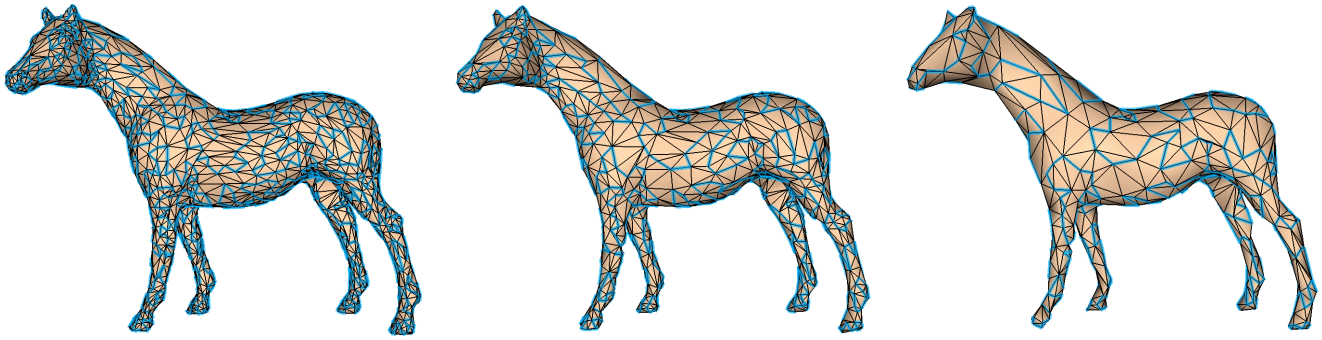
Efficient Compression and Rendering of Multi-Resolution Meshes

Zachi Karni[1] Alexander Bogomjakov[2] Craig Gotsman[3]

Center for Graphics and Geometric Computing

The Faculty of Computer Science

Technion – Israel Institute of Technology



(a) (b) (c)

Figure 1: Locality and continuity-preserving vertex sequences at different resolutions: (a) n=2,048 vertices. (b) After 1,024 edge collapses, n=1,024 vertices. (c) After another 512 edge collapses, n=512 vertices.

Abstract

We present a method to code the multiresolution structure of a 3D triangle mesh in a manner that allows progressive decoding and efficient rendering at a client machine. The code is based on a special ordering of the mesh vertices which has good locality and continuity properties, inducing a natural multiresolution structure. This ordering also incorporates information allowing efficient rendering of the mesh at all resolutions using the contemporary vertex buffer mechanism. The performance of our code is shown to be competitive with existing progressive mesh compression methods, while achieving superior rendering speed.

Keywords: progressive compression, wavelets, geometry coding, rendering.

1 INTRODUCTION

In a typical Internet-based client-server 3D application, the two most important requirements are to transmit a geometric model quickly from the server to the client, and to render the model at interactive frame rates at the client. One way of achieving fast transmission is to employ a compact progressive representation of the mesh, meaning that a rough version of the mesh is reconstructed after a small number of bits have been received at the client, and this gradually improved as more bits arrive. The objective is to achieve the best fidelity possible with the smallest number of bits.

3D meshes consist of two main components: geometry and connectivity. Geometry describes the coordinates in 3D space of the mesh vertices, and connectivity describes the manner in which these vertices are connected in faces to form the mesh surface. In a progressive model, a rough version of the mesh will typically contain a smaller number of vertices and faces than the original mesh, so both the geometry and connectivity components will be smaller than those of the full resolution version. This reduction in polygon count alleviates the rendering load on the client at the beginning of the transmission, but an increase in rendering load is inevitable as the mesh evolves.

[1]zachik | [2]alexb | [3]gotsman]@cs.technion.ac.il

To provide better frame rates, manufacturers of modern graphics hardware have introduced vertex buffers into the graphics pipeline, which allow reuse of the mesh vertex data using a FIFO cache. The reuse saves geometric projection and vertex shading operations. Smart use of the vertex buffer (through extensions to OpenGL and Direct3D) can potentially accelerate rendering rates by a factor of six. This is achieved by rendering the mesh faces in a special order which maximizes the cache hits. Finding this optimal *rendering sequence* is not easy, and has been the subject of recent research. In any case, it is time-consuming and must be done at the server in a preprocessing stage. This means, however, that the rendering sequence information must be integrated into the transmitted mesh code, possibly lengthening it. This is the price paid for efficient rendering at the client. Furthermore, since the mesh evolves at the client decoder, with concomitant increase in face count, the rendering sequence should also evolve with it, facilitating efficient rendering at any resolution. All progressive coding methods for 3D meshes reported to date do not provide any support for efficient rendering.

The objective of this work is a method to code a 3D mesh in a manner that incorporates in a single bit stream both information on the multiresolution structure of the mesh and its rendering sequence at all resolutions. We will show that the efficiency of this bit stream is comparable to those of existing methods for progressive mesh compression, and at the same time provides a significant benefit in rendering, and is easy to implement. There is an intimate connection between our method and one-dimensional Haar wavelet coding. Furthermore, apart from a preliminary quantization of the mesh geometry, the decoding is completely without loss, namely the mesh reconstructed at highest resolution at the decoder is *identical* in connectivity and geometry to the original at the encoder.

2 RELATED WORK

2.1 Progressive Coding of Meshes

Progressive meshes were first introduced by Hoppe [15], whose main objective was to accelerate the rendering process by using low-resolution versions of the mesh when full resolution was not justified, e.g. when the viewer was distant from the object. The

refinement of the mesh was achieved through a series of so-called *vertex splits,* each split adding one vertex, two faces and three edges to the mesh. Each split in the sequence was carefully chosen using a *geometric* criterion in order to maximize the gain in the geometric approximation to the full-resolution mesh.

Realizing that progressive meshes were also useful for efficient transmission, Taubin et al. [25] showed how to code them in a compact manner. They concentrated mostly on coding the connectivity, using a technique called forest-splits. The commercial Metastream format [1] allowed mesh refinement through the more general mechanism of *vertex-insertion*. Pajarola and Rossignac [23] used vertex splits, and Cohen-Or et al. [8] also used vertex insertion. In the latter two methods, the geometry was coded by predicting vertex coordinates based on its neighbors. Bajaj et al. [3] use the more general *triangle-split* operator.

A rich area of research is the coding of single-resolution meshes. The main emphasis has been placed on coding the connectivity component, although it is obvious that the code is dominated by the geometric component. For coding connectivity, a wealth of algorithms exist, all of which code the edge structure by "conquering" the mesh, one element at a time. Depending on the type of element, the algorithms may be loosely classified into three main categories: face (e.g. [24]), edge (e.g. [18]) and vertex-based (e.g. [26]). The vertex-based strategies seem to be the most efficient.

