Edgebreaker: Connectivity compression for triangle meshes

Jarek Rossignac
GVU Center, Georgia Institute of Technology

Abstract

Edgebreaker is a simple scheme for compressing the triangle/vertex incidence graphs (sometimes called connectivity or topology) of three-dimensional triangle meshes. Edgebreaker improves upon the worst case storage required by previously reported schemes, most of which require O(nlogn) bits to store the incidence graph of a mesh of n triangles. Edgebreaker requires only 2n bits or less for simple meshes and can also support fully general meshes by using additional storage per handle and hole. Edgebreaker's compression and decompression processes perform the same traversal of the mesh from one triangle to an adjacent one. At each stage, compression produces an op-code describing the topological relation between the current triangle and the boundary of the remaining part of the mesh. Decompression uses these op-codes to reconstruct the entire incidence graph. Because Edgebreaker's compression and decompression are independent of the vertex locations, they may be combined with a variety of vertex-compressing techniques that exploit topological information about the mesh to better estimate vertex locations. Edgebreaker may be used to compress the connectivity of an entire mesh bounding a 3D polyhedron or the connectivity of a triangulated surface patch whose boundary needs not be encoded. Its superior compression capabilities, the simplicity of its implementation, and its versatility make Edgebreaker particularly suitable for the emerging 3D data exchange standards for interactive graphic applications. The paper also offers a comparative survey of the rapidly growing field of geometric compression.

Introduction

Interactive 3D graphics already plays an important role in manufacturing, architecture, petroleum, entertainment, training, engineering analysis and simulation, medicine, and science. It promises to revolutionize electronic commerce and many aspects of human-computer interaction. For many of these applications, 3D data sets are increasingly accessed through the Internet. The number and complexity of these 3D models is growing rapidly, due to improved design and model acquisition tools, to the widespread acceptance of this technology, and to the need for higher accuracy. In many of these applications, human productivity or satisfaction would be significantly enhanced by the possibility of an immediate access to remotely located 3D data sets for visual inspection or manipulation. Even when image-based rendering [21, 20, 5] and progressive transmission techniques [12, 14] for adaptive resolution graphics are used to reduce the fraction of the 3D representation that must be transferred at any given time, geometry transfer remains the bottleneck. The anticipated phone and network bandwidth increases will not, by themselves, suffice to offset the explosion of the complexity and popularity of 3D models. Consequently, it is urgent to develop optimal bit-efficient formats and associated compression and fast decompression algorithms for 3D models.

Although many representations have been proposed for 3D models [28] polyhedra (or more precisely triangle meshes) are the de facto standard for exchanging and viewing 3D data sets. This trend is reinforced by the wide spread of 3D graphic libraries (OpenGL [24], VRML [3]) and other 3D data exchange file formats, and of 3D adapters for personal computers that have been optimized for triangles. Graphic subsystems can convert polygons and curved surfaces into an equivalent (or approximating) set of non-overlapping triangles, which may be rendered efficiently using hardware-assisted rasterizers [26, 24]. But to avoid the cost of this runtime conversion, most applications precompute and store the triangle meshes. Therefore, triangle count is a suitable measure of a model's complexity and triangle-meshes are an appropriate target for current efforts on compression [29].

A triangle mesh may be represented by its vertex data and by its connectivity. *Vertex data* comprises coordinates of all the vertices and optionally the coordinates of the associated normal vectors and textures. In its simplest form, *connectivity* captures the incidence relation between the triangles of the mesh and their bounding vertices. It may be represented by a triangle-vertex incidence table, which associates with each triangle the references to its three bounding vertices. For all meshes that are homeomorphic to a sphere, and in fact for most meshes in practice, the number of triangles is roughly twice the number of vertices. Consequently, when pointers or integer indices are used as vertex-references and when floating point coordinates are used to encode vertex locations, connectivity data consumes twice more storage than vertex coordinates. Furthermore, for most applications, vertex data may be compressed down to about a tenth of the uncompressed connectivity data, with an average of 12 bits per vertex location and 6 bits per vertex normal [4, 33, 37]. Consequently, we need to deploy aggressive schemes for compressing the triangle-vertex incidence table, from which most popular boundary data structures may be easily derived.

It is possible to hide some or all of the connectivity cost in the vertex encoding. For example, one could use some automatically computed triangulation as a first guess for connectivity and then only encode the necessary transformations that produce the correct connectivity. Another approach, proposed for 2D triangulations by Denny and Sohler [7], would be to encode the vertices in a specific order, which, when compared to a lexicographical (left-to-right) sorting of

these vertices, defines a permutation number. That number is a sequence of bits, which, for sufficiently large n, suffices to identify one among the possible labeled planar triangulated graphs of n vertices. Unfortunately, these approaches are not compatible with the schemes mentioned above for compressing vertex data. Indeed, these schemes require access to the connectivity information for predicting the data for each new vertex from previously encoded neighbors. They use variable length codes for the coordinate corrections between the predicted and the actual data. The better the predictions, the shorter the codes. The lack of connectivity information and an encoding of vertices in an order that does not capture some of their proximity relations would considerably increase the storage needed to encode vertex data, which already is the bottleneck when previously proposed connectivity compression schemes are used [6, 33].

To meet these two objectives, we have developed a new compression scheme, called Edgebreaker. It encodes into $2t$ bits or less the connectivity of any mesh of t triangles that is homeomorphic to a sphere. The encoding is independent of vertex locations. Previously proposed approaches require more storage or even exhibit non-linear asymptotic worst-case storage complexity. More general meshes with b exterior edges and h handles require $2t+b+(\log_2(h)+\log_2(t)+k)h$ bits or less, where k is a small constant. In practice, for meshes with relatively few handles and few bounding edges, the compressed data requires between 1.5 and 2 bits of storage per triangle. This ratio may be even lower for compressing patches with a complex bounding loops and relatively few interior vertices. Their results do not rely on statistic-based entropy or arithmetic coding schemes, which in general perform poorly on small or irregular data sets. Consequently, Edgebreaker is suitable for compressing all models and particularly attractive for compressing large catalogs of small models for remote instant access without overhead.

Edgebreaker organizes the vertices of the model along a spiraling vertex-spanning tree that is almost identical to the vertex-traversal orders produced by several recently proposed compression schemes [33, 37, 10]. Therefore, our connectivity compression technique may be trivially combined with several previously proposed schemes for compressing vertex data.

The rest of this paper is organized as follows: we start by defining our *Terminology and Notation* and proceed to *a Comparative Analysis of Prior Art*; we introduce the Edgebreaker approach by first focusing on simple meshes that are homeomorphic to a half-sphere and provide the details of *Compressing Simple Meshes*, the *Compressed Format*, and Decompressing Simple Meshes; then we discuss Extensions to More General Triangle Meshes.

Terminology and notation

To define our notation and domain, we use simple concepts of topology. Their precise definitions may be found in [22] or other textbooks on this topic. Let |X| denote the number of elements in the set X. Let T denote a set of topologically closed triangles T_i, for integer i in [1..|T|]. $\{T_i\}$ is the closed pointset of T_i. {T} is the union of these pointsets for all triangles in T. Let V denote the set of the vertices of T. For simplicity, and without loss of generality, we assume that the vertices of V may be uniquely identified by integer numbers between 1 and |V|. The connectivity may be represented by a triangle-vertex incidence table, which associates each triangle with three integer labels that reference its bounding vertices.

The compression algorithm described here is restricted to manifold representations of triangle meshes. In a manifold mesh, each edge is bounding one or two triangles and the star of each vertex **v** (i.e., its incident triangles and edges) remains connected, when **v** is removed. By replicating some of their non-manifold vertices, non-manifold meshes may be represented using data structures for manifold meshes and may hence be processed by our compression algorithms.

Edges that bound two triangles are called *interior edges*. Edges that bound exactly one triangle are called *exterior edges* and their union is denoted b{T} and called the *boundary* of {T}. The connected components of b{T} are one-manifold polygonal closed curves, called *loops*. Vertices of T that do not bound any exterior edge are called *interior vertices*. The set of all interior vertices is denoted V_I. The other vertices are called *exterior vertices* and their set is denoted V_E.

A choice of the cyclic order for the bounding vertices of a triangle X defines an orientation for X and imposes orientations on its bounding edges. The orientations of two adjacent triangles are compatible, if they impose opposite orientations on their common edge. A manifold mesh is *orientable*, if and only if there exists a choice of orientations that makes all pairs of adjacent triangles compatible. In this paper, we assume that the mesh is always orientable. Non-orientable surfaces maybe cut into orientable pieces by replicating interior edges by pairs of coincident exterior ones.

We define a *simple mesh* to be a triangle mesh that forms a connected, orientable, manifold surface that is homeomorphic to a sphere or to a half-sphere. Such meshes have no handles and have either no boundary or have a boundary that is a connected, manifold, closed curve, i.e.: a simple loop. The core of this paper deals with the compression of connectivity graphs that may—but need not—be imbedded in such a way that they represent the connectivity of simple meshes.

The Euler equation for simply meshes yields $t-e+v=1$, where t is the number of triangles, |T|, where v *is the total number of vertices*, $|V_I|+|V_E|$, and where e is the total number of the external and internal edges. Since there are $|V_E|$ external edges and $(3|T|-|V_E|)/2$ internal edges, we obtain by substitution: $|T|-|V_E|-3|T|/2+|V_E|/2+|V_I|+|V_E|=1$ and $|T|=2|V_I|+|V_E|-2$. When $|V_E|<<|V_I|$, there are approximately twice more triangles than vertices.

Comparative analysis of prior art

In this section, we first summarize the most relevant approaches for compressing vertex data and then review previously published schemes for compressing the connectivity of triangle meshes. We also propose improvements to some of these techniques and discuss their

expected or worst case *connectivity cost*. For compressed formats, where connectivity information is combined with vertex data, we define the connectivity cost to be the difference between the overall storage cost and the cost that would be necessary for compressing the vertex data alone. We organize the prior art of compressing connectivity into five categories: uncompressed data structures, triangle strips, vertex insertion, graph encoding, and vertex permutations.

