Compressing Large Polygonal Models

Jeffrey Ho [*] Kuang-Chih Lee [†] David Kriegman [‡]

Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, IL 61801

Abstract

We present an algorithm that uses partitioning and gluing to compress large triangular meshes which are too complex to fit in main memory. The algorithm is based largely on the existing mesh compression algorithms, most of which require an 'in-core' representation of the input mesh. Our solution is to partition the mesh into smaller submeshes and compress these submeshes separately using existing mesh compression techniques. Since a direct partition of the input mesh is out of question, instead, we partition a simplified mesh and use the partition on the simplified model to obtain a partition on the original model. In order to recover the full connectivity, we present a simple scheme for encoding/decoding the resulting boundary structure from the mesh partition. When compressing large models with few singular vertices, a negligible portion of the compressed output is devoted to gluing information. On desktop computers, we have run experiments on models with millions of vertices, which could not be compressed using standard compression software packages, and have observed compression ratios as high as 17 to 1 using our technique.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—object representations; I.3.6 [Computer Graphics]: Methodology and Techniques—graphics data structures and data types; E.4 [Coding and Information Theory]: Data compaction and compression

Keywords: compression algorithms

1 INTRODUCTION

With the recent and rapid advances in digital acquisition technology, meshes with millions if not billions of vertices are becoming increasingly common. The most celebrated examples are the range scans of Michelangelo's sculptures, made by Stanford's Digital Michelangelo project, which contain up to two billion triangles. These massive datasets pose great challenges to virtually all existing mesh processing tools such as rendering, editing, simplification, and compression. The large memory requirement for handling these datasets renders many of these mesh-processing tools ineffective on a computer with modest size RAM (i.e. a desktop PC). Existing mesh compression and, to a lesser extent, decompression algorithms are only effective if a representation of the mesh's entire topological and geometric structures (and other attributes) is small enough to fit 'in-core'. Yet for a mesh with a few million vertices, one faces the possibility that there is insufficient memory on a desktop computer for the entire model.

Our approach toward compressing these large models is straightforward: to partition the mesh into submeshes of smaller sizes, depending on the available local memory, and then to compress them separately using the existing mesh compression algorithms. Extra effort is also needed to encode and decode the connectivity information that glues (or stitches) the various boundaries resulted from the mesh partition. Certainly, there are many ways to partition a mesh. However, from the compression standpoint, it is desirable that 1) each region of the partition is "localized" somewhere in the model, and 2) the boundary of each region is as simple as possible. The motivation is that submeshes within a localized region would share similar geometry and therefore, minimize the size of the compressed geometry output code. The simple boundary assures that the boundary encoding would also be minimized. In addition, the partition should in general be balanced in the sense that each patch of the partition should contain roughly the same number of vertices or faces, or other geometric primitives. Since the original mesh is assumed to be too large to be represented in-core, we are faced with the problem of how to produce partitions that satisfy these criteria. One obvious partitioning approach is to use the coordinate axes or other linear functions of the ambient space to partition the mesh. However, without knowing the approximate shape or geometry of the model, a direct approach is generally quite hazardous. We propose the simple idea of using a weighted graph, which is derived from a simplified mesh of the input model, as a guide for partitioning the original mesh. Therefore, the compression starts with an initial pass over the input file so that this weighted graph can be computed. It is assumed that the weighted graph is small enough to partition it directly. If N is the ratio between the size of the model representation and the locally available memory, we partition the weighted graph into N parts and use the partition on the graph to induce a partition on the input model. We make another N passes over the input file such that each submesh (region of the partition) is reconstructed and compressed separately.

To encode/decode the full connectivity, we propose an encoding/decoding scheme for the boundaries resulting from the partition. The problem is similar to the one studied in [5], but with several important differences. The idea is simple. Because of the partition, most of the vertices that need to be identified are the boundary vertices of the submeshes. Using this as the working hypothesis, our method is similar to the variable-length method described in [5] and is optimized for our particular circumstance.

Since the compression algorithm requires $N + 1$ passes over the input file, its performance in speed is dominated by disk I/O. This becomes rather unattractive if N and the file size are both large. The decoder, on the other hand, only makes one pass over the compressed file, which is assumed to be smaller than the original file; therefore, its speed is less effected by the size of the original file. This makes the compression algorithm asymmetric, although it is generally preferable to have a faster decoder.