Alliez and Desbrun [2] show how to code progressive meshes using a somewhat complicated extension of the single-resolution vertex-based connectivity-coding method of Touma and Gotsman [26]. Here, as in the method of Cohen-Or et al. [8], the refinement procedure is by connectivity-based vertex insertion, meaning that the next vertex to be inserted in the refinement process is chosen based on its connectivity properties, rather than its geometric properties. This allows optimization of the connectivity portion of the code at the possible expense of geometric fidelity.

Khodakovsky et al. [21] postulated that the precise connectivity structure of a mesh is unimportant as long as the geometric shape of the mesh is preserved. Based on that, they *remesh* a model to have subdivision connectivity, so that wavelet-style decomposition may be applied to the new mesh. This gives a natural multiresolution mesh decomposition, but at the price of a totally different connectivity.

Devillers and Gandoin [11] build a simple multiresolution structure on the geometry of the mesh using a three-dimensional k-D tree. Given a set of points in 3D space, this set may be recursively partitioned into two halves (alternating between the *x,y,z* dimensions), represented by two sons in a binary tree, and each subset of points in the tree nodes represented by the centroid of the 3D cell corresponding to that node. The mesh at that intermediate resolution is constructed on these centroids. Interestingly enough, it turns out that each recursive partition of a cell into two smaller cells is equivalent to a vertex split in the connectivity structure.

The spectral compression method of Karni and Gotsman [19] compresses the mesh geometry by projecting it on orthogonal basis functions which are the eigenvectors of the mesh connectivity Laplacian matrix. Ordering the eigenvectors by their eigenvalues ranks them by importance (analogous to frequency of Fourier basis functions), and for many meshes, in particular smooth ones, the projection coefficients decay rapidly in this order. Thus it is possible to progressively build up an approximation of the mesh geometry by incrementally adding more and more basis vectors to the sum. However, the connectivity remains at full resolution during the entire process, hence the number of triangles is maximal, even though the mesh is "low-resolution" in terms of the basis expansion. Another major drawback of the spectral approach is that it is impractical to compute the eigenvectors of large mesh connectivities, especially at the decoder. Although this may be alleviated somewhat by partitioning the mesh into a number of submeshes, that solution introduces artifacts along the boundaries of the submeshes. An attempt to circumvent the computational complexity problem was made by Karni and Gotsman [20] by using a fixed (regular) connectivity structure to construct the basis functions, and map the true connectivity to the fixed one such that the neighborhood relationships are preserved as much as possible. Since the mesh is fixed, the basis vectors may be precomputed. This method was shown to achieve reasonable results, but did not eliminate the use of full connectivity, so the spectral method is not truly progressive.

Comparison of different progressive compression methods is not easy. The main difficulty lies in objectively quantifying the loss in mesh quality at low resolutions – the so-called *distortion* – as a function of the code length – the so-called *rate*. Many of the works to date have not published any rate-distortion curves, probably for lack of an objective distortion measure. In the mesh simplification literature, the *Metro* tool [7] for measuring the Hausdorff distance between two meshes has been used to measure distortion, and we have adopted this in our work, as have some others.

In terms of rates, it is obvious that the number of bits that will have to be transmitted before the mesh can be progressively restored to its original form will be greater than the number of bits required to code the mesh as a single-resolution model. However, a good progressive code should give a very good approximation very rapidly at the beginning of the transmission, so in many applications the transmission may be stopped long before it is complete.

2.2 Mesh Rendering Sequences

Triangles strips are a well-known method of accelerating low-level rendering of triangle meshes by reusing the last two vertices in the pipeline in order to form the next triangle to be rendered. Modern graphics hardware (e.g. the GeForce family of graphics cards for PCs and the Elite3D family of Sun workstations) has taken this one step further by providing a large *vertex buffer* in the form of a vertex cache, first proposed by Deering [9]. This buffer allows a larger (typically 16) number of vertices to be reused, but to maximize benefit from this means that the triangles must be rendered in a very special order. This special order is called a *rendering sequence* for the mesh. The performance of a given rendering sequence is measured by its ACMR (Average Cache Miss Ratio), which is the average number of cache misses incurred per triangle while rendering the mesh in the order prescribed by the rendering sequence. It can be anywhere between 0.5 and 3. From a theoretical point of view, Bar-Yehuda and Gotsman [5] have shown that a cache of size $\Omega(\sqrt{n})$ is necessary to render *any* n-vertex mesh with maximal vertex reuse, i.e. ACMR = 0.5. They further showed that an ACMR of $0.5+O(1/k)$ is achievable for a cache of size k. Hoppe [16] showed how to construct good rendering sequences when the cache behavior is FIFO (as in the GeForce cards) and the cache size is known. Bogomjakov and Gotsman [6] showed how to construct so-called *universal* rendering sequences for FIFO caches, which are suitable for *any* cache size, and once generated for a mesh at maximal resolution, may be easily updated on-the-fly to render any lower resolution version of the mesh obtained by a sequence of vertex removals from the original. The key idea of their construction is that the rendering sequence be a *locality-preserving* ordering of the mesh faces, which is built recursively and traverses the mesh faces in a "winding" order at all scales.