Vertex data compression techniques

Typically, tessellated models are used for visualization, interference detection, or finite-element analysis. They are often approximations of curved shapes, which may have to be represented with higher degree surfaces for manufacturing and more advanced simulation and analysis applications. Even when a model represents a shape that is polyhedral by nature, the accuracy of the model is often limited during its creation by numeric round-off errors in the computation of geometric intersections, by the limited resolution of input techniques during design, or by measurement errors. Applications for which such numeric inaccuracies or such crude polyhedral approximations of curved shapes are acceptable do not in general require that vertex coordinates be stored with full floating point precision, as long as the geometry preserves the important topological and adjacency relations.

Following [6, 4, 33], we suggest to represent the vertex coordinates with k bits each, as integers between 0 and 2^k -1, defined over the smallest axis-aligned box that contains the model. For example, 10-bit quantization (k=10) will result in better than 0.5mm accuracy for any part of a car engine. Note that the quantization needs not be uniform for the entire model, but may be adjusted locally depending on the smallest triangle size or largest surface curvature [4]. Different accuracy levels for vertex data should be associated with different levels-of-detail for the mesh complexity, so as to integrate a progressive vertex data accuracy refinement with a progressive download of increasingly detailed approximations of the mesh [19]. Consequently, the coordinates of a vertex may often be stored using less than 30 bits, instead of 3 floating point numbers.

Furthermore, the storage for vertex data may be considerably reduced by using variable length encoding [6, 13, 33, 37, 19]. Indeed, if the compression and decompression algorithms compute identical estimates for the location of each vertex, it suffices to encode the corrective displacement vectors: The decompression algorithm will estimate the location of the next vertex and simply add it to the corrective vector. If the vertex coordinates are quantized to a small number of bits and if the estimates are good, many of the coordinates of the corrective vectors will be small integers. Entropy coding or other variable length schemes replace the frequently occurring integers with shorter codes. Thus, in highly tessellated models with quantized coordinates, compression ratios for V depend primarily on the precision of the vertex estimates.

For example, Taubin and Rossignac [33] have used vertex estimators based on a few ancestors in a vertex-spanning tree, whose edges correspond to some of the edges of the mesh. Each new vertex is expressed as $a\mathbf{A}+b\mathbf{B}+c\mathbf{D}+d\mathbf{D}+\mathbf{E}$, where **A**, **B**, **C**, and **D** are the successive ancestors of v in the vertex-spanning tree, where **E** is the corrective vector than must be encoded, and where a, b, c, and d are scalar values computed to minimize the **E**'s over the entire mesh. For highly complex models with finely tessellated surfaces, their technique approaches 12 bits per vertex, which represents an average of only 4 bits per coordinate (or 6 bits per triangle). Touma and Gotsman [37] suggested to use the estimate $\mathbf{A}+\mathbf{CB}$ for the third vertex of the triangle incident upon the gate g, where **A** and **B** are the vertices of g and **C** is the third vertex of the other previously processed triangle incident upon g.

To provide a meaningful basis for comparing the storage costs of the various coding schemes, we assume in this section: that it suffices to represent vertex coordinates with an accuracy of 1/1024 with respect to the overall dimension of the model; that $|V_E| \ll |V|$; and that there are no holes or handles. To simplify notation, v will stand for $|V|$ throughout this section.

Uncompressed data structures

Storing each triangle independently of all other triangles as the list of 10-bit integer coordinates for each one of its three vertices would require 90 bits per triangle. In such a simple representation, the connectivity is not coded explicitly, but can be recovered through geometric tests on the vertex locations. The location of a vertex is repeated on average 6 times. Thus, the storage used in excess of the vertex location is 60 bits per vertex or approximately 30 bits per triangle.

To avoid storing multiple representations of each vertex, we could store the vertex data table in a sequence and store connectivity as a sequence of triangle descriptors, each triangle been represented by 3 integer numbers that each identify the position of a vertex in the above vertex sequence. We would need at most $\lceil \log_2(v) \rceil$ bits per reference, where $\lceil x \rceil$ denotes the lowest integer greater than x. For simplicity, we will omit the "$\lceil\ \rceil$" from the formulae in the remainder of the paper. With this scheme, the connectivity cost is $3\log_2(n)$ bits per triangle. When the vertex data does not include anything but the vertex coordinates, this solution becomes more expensive than storing independent triangles for models with more than 512 vertices.

The advantages of both schemes may be combined by storing only the triangles, each represented by 3 vertex descriptors. Each vertex descriptor would start with a one-bit switch indicating whether this is a new vertex for which the three coordinates and other vertex data follow or whether the vertex has already been encountered, in which case the rest of the vertex descriptor contains $\lceil \log_2(i) \rceil$ bit which identify one of the i previously encountered vertices. This representation requires a total of $31v+5v\log_2(v)$ bits. If we subtract $30v$ for the vertex coordinates, the connectivity cost is $v(5\log_2(v)+1)$, or approximately $2.5\log_2(v)+0.5$ bits per triangle.

The decompression algorithm may keep track of which vertices are interior to the part of the mesh that has been recovered so far and which vertices are still exposed, i.e., are on the boundary of that part. Because only the exposed vertices need to be referenced, it is suitable to reduce their number. Bar-Yehuda and Gotsman have proposed a technique for visiting

the triangles in an order which guarantees that no more than $13(v)^{0.5}$ vertices are exposed at any given time [2]. Using such an improvement would lead to a connectivity cost of $1.25\log_2(v)+9.25$ bits per triangle.

Triangle Strips

A representation based on triangle strips, supported by OpenGL [24] and other graphic libraries, is used to reduce the number of times the same vertex is transferred and processed by the graphics subsystem. Basically, in a triangle strip, a triangle is formed by combining a new vertex description with the descriptions of the two previously sent vertices, which are temporarily stored in two buffers. Each new triangle, X, shares an edge with the previous triangle in the strip. Using a convention to orient the surface of the strip, we can label the other two "free" edges of X as the *left* and the *right* edge. One bit per triangle suffices to indicate whether the triangle is incident upon the left or the right edge of the previous triangle. The first two vertices are the overhead for each strip, so it is desirable to build long strips, but the automation of this task remains a challenging problem [9]. Instead of using such a left/right bit, OpenGL requires to alternate between left and right edges throughout the strip (see Fig. 1). Note that two consecutive right or left "moves" may be implemented in OpenGL by encoding a vertex twice without breaking the strip.

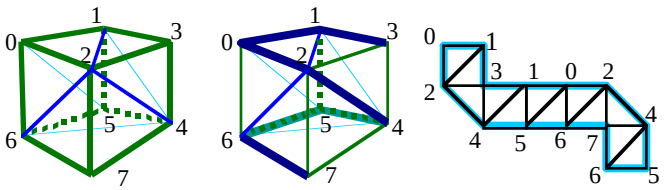

Figure 1: *The triangulated boundary of a polyhedron (left) may be cut (thick blue edges, center) into a flat triangulated polygon without interior vertices (right). When this polygon has no bifurcation, it may be represented by a single triangle strip, where triangles are attached to free edges of the previous triangle. OpenGL requires to alternate between the left and the right free edge Except for the first and last ones, each vertex has 3 incident edges (right). Note that, each vertex, except for 3 and 7, which are the end-points of the cut, is encoded twice. Indeed, in general vertices appear either in the boundary of two separate strips or are used by non-consecutive triangles within the same strip.*

Let us improve the triangle strip format and avoid vertex replication by using as above, in lieu of a replicated vertex, a reference to a previously decoded one. Assuming strips of length k«1 and using one bit per triangle to indicate whether the next triangle is attached to the left or the right edge of the current one and another bit per triangle to indicate whether the next vertex has been already encoded, the connectivity cost becomes 2 bits per triangle plus an average of $5/6\log_2(v)$ bits per triangle for the references to previously decoded vertices. The total connectivity cost of our modified triangle strip approach is $(5/3)v\log_2(v)$ bits.

Deering's compressed format [6] was designed for decompression by a hardware graphics adapter with very limited memory. Thus, random access to previously decompressed vertices was out of the question. Instead, Deering uses a 16-register cache to store 16 of the previously decoded vertices for subsequent references. When a previously decoded vertex no longer in cache is needed, a new instance of it must be decoded. Chow has proposed a technique for traversing the triangles of a mesh in an order that exploits Deering's architecture [4]. He sweeps a front of edges in a spiraling pattern to avoid creating isolated vertices. The generalized triangle strip is formed by connecting the triangle-corridors defined by two consecutive positions of the swept front. We estimate that on average one vertex out of eight must be encoded twice.

Deering's cache idea could be adapted for software decompression by providing access to all previously decoded vertices. Seven out of eight vertices would be in the cache when needed more than once and could be identified using 4 bits. One out of eight reused vertices would be in main memory and would require a reference of $\log_2(v)$ bits. So the average cost of identifying a reused vertex is $(7*4+ \log_2(v))/8$, plus one bit to distinguish the cached vertices from those in RAM. The connectivity cost would also include one bit per vertex to indicate whether each newly decoded vertex should be saved in the cache and two bits per triangle to indicate how to form the next triangle. Assuming that each vertex is used twice, the total connectivity cost would be $7.5+0.125\log_2(n)$ bits per vertex.

Progressive Vertex Insertion

Hoppe's Progressive Meshes [13] permit to transfer a 3D mesh progressively, starting from a coarse mesh and then inserting new vertices one by one. Instead of a vertex insertion to split a single triangle, as suggested in [8] for convex polyhedra, Hoppe applies a vertex insertion that is the inverse of the edge collapse operation used in many mesh simplification techniques [12, 27, 11]. A vertex insertion identifies a vertex **v** and two of its incident edges. It cuts the mesh open at these edges and fills the hole with two triangles. Vertex **v** is thus split into two vertices. In Hoppe's scheme, each vertex is transferred only once. The connectivity cost for each vertex is the identification of one of the previously transferred vertices (on average more than $0.5\log_2(v)$) plus the cost of identifying two of the incident edges (5 bits are sufficient if no vertex is bounding more than 32 edges). Thus, the connectivity cost per vertex would be more than $5+0.5\log_2(v)$.