For mesh simplification, the recently introduced out-of-core simplification technique [13] provides an elegant solution for memoryless simplifications. The out-of-core method makes one pass over the input mesh data file and it only retains the data needed for computing the simplified mesh; therefore, the memory used in processing the mesh is kept at minimum. The usual text-based compression techniques such as the GZIP or [21] also share this characteristic. These compression methods usually make one or two passes on the input file, depending on whether the compression statistics is extracted from the document before the compression. The memory

[*] j-ho1@dizzy.ai.uiuc.edu.
[†] klee10@uiuc.edu
[‡] kriegman@uiuc.edu

usage for these methods are generally small and for GZIP or LZW compression method, it depends on the size of the windows used in computing predictions. Comparing with the two cases above, the situation for mesh compression is considerably more complicated. The out-of-core simplification ignores the local connectivity information. However, local connectivity is one of the properties that is to be encoded (or decoded) by the mesh compression algorithm. The prediction used in text-based compression is simply unidirectional, i.e. toward the end of the file. For the mesh compression, which involves the 3D geometry, the relation between the primitives, i.e. the vertices and faces, is multi-directional. In an abstract sense, all existing mesh compression algorithms aim to produce some type of linear ordering (based on spatial proximity) on the mesh (vertex or face traversals); this requires a both global and local structure of the model and hence a large in-core representation of the mesh.

This paper is organized as follows. In the next section, we briefly review some of the major work in mesh compression. In section three, we describe our partition and glue schemes, and experimental results are reported in the concluding section.

2 PREVIOUS WORK

The process of compressing a mesh generally consists of two steps: a preprocessing step and the actual compression. Since virtually all the existing mesh compression algorithms (with the exception of [1]) require the input mesh to be a manifold, a preprocessing step is necessary to detect singular points on the input model (if they exist) and convert a non-manifold input mesh into a manifold one.

2.1 Converting Non-Manifold Meshes

The method for converting non-manifold meshes into manifold meshes is studied in [6]. The main idea is to separate the local branches at each singular vertex by duplicating it. The resulting manifold mesh will have the same number of (non-degenerate) triangles as the original mesh but with more vertices. In addition, the manifold mesh will typically have more than one connected component. Each regular vertex in the original mesh corresponds to one vertex in the manifold mesh while each singular vertex corresponds to more than one vertex in the manifold mesh. The correspondence is recorded in a vertex clustering array. In [5], two efficient methods for encoding/decoding this vertex clustering array were proposed: a stack-based method, which directly encodes and decodes the array and a variable-length method, which is more efficient if the vertex clustering array contains long chains of (singular) vertices with consecutive decoding orders of traversal.

2.2 Mesh Compression

Many mesh compression algorithms have been proposed and investigated in the past five years. They differ mainly in the way the mesh connectivity is encoded/decoded. Due to its discrete nature, the connectivity encoding is generally combinatorial in nature and different algorithms give different recipes for visiting each triangle or vertex of the mesh. The idea of using triangle strips for compression appeared in Deering's work [3]. It was later generalized and extended in various directions [2, 12, 7], which includes the IBM's topological surgery [18, 19] and Rossignac's Edgebreaker [15]. Somewhat differently, Touma and Gotsman's method gives a recipe for making a vertex traversal. They encode the valences of the vertices and record the merging and splitting of the boundary of the region the algorithm has visited. As a by-product of the mesh traversal, all of the methods cited above produce an ordering of the vertices, the decoding order.

Using the decoding order defined by the connectivity encoding, a predictive coding can be developed for compressing mesh geometry and other attributes. Typically, a bounding box for the model is used to define the coordinate quantization. The bounding box is divided into uniform grid cells, and the positions of the vertices are normalized within each cell. Using the decoding order of traversal, the position of a vertex can be predicted by the positions of previously decoded vertices. The difference between the actual position and the predicted position is encoded as an integer. The most successful prediction rule so far is the parallelogram rule of Touma and Gotsman [20], which predicts the position of a vertex as the fourth vertex on a parallelogram formed by the vertex and three of its "neighboring" vertices. The size of the output code is further reduced by the use of an entropy encoder, which is applied to all data, namely connectivity, geometry and other properties.

