings Visualization 99*, pages –. IEEE, Computer Society Press, Los Alamitos, California, 1999.
- [RR96] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. *IEEE Computer Graphics Forum*, 15(3):C67–C76, August 1996.
- [Ros99] Jarek Rossignac. Edgebreaker: Compressing the incidence graph of triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, January-March, 1999.
- [RB93] 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.
- [Sal98] David Salomon. *Data compression: the complete reference*. Springer-Verlag, New York, 1998.
- [Sie88] Andrzej Sieminski. Fast decoding of the huffman codes. *Information Processing Letters*, 26(5):237–241, January 1988.
- [TGHL98] Gabriel Taubin, André Guézic, William Horn and Francis Lazarus. Progressive forest split compression. In *Proceedings SIGGRAPH 98*, pages 123–132. ACM SIGGRAPH, 1998.

## References

---

- [TR98] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.
- [TG98] Costa Touma and Craig Gotsman. Triangle Mesh Compression. In *Proceedings Graphics Interface 98*, pages 26–34, 1998.