Taubin et al. [35] proposed to group Hoppe's splits into refinements. Each refinement doubles the number of triangles at an average expected cost of 3.5 bits per triangle. Each refinement of their progressive forest split method identifies a set of cut edges, which are grouped into maximally connected components and stored as spanning vertex-trees. Removing the cut edges of a tree produces a topological hole that is a simple polygon, whose boundary is known. The triangulation of the interior of that polygon, which does not contain any interior vertices, may be encoded using a simplified version of the Topological Surgery of Taubin and Rossignac [33], which, as discussed later, exhibits a non-linear worst case behavior.

Li and Kuo [19] combine progressive transmission of connectivity refinements, which insert new vertices one at a time, with progressive transmission of vertex data, which adds resolution to the vertex coordinates. They use a simple vertex-decimation scheme to produce series of operations that decrease the levels-of-detail of the model. They encode the inverse connectivity refinement operations by identifying the base triangle where the vertex must be inserted, and by labeling the surrounding edges to indicate which of these must be flipped to restore the correct incidence. This approach leads to an average connectivity cost of $\log_2(v)+10$ bits per vertex.

Graph encoding

The adjacency information for a simple mesh without boundary is a labeled triangulated planar graph and may be represented by the triangle-vertex incidence table using $3\log_2(v)$ bits per triangle. However, if we adopt a convention for constructing a vertex-spanning tree of such graphs and if we accept to label the vertices with integers that correspond to the order in which they are visited by a traversal of this spanning tree, our compression problem may be reduced to one of computing a bit-efficient representation of an unlabeled triangulated planar graph. Turan has shown that the structure of a labeled planar graph may be encoded using slightly less than $12v$ bits [38]. Having a constant number of bits per vertex has a significant advantage over the previous approaches, which all include a $\log_2(v)$ factor, especially for highly complex meshes. Turan builds a vertex-spanning tree and uses it to represent the boundary of a topological polygon of $2v-2$ edges. The structure of this tree is encoded using $4v-4$ bits. There are at most $2v-5$ edges that do not belong to the vertex-spanning tree. These may be encoded using 4 bits each. The overall connectivity cost is thus, $12n-24$ bits.

A triangle-spanning tree of T is a binary tree, whose nodes correspond to all the triangles of T and whose edges correspond to some of the interior edges of T. A depth-first traversal of such a spanning tree corresponds to a walk on the entire mesh that starts at the root triangle and recursively visits the neighboring triangles that have not been previously visited. The spanning tree may be encoded using 2 bits per triangle as follows. Each triangle is visited by coming to it from an adjacent, previously visited neighbor triangle. Because the surface is an orientable manifold, the other two edges may be uniquely labeled as the *left* and the *right* edge. We can use one bit for each one of these edges to indicate whether they are to be broken or not during the traversal (i.e., whether they connect triangles that hold a parent-child relation in the triangle spanning tree).

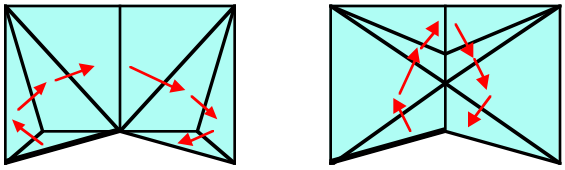

Figure 2: *The two meshes have identical boundaries and triangle-spanning trees, which are shown by red arrows and described by the sequence of moves: left, right, left, right, right, left, right.*

If we could always derive a complete representation of the connectivity from the spanning tree of the triangles of T, we would have attained our objectives and would have a very simple scheme for encoding simple meshes using 2 bits per triangle. Unfortunately, the triangle-spanning tree does not capture by itself the entire topology of the incidence graph (see Fig. 2 for a counterexample).

The Topological Surgery method recently developed by Taubin and Rossignac [33, 34] encodes both a vertex-spanning tree and its dual triangle-spanning tree. Cutting through the edges of the vertex-spanning tree produces a triangulated surface that is a simple mesh without internal vertices and thus may be completely represented by a triangle-spanning tree. As demonstrated above, encoding that tree would not, by itself, suffice to represent the connectivity. Taubin and Rossignac encode both the triangle-spanning tree and the topology of the vertex-spanning tree. Together, these two graphs provide enough information to recover the connectivity of the mesh. Basically, the vertex-spanning tree matches pairs of edges in the boundary of the polygon defined by the triangle-spanning tree. Taubin and Rossignac encode both trees using a run length method. Each sequence of consecutive ancestors with a single child is grouped into a run and encoded by simply storing its length, using $\lceil n \rceil$ bits. Two bits per branching nodes are used to capture the topology of the tree. For pathological cases, with a non-negligible proportion of multi-child nodes, this approach does no longer guarantees a linear storage cost, but for complex meshes, the cost of encoding both trees may amount to less than a bit per triangle.

The author has proposed a variation of the above Topological Surgery method [31], where, instead of using a run-length encoding of the vertex- and triangle-spanning trees, one uses 2 bits per vertex to encode the vertex-spanning tree (one bit indicates the presence of a child while the other bit indicates the presence of a right sibling) and 2 bits per triangle to encode the triangle-spanning tree (one bit indicates the presence of a right child, while the other bit indicates the presence of a left child). With twice more triangles than vertices, the guaranteed worst case connectivity cost of this representation is 3 bits per triangle.

Following Tutte's studies [39] and using an enumeration theorem, Itai and Rodeh [15] show that any unlabeled rooted non-separable triangulated planar graphs of n vertices (i.e., the incidence graph of triangle meshes homeomorphic to a sphere) may be represented by 4n bits. Furthermore, they propose a linear algorithm for constructing a representation of any labeled planar graph using at most $1.5n\log_2(n)+6n+O(\log_2(n))$ bits, while the theoretical minimum is $n\log_2(n)+O(n)$. Their approach uses a triangle as the initial outer loop and then shrinks that loop by removing one triangle at a time. They always delete the triangle that is incident to the smallest vertex v_1 in the outer loop and is bounded by the outer loop edge that starts at v_1. They distinguish four cases: (1) The third vertex precedes v_1 in the outer loop; (2) It follows the successor of v_1; (3) It is somewhere else in the outer loop; and (4) it is not on

the outer loop. Operations (3) and (4) each require $\log_2(v)$ bits to identify a vertex in the not yet processed part of the mesh.

A variation of Itai and Rodeh's method was recently reported by Gumhold and Strasser [10]. It is closely related to the Edgebreaker method reported here. Although developed independently, both the Edgebreaker and the method of Gumhold and Strasser perform the same traversal of the mesh and, at each step, remove a triangle and encode the necessary information to reconstruct the triangle by distinguishing several cases that include the four cases of Itai and Rodeh. Edgebreaker uses the letters L, R, S, and C to identify cases 1 through 4 of Itai and Rodeh. Gumhold and Strasser add the case where a boundary edge is reached. Edgebreaker does not need to distinguish this case, since it encodes the bounding loop at the beginning of the vertex array. However, Edgebreaker adds the case E, which corresponds to the situation where the current triangle is not adjacent to any other remaining triangle. Both these approaches avoid the $\log_2(v)$ bits cost associated with case (4) of Itai and Rodeh by encoding the vertices in the order in which they are used by case (4). With each case (3) operation, Gumhold and Strasser must encode the reference to a vertex in the current boundary, which requires $\log_2(v)$ bits and makes their storage costs a non-linear function of v. Note that Edgebreaker uses a decompression preprocessing step to compute these vertex-references from the sequence of symbols, and therefore exhibits a linear storage cost. For common meshes, Gumhold and Strasser report compression results between 1.7 and 2.15 bits per triangle using Huffman encoding of the bit stream.

Keeler and Westbrook [16] improve on Turan's results and propose a technique for encoding planar graphs with $4.6v$ bits. They also build a triangle-spanning tree. Each triangle of the tree, except the root, shares an edge with its parent and may have zero, one, or two children and thus two, one, or zero free edges. They append free edges to the leaves of the triangle-spanning tree and label them. Encoding the graph and the labels requires an average of $1+\log_2(3)/3$ bits per edge. The authors suggest a coding scheme based on a series of graph transformations.

Touma and Gotsman [37] also encode the vertices along the vertex-spanning tree in the same order as Taubin and Rossignac, Gumhold and Strasser, and Edgebreaker. They distinguish only two cases, which correspond to the cases (3) and (4) of Itai and Rodeh and to the Edgebreaker's cases S and C. The other cases are not encoded. Instead, Touma and Gotsman encode the degree of each vertex, i.e., the number of incident edges and use it to automatically identify the other cases. During decompression, they keep track of the number of the already decoded triangles that are incident upon each vertex and are thus capable of identifying the R, L, and E triangles automatically. For highly tessellated regular models, where the degree of the vertices follows almost regular patterns, they report compression results of less than a bit per triangle using Huffman encoding. However, for smaller and less regular meshes, the required storage may easily exceed 2 bits per triangle. As Itai and Rodeh and as Gumhold and Strasser, they require that with each S operation be associated a vertex reference, which requires $\log_2(v)$ bits, prior to Huffman compression.

Vertex permutation

Inspired by [17] and improving on [23, 32], Denny and Sohler have recently proposed a technique for encoding the incidence of *planar* triangulations of sufficiently large size as a permutation of the vertices in V [7]. They show that there are less than $2^{8.2v+O(\log v)}$ valid triangulations of a planar set of v points, and that for sufficiently large v, each triangulation may be associated with a different permutations of these points (there are approximately $2^{v\log(v)}$ such permutations). Their approach requires first transmitting an auxiliary triangle that contains the entire set and then the vertices of V in a suitable order, computed by the compression algorithm. The decoding process receives the vertices in batches, sorts them lexicographically, computes a permutation number by comparing the order in which the vertices were received with their lexicographic order, then sweeps over the current triangulation from left to right. At each vertex of the current batch, it identifies the enclosing triangle [18] and the vertex is inserted according to the incidence relation derived from the bit string that encodes the permutation number. Compression constructs the successive batches through repetitive plane-sweeps, during which vertices are removed incrementally and the resulting holes re-triangulated. For each point, the information needed to reconstruct that triangulation is encoded in the permutation of the vertices of the batch. The batches are decompressed in inverse order. Although for sufficiently complex models the cost of storing the connectivity is null, the unstructured order in which the vertices are received and the absence of the incidence graph during their decompression makes it difficult to combine this approach with the predictive techniques for vertex encoding discussed earlier.