The work cited above all faithfully encode the connectivity of the mesh while its geometry is encoded in a lossy way with error controlled by the coordinate quantizations. However, recently there emerges a new type of "appearance-based" mesh compression [11, 8]. These methods generally involve re-meshing the input model into a semi-regular mesh. Standard multi-resolution signal processing tools, such as wavelets, are used to compress the geometry and other attributes. Therefore, both the connectivity and geometry are compressed in a lossy way; however, the main concern here is the fidelity of the compressed model's appearance. A more radical departure from traditional mesh compression is the Qsplat data structure of [17]. The data structure is used for rendering a massive mesh data file, and it can be considered as a form of mesh compression where all the connectivity information is ignored. In this paper, we follow the more traditional type of mesh compression where the mesh connectivity is encoded losslessly.

3 COMPRESSING LARGE MESHES

The mesh compression algorithm presented here builds on existing mesh compression algorithms, in particular [20, 15]. Our solution to compressing meshes too large to fit in-core is to partition the input mesh into submeshes small enough to apply an existing mesh compression algorithm [20]. If the complete connectivity is required, our method also provides a simple and efficient scheme to encode/decode the information needed to glue various submeshes together to recover the full connectivity. Our main contributions are: 1) a simple partitioning scheme, using a simplified mesh graph, which produces good partitions for compression, and 2) an encoding/decoding scheme for gluing the submeshes.

The recent work of Karni and Gotsman [9] appears to be the first time that mesh partitioning was used in mesh compression. However, the functionality of the partition in their work is different. In [9], the connectivity is encoded and decoded separately from the geometry, and it is used to define a mesh partition such that the geometry of each submesh is encoded and decoded independently using a spectral method based on the mesh Laplacian. Furthermore, since connectivity is used to compute the partition, there is no need to glue submeshes. In our case, it is not possible to use the full connectivity to define the partition; in fact, the partition is only defined on a simplified mesh graph. On the decoding side, we need to glue the submeshes together in order to recover the full connectivity.

3.1 Mesh Partitioning

The main goal of the mesh partitioning is to divide the input mesh into submeshes of roughly equal size. The size of course depends on the available memory and the amount of memory required by algorithms being used to compress the individual submeshes, and the later differs with different implementations. However, from

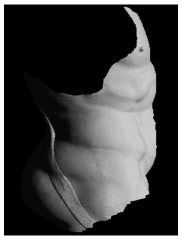
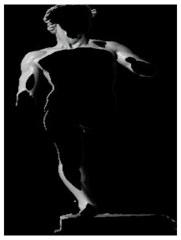
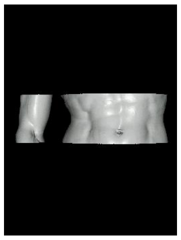
a.  b.  c. 

Figure 1: Examples of mesh partition. a. This partition is induced from a simplified mesh; b, c. Partitions using z and y axes, respectively. The original model is the statue of David from Stanford's Digital Michelangelo Project

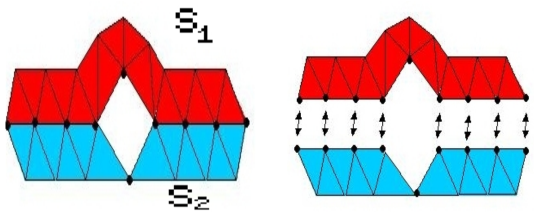


Figure 2: After the partition, the boundary of S_1 contains both the vertices on the cut boundary and the vertices on the actual boundary. When gluing S_1 and S_2 together to recover the original mesh, only the vertices on the cut boundary are identified.

the compression standpoint, it is also desirable that 1) the resulting boundary structure should be simple, and 2) each region of the partition should be localized in the model. A direct partitioning of the model using for example the coordinate axes or other linear functions of the ambient space generally does not have this property. See Figure 1.

