Simplification and Compression of 3D Meshes

Craig Gotsman[1], Stefan Gumhold[2], and Leif Kobbelt[3]

[1] Computer Science Department, Technion, Haifa, Israel
[2] WSI/GRIS, University of Tübingen, Germany
[3] Computer Graphics Group, RWTH-Aachen, Germany

Abstract. We survey recent developments in compact representations of 3D mesh data. This includes: methods to reduce the complexity of meshes by simplification, thereby reducing the number of vertices and faces in the mesh; methods to resample the geometry in order to optimize the vertex distribution; methods to compactly represent the connectivity data (the graph structure defined by the edges) of the mesh; methods to compactly represent the geometry data (the vertex coordinates) of a mesh.

1 Introduction

Interactive display of three-dimensional content is an important component for applications in electronic commerce, medical and scientific visualization, engineering analysis and the game industry. Large amounts of 3D content are processed and transmitted over the Internet. Among several representations polygonal meshes are used most often as surface representation because of their wide support in VRML, OpenGL and other file formats and graphics libraries.

A *polygonal mesh* consists of three different kinds of *mesh elements*: vertices, edges and faces. The information describing the mesh elements consists of the *mesh connectivity* and *mesh geometry*. The mesh connectivity describes the incidence relations between the mesh elements. The incidence relations specify for each face the vertices and edges on the bounding loop, for each edge the end vertices and the faces to which the edge is incident, and for each vertex the incident edges and faces. Two vertices or two faces are called *adjacent*, if there exists an edge incident to both. The mesh geometry specifies a position in space for each vertex.

In this chapter we survey three approaches to reducing the complexity of polygonal meshes. Section 2 introduces simplification algorithms that reduce the number of elements in a mesh. Section 3 surveys recent algorithms that compactly encode the incidence relations of the mesh elements. Finally, Section 4 describes coding techniques for the mesh geometry. The remainder of the introduction describes polygonal meshes in more detail and introduces the basic coding techniques used in the following sections.

1.1 Meshes

Definitions. A mesh is called *manifold* if each edge is incident to only one or two faces and the faces incident to a vertex form a closed or an open fan. Non-manifold meshes can be cut into manifold meshes by replicating vertices with more than one fan and edges incident to more than two faces.

The orientation of a face is the cyclic order of the incident vertices. There are two possible orientations for each face (clockwise and counter-clockwise). The orientation of two adjacent faces is *compatible*, iff the two vertices of the common incident edge are in opposite order. A mesh is called *orientable*, iff there exists a choice of face orientations that makes all pairs of adjacent faces compatible.

The Euler formula (see [50, p. 145] for an introduction to the more general Euler-Poincaré formula) describes the relationship between the number of vertices v, edges e, faces f, and the topological type of an orientable manifold mesh. If s is the number of connected components, g the genus and b the number of border loops of the mesh, then

$$v - e + f = 2(s - g) - b =: \chi, \tag{1}$$

where χ is called the *Euler characteristic* of the mesh. A mesh has *genus* g iff one can cut the mesh along $2g$ closed loops without disconnecting the mesh. The sphere has genus zero and the torus has genus one. Any mesh of genus g can be continuously deformed into a sphere with g handles. In the special case of a closed manifold triangular mesh each edge has exactly two incident triangles and each triangle three incident edges: $2e = 3f$. This yields $2v - f = 2\chi$ and as χ is typically small, $f \approx 2v$.

A mesh is called *simple*, iff it is connected, orientable, manifold, of genus zero and has no more than one border loop.

Standard Mesh Representation. The mesh geometry is mostly represented as fields indexed by the mesh elements. For example, the vertex locations are stored in a vector-valued field indexed over the vertices.

The connectivity is often defined by a *face vertex incidence table.* For each face an oriented cyclic list of indices of the incident vertices is given. In VRML the `IndexedFaceSet`-node defines the face-vertex incidence table by one list of indices, where the index -1 separates different faces. The order of the vertex indices in a face also defines the face-edge incidences. Between each two successive vertices there must be an edge in the face loop. All face-edge incidences of one edge can be collected in linear time by bucket sorting of the reference pairs of incident vertices. This approach can be used to efficiently build a data structure, such as the half-edge [50] or winged-edge [3] data structures that allow for enumeration of incident and adjacent mesh elements in time proportional to the number of enumerated elements.