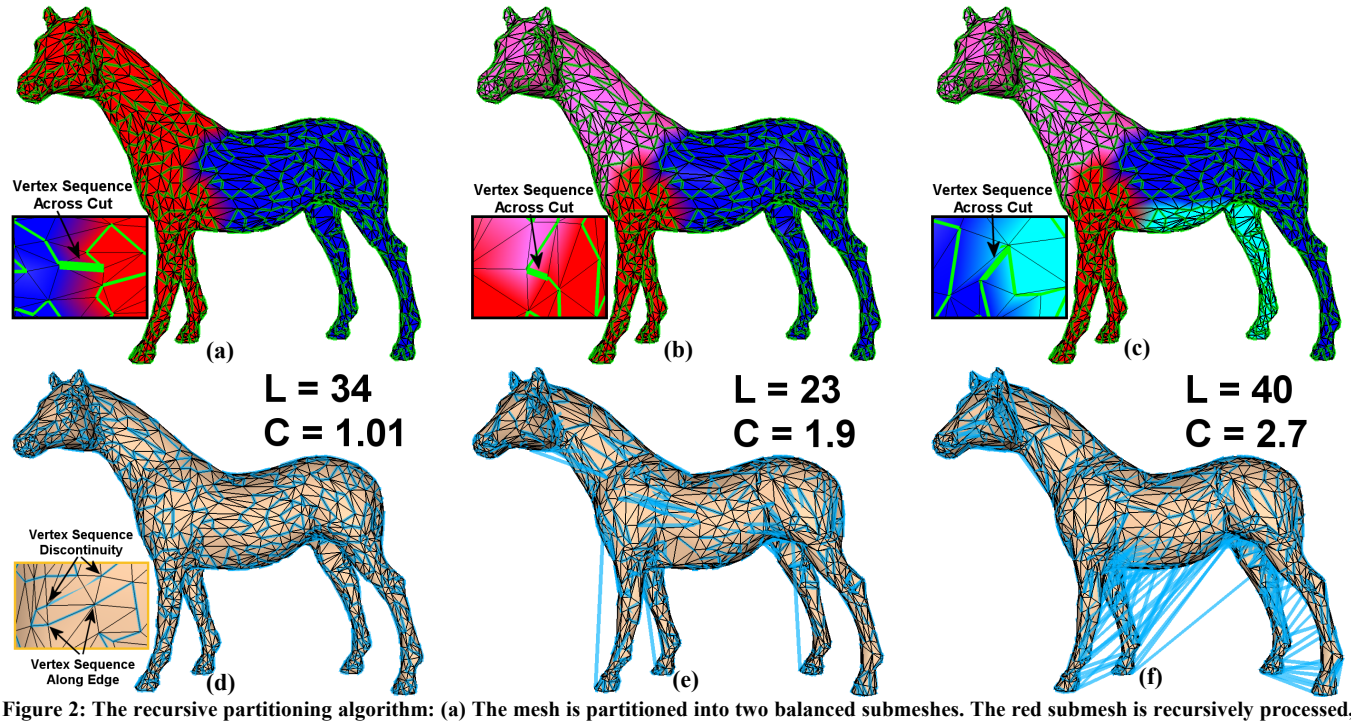


Figure 2: The recursive partitioning algorithm: (a) The mesh is partitioned into two balanced submeshes. The red submesh is recursively processed, and then the blue. (b) The red submesh of (a) is recursively processed by partitioning into red and pink submeshes. (c) The blue submesh of (b) is recursively processed by partitioning into blue and cyan submeshes. (d) Cyan vertex sequence and corresponding measures of locality and continuity generated by our recursive partitioning algorithm on the horse mesh of 2,048 vertices. (e) Vertex sequence generated by MLA algorithm. (f) Vertex sequence generated by GPS algorithm.

2.3 Combining Compression with Efficient Rendering

Progressive meshes are useful both for rapid transmission and for efficient rendering. A low resolution version of the mesh can be a good approximation of the original, and at the same time require fewer bits to represent it and ease the rendering load on the graphics hardware since its polygon count is relatively low. Combining this with optimized rendering of the mesh using a vertex cache is an all-round win.

All relevant work to date addresses only the problem of efficiently coding a progressive representation of a triangle mesh generated using some connectivity-based or geometry-based simplification algorithms, and does not support vertex cache rendering. Hence they cannot benefit from this potential rendering speedup. Nevertheless, if the simplification algorithm used to generate the progressive mesh is superior (for example, geometry-based), for a given bit rate and distortion error – the algorithm might be able to supply an approximation with a very low polygon count. So while the code does not support the efficient use of a vertex cache, this may be somewhat compensated for by the lower polygon count.

Thus, in progressive compression schemes there is a multi-dimensional tradeoff between distortion, bit rate and rendering speed. The rendering speed manifests in the number of vertex cache misses incurred while rendering, which is a function of both the polygon count of the mesh and the ACMR of the rendering sequence used. The essence of our method is to provide a rate-distortion tradeoff which is comparable to others, but with a large benefit in rendering speed, due to the much superior rendering sequence implicit in the code.

A conventional progressive mesh method might attempt to run a simple rendering sequence generator in real-time at the decoder as postprocessing after the mesh is decoded, and in this way gain some rendering performance independent of the compression. While this is theoretically possible, in practice even the simplest rendering sequence generator cannot achieve real-time performance. Also, since the mesh is rapidly being refined as bits are received, this rendering sequence must be constantly regenerated or at least updated on-the-fly, to adapt to the new resolution. The method of Bogomjakov and Gotsman [6] to generate a universal rendering sequence suitable for all resolutions is not applicable here, since that method runs with the *highest* resolution connectivity as input. This is not available at the decoder until the entire bit sequence has been received.

A work which is somewhat relevant is that of Isenburg [17]. This describes a method to encode traditional triangle strip information into a single-resolution compressed mesh bit sequence, so that it is available at the decoder for efficient rendering. Again, the reason to incorporate this information into the bitstream, as opposed to generating it independently at the decoder, is that hi-quality triangle strips cannot be generated at interactive rates.

3 LOCALITY AND CONTINUITY-PRESERVING VERTEX SEQUENCES

A 3D surface mesh has an inherent 2D structure, especially if it is manifold. However, the connectivity is usually irregular, so no regular 2D operations (e.g. Fourier, wavelet decomposition) can be applied. If it were possible to embed the connectivity structure in a lower dimensional space, it might be possible to manipulate it more easily. One way to do this is to embed the mesh on the one-dimensional grid (i.e. the integers), such that the 2D neighborhood relationships are preserved as much as possible. This means finding a *vertex sequence*, an ordering of the mesh vertices $\pi: V \rightarrow \{1,..,n\}$ in the mesh $<V,E>$, which minimizes the following measure of *average locality* :

$$L(\pi) = \frac{1}{|E|} \sum_{(i,j)\in E} |\pi(i) - \pi(j)|,$$

meaning that there are not too many long edges between the connected grid points. Variants of this problem are well known in a