By partitioning the mesh into N parts, we mean that each vertex of the mesh can be assigned a number, its region index, from 1 to N. The assignment for the vertices induces an assignment for the triangles: the region index for each triangle is defined as the minimum of the region indices of its three vertices. The K^{th} submesh S_K is simply the union of all triangles with region index K, and S_K does not contain any vertex with region index less than K. See Figure 2. The intersection of two submeshes is either empty or consists of just vertices and edges, and we call these non-empty intersections the *cut boundaries* to distinguish them from the real boundary.

The geometry of the input model can be condensed into a weighted graph G using spatial vertex clustering. The result is a map from the set of the vertices to the nodes of G. The weight at each node simply counts the number of vertices mapped to the node while the weight on each edge counts the number of triangles with vertices mapped to the edges' two nodes. If G is small enough, we can directly partition G, and in turn, the partition of G induces a partitioning on the input mesh.

The vertex clustering algorithm that we use is similar to the one in [16]. It requires a bounding box for the input mesh, and the same bounding box is used to define the global coordinate quantization that is applied to all submeshes. The bounding box is divided into a number of uniform grid cells. Given a triangle t from the input mesh, we obtain its vertex coordinates. For each vertex, we determine the grid cell that the vertex falls in. If this cell has not been visited, a new node in G is created. If two of the triangle's vertices belong to the same cell u and the other vertex belongs to a different cell v, a weight on the edge uv in G is increased by 1. If all the vertices of t belong to the same node, the node's weight is also increased by 1. G should be as small as possible; however, our primary concern is to have a balanced partition on the input mesh. If G is too coarse, the induced partition on the input mesh may not be appropriately balanced. In our implementation, the bounding box is divided into 64 to 128 equal parts in each coordinate axis, and G typically contains fewer than 40000 nodes.

After the initial pass, we have a weighted graph G that records roughly the spatial distribution of the vertices of the input mesh. The weighted graph can be partitioned using a standard graph partitioning package such as METIS [10]. The result of applying METIS to G is a balanced partition of G with small (but not minimal) edge cuts. A balanced partition for G means that the sum of the weights of all nodes in the different regions are approximately the same. Since the weight of the nodes simply counts the number of triangles collapsed into the cell, the balanced partitioning on G translates into a balanced partitioning on the input mesh. A partition of G with small edge cuts also translates into a partitioning of the input mesh with "simple" boundary structure. Table 1 shows two different partitions of David from the Digital Michelangelo Project, and the number of vertices on the cut boundaries.

The data structure for the partitioning of G will remain throughout the entire encoding process. We make another N passes over the input file, and each time a submesh S_i is constructed in-core. Again, given a triangle t, we obtain its three vertices and identify the nodes of the graph these vertices belong to. A vertex's region index is simply the region index of its parent node in G. If t is determined to have a region index equal to i, it is retained; otherwise, it is discarded. After S_i has been compressed, we free the memory holding the structure for S_i and proceed to compress S_{i+1}.

At this point, we have two options, we can proceed directly to compress each submesh independently and hence ignore the cut boundaries or we can proceed to encode the cut boundaries and hence compress each submesh separately but not independently. The main difference between the two approaches is that the positions of the vertices on the cut boundaries are encoded multiple times if each submesh is compressed independently. If the number of submeshes is small, the vertices on the cut boundaries are usually less than 1% of the total number of vertices. Of course, there will always be some unfortunate cases where a small number of submeshes results in a large number of vertices on the cut boundaries. However, our partitioning scheme is specifically designed, for most cases, to have a small number of vertices on the cut boundaries, as shown in Table 1. If the submeshes are compressed independently, the compressed output represents a triangular mesh M' which contains the same number of triangles as the original mesh M but with more vertices. Since G is determined only by the triangles, the G for M and M' are identical. Hence, if we compress M' with N partitions, the compressed output this time will faithfully represent M', i.e. the number of vertices will not increase as long as N is the same. Therefore, on the same desktop PC, repeatedly compressing and decompressing the input model will not increase the number of vertices indefinitely.

Partition	N = 14	N=5
x	19063	57435
y	22371	6123
z	98081	31066
G	13870	6332

Table 1: The number of vertices on the cut boundaries. Comparisons between partitions using the coordinate axis and the weighted graph G. The partitions using weighted graph are shown in Figure 1 of the color plate.