Compressing Simple Meshes

We focus in this section and the next two on simple meshes. Then, we explain how to generalize our scheme to non-manifold triangulated surfaces with an arbitrary number of handles and several bounding loops.

The Edgebreaker compression algorithm performs a series of steps. Each step removes one triangle from the current mesh. At each stage, the remaining portion of the mesh is composed of one or several regions, denoted R_i, which are simple meshes. Technically, each region is the union of triangles of T, whose interior is contained in one maximally connected component of the interior of the union of the remaining triangles. Note that two regions may share a vertex, but not an edge. The edges bounding each region form a closed manifold polygonal curve, called loop, which has no self-intersections. One edge of each loop is called a gate. A stack contains references, S_0, S_1, S_2,… to all the gates. The top of the stack, S_0, references the active gate, g. Let R_0 be the region incident upon g and let B denote the bounding loop of R_0. Note that B contains g. This notation is illustrated in Fig. 3. Note that for

simple meshes, the initial configuration has a single region and a single loop.

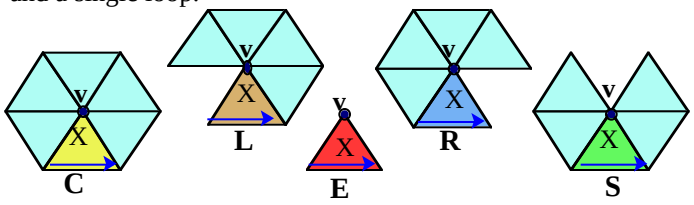

Figure 3: *During compression, the top of the stack, S_0, points to half-edge g, called the gate, which identifies the boundary B of the active mesh R_0. The only triangle that is incident upon g (shown in green) will be removed from R_0. When present, the other entries in the stack point to half-edges included in the bounding loops of regions that will be compressed later. These will become gates when they are popped to the top of the stack. Note that R_0 may later be split into separate regions, which will be tracked using the stack.*

At each step, Edgebreaker identifies the unique triangle, X, that is part of R_0 and is incident upon g. Let **v** be the only vertex of X that does not bound g. Edgebreaker analyzes the relation that **v** has with respect to B and g, distinguishing 5 cases labeled C, L, E, R, and S (see Fig. 4).

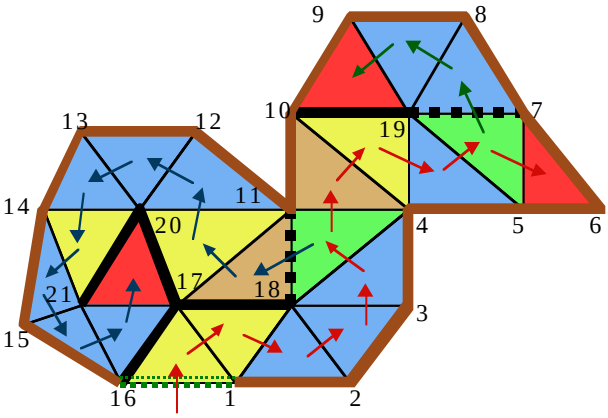

Figure 4*: The triangle X, identified because it is the only triangle in the remaining portion of the mesh that is incident upon the gate g, will be removed. The associated operation is of type C if the third vertex* ***v*** *of X that is not bounding g has not yet been visited. If* ***v*** *has been visited, then it is included in B. If it both immediately follows the end-vertex of and immediately precedes the start vertex of g along B, then X is the last triangle of R_0 and we record an E operation. If* ***v*** *immediately follows g, but does not precedes it, we record an R operation. If* ***v*** *does not immediately follow g but precedes it, we record an L operation. Finally, if* ***v*** *lies in B but is not the vertex that immediately precedes or follows the vertices of g, then we record an S operation.*

The selection of the appropriate case may be performed by the following sequence of tests:

IF **v** ∉ B THEN case C
  ELSE IF **v** follows g
    THEN IF **v** precedes g THEN case E ELSE case R
    ELSE IF **v** precedes g THEN case L ELSE case S

Edgebreaker constructs a compression history H by appending op-codes selected from the set {C, L, E, R, or S} to identify the successive steps that must be used to reconstruct the mesh during decompression. Edgebreaker also builds a list **P** of vertex identifiers, in the order in which they are reached by C operations as the third vertex, **v**, of the triangle incident upon the gate. This list will define the order in which the vertices will be encoded. The history H will be compressed using binary codes or any desired compression scheme. Surprisingly, as demonstrated in the section on *Decompression Algorithms*, the information contained in H suffices to recover the labeled planar triangulated graph that represents the connectivity of T. The vertices referenced by the graph are labeled with integer indices (1, 2, 3…) that represent the order in which the corresponding vertex data will be recovered at decompression. During compression, it suffices to encode the vertices in the order of their references in **P**. For meshes with boundary, **P** is initialized to the references of the vertices of the initial loop B as they are encountered by walking around it, starting with the end-vertex of the gate. Fig. 5 illustrates this process.

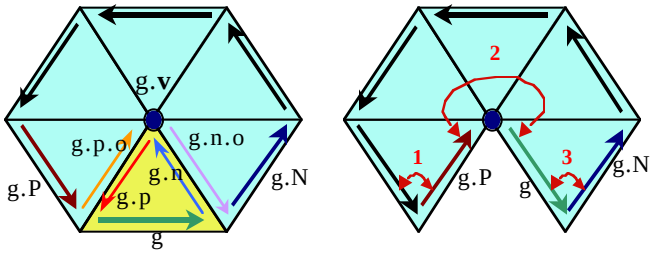

Figure 5*: This mesh may represent the final stages of the compression of a large region in the mesh or the full compression of a small simple mesh with boundary. Starting at gate g, Edgebreaker removes triangles by following the dark arrows: first red, then green, then blue. Triangles are color-coded as in Figure 4 indicating the type of the associated operation(C yellow, R blue, S green, L brown, and E red). The history is H=CCRRRSLCRSERRELCRRRCRRRE. The thick dotted dark green line is the gate. The rest of the boundary is shown with a thick brown line. The thick black lines identify edges that have never been gates. Together, the thick solid lines define a vertex-spanning tree rooted at the end-vertex of g and cutting the surface into a topological polygon. The thick black dotted edges are gates that have been on the stack. The vertices are marked with integer indices that indicate the order in which their references are to be included in* ***P****. Note that each interior vertex corresponds to a C operation (yellow triangle).*

To clarify some implementation details, we introduce a simple data structure for storing both the connectivity of the mesh and the links between the successive edges in the bounding loops. This data structure is based on the concept of a half-edge used in many polyhedral representations (see the author's survey [28]). A half-edge h is the association of an edge e of T with a triangle X incident upon e. Note that each half-edge is oriented and that to each internal edge of T correspond two half-edges with opposite orientation, each induced by the corresponding

triangle. With each half-edge h, we associate the following entities (see Fig. 6):

- h.s is the starting vertex for h
- h.e is the ending vertex for h
- h.v is the third vertex of X that does not bound h
- h.n is the half-edge that follows h in the boundary of X
- h.p is the half-edge that precedes h in the boundary of X
- h.o is the opposite half-edge (When e is an interior edge, h.o associates e with the other incident triangle.)
- h.N is the half-edge that follow h in B that contains h
- h.P is the half-edge that precedes h in B that contains h

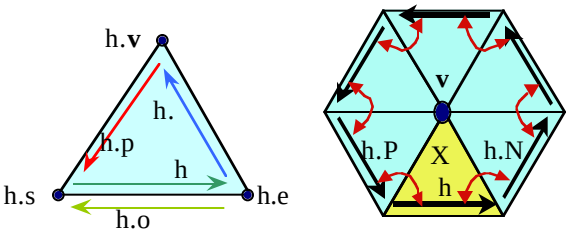

***Figure 6**: A half-edge, h, green arrow (left), points to its starting and ending vertices, h.**s** and h.**e** and to the opposite vertex h.**v**. The other two edges of the triangle associated with h are denoted h.p and h.n (red and blue arrows). The opposite half-edge, h.o (lime arrow) provides access to the adjacent triangle when it exists. Curved red bi-directional arrows (right) represent links h.P and h.N that organize the bounding black half-edges of B into a doubly-linked cyclic list.*

For clarity, we have extended the object-oriented notation to reference the various fields associated with a half-edge structure. In the algorithms presented below and in Figs. 7 through 11, we assume the following semantics: The assignment h.x=y changes the content of the field h.x associated with h so that it points to y. For example, if these references were stored as parallel arrays of integer indices, the statement h.x=y would be coded as x[h]=y and the statement g.n.o.P=g.p.o could be coded as P[o[n[g]]]=o[p[g]]. Efficient techniques for storing and constructing such tables from a triangle-vertex incidence array are suggested in [30].

The compression algorithms also uses binary flags, **v**.m and h.m, to mark each previously visited vertex **v** and each half-edge h that is in a bounding loop of the remaining portion of the mesh. The vertex-flags are used to distinguish between C and S cases without having to traverse B. The edge-flags are used during S operations to accelerate the process of finding the bounding half-edge b such that g.**v** is b.**e**. The notation **P**=**P**|**v** means that the reference to vertex **v** is appended to the list **P** and H=H|C means that the op-ode for the C operation is appended to the history H. We also use the # sign to start inline comments.