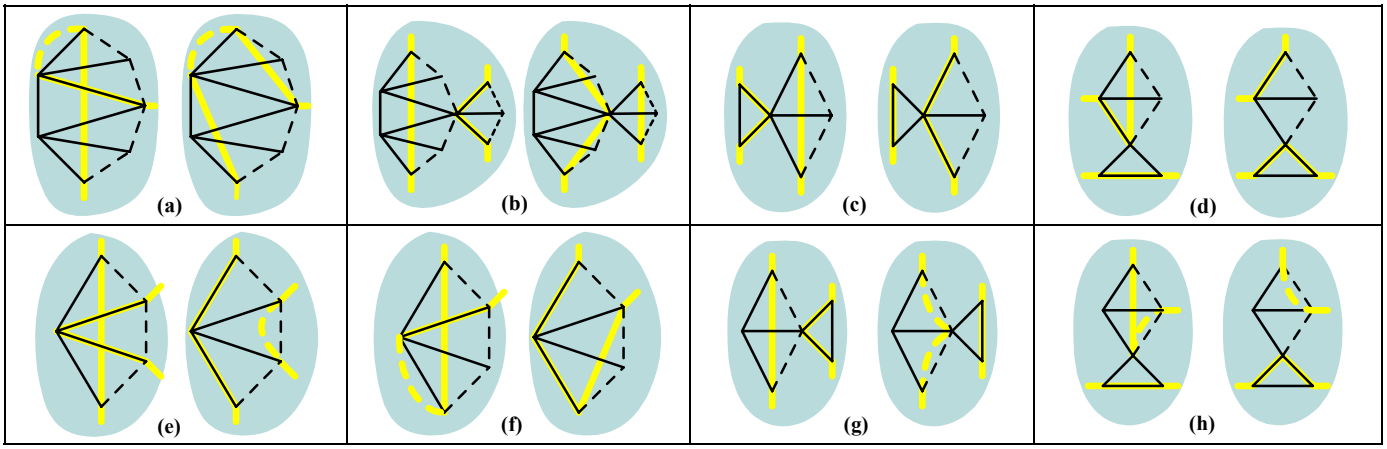


Figure 3: Local reduction of jumps in a vertex sequence. Left in pair is before reduction. Right in pair is after reduction. Yellow line is vertex sequence. Dashed portion is an arbitrary vertex subsequence connecting the two vertices (with other vertices possibly inbetween). Dashed black line means that other vertices (and other edges) may be present. (a) Jump of size three reduced to two jumps of size two. (b) Jump of size three reduced to three jumps of size two. (c)-(d) Jumps of size two which are completely eliminated. (e)-(h) Jumps of size two whose cut is reduced.

number of contexts. In graph theory, this problem is called *minimal linear arrangement* (MLA), and is known to be NP-hard. Various approximation strategies [4] exist, and the *recursive-partitioning* technique we use is similar in spirit to those employed in graph algorithms, inspired by the classical recursive space-filling-curve constructions on a regular grid (e.g. the Hilbert and Peano curves) which have been shown to preserve locality [13].

To construct a vertex sequence with good locality properties, process the mesh as follows: At the beginning, all vertices are unmarked. Partition the mesh into two balanced connected submeshes with a minimal edge-cut, meaning that each submesh contains approximately half of the vertices of the original mesh, and the number of edges straddling the partition is small. Select a *transition* from an unmarked vertex in the first submesh to an unmarked vertex in the second, preferably along an edge, and mark these two vertices. This pair of vertices will be adjacent in the final vertex sequence. Recursively process the first submesh to generate the vertex subsequence preceding this pair of vertices, and then recursively process the second submesh to generate the vertex subsequence succeeding this pair. Terminate when the submeshes are very small. The resulting vertex sequence can be shown to have good locality properties. For a mesh with irregular connectivity, partitioning the mesh into two balanced submeshes with a small edge-cut can be done using the MeTiS graph partitioning package [21]. MeTiS runs in time linear in the mesh size. Unfortunately, MeTiS does not guarantee that the submeshes will be connected, which is important for our application. Hence, in the cases where MeTiS generated an unconnected submesh, we ran a simple partitioner of our own (based on BFS) that does not produce as good balance and edge-cut, but does guarantee connectedness. See Figure 2a-c for an illustration of the recursive partitioning process.

In numerical matrix theory, the related measure:

$$L'(\pi) = \max_{(i,j)\in E} |\pi(i) - \pi(j)|$$

is known as the *bandwidth* of the graph adjacency matrix after reordering. This means that the non-zero entries of the matrix are not too far away from the matrix diagonal. The well known heuristic Gibbs-Poole-Stockmeyer (GPS) algorithm [12] for minimizing bandwidth is implemented in many numerical packages, and may be used as a crude approximation for minimizing L.

The vertex sequences generated by our algorithm are not necessarily *continuous*, in the sense that every two adjacent vertices in the sequence will not necessarily be connected by an edge of the mesh (otherwise the sequence would be a Hamiltonian path). As we will see later, it is advantageous for the sequence to contain a minimal number of *jumps* – adjacent vertices in the sequence not connected in the mesh by an edge. Thus, not only do we want to minimize $L(\pi)$, but also the following measure of *average continuity*:

$$C(\pi) = \frac{1}{|V|}\sum_{i=1}^{n} d(\pi^{-1}(i), \pi^{-1}(i+1)),$$

where $d(u,v)$ is the topological distance between vertices u and v in the mesh connectivity graph (the size of the *jump*). Standard MLA and GPS implementations do not attempt to minimize the sequence continuity. Our vertex sequence generator optimizes C in a separate phase, after generating the sequence with good locality properties. First, we observe that a jump may occur (only) if for every edge straddling the cut at some recursion level, at least one of the vertices incident on the edge has already been marked at previous levels. However, since no more than two vertices can already be marked per submesh (the transition vertices to other submeshes), because of their connectedness, this can happen only if there was just one edge straddling the partition, and at least one of its vertices is marked. In this case it is easy to guarantee that the size of the jump will never be larger than three by selecting a *neighbor* of the marked vertices as the transition vertex. In typical meshes, jumps of size two occur in about 10% of the vertex sequence but jumps of size three are very rare.

The second phase of our vertex sequence generation algorithm attempts to fix these jumps by performing local changes in the vertex ordering. There are only two types of size three jumps, depending on whether the jump "crosses" the rest of the sequence or not. Both of the two types of length three jumps can always be reduced to several jumps of length two; see Figure 3a-b.

Define the *cut* of a size two jump to be the minimal number of mesh edges cut by the jump. The minimal cut of a jump is one, and if d is the degree of the vertex that is being "jumped over", the cut is no larger than $\lfloor (d\text{-}2)/2 \rfloor$ for an internal vertex. There are several cases of jumps of size two, some which may be reduced immediately, as illustrated in Figure 3c-d. Others can be reduced to jumps of size two, but with smaller cuts, which in turn may be further reduced, as illustrated in Figure 3e-h. If the jump does not match any of these cases, it cannot be reduced by a local procedure. After this optimization procedure, the number of jumps of size two is typically less than 1% of the number of vertices.

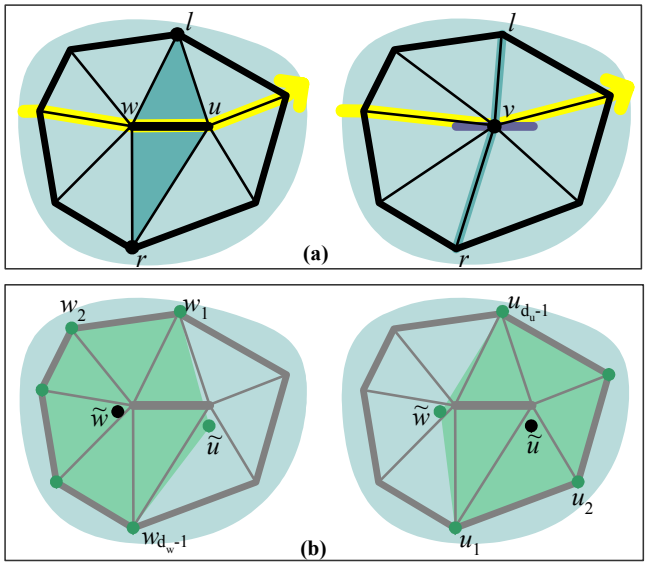


Figure 4: Vertex geometry prediction. (a) Edge (*w*,*u*) along yellow vertex sequence collapsed at encoder to vertex *v*. (b) *u* and *w* predicted to be at centroids of neighbors.