3.2 Gluing the Cut Boundaries

If the full connectivity is required, we are obliged to encode/decode the cut boundary structure, i.e. how the vertices on the various cut boundaries should be identified. A straightforward method is to directly encode the gluing. If v and u are vertices that should be identified and u is decoded after v, we can associate with u an integer index that identifies which previously decoded vertices should u be identified with. Since the input mesh is supposed to contain millions of vertices and the number of vertices on the cut boundary are usually more than a few thousand, the integer index would require more than 10 bits to code. Therefore, hard coding the gluing is not likely to produce results better than encoding the vertices multiple times. Since the cut boundaries are the direct result of mesh partitioning, most of the vertices on the cut boundaries are in fact on the boundary of the submeshes containing them. As can be seen from Figure 1, the cut boundaries are usually very long, and this suggests some type of variable-length encoding is appropriate. For vertices that do not belong to a submesh's boundary, we simply encode them directly.

The gluing (or stitching) problem has been studied previously in [5], and there are differences between Gueziec's case and ours. First, in [5], the vertices that need to be identified are the singular vertices, and they generally reside on the interior of the mesh rather than on the boundary. Therefore, the only ordering of the vertices that can be used for the variable-length encoding is the decoding order of the vertices. In our case, an ordering for most of the vertices on the cut boundaries can be provided by the orientations of submeshes' boundaries. Second, in [5], at the time of encoding, the complete information on how to cluster vertices is available (the vertex clustering array in [5]) while in our case, the information on vertex identifications is only revealed incrementally.

Since we are using the boundary information to encode the gluing, this requires that, on the decoding side, the connectivity is decoded before the gluing can start. On the decoding side, the connectivity is decoded first, and this is followed by gluing. The geometry is decoded last, following the same traversal as the connectivity decoding. A table containing the structure of the mesh partition is encoded as a function $p(k)$ for $1 \leq k \leq N$, and it is placed at the beginning of the compressed file. For each $1 \leq k \leq N$, the number $p(k)$ is defined as the largest integer such that the intersection between submeshes S_k and $S_{k+p(k)}$ is non-empty.

3.3 Cut Boundaries Encoding

Suppose we are encoding an input mesh that has been partitioned into N parts. Let $S_1 \cdots S_N$ denote the resulting submeshes, and we compress them in this order. When compressing a submesh, we need to 1) identify vertices in the current submesh that are glued to vertices in the previous submeshes and 2) identify vertices in the current submesh that will be glued to vertices in the forthcoming submeshes.

At any moment, a list of vertices, CL, is maintained. These are the vertices which are "waiting to be glued". Using the cut boundaries, we can impose extra geometric structure on CL so that it describes a collection of line segments, close loops and single points. See Figure 3. In the actual implementation, CL is a collection of doubly-linked lists of vertices and lists with a single vertex.

The data structure for the vertices contains the following fields:

1. *rID* denotes the region it belongs (according to the weighted graph G),
2. *cID* denotes the last submesh it is on,
3. *dID* denotes the decoding order of the given vertex in submesh S_{cID}.

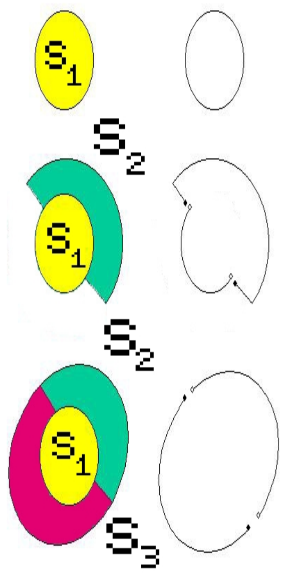


Figure 3: The evolution of CL: after S_1 is processed, only the vertices on its boundary are retained. At this moment, CL consists of these vertices. After S_2 is processed, CL consists of two segments. Each segment is a part of the cut boundary of S_1 or S_2. After S_3 is processed, CL again consists of two segments. The cut boundary of S_1 has been processed and its boundary structure is deleted from the memory.

For each submesh S_k, let V_k denote the set of its vertices. Let P_k denote the set of vertices in V_k that have to be glued to some vertex in a previous mesh S_i with $i < k$. We also let N_k denote the vertices in V_k that will be identified with some future vertices. See Figure 2 of the color plate. The intersection $N_k \cap P_k$ is generally non-empty.