During the initialization part of the compression process, Edgebreaker loads into **P** the references to the vertices encountered by marching along the initial bounding loop, starting from the end-vertex of the gate. It also marks the bounding edges and vertices and sets the .P and .N links for all bounding edges. It initializes the stack to point to the half-edge that is the initial gate.

Then compression identifies the operation type using:

```
IF NOT g.v.m THEN case C          # v not marked
   ELSE IF g.p==g.P               # left edge of X is in B
        THEN IF g.n==g.N THEN case E ELSE case L
        ELSE IF g.n==g.N THEN case R ELSE case S
```

and then performs the corresponding changes to the half-edge data structure, as explained in Figs. 7 to 11.

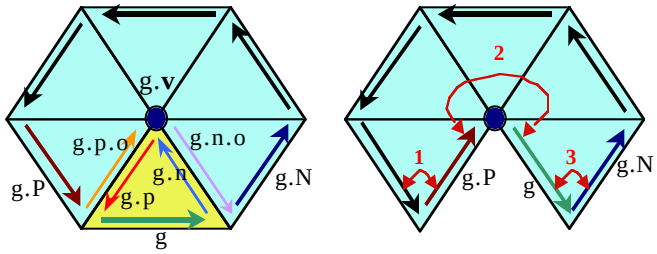

***Figure 7**:* **Case C**

The initial mesh (left) corresponds to a C case. The result (right) is obtained by creating 3 bi-directional links in B (two-headed red arrows). Thicker arrows show bounding half-edges.

```
H=H|C;                          # append C to history
P=P|g.v;                        # append v to P
g.m=0; g.p.o.m=1;               # update flags
g.n.o.m=1; g.v.m=1;
g.p.o.P=g.P; g.P.N=g.p.o;       # fix red link 1 in B
g.p.o.N=g.n.o; g.n.o.P=g.p.o;   # fix red link 2 in B
g.n.o.N;=g.N; g.N.P=g.n.o       # fix red link 3 in B
g=g.n.o; StackTop=g;            # move gate
```

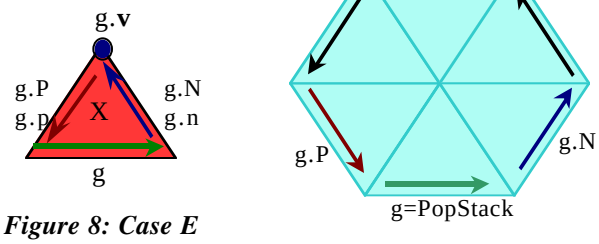

Figure 8: Case E

```
H=H|E;                          # append E to the history
g.m=0; g.n.m=0; g.p.m=0;        # unmark edges
PopStack; g=StackTop;           # pop stack: next region
```

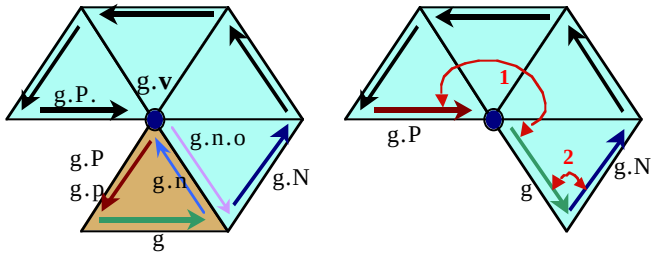

***Figure 9**:* **Case L**

```
H=H|L;                          # append L to history
g.m=0; g.P.m=0; g.n.o.m=1;      # update marks
g.P.P.N=g.n.o; g.n.o.P=g.P.P;   # fix red link 1 in B
g.n.o.N=g.N; g.N.P=g.n.o;       # fix red link 2 in B
g=g.n.o; StackTop=g;            # move gate
```

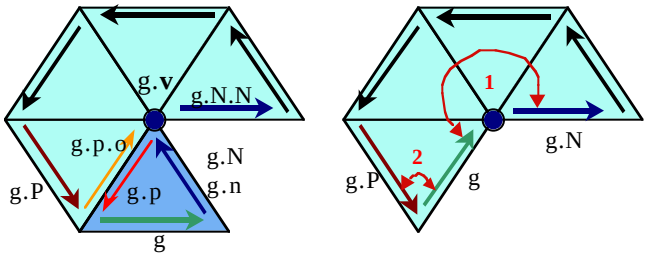

Figure 10: **Case R**

```
H=HlR;                          # append R to history
g.m=0; g.N.m=0; g.p.o.m=1;      # update marks
g.N.N.P=g.p.o; g.p.o.N=g.N.N;   # fix red link 1 in B
g.p.o.P=g.P; g.P.N=g.p.o;       # fix red link 2 in B
g=g.p.o; StackTop=g;            # move g
```

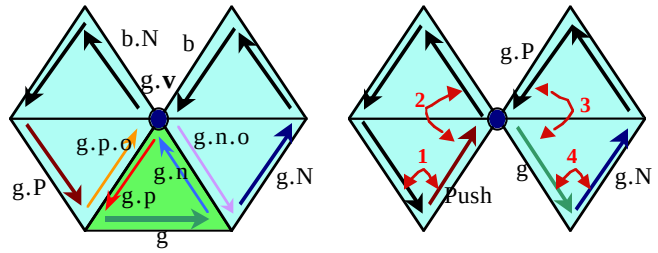

Figure 11: **Case S**

```
H=HlS;                          # append S to history
g.m=0; g.p.o.m=1; g.n.o.m=1;    # update marks
b=g.n;                          # initial candidate for b
WHILE NOT b.m DO b=b.o.p;       # turn around v to marked b
g.P.N=g.p.o; g.p.o.P=g.P;       # fix red link 1 in B
g.p.o.N=b.N; b.N.P=g.p.o;       # fix red link 2 in B
b.N=g.n.o; g.n.o.P=b;           # fix red link 3 in B
g.n.o.N=g.N; g.N.P=g.n.o;       # fix red link 4 in B
StackTop=g.p.o; PushStack;      # push g.p.o on stack
g=g.n.o; StackTop=g             # move g
```

Because the preconditions for the L, R, C, S, and E operations are mutually exclusive and cover all possible cases, and because these operations all decrement the triangle count in T, the compression process removes all triangles of T and always terminates.

Compressed format

The compressed format contains a few *Selectors*, the *History*, and the *Vertex Data*. We discuss them in reverse order.

Vertex data

We distinguish three situations:

1. To compress an isolated, triangulated surface, we may need to encode a simple mesh along with its boundary.
2. To compress the subset of a larger triangle mesh, we need to encode the connectivity and interior vertices of a simple mesh, but need not encode its bounding vertices, because they will be already available prior to decompression. This, for example may be the case when we compress the refinement of a specific feature in a larger mesh represented at a coarse level-of-detail [14].
3. To encode the triangulated boundary of a simply connected manifold solid, we need to encode a closed mesh without boundary.

For situation 1, the vertex data stream starts with the exterior vertices listed in the order in which they occur around B, starting with the end-vertex of g. For situation 2, that list is omitted. For situation 3, an edge is selected as the initial gate and its end- and start-vertices are first coded in the vertex data stream.

The rest of the vertex data stream contains data for the interior vertices, encoded in the order specified in the remaining part of **P**. The binary format that is used for encoding vertex data depends on which compression scheme is used.

History

The best coding strategies for the history, H, depends on the size of the mesh and on the ratio $|V_E|/|V_I|$.

For very complex meshes, the most effective option is to temporarily store H as a sequence of symbols from the set {C, L, E, R, S} and to compute, as a post-processing compression step, an optimal custom scheme for each individual mesh. Some agreed-upon convention could then be used to include the description of the particular coding scheme before H in the compressed format. An alternative is to use progressive coding schemes [41, 40, 25]. A number of general-purpose data compression schemes may be used for this purpose and will not be further discussed here. They may yield very high compression rations for large regular meshes, but often perform poorly for large and irregular meshes and for small meshes. Instead we focus our discussion on practical schemes that are effective for small meshes and on demonstrating that for simple meshes, Edgebreaker provides the best guaranteed worst case compression.

If we use fixed binary op-codes with 1 bit to encode each C operation and 3 bits to encode each other operation (for example, we use 0 for C, 100 for S, 101 for R, 110 for L, and 111 for E), the total number of bits needed to encode H is $c=|C|+3(|S|+|L|+|R|+|E|)$, where $|X|$ denotes the number of X-type operations in H. Because there is a one-to-one association between the vertices of V_I and the triangles processed by a C operation, we have $|C|=|V_I|$.

Hence, $|S|+|L|+|R|+|E|=|T|-|C|=|T|-|V_I|$ and $c=|V_I|+3(|T|-|V_I|)$, which yields: $c=2|T|+(|T|-2|V_I|)$. Given that $|T|-2|V_I|=|V_E|-2$, we obtain $c=2|T|+|V_E|-2$. Consequently, for simple meshes with a relatively simple initial boundary, we have $|V_E|<<|V_I|$, leading to $|V_E|<<|T|$, and to $c\approx 2|T|$.

To encode a simple mesh without boundary, such as the entire surface that bounds a manifold 3D solid, it suffices to "cut open" one of its edges, declare it to be the initial loop, B, and include the encoding of its two vertices at the top of the vertex list, as discussed above. In that case, $|V_E|=2$ and $c=2|T|$, which is exactly 2 bits per triangle. The connectivity of such meshes is a planar triangulated graph. Thus, we have introduced a new representation of such graphs, which is more compact than previously proposed solutions [39, 15, 38, 23, 16].

The CL and CE sequences of operations correspond to situations where two triangles are identical (have the same vertices). By definition, these situations are impossible in simple meshes. We can exploit this constraint to increase the expected compression ratio of Edgebreaker by using a slightly more complex coding scheme. We use two different code sets: the general code set proposed above for operations that do not follow a C operation and a special code set for operations that follow a C. The special code set is still 0 for C, but reduces to a 2-bit op-code for the other two operations: 10 for S, and 11 for R. In the worst case, with long sequences of consecutive C's, this encoding method has no effect on the bit-count. At best however, when all C's are separated, it reduces the bit-count to an average of 1.5 bit per triangle (because there are as many C's as other operations).