Figure 2d-f shows the vertex sequence generated by our recursive partitioning algorithm on the horse mesh of 2,048 vertices, as compared to those generated by the MLA or GPS algorithms, with their corresponding values of the measures L and C. As is to be expected, the MLA algorithm achieves the best value of L, at the expense of the value of C. The GPS algorithm does not do well on either of the measures. Our recursive partitioning algorithm achieves a good balance between the two. In terms of run time, our unoptimized software for the vertex sequence generator runs for a few seconds on a mesh of 10,000 vertices using a contemporary PC (700 MHz Pentium III with 128MB RAM).

4 MESH CODING

4.1 Multiresolution Geometry Coding

The mesh geometry is quantized once before any coding is applied, typically to 10-12 bits per coordinate. Thereafter the vertex coordinates may be considered integers in a finite range. Our objective is to code this information in a *lossless* manner, meaning that when the full resolution mesh is reconstructed at the decoder, its vertices will have coordinate values identical to those immediately after the quantization at the encoder. The connectivity information will also be perfectly reconstructed.

The coding proceeds by first generating a vertex sequence with good locality and continuity properties, as described in Section 3. Once the mesh vertices have been ordered in this sequence, it is possible to apply a variety of one-dimensional methods to code the three one-dimensional geometric signals – *x*,*y*,*z*. The locality and continuity properties of the ordering guarantee that much of the correlation in the geometry of vertices adjacent in the connectivity graph is retained in the one-dimensional signals. Edge collapses are induced by sequentially collapsing two adjacent vertices in the sequence to their centroid (see Figure 4a). If n is a power of 2, each scan of the sequence reduces the number of vertices by a factor of two, so by scanning the vertex sequence $\lceil \log_2 n \rceil$ times, where n is the number of vertices, all vertices in the end collapse to one point. This procedure may also be described by a binary tree, whose leaves are the mesh vertices, in which each two collapsed vertices create a father node. Each collapse "short-cuts" the (almost) edge path described by the vertex sequence by one edge.

At the decoder, each packet of bits received indicates how some vertex (in the predefined sequence) is to be split to refine the mesh by replacing the vertex with two new ones, increasing the length of the vertex sequence by one. Thanks to the construction of the sequence, the updated vertex sequence maintains the properties of locality and continuity at all scales; see Figure 1.

To code the precise location of the two vertices introduced by a split at the decoder, we use a prediction method similar to that of Pajarola and Rossignac [23]. In a nutshell, if a vertex v splits into vertices u and w, we may write the following two linear equations for $\tilde{u}$ and $\tilde{w}$, the predicted locations of u and w at the encoder:

$$\tilde{u} = \tfrac{1}{d_u}(\tilde{w} + u_1 + u_2 + .. + u_{d_u - 1})$$

$$\tilde{w} = \tfrac{1}{d_w}(\tilde{u} + w_1 + w_2 + .. + w_{d_w - 1}).$$

Here u_i is the neighbor of u, w_i the neighbors of w, d_u the degree of u and d_w the degree of w (see Figure 4b). This is based on the belief that a vertex is located approximately at the center of its neighbors in the mesh. The (3D vector) prediction error that is coded is

$$e = (\tilde{w} - \tilde{u}) - (w - u)$$

after rounding off $\tilde{u}$ and $\tilde{v}$ to the nearest integers. Note that this implies that e is an integer. Hence the geometry code is just a sequence of these e's, one per edge collapse (i.e. one per vertex). This sequence of values is entropy coded, using an adaptive Huffman coder.

At the decoder, the same two equations for $\tilde{u}$ and $\tilde{v}$ are solved and results rounded off, e is retrieved from the bit stream, and the following two additional linear equations solved for the true u and w, based on the existing vertex v:

$$\tfrac{1}{2}(u + w) = v$$

$$w - u = (\tilde{w} - \tilde{u}) - e.$$

4.2 Connectivity Coding

First observe that the vertex sequence used to code the geometry already encodes approximately one third of the connectivity information (since |E|=3|V| for a typical manifold triangle mesh), so the identity of the edge introduced at the decoder by the vertex split is known. This means that we already gain some savings

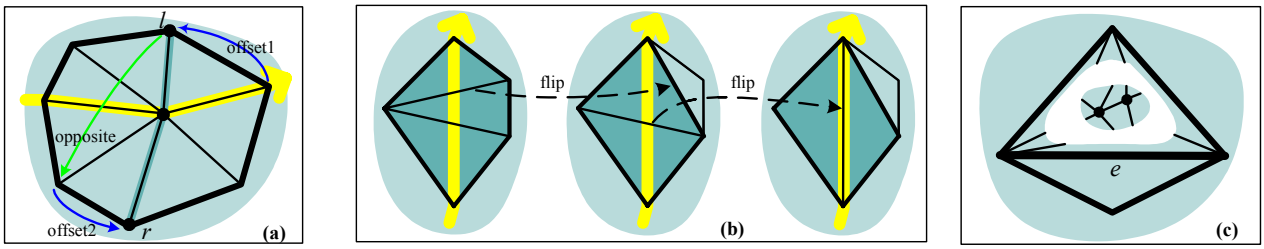


Figure 5: Connectivity coding. (a) Predicting and coding the positions *l* and *r* of the two edges affected by the split operation as two integer offsets. (b) Coding a non-continuous split as a sequence of edge flips. (c) Collapsing edge *e* is illegal, as then all the triangles interior to the top outer triangle will collapse.

relative to other progressive coding methods (e.g. that of Pajarola and Rossignac [23]), who code this information explicitly. But we still have to code the identity of the other edges affected by the vertex split. This may be done using two integer values, denoting the indices of the two affected edges in the "star" of the vertex, i.e. the set of edges incident on the split vertex. It is even possible to predict these values effectively, and thereby reduce the entropy of their distribution. We do this by observing that the two edges adjacent on the vertex which expand to triangles when the vertex is split are usually opposite in the vertex star. Hence the identity of these edges in the star may be coded by the (counter-clockwise) offset of the first edge relative to the sequence edge in the star (l), predicting the second edge to be its opposite (at offset $\lfloor d/2 \rfloor$ from the first), and then the offset of the true edge relative to this predicted position (r); see Figure 5a.