To identify N_k is easy, they are the vertices in V_k with $rID > k$. P_k, by definition, is simply $V_k \cap CL$.

Before connectivity encoding, the boundary structure of S_k is computed. A vertex v is on the boundary of S_k if a small neighborhood of v in S_k is homeomorphic to the half disc and a boundary edge is an edge in S_k shared by only one triangle, whose orientation induces an orientation on the edge. Let ∂S_k denote the set of boundary vertices of S_k. By computing the boundary structure of S_k, we mean that the set ∂S_k can be partitioned into disjointed subsets such that u and v belong to the same subset if there exists a sequence of (oriented) boundary edges $\{e_1, ...e_n\}$ connecting u and v. Geometrically, each subset (with the connecting edges) forms a line segment or a closed loop, or a single point. The boundary structure induces a geometric structure on the sets N_k and P_k, i.e. we can form line segments, loops with vertices in N_k and P_k. Similar to CL, in our implementation, N_k and P_k are collections of doubly-linked lists and lists with single vertex.

Next, we identify vertices in P_k which are "essential" to gluing S_k to the previous submeshes. See Figure 4. By definition, each vertex v in P_k will be identified with a vertex v' in CL. The correspondence is one-to-one, i.e. no two vertices in P_k will be glued to the same point in CL. If $v \in P_k$ but not in ∂S_k, we simply find the corresponding vertex v' in CL, and we record five integers for v. The first two are the *cID dID* of v', and the other three are all zeros. We remove v from the set P_k and put it into a list T. Once a vertex in CL has been selected for gluing, it is removed from CL, and the corresponding geometric structure it belongs to is modified.

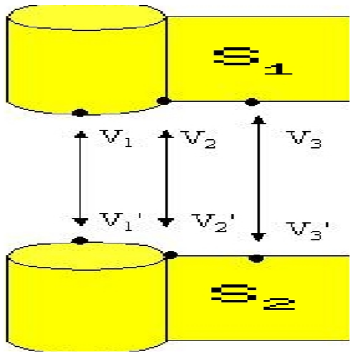


Figure 4: Gluing two cut boundaries. Vertices v_1 and v_3 are boundary points of S_1. v_2 is not a boundary point according to our definition. The circles can be glued together by specifying only 1) the pair v_1 and v_1', 2) the number of vertices on the circle (excluding v_2) and 3) the orientations of the circles. The line segments containing v_3 and v_3' can be coded similarly. The pair v_2 and v_2' is coded directly.

For instance, if v' belongs to a loop, v' is removed from CL, and the loop is changed to a line segment.

For each $v \in P_k \cup \partial S_k$, let v' be the vertex in CL that is to be identified with v. If L_{P_k} and L_{CL} are the line segments in P_k and CL containing v and v', we simply find the longest (oriented) line segment L in L_{CL} containing v such that by traveling down L, we can identify vertices on L with vertices on L_{CL}. For v, we record the following five numbers: the first two numbers identify the vertex v', the *cID dID* of v'. The next two number are the forward length fl and backward length bl. These are the topological distance between v and the two endpoints of L (recall that L is oriented, so starting at v, we can travel forward and backward on L according to its orientation). If L is a loop, bl is set to -1. The last number compares the orientations between L and L_{CL}, it is 1 if they are compatible and -1 otherwise. v is removed from P_k and added to the list T. We remove all other vertices u on L from P_k and their corresponding vertex u' is also removed from CL without further processing. The process is terminated when P_k is empty.

During the mesh traversal phase of the compression, these records are inserted into the symbol sequence at the appropriate places. In Rossignac's Edgebreaker, there are five symbols for connectivity encoding: S(S*), R, L, E, C. We add two more, GLUE1 and GLUE2. The symbol GLUE1 is followed by 5 integers while GLUE2 stands by itself.

Whenever a vertex v in the list T is first encountered during mesh traversal, GLUE1 is inserted into the symbol sequence. The five integers that follows are the five numbers we described above. Whenever, a vertex $v \in N_k$ and $v \notin \partial S_k$ is first encountered, GLUE2 is inserted. The functionality of GLUE1 has been explained above. The functionality of GLUE2 will be explained later.