When Edgebreaker is used to compress small surface patches with a relatively large number of edges in their boundary, the above binary codes will never exceed an average of 3 bits per triangle, but are not optimal. Because, in such cases, the R operation is the most frequent in the sequence, the op-codes, proposed earlier, should be replaced by others, where R is a one-bit code (say 0) and the other four operations have 3-bit codes. Under these new conditions, c=3|T|-2|R|, which implies that, if most of the triangles correspond to an R operation (which is the case for a fan of triangles), the sequence representing G may be compressed down to 1 bit per triangle.

For meshes that do not fall in these two categories (interior-heavy or boundary-heavy), we suggest a post-processing compression step, which would compute the optimal op-code assignment for each operation, taking into account their frequencies and the constraints on impossible sequences. The resulting codes would be transmitted before H, using some convention. For example, we may use length-value tuples (2 bits to encode the bit length followed by the actual code) for all the 8 cases discussed earlier: C following a C, R following a C, E following a C, and occurrences of C, R, L, S, and E that do not follow a C. This table will take at most 42 bits.

Selectors

The compressed format may start with selectors, indicating whether the external vertices are included and which coding method is used for the history.

Decompressing simple meshes

The decompression algorithm receives a binary encoding of the history, H, which contains only the sequence of op-codes generated by the compression algorithm described above. It produces a triangle table, where each triangle is represented by three labels. These labels are consecutive integers assigned to vertices in the order in which the vertices are first encountered by the decompression algorithm. Note that it is the same order as the one in which they are first encountered by the compression algorithm.

Decompression performs two traversal of the input stream: *Preprocessing* computes |T|, $|V_E|$, $|V_I|$, and the offsets for all the S operations, which are stored in the offset table O; *Generation* creates the triangles in the order in which they were deleted by the compression process and, for each triangle, stores the labels of its 3 vertices. To compute the correct labels at decompression, we simply increment an integer vertex-counter, c, each time we encounter a C operation and use c as the label of the new vertex, g.**v**. We provide the details for both phases and illustrate them on an example.

Preprocessing

The Edgebreaker decompression preprocessing phase reads the input stream, i.e.: an encoding of the sequence of op-codes, decodes the op-codes one at a time, stores them in H for the *Generation* phase, and performs the actions described below. This process continues until the number of encountered E op-codes exceeds the number of encountered S op-codes.

It use the following variables and data structures:

- t, initialized to zero, tracks the total number of operations. The final value of t is the triangle count.
- d, initialized to zero, tracks the value |S|–|E|. d becomes negative after processing the last E operation of H.
- c, initialized to zero, tracks the number of C operations encountered so far. The final value of c *is* $|V_I|$.
- e, initialized to zero, tracks the value 3|E|+|L|+|R|–|C|–|S|. The final value of e *is* $|V_E|$. Values of e resulting from S operations will be pushed on the stack.
- s, initialized to zero, tracks the number of S operations encountered so far. We use s to relate e to the corresponding S, when e is pushed onto the stack.
- An initially empty stack, where we save (e,s) pairs resulting from S operations and use them during the corresponding E operations to compute the offset.
- O, an initially empty table of offsets.

At each step depending on the op-code, Edgebreaker performs the following operations:

Case S: e–=1; s+=1; push(e,s); d+=1;
Case E: e+=3; (e',s')=pop; O[s']=e–e'–2; d–=1;
IF d<0 THEN stop. # This is the end of the history
Case C: e–=1; c+=1;
Case R: e+=1;
Case L: e+=1;

At the end of the preprocessing phase, |T|=t, $|V_I|$=c, $|V_E|$=e, and O contains the desired offsets, sorted in the order in which the corresponding S operations occur in H. The remainder of this subsection explains why this simple procedure produces the desired results.

|T|=t, because each operation corresponds to a different triangle. Since only C operations require the introduction of new vertices, $|V_I|$=|C|. Deriving the count of external vertices, $|V_E|$, is slightly more complex. We know that at the end of the whole decompression process, the boundary of the remaining region of T must have zero edges, because this region is empty. If we can extract from H how many edges have been added or deleted by the steps of the compression process, we will know the initial length of B. Each operation alters the total count of edges (and thus of vertices) in B as follows. R deletes two edges from B, but exposes a new one, thus decreases the edge count by 1. L does the same. E removes 3 edges. C and S increase the edge-count by 1 because they each

remove one edge from B and insert two new edges. We can track the total count regardless of the topological changes in the boundary that may be produced by S operations. These edge-counts changes lead to the following formula: The number of edges, and hence of vertices, in the initial bounding loop, B, is 3|E|+|L|+|R|−|C|−|S|.

To compute the offsets, we first note that S and E operations are paired and work as parentheses in the following sense: any sub-string of H that starts at an S and finishes at the corresponding E operation encodes the incidence graph of a simple mesh that is a subset of T. We use *e* to track the value of the expression 3|E|+|L|+|R|−|C|−|S| for the already processed subset of H. The difference between the values of *e* at the E operation and the value of *e* at the corresponding S operation is the number of vertices in the boundary of that subset. We subtract 2 from that number, because we are not counting the two vertices of g as part of the offset.

To keep track of matching S and E operations, we use a stack, which we push for S operations and pop for E operations. The stack is used to save the *e* value for each S. When the stack is popped, we will subtract this value from the current value of *e*, which is associated with the matching E.

The order in which S operations are encountered in H may be different from the order in which the matching E operations occur. Because each offset value is only computed when an E operation is reached, we cannot simply save it in the next available entry in O. Therefore, we associate with each *e* pushed on the stack the corresponding value of *s*, which identifies the S operations. When popped from the stack, this *s* value, denoted *s'*, is the index of the entry in O for the offset *e*−*e'*−2, where *e'* is the value of *e* that was pushed on the stack along with *s'*.

We illustrate the preprocessing phase of decompression on H=CCRRRSLCRSERRELCRRRCRRRE, which corresponds to meshes with a connectivity homeomorphic to the mesh of Fig. 5. After each operation, the table below indicates the resulting values of the variables: *d*, *c*, *e*, and *s*, as defined above. The variables *e'* and *s'* refer to the content of the top of the stack. Note that this process first enters an offset value of 1 in O[2] and then enters an offset value of 6 in O[1]. The final value of *e* indicates 16 vertices in the external loop. The final value of *c* indicates 5 internal vertices. We did not include the triangle count *t*, which is incremented from 1 to 24.

	C	C	R	R	R	S	L	C	R	S	E	R	R	E	L	C	R	R	R	C	R	R	R	E
d	0	0	0	0	0	1	1	1	1	2	1	1	1	0	0	0	0	0	0	0	0	0	0	-1
c	1	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	5	5	5	5	5
e	-1	-2	-1	0	1	0	1	0	1	0	3	4	5	8	9	8	9	10	11	10	11	12	13	16
s	0	0	0	0	0	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
e'						0	0	0	0	0	0	0	0											
s'						1	1	1	1	2	1	1	1											
O[*s*]											1		6											

Generation

The generation phase allocates a table of triangle-vertex incidence relations, denoted TV, of |T| entries, each will combine the three integer vertex labels of a triangle. Then, it initializes the vertex counter, *c* to $|V_E|$, so that references to external vertices precede references to internal vertices. It constructs the bounding loops, B, a circular doubly-linked list of $|V_E|$ edges, each edge, G, containing the pointer G.P to its predecessor and G.N successor in the loop, and an integer label, G.**e**, which identifies the end-vertex of G. The labels are initialized with increasing integers between 1 to $|V_E|$. Edgebreaker also creates a stack of references to edges and initializes it to a single entry, which refers to the first edge in the loop. The top of the stack is called the gate, and will be denoted G. We use upper case letters for the edges of B to distinguish them from the half-edges used during compression. Finally, Edgebreaker initializes a triangle counter, *t*, and *s*, the counter of S operations, to zero.

After this initialization, Edgebreaker traverses H and, for each operation, it increments *t*, stores the labels of the 3 vertices of the current triangle X in entry number t of the table of triangles, and updates B, G, and the stack, if necessary.

At each stage, the gate, G, already identifies two of the vertices of the current triangle, X. These are G.P.**e** and G.**e**. The computation of the third vertex depends on the current op-code. Depending on the current op-code, we perform the operations listed below. We use the following notation: x++ returns the x value and then increments x while ++x increments x first and returns the result. The terms "REPEAT" and "*n* TIMES" delimit a block of instructions. For simplicity, we do not include operations that release the memory allocated during the process.

Case C: TV[++*t*]=(G.P.**e**, G.**e,** ++*c*);.
New Edge A; A.**e**=c;
G.P.N=A; A.P=G.P;
A.N=G; G.P=A;

Case R: TV[++*t*]=(G.P.**e**, G.**e,** G.N.**e**);
G.P.N=G.N; G.N.P=G.P;
G=G.N;

Case L: TV[++*t*]=(G.P.**e**, G.**e,** G.P.P.**e**);
G.P=G.P.P; G.P.P.N=G;

Case E: TV[++*t*]=(G.P.**e**, G.**e,** G.N.**e**); G=pop;

Case S: D=G.N; REPEAT D=D.N; O[++s] TIMES;
TV[++*t*]=(G.P.**e**, G.**e,** D.**e**);
New Edge A; A.**e**=D.**e**;
G.P.N=A; A.P=G.P;
pop; push(A);
A.N=D.N; D.N.P=A;
G.P=D; D.N=G;
push(G);

Although in practice only a small fraction of operations are of the S type and the average length of the loop is proportional to the square root of |V| (see [10] for a discussion of this issue), the use of a linked list for B implies a linear cost for each S operation, and hence makes the asymptotic worst case complexity of the computational cost of decompression $O(v^2)$. This cost may be reduced to $O(v \log v)$ by maintaining a balanced binary search tree [1], rather than a doubly linked list, for representing the sequence of vertices in the active loop, B. On hardware platforms that support fast block memory transfer operations, an attractive alternative is to use a linear array to store the ordered set of index-references to the vertices of B and to perform insertions and deletions via block memory transfers. This latter solution turns updating B and accessing the offset vertex into constant cost operations, at least for models of moderate size.