However, the vertex sequence might not be continuous along a mesh edge, and the corresponding split at the decoder would create a redundant edge unless this fact is recorded in the code. In this case we code a series of edge flip operations that transform the illegal split to a legal one (see Figure 5b) and are performed in reverse at the decoder.

Other cases where an edge collapse is illegal occur; see Figure 5c. These pathological collapses are not allowed, as they would create degenerate triangles, and this fact is also recorded in the code. The edges accumulated in this manner are left in a "base mesh" which is coded separately as part of the bit stream header, using the Touma-Gotsman [26] mesh coder.

Adaptive Huffman entropy coding of all the relevant symbols results in an average of 4.5 bits/split, which means that connectivity coding costs 4.5 bits/vertex. With better (arithmetic) low-level coding, this can probably be reduced to the true entropy of the symbol distribution, which was measured to be less than 4 bits/vertex. This contrasts with the 7 bits/vertex (3.5 bits/triangle) reported for the CPM method of Pajarola and Rossignac [23].

5 GENERATING THE RENDERING SEQUENCE

When the mesh is partially decoded to some resolution at the decoder, the mesh vertices are available, ordered in a sequence corresponding to that resolution with good locality and continuity properties. This is thanks to the vertex sequence (as in Figure 1a-c). To render the mesh efficiently, this ordering of the mesh vertices must be converted into a rendering sequence for the mesh faces, as described in Section 2.2. The following simple method gives very good results: Scan the mesh along the vertex sequence, and at each vertex, in turn, render all faces incident on this vertex, starting at the faces incident on the edge from which the vertex was approached (along the vertex sequence). By marking the faces, avoid rendering those which were already rendered when previous vertices were visited. It is important to notice that both the decoding and rendering may be done in one pass, as all triangles of the rendering sequence generated as described above will be available when they are needed. Since the vertex sequence has good locality and continuity properties at all resolutions, it stands to reason that the rendering sequence will also have a good ACMR at all resolutions; see Section 6 for experimental results.

6 EXPERIMENTAL RESULTS

Our encoder is relatively simple to implement and our (non-optimized) software encodes a 10,000 vertex mesh in a few seconds on a contemporary PC (700MHz Pentium III with 128MB RAM). The decoder is also very simple, and runs in real-time.

Since the rationale behind working with multi-resolution models is that lower resolution is good for both coding and rendering,

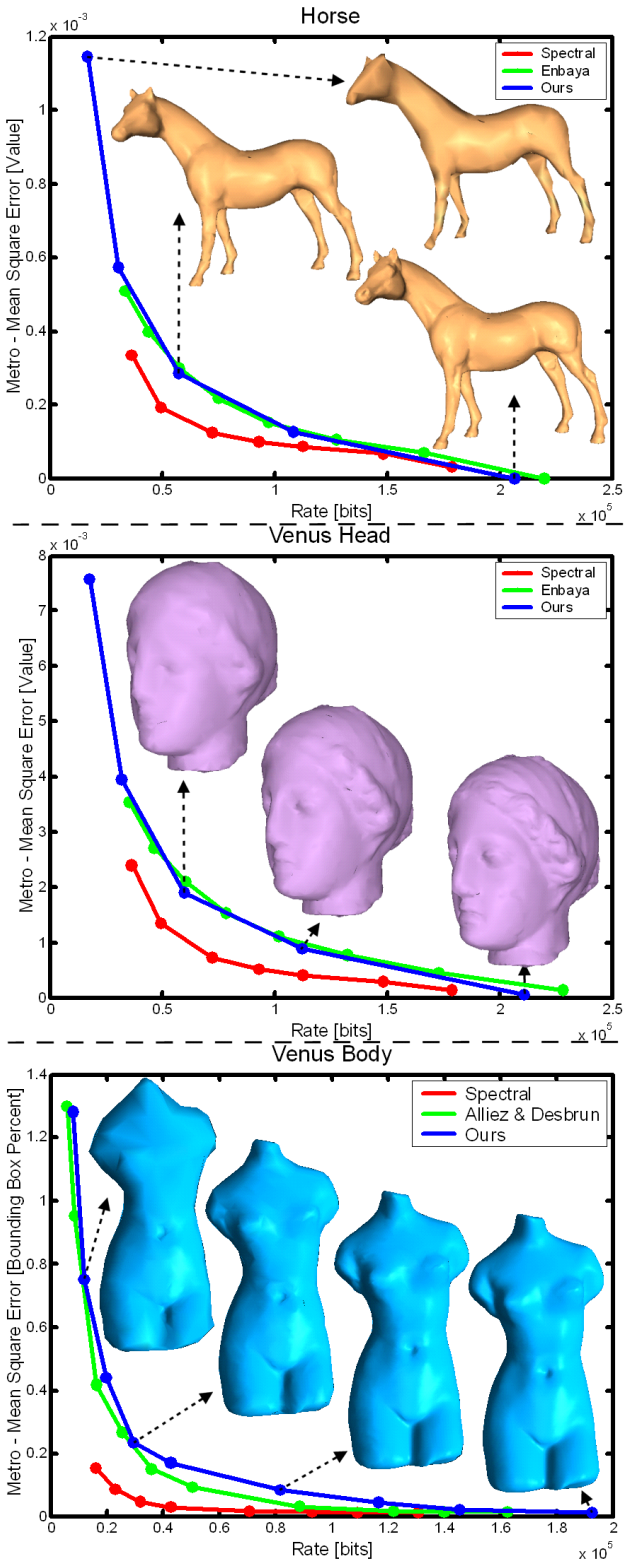


Figure 6: Rate-distortion curves comparing coding efficiency of various progressive mesh compression algorithms.

especially if it is only at a minor cost in quality, the ultimate comparison of our methods with others is through the behavior of various performance measures as a function of the mesh distortion. Hence we measured, as a function of the distortion, the bit rate, the number of triangles, and the number of cache misses at a variety of points along the resolution scale. The first measures compression efficiency, the second measures simplification efficiency, and the third measures rendering efficiency. We measured these for ours and others algorithms for which we were able to obtain results or working sofware. Unfortunately, these included only those of Alliez and Desbrun [2] and of Cohen-Or et al [8]. The latter is available through a commercial version of their algorithm implemented in the products of Enbaya Ltd.