After compression is finished, all vertices in S_k are deleted except those belonging to N_k. For each $v \in N_k$, its *cID* is changed to k and *dID* is changed to v's decoding order on S_k. And finally, N_k is appended to CL.

3.4 Decoding the Cut Boundaries

On the decoding side, the process is reversed. For each S_k, its connectivity is decoded first. After the boundary of S_k has been computed, we do the gluing, and geometry decoding follows in the same traversal as the connectivity decoding. During the connectivity decoding, any vertex marked by GLUE1 and GLUE2 is put into a list T. These are the vertices needed for gluing. A list of vertices CL, which contains vertices that will be identified later, is maintained throughout. After the connectivity of S_k has been decoded, we compute its boundary structure. For every vertex in S_k marked by GLUE2, we use the five integers described above to identify vertices in S_k with vertices in S_i with $i < k$. The geometry is decoded after the gluing is finished. After S_k has been decoded, we delete all its vertices (and faces) from the memory except those that will be used in the future for gluing. These vertices are the vertices on its boundary and those marked by GLUE2. The main point is that, on the decoding side, we do not recover the exact N_k. Instead, N_k is defined as the union of the vertices on the boundary of S_k and the vertices marked by GLUE2. Therefore, there are vertices in N_k that will not be glued to anything; however, all the vertices that will be needed for gluing are in N_k. This may create a problem if N is large and hence many boundary vertices have to be kept in memory. The problem can be greatly reduced by using the function p decoded at the beginning. Vertices created during the decoding of S_k can be deleted once the the submesh $S_{k+p(k)}$ has been decoded since, by definition, the intersection between S_k and S_i is empty if $i > k + p(k)$

a. b. c.

Figure 5: Three of our test models: a. Lucy from Stanford 3D Scanning Repository. 28,055,742 triangles and 14,027,872 vertices. b. David from the Digital Michelangelo Project. 8,254,152 triangles and 4,129,614 vertices. c. A part of St. Matthew (part 4) from the Digital Michelangelo project. 36,550,076 triangles and 18,306,508 vertices

4 RESULTS AND DISCUSSION

For compressing submeshes, We have used Rossignac's Edgebreaker [15] for connectivity encoding and the parallelogram rule of Touma and Gotsman [20] for geometric prediction. In our implementation, an "in-core" representation of an input mesh requires approximately 172 byes per vertex. This includes the data structures needed for compression and converting a non-manifold mesh into a manifold one. The output from the geometry and connectivity encodings are entropy encoded using the arithmetic compression software of [21]. The results reported in Table 2 were gathered on a 450 MHZ machine with 256 MB of RAM. We limit the size of each submesh to about one million vertices. All three models can't be compressed using Virtue3D's Virtuoso Optimizer on a 850MHZ PC with 516MB of RAM. In this early and non-optimized implementation, the running times for compressing David and Lucy are approximately 1.5 hours and 8 hours, respectively. The performance in speed is dominated mostly by the disk I/O. The running times for decompressing both models are approximately 8 and 30 minutes, respectively.

The gluing scheme described in the previous section is quite effective on these large models since the cut boundaries are usually long and contain mostly regular boundary points. Only a few vertices are marked with GLUE1 and GLUE2. These are mainly the results of singular points and small holes in the models. Due to their large number of vertices, we have set the coordinate quantizations to be at least 16 bits per coordinate. For the three models listed in Table 2, the geometry is compressed to about 16 bits per vertex at 16 bits per coordinate quantization. Note that with millions of

Model	Uncompressed File Size	Quantization	Compressed Connectivity	Compressed Geometry	Total	Compression Ratio
David	173.3MB	16	1.34MB (2.6)	8.7 MB (16.8)	10.1MB	17
David	173.3MB	18	1.34MB (2.6)	11.7MB (22.8)	13 MB	13
Lucy	533MB	16	4.9MB (2.8)	30.7MB (17.5)	35.6MB	15.2
Lucy	533MB	18	4.9MB (2.8)	41.0MB (23.4)	46 MB	11.5
St. Matthew(IV)	768MB	16	6.18MB (2.7)	37.7MB (16.5)	43.9MB	17.86
St. Matthew(IV)	768MB	18	6.18MB (2.7)	49.8MB (21.8)	56MB	13.7