Again, we illustrate the *Generation* phase of the decompression process on the initial part of the same sequence of operations, CCRRRSLCRSERRELCRRRCRRRE, produced by the compression of the mesh in Fig. 5. Given that $|V_E|$=16, we start with an initial loop of 16 edges containing the labels 1, 2, .3… The first edge is the gate G, which is associated with its end-vertex labeled 1. *c is initialized to 16.* The first C operation fills TV[1] with the three values: 16, which is G.P.**e**; 1, which is G.**e**; and 17, which is the result of incrementing *c*. We also create a new edge, A, and store 17 as its label. Then we insert A before G by updating the pointers as follows: G.P.N=A; A.P=G.P; A.N=G; G.P=A. The second C operation creates triangle (17, 1, 18) and inserts another edge before G with the label 18. Then the first R operation creates triangle (18,1,2), deletes the gate from the loop and makes the edge labeled 2 the current gate. The second and third R operations create the triangles (18,2,3) and (18,3,4) reducing the loop to contain the vertex sequence {4,5,6,7,8,9,10,11,12,13,14,15,16,17,18}. Because O[1] is 6, the first S operation skips the six vertices 5, 6, 7, 8, 9, and 10. Then it fills triangle number 6 with the labels (18,4,11) and splits the loop into {11,12,13,14,15,16,17,18} and {4,5,6,7,8,9,10,11}. The bottom of the stack points to the edge (18,11), the first one in the first loop. The top of the stack points to edge (11,4), the first one in the second loop. Then the L operation creates triangle (11,4,10) and deletes the last edge of the second loop. At this point G is the edge (10,4). This process continues, reaching the second S, at which point there are 3 entries in the stack pointing to the loops: {11,12,13,14,15,16,17,18}, {7,8,9,10,19}, and {5,6,7}, which includes the current gate (7,5). The next operation, E, creates the triangle {7.5.6} and pops the stack, so that the new gate is edge (19,7). The next 3 operations, R, R, and E create the 3 triangles of that region. We pop the stack and have edge (18,11) as gate. The remaining portion of H, which now contains LCRRRCRRRE, is processed similarly.

General triangle meshes

The Edgebreaker approach, as presented so far, is capable of encoding the connectivity of any planar triangle graph with zero or one hole. We describe here how to extend the Edgebreaker compressed format and the compression and compression algorithms, so as to support meshes with multiple holes and with one or more handles.

Holes

In a mesh with several holes, i.e.: with more than one bounding loop, the compression algorithm may find a triangle, whose third vertex, g.n.**v**, lies on the boundary of a hole rather than on the current loop. Instead of splitting the current loop into two, we merge it with the hole by opening both loops at their common vertex and reconnecting them into a single cyclic list. We use the symbol M to identify such cases in the history. To distinguish these situations from the S cases, we initialize the .m marks of all vertices and edges of the initial loop to 1 and of all other external vertices and external edges to 2. Internal vertices and edges are still marked with zero. This assignment may be easily performed by traversing the half-edge data structure and, each time an unmarked external edge is encountered, by following the loop that contains it and by marking all the vertices and edges with a 1 for the first loop and with a 2 for all subsequent loops. We assume for simplicity that the union of all external edges forms a manifold curve with one component per hole. Surfaces with non-manifold boundary may be converted to such a representation by replicating their non-manifold vertices.

C cases now correspond to situations where g.**v**.m==0. M cases correspond to situations where g.**v**.m==2, all other cases correspond to situations where g.**v**.m==1 and may be distinguished as before. Each time a hole is reached, the references to all of its vertices are appended to **P**, starting with the contact vertex g.n.**v**. In addition, the extended compression algorithm associates with each M operation the length *l* of the corresponding hole by appending it to a list of hole length, *L*. The processing of the M operation during compression is illustrated Fig. 12.

During the preprocessing phase of the extended decompression algorithm, each time an M operation is reached, to correctly compute the offset table, the value *l*+1, rather than 1, must be subtracted from the edge-count *e*.

Then, during the generation phase, at each M operation, after the bounding loop has been merged with the hole, as shown Fig. 12, it contains the *l* edges of the hole inserted just before g. The vertex label for the last one of these edges is *c*+1. The labels of the first *l*–1 of these edges are assigned by successive increments of *c*. The pseudo-code for the M operation follows:

```
TV[++t]=(G.P.e, G.e, c+1);      # new triangle with the contact point
  D=G.P;                        # initialize an end-of-list edge pointer
  REPEAT                        # insert l+1 edges after G.P
    New Edge A;                         # create new edge
    D.N=A; A.P=D; A.e=++c;              # link it after D and set label
    D=A;                                # move end-of-list
    l TIMES;                            # last edge has wrong label
  New Edge A;                           # create new edge
  D.N=A; A.P=D; A.e=c–l+1;              # link it after D and set label
  A.N=G; G.P=A;                         # link end-of-list to G,
```

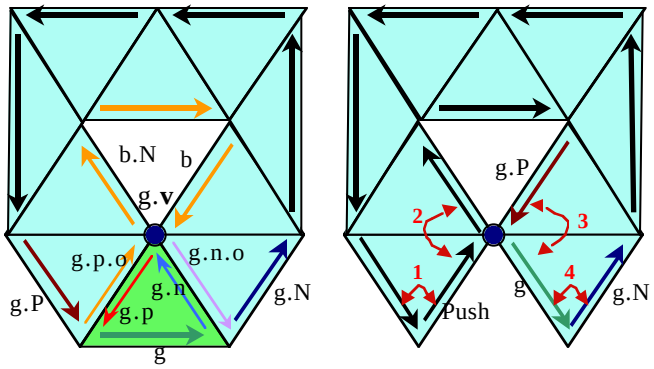

Figure 12: **Case M**

```
H=HlM;                          # append M to history
g.m=0; g.p.o.m=1; g.n.o.m=1;    # update marks
b=g.n;                          # initial candidate for b
WHILE b.m≠2 DO b=b.o.p;         # turn around v
g.P.N=g.p.o; g.p.o.P=g.P;       # fix red link 1 in B
g.p.o.N=b.N; b.N.P=g.p.o;       # fix red link 2 in B
b.N=g.n.o; g.n.o.P=b;           # fix red link 3 in B
g.n.o.N=g.N; g.N.P=g.n.o;       # fix red link 4 in B
g=g.n.o; StackTop=g             # move g
b=b.N;          # start here the traversal of the new edges
l=0;
WHILE b.e≠g.s DO {      # until old g.v. is reached do
    b.m=1; b.s.m=1;             # mark hole
    l++;                        # counts length of hole
    P=P|b.s;            # append new vertex reference to P
    b=b.N };            # move to next edge around hole
 L = L | l ;            # appends the length of hole to L
```

The addition of the M operation requires changing our coding scheme. A simple approach would be to use 4-bit codes for S and M (for example, S could be 1000 and M could be 1001. This solution adds |S| bits to the overall compressed representation in addition to the cost of encoding the M operations. If we expect the number of holes to be small compared to |S|, it is advantageous to use the same code for all M and S operations, and to distinguish them by including, in the compressed format, right before the encoding of H, a representation of an M-table. The M-table contains two entries for each M operation. The first entry is the number of S operations encountered since the previous M operation or since the beginning of H for the first M operation. Each one of these numbers indicates how many consecutive S operations should not be treated as M operations. The second entry in the M-table is the length *l* of the boundary of the corresponding hole.

Handles

An S operation splits the current bounding loop into two parts, which may be independently processed by the compression and by the decompression algorithms. However, when an S operation is performed as described earlier on a mesh where the current boundary wraps around a handle, a new hole is created, instead of a separate component of the mesh. Indeed, the current loop is split into two loops without disconnecting the remaining portion of the mesh. Such a split may, for example, transform a toroidal surface with one hole into a cylindrical surface (a mesh with two holes). Several such holes may be created during compression. They will be merged into the current loop by subsequent M operations, as described above. However, they will not necessarily be merged in the order in which they have been created.

We modify the S operation as follows. When the current loop is split into the left and right sub-loops (Fig. 11), we mark the vertices and edges of the left sub-loop with a 3. The first half-edge of this left sub-loop is pushed onto the stack along with the location of the corresponding S op-ode in H.

During compression, when we later reach a vertex marked with a 3, we perform a different merging operation identified by the op-code M', which merges the two loops, but does not append to **P** the references to the vertices of the hole. M' also computes the position, *p*, of the associated gate in the stack and the offset, *o*, between that gate and the reached point. *p*, *o*, and the number of S operations encountered after the previous M' operation are stored in the M'-table.

During decompression of an M' operation, Edgebreaker performs the following steps:

```
D=remove(p);            # fetches and removes entry p from
  stack
REPEAT D=D.N o TIMES;   # find edge to connecting vertex
TV[++t]=(G.P.e, G.e, D.e);  # new triangle with the contact point
New Edge A;             # create new edge
G.P.N=A; A.P=G.P;       # link 1
A.N=D.N; D.N.P=A;       # link 2
D.N=G; G.P=D;           # link 3
```

Boundaries of solid models with triangulated surfaces may have handles, but do not have holes. Therefore, when dealing with solid models, it is not necessary to support holes and the operations of type M' may be encoded by using the same op-code as for the S operations. The entries of the M'-table may be used to distinguish them, as discussed above for holes.

For meshes with handles and holes, both types of operations must be supported. Therefore, during compression, the vertices may be marked with a 0, 1, a 2 or a 3. These markings permit to distinguish between C, S, M and M' operations. Again, we suggest to use a combined M-M'-table to distinguish between S, M, and M' operations and to encode the associated parameters. With this convention, the history still only contains C, L, E, R, and S op-codes. For edges with a small number of handles and holes relative to the number of triangles, the storage cost of the M-M'-table has little effect on the connectivity cost per triangle.