Distortion was measured as the Hausdorff distance between the reconstruction at a given bitrate and the original model, as given by the Metro software [7], and the bitrate as the size of the file generated by our software. Figure 6 shows rate-distortion curves of our algorithm and various other algorithms on a few well-known 3D models where the geometry has been pre-quantized to 10 or 12 bits per coordinate. Our belief that the (not-so-practical) spectral method of Karni and Gotsman [19] should give the best rate-distortion curves is verified (using 16 bit quantization of the spectral coefficients). Our method exhibits performance comparable to that of Cohen-Or et al. on the horse and Venus head meshes, but the Allliez and Desbrun method performs slightly better than ours on the Venus body mesh. This is probably due to the high degree of regularity of this mesh, of which that algorithm takes advantage. In comparison, single-resolution coding algorithm of Touma and Gotsman [26] requires 183,912 bits for the horse mesh of 8,192 vertices (geometry quantized to 12 bits/coord) and 194,208 bits for the Venus head mesh of 8,192 vertices (quantized to 12 bits/coord). This is just 10% less than our (and Cohen-Or et. al.'s) progressive code size. On the other hand, that algorithm required 78,464 bits for the Venus body mesh of 11,362 vertices (quantized to 10 bits/coord), which is 60% less than our (and Alliez and Desbrun's) progressive code size. Again, this is probably due to the significant regularity in both the connectivity and geometry of the mesh, which is exploited by the Touma and Gotsman algorithm.

The number of triangles vs. the distortion for the various algorithms is shown in Figure 7. It is evident that the performance is comparable, so our (connectivity-based) simplification algorithm is not much worse than those of Cohen-Or et al. and Alliez-Desbrun.

The number of cache misses for a 16-entry cache vs. the distortion is shown in Table 1, for our method and that of Alliez and Desbrun (it was not possible to generate these for the algorithm of Cohen-Or et al.). This is a measure of rendering efficiency and is where we have a significant advantage over the other schemes. Since there is no rendering sequence information explicitly present in any of the codes except ours, we assumed that the coding algorithm of Alliez and Desbrun was given, as input, meshes whose triangles had been ordered in a rendering sequence generated by our algorithm at full resolution. We then measured the number of cache misses incurred by the renderer after the meshes were decoded at various resolutions, and the triangles rendered in the order that the decoder generated. As can be seen from the table, despite our method generating a larger number of triangles for a given distortion level, the total number of cache misses incurred while rendering our model is nonetheless smaller by a factor of anywhere between 1.4 and 1.9. This is translated immediately into a comparable gain in rendering speed.

Figure 8 illustrates the distribution of the cache misses on a model when decoded and rendered using our algorithm. The resulting ACMR was typically 0.7 for most models at all resolutions. This

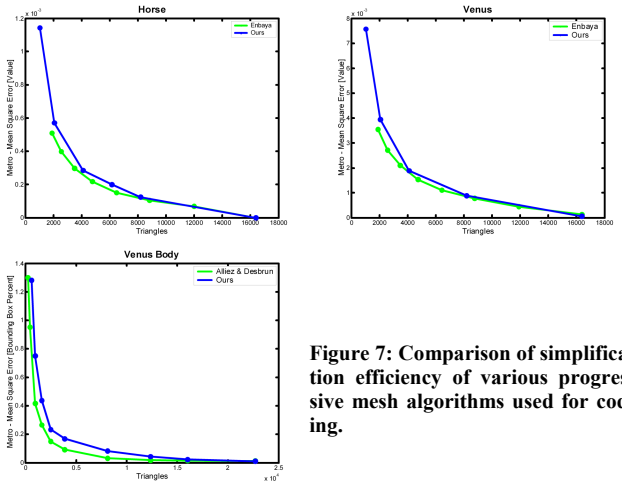


Figure 7: Comparison of simplification efficiency of various progressive mesh algorithms used for coding.

is an excellent result, as the algorithm of Bogomjakov and Gotsman [6], which is optimized to produce good rendering sequences (independent of coding considerations), usually did not do much better than this even when run separately at each resolution. Using these rendering sequences in practice accelerated the frame rate (relative to unordered face rendering) by a factor of up to 3.

Distortion	Ours		Alliez & Desbrun	
	Triangles	Cache Misses	Triangles	Cache Misses
0.744	950	**690**	631	**1290**
0.433	1620	**1173**	934	**1656**
0.230	2510	**1832**	1885	**3061**
0.166	3878	**2801**	2372	**3916**
0.079	8096	**5828**	4810	**11058**
0.039	12368	**8920**	7605	**14744**
0.018	16002	**11531**	9062	**16664**

Table 1: Comparison of rendering efficiency of various progressive mesh algorithms used for coding the Venus body model.

7 SUMMARY AND DISCUSSION

We have presented a method to code a multiresolution structure of a 3D mesh which allows progressive decoding and also efficient rendering of the mesh at all resolutions. Ordering the vertices in a special manner, and then collapsing every two adjacent vertices to their centroid according to this sequence may be considered one-dimensional wavelet transform coding of the mesh geometry using the Haar wavelet as basis functions. These basis functions give this effect because of their piecewise-constant character, which recursively force every two mesh vertices adjacent in the ordering to collapse to one location, inducing a binary tree structure on the coded prediction errors. However, strictly using the wavelet interpretation would mean coding the "prediction error" $e'=(u+w)/2-v$, using the terminology of Section 4.1, whereas, in practice, we use a more sophisticated prediction scheme, taking into account not just the vertices u and w adjacent to v on the vertex sequence, but other vertices adjacent on the mesh. These are identified thanks to the connectivity code. The Haar wavelets are the most primitive possible, and it would be interesting to examine the meaning and effect of using higher-order wavelet bases.

There is an interesting connection between graph partitioning, graph embedding and spectral graph theory. It is well known [14] that a good graph partition may be obtained by computing the eigenvector of the graph Laplacian with the smallest non-zero eigenvalue, and computing the median of its coordinates. Those vertices represented by entries of the eigenvector below the me-

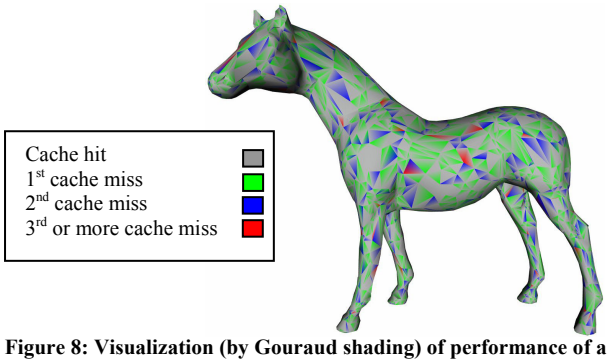


Figure 8: Visualization (by Gouraud shading) of performance of a 16-vertex cache using the rendering sequence derived from the vertex ordering at resolution *n*=2,048. Each vertex in each triangle is colored according to whether it incurs a cache hit or miss. Each vertex must have at least one incident triangle in which it is colored green (the first time the vertex is rendered), so green and a lot of gray is a good sign. The resulting ACMR is 0.7.

dian may be taken to be one subset of the partition, while those above the median are the other subset. This is equivalent to embedding the graph vertices on the real line using the eigenvector as a coordinate vector and using this embedding to generate a balanced partition with small edge-cut by partitioning them at the point of the median. What we do in our embedding algorithm is exactly the *opposite*: we use MeTiS to partition the mesh, and then use this partition to obtain a one-dimensional embedding. The main difference is that we embed the vertices on the integers, as opposed to arbitrary real values, and apply the partitioning in a recursive manner. It is, therefore, not surprising that "projecting" the geometry on a "basis" generated in this way results in "coefficients" (actually, prediction errors) which decay rapidly, similar to the behavior of spectral coefficients.