Table 2: Compression Results. Numbers in Parenthesis are bits per vertex. David contains 8,254,152 triangles and 4,129,614 vertices. Lucy contains 28,055,742 triangles and 14,027,872 vertices. St. Matthew (IV) has 36,550,076 triangles and 18,306,508 vertices

vertices, the models that we used here are generally several orders of magnitude larger than those used in mesh compression literature in the past (e.g. [20]). Therefore, the quantizations required for our test models are more refined (more bits per vertex, typically 17 bits per vertex) than those used in reporting compression results before (typically 10 bits per vertex).

Mesh compression and mesh simplification have been hot topics for research in the past few years and both fields have reached a certain degree of sophistication. The algorithm we proposed in this paper can be considered as a "simplification-based" mesh compression. The way we use the simplified mesh is to partition it and use the partition on the simplified mesh to induce a balanced partition on the input mesh, which is our primary goal. Therefore, we have completely ignored the geometry of the simplified mesh. It is interesting to see if the geometry of the simplified mesh can be used to further reduce the compressed geometry of the input model. Perhaps, non-linear prediction rules, based on the curvature of the simplified mesh, can be developed to more efficiently compress the geometry. To proceed further in this direction, a more refined and geometry-oriented partition scheme, such as [4, 14], is probably required.

5 ACKNOWLEDGEMENT

The authors would like to thank Marc Levoy for providing the David and St. Matthew datasets. The first author was supported by the National Science Foundation under grants EIA 00-04056 and CCR 00-86094. These findings do not necessarily reflect the views of NSF.

References

[1] V. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitray triangular meshes with properties. *Proceedings of the Data Compression Conference, Snowbird,* 1999.

[2] M. Chow. Optimized geometry compression for real-time rendering. *Visualization 97*, pages 415–421, 1997.

[3] M. Deering. Geometric compression. *Proc. SIGGRAPH*, pages 13–20, 1995.

[4] M. Eck, T. DeRose, T. Duchamp, and H. Hoppe. Multiresolution analysis of arbitrary meshes. *Proceedings of SIGGRAPH 95*, pages 173–182, 1995.

[5] A. Gueziec, F. Rossen, G. Taubin, and C. Silva. Efficient compression of non-manifold polygonal meshes. *In Visualization 99*, pages 73–80, 1999.

[6] A. Gueziec, G. Taubin, F. Lazarus, and W. P. Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. *In Visualization 98*, pages 383–390, 1998.

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

[8] I. Guskov, K. Vidimce, Wim Sweldens, and Peter Schroder. Normal meshes. *Proceedings of SIGGRAPH 00*, pages 95–102, 2000.

[9] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. *Proceedings of SIGGRAPH*, pages 279–286, 2000.

[10] G. Karypis and V. Kumar. *Software available at http://www-users.cs.umn.edu/ karypis/metis/index.html*.

[11] A. Khodakovsky, Peter Schroder, and Wim Sweldens. Progressive geometry compression. *Proceedings of SIGGRAPH 00*, pages 271–278, 2000.

[12] J. Li and C.C. Kuo. Progressive coding of 3D graphics models. *Proceedings of the IEEE*, pages 1052–1063, 1998.

[13] P. Lindstrom. Out-of-core simplification of large polygonal models. *Proceedings of SIGGRAPH 00*, pages 259–262, 2000.

[14] J. Maillot, H. Yahia, and A. Verrous. Interactive texture mapping. *Proceedings of SIGGRAPH 93*, pages 27–34, 1993.

[15] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transaction on Visualization and Computer Graphics, 5(1)*, 1999.

[16] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. *Modeling in Computer Graphics*, pages 455–465, 1993.

[17] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. *Proceedings of SIGGRAPH 00*, pages 343–352, 2000.

[18] G. Taubin, W. Horn, F. Lazarus, and J. Rossignac. Geometry coding and vrml. *IEEE Special Issue on multimedia signal processing*, 1998.

[19] G. Taubin and J. Rossignac. Course on 3D geometry compression. *Proceedings of SIGGRAPH*, 1999.

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

[21] I. H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes:Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, 1999.