Summary of contributions

We have presented a new compression/decompression technique for coding the connectivity, i.e., the triangle/vertex incidence graph, of arbitrary triangle meshes. The research contributions reported in this paper have both a theoretical and a practical value. They include an extensive survey of prior art, an improvement over the best known encoding of planar triangulated graphs, i.e., an algorithm which guarantees the lowest known upper bound on the connectivity storage cost, and a detailed description of very effective simple compression and decompression algorithms for the connectivity of a large class of triangle meshes.

Our survey discusses theoretical contributions on labeled and unlabeled planar triangulated graphs and practical implementations of compression algorithms developed for graphic applications [36]. We also propose several variations, which improve or combine these approaches in novel ways and have identified the associated connectivity cost, which we use as a basis of comparison.

Through the introduction of a novel decompression technique, which in one pass automatically extracts the offsets of S operations from the compression history, H, we have eliminated the need to encode these offsets explicitly and thus have achieved a linear connectivity cost for meshes with a constant number of handles and holes. This result improves over recently published, independently developed approaches [37, 10], which exhibit an asymptotic O($v$log$v$) connectivity cost, even for simple meshes without holes or handles. On a more theoretical aspect, we have also improved on all previous work on coding planar triangulated graphs [39, 15, 38, 23, 16] by providing a linear code with the lowest constant: a guaranteed 2 bits per triangle or less.

On a practical side, the Edgebreaker technique introduced here offers a simpler to implement and, in many cases, more effective alternative to previously proposed connectivity coding schemes [6, 13, 33, 34]. The compression algorithm, which we describe in details, uses a half-edge data structure and simply traverses the mesh from one triangle to a neighboring one, recording the history: a simple list of the op-codes for the C, E, R, L, and S operations. The decompression algorithm traverses the history once to compute the number of internal and external vertices and the offset for each S operation. Then it recreates the triangles, one at a time, in the order in which they have been visited by the compression algorithm. It labels the vertices with successive integers. This labeling of the vertices is also computed as a byproduct of the compression process and defines the order in which vertex data should be compressed.

Edgebreaker may be easily combined with a variety of vertex data compression schemes [33, 37] based on vertex estimates that are derived from the incidence graph and from the location of previously decoded vertices.

Edgebreaker compresses the connectivity of simply-connected manifold triangle meshes down to between 1.5 and 2 bits per triangle. By allowing additional bits, the basic technique is extended to support triangle meshes with holes and handles. For highly complex meshes, these results are comparable to those reported by other authors [33, 37, 10], who through Huffmann coding may achieve lower bit counts, especially for meshes with an almost regular topology [37]. Because general purpose statistical compression schemes perform poorly on smaller or very irregular data, these compression methods require more bits per triangle when dealing with simpler or irregular triangle meshes. Edgebreaker does not rely on statistical methods and thus guarantees its low bit counts. It is therefore a practical solution for compressing both large and small meshes.

In conclusion, Edgebreaker provides a simple and effective connectivity compression tool for a variety of 3D applications.

We believe that the area of 3D compression will grow significantly over the next few years. We plan to focus on the integration of Edgebreaker's connectivity compression with progressive methods [14, 35] for connectivity refinement and with methods for the progressive refinement of vertex-accuracy [19]. We also plan to explore variations of Edgebreaker's format that are suitable for hardware decompression and graphic acceleration and would offer easier to program compression algorithms and more compact formats than currently available methods [6, 4].

Acknowledgments

The author thanks Andrzej Szymczak from Georgia Tech for pointing out that the CL and CE combinations are impossible, Antonio Haro from Georgia Tech for developing an early implementation of the Edgebreaker algorithms, and Gabriel Taubin from IBM Research, and Leonard Schulman, Peter Lindstrom, Renato Pajarola and Greg Turk from Georgia Tech for their comments on this work and for their input regarding data compression schemes.

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

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

[3] R., Carey, G. Bell, C. Martin, The Virtual Reality Modeling Language ISO/IEC DIS 14772-1, April 1997, http://www.vrml.org/Specifications.VRML97/DIS.

[4] M. Chaw, Optimized Geometry Compression for Real-time Rendering, Proc. IEEE Visualization'97, pp. 347-354, Phoenix, AZ, October 19-24, 1997.

[5] L., Darsa, B.Costa Silva, and A. Varshney, Navigating static environments using image-space simplification and morphing, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 7-16, April 1997.

[6] M. Deering, Geometry Compression, Computer Graphics, Proceedings Siggraph'95, 13-20, August 1995.

[7] M. Denny and C. Sohler, Encoding a triangulation as a permutation of its point set, Proc. of the Ninth Canadian

Conference on Computational Geometry, pp. 39-43, Ontario, August 11-14, 1997.

[8] D. Dobkin and D. Kirkpatrick, A linear algorithm for determining the separation of convex polyhedra, Journal of Algorithms, vol. 6, pp. 381-392, 1985.

[9] F. Evans, S. Skiena, and A. Varshney, Optimizing Triangle Strips for Fast Rendering, Proceedings, IEEE Vizualization'96, pp. 319--326, 1996.

[10] S. Gumhold and W. Strasser, Real Time Compression of Triangle Mesh Connectivity. Proc. ACM Siggraph 98, pp. 133-140, July 1998.

[11] P. Heckbert and M. Garland, Survey of Polygonal Surface Simplification Algorithms, in *Multiresolution Surface Modeling Course*, ACM Siggraph Course notes, 1997.

[12] H. Hoppe, T, DeRose, T, Duchamp, J, McDonald, and W, Stuetzle, Mesh optimization, Proceedings SIGGRAPH'93, pp:19-26, August 1993.

[13] H. Hoppe, Progressive Meshes, Proceedings ACM SIGGRAPH'96, pp. 99-108, August 1996.

[14] H. Hoppe, View Dependent Refinement of Progressive Meshes, Proceedings ACM SIGGRAPH'97, August 1997.

[15] A.Itai and M. Rodeh, Representation of Graphs, Acta Informatica, No. 17, pp. 215-219. 1982.

[16] K. Keeler and J. Westbrook, Short Encodings of Planar Graphs and Maps, Discrete Applied Mathematics, No. 58, pp. 239-252, 1995.

[17] D. Kirkpatrick, Optimal search in planar subdivisions, SIAM Journal on Computing, vol **12**, pp. :28-35, 1983.

[18] D.T. Lee and F.P. Preparata, Location of a point in a planar subdivision and its applications. SIAM J. on Computers, 6:594-606, 1977.

[19] J. Li, and C.C Kuo, Progressive Coding of 3D Graphic Models, Proceedings of the IEEE, pp. 1052-1063. June 1998.

[20] Y. Mann and D. Cohen-Or, Selective Pixel Transmission for Navigation in Remote Environments, Proc. Eurographics'97, Budapest, Hungary, September 1997.

[21] W., Mark, L. McMillan, and G. Bishop, Post-rendering 3D warping, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 7-16, April 1997.

[22] W. Massey, *Algebraic Topology: An Introduction*, Harcourt, Brace& World Inc., 1967.

[23] M. Naor, Succinct representation of general unlabeled graphs, Discrete Applied Mathematics, vol. 29, pp. 303-307, North Holland, 1990.

[24] J. Neider, T. Davis, and M. Woo, OpenGL Programming Guide, Addison-Wesley, 1993.

[25] M. R. Nelson, LZW Data Compression, Dr. Dobb's Journal, October 1989.

[26] A. Rockwood, K. Heaton, and T. Davis, Real-time Rendering of Trimmed Surfaces, Computer Graphics, 23(3):107-116, 1989.

[27] R. Ronfard. and J. Rossignac, Full-range approximation of triangulated polyhedra, Proc. Eurographics'96 , Computer Graphics Forum, pp. C-67, Vol. 15, No. 3, August 1996.

[28] J. Rossignac, Through the cracks of the solid modeling milestone, *From Object Modelling to Advanced Visual Communication*, Eds. Coquillart, Strasser, Stucki, Springer-Verlag, pp. 1-75, 1994.

[29] J. Rossignac, Geometric Simplification and Compression, in *Multiresolution Surface Modeling Course*, ACM Siggraph Course notes 25, Los Angeles, 1997.

[30] J. Rossignac, Edgebreaker: Compressing the incidence graph of triangle meshes, Jarek Rossignac, GVU Technical Report GIT-GVU-98-17, Georgia Institute of Technology, http://www.cc.gatech.edu/gvu/reports/1998.

[31] J. Rossignac, 3D Geometry Compression: Just-in-time upgrades for triangle meshes, in *3D Geometry Compression*, Course Notes 21, Siggraph 98, Orlando, Florida, July 18-24, 1998.

[32] J. Snoeyink and M. van Kerveld, Good orders for incremental (re)construction, Proc. ACM Symposium on Computational Geometry, pp. 400-402, Nice, France, June 1997.

[33] G. Taubin and J. Rossignac, Geometric Compression through Topological Surgery, ACM Transactions on Graphics, Volume 17, Number 2, pp. 84-115, April 1998.

[34] G. Taubin, W. Horn, F. Lazarus, and J. Rossignac, Geometry Coding and VRML, Proceedings of the IEEE, pp. 1228-1243, vol. 96, no. 6, June 1998.

[35] G. Taubin, A. Gueziec, W. Horn, F. Lazarus, Progressive Forest Split Compression, Proc. ACM Siggraph 98, pp. 123-132, July 1998.

[36] G. Taubin and J. Rossignac, *3D Geometry Compression*, Course Notes 21, Siggraph 98, Orlando, Florida, July 18-24, 1998.

[37] C. Touma and C. Gotsman, Triangle Mesh Compression, Proceedings Graphics Interface 98, pp. 26-34, 1998.

[38] G., Turan Succinct representations of graphs, Discrete Applied Math, 8: 289-294, 1984.

[39] W.T. Tutte, The Enumerative Theory of Planar Graphs. In *A Survey of Computational Theory*, J.N. Srinivasan et al. (Eds.). North-Holland, 1973.

[40] T Welch, A Technique for High-Performance Data Compression, Computer, June 1984.

[41] J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, May 1977.