We have chosen to embed the mesh in a regular one-dimensional structure, which is easy to manipulate. It is certainly possible to embed the mesh in a regular two-dimensional structure, and possibly gain extra efficiency. This was done by Karni and Gotsman [19] for spectral compression, but seems to significantly complicate the process by introducing too many new degrees of freedom. Using a one-dimensional embedding also allows the methods to extend easily to meshes with boundaries and non-zero genus. Non-manifold meshes have to be treated specially, by cutting them into a number of smaller manifold pieces, a standard practice also in other work.

The order of vertex splits at the decoder (hence also edge collapses at the encoder) is uniquely determined by the vertex ordering and its locality properties. This means that the mesh will evolve at the decoder in a somewhat non-uniform manner. For example, towards the end of the transmission, one half of the mesh will already be completely at full resolution while the other half will not yet be these. It makes more sense to spread the refinement process uniformly over the mesh. It is possible to immediately incorporate this feature into our method, as long as the order of vertex splits over the vertex sequence is deterministic, so it can be applied at the decoder with no extra information. One way of doing this is to use the *geometric* information accumulated so far at the decoder (and encoder) at any point in time to determine the identity of the next vertex to be split. This will maximize the geometric fidelity of the result, and possibly improve the rate-distortion and rendering-distortion tradeoffs significantly. Note, however, that these reorderings of the vertex sequence can only be done within levels of the binary tree representing the sequence.

References

[1] Abadjev, V., del-Rosario, M., Lebedev, A., Migdal, A. and Pashaver, V. Metastream. *Proceedings of VRML, 1999.*

[2] Alliez, P. and Desbrun, M. Progressive Encoding for Lossless Transmission of 3D Meshes. *Proceedings of SIGGRAPH, 195-202, 2001.*

[3] Bajaj, C., Pascucci, G. and Zhuang, G. Progressive Compression and Transmission of Arbitrary Triangular Meshes. *Proceeding of Visualization, 307-316, 1999.*

[4] Bar-Yehuda, R., Even, G., Feldman, J. and Naor, J. Computing an Optimal Orientation of a Balanced Decomposition Tree for Linear Arrangement Problems. *Journal of Graph Algorithms and Applications,5(4):1-27, 2001.*

[5] Bar-Yehuda, R. and Gotsman, C. Time/Space Tradeoffs for Polygon Mesh Rendering. *ACM Transactions on Graphics 15(2), 141-152, 1996.*

[6] Bogomjakov, A. and Gotsman, C. Universal Rendering Sequences for Transparent Vertex Caching of Progressive Meshes. *Computer Graphics Forum, 21(2), 137-148, 2002.*

[7] Cignoni, P., Montani, C., Rocchini, D. and Scopigno, R. Metro: Measuring Error on Simplicial Surfaces. *Computer Graphics Forum 17(2), 167-174, 1998.*

[8] Cohen-Or, D. Levin, D. and Remez, O. Progressive Compression of Arbitrary Triangular Meshes. *In Proceedings of Visualization, 67-72, 1999.*

[9] Deering, M. Geometry Compression. *Proceedings of SIGGRAPH, 13-20, 1995.*

[10] Denny, M.O. and Sohler, C. Encoding a Triangulation as a Permutation of its Point Set. *Proceedings of the Canadian Conference on Computational Geometry, 1997.*

[11] Devillers, O. and Gandoin, P.M. Geometric Compression for Interactive Transmission. *Proceedings of IEEE Visualization, 319-326, 2000.*

[12] Gibbs, N.E., Pole, W.G. Jr. and Stockmeyer, P.K. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis 13(2), 236-250, 1976.*

[13] Gotsman, C. and Lindenbaum, M. On the Metric Properties of Discrete Space-Filling Curves. *IEEE Transactions on Image Processing 5(5), 794-797, 1996.*

[14] Hall, K.M. An r-dimensional Quadratic Placement Algorithm. *Management Science,(17), 219–229, 1970.*

[15] Hoppe, H. Progressive Meshes. *Proceedings of SIGGRAPH, 99-108, 1996.*

[16] Hoppe, H. Optimization of Mesh Locality for Transparent Vertex Caching. *Proceedings of SIGGRAPH, 269-276, 1999.*

[17] Isenberg, M. Triangle Strips Compression. *Proceedings of Graphics Interface, 197-204, 2000.*

[18] Isenberg, M. and Snoeyink, J. Face Fixer: Compressing Polygon Meshes With Properties. *Proceedings of SIGGRAPH, 263-270, 2000.*

[19] Karni, Z. and Gotsman, C. Spectral Compression of Mesh Geometry. *Proceedings of SIGGRAPH, 279-286, 2000.*

[20] Karni Z. and Gotsman C. 3D Mesh Compression Using Fixed Spectral Bases. *Proceedings of Graphics Interface, 1-8, 2001.*

[21] Karypis, G. and Kumar, V. MeTiS - a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Version 4, University of Minnesota, *1998.*

[22] Khodakovsky, A., Schroeder, P. and Sweldens, W. Progressive Geometry Compression, *Proceedings of SIGGRAPH, 271—278, 2000.*

[23] Pajarola, R. and Rossignac, J. 2000. Compressed Progressive Meshes. *IEEE Transactions on Visualization and Computer Graphics 6(1), 79-93.*

[24] Rossignac, J. Edgebreaker: Connectivity Compression for Triangle Meshes. *IEEE Transactions on Visualization and Computer Graphics 5(1), 47-61, 1999.*

[25] Taubin, G., Gueziec, A., Horn, W. and Lazarus F. Progressive Forest Split Compression, *Proceedings of SIGGRAPH, 1998.*

[26] Touma, C. and Gotsman, C. Triangle Mesh Compression. *Proceedings of Graphics Interface, 26-34, 1998.*