1.2 Coding techniques

The mesh compression schemes to be introduced in Sections 3 and 4 translate a mesh into sequences of symbols and/or indices. If the minimum index is $i_{\min}$ and the maximum index $i_{\max}$, the coding of indices can be reduced to the coding of symbols from an alphabet $\{\sigma_{i_{\min}}, \sigma_{i_{\min}+1}, \ldots, \sigma_{i_{\max}}\}$. In the general case we assume an alphabet $\mathcal{A} = \{\sigma_1, \ldots, \sigma_a\}$ to be given. The compression schemes translate the mesh connectivity and/or geometry into a sequence $S_n = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_n}$ of n symbols. As each symbol can be encoded with $\lceil \log_2 a \rceil$ bits, the sequence S_n can be encoded in $n \lceil \log_2 a \rceil$ bits. Suppose now that half of the symbols in S_n are σ_1. Then the sequence can be encoded in only $n + \frac{n}{2} \lceil \log_2 (a-1) \rceil$ bits by first coding one bit for each symbol noting whether the symbol is σ_1 and only when this is not the case, encoding the symbol as before. Hence it is possible to encode a sequence of symbols more efficiently, if the probability of some symbols is higher than that of others. Let $\#_i(S_n)$ be the number of symbols σ_i in the sequence S_n, and let $p_i(S_n) = \#_i(S_n)/n$ be the probability of symbol σ_i. The probabilities define the distribution of the symbols in a sequence. The following two coding schemes exploit the distribution of the symbols in the general case. The average amount of bits per symbol spent by an optimal coding scheme for a given symbol distribution p_i is called the *entropy* and amounts to

$$-\sum_{i=1}^{a} p_i \cdot \log_2 p_i \text{ bits per symbol.} \tag{2}$$

Huffman Coding. Huffman [30] devised a coding scheme that assigns to each symbol σ_i a bit code c_i of variable length. For unique decoding it is essential that no bit code is a prefix of another code. This can be guaranteed by defining the bit codes from a binary tree with the symbols as leaves. The path from the root to a symbol σ_i on the k-th tree level defines the bit code c_i as a sequence of k bits, where each bit tells if the path moves from the current node to the left or right child. Two of the codes defined in this way can violate the prefix condition, iff one symbol is within the defining path of another, which is impossible as all symbols are leaves. The binary tree is built from a forest of leaves (σ_i, p_i) such that the probability of the symbols is balanced. As long as there is more than one tree in the forest, a new node is built with the two trees of smallest probability as children and the sum of the child probabilities as probability. Figure 1 (a) illustrates the tree construction on an example, where the symbol counts were used instead of the probabilities. Huffman showed that this procedure yields the tree with the most efficient coding of the symbol sequence. To avoid having to encode the binary tree and compute the symbol probabilities in a preprocessing step, Cormack and Horspool [12] describe an algorithm that generates the Huffman codes on the fly at the encoder and the decoder. It builds the binary tree over the symbol counts $\#_i$, which are all initialized to zero. As each symbol σ_i of the sequence is encoded, the count $\#_i$ is incremented and the binary tree updated.

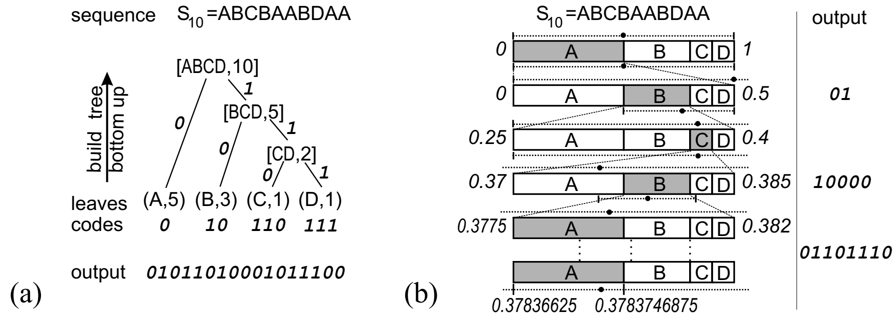


Fig. 1. Illustration of a) Huffman and b) arithmetic coding.

Arithmetic Coding. Huffman coding is only optimal if the symbol probabilities are negative powers of two. In arithmetic coding no fixed bit codes are assigned to the symbols. Instead coding is done in two phases. In the first phase the symbol probabilities are used to recursively define an interval subdivision of $[0, 1]$ as illustrated in Figure 1 (b). In the first line the interval $[0, 1]$ is subdivided into four blocks – one for each symbol – where the width of the block is equal to the symbol probability. The first A in the sequence defines the subinterval $[0, 0.5]$, which is taken as the new base interval and split again into four blocks. The symbol B specifies the interval $[0.25, 0.4]$ and so on. The complete sequence results in a very tiny *target interval* around 0.37837, which uniquely represents the symbol sequence.

In the second phase the target interval is encoded via a binary subdivision that initializes its *subdivision interval* to $[0, 1]$, and moves into the lower or upper half until the center of the subdivision interval hits the target interval. Figure 1 (b) also illustrates the interval subdivision of the second phase after every line as if coding would stop. Above each line the subdivision interval resulting from the previous line is illustrated by a dotted line with its frontiers as horizontal bars and its center as the black dot. In the right column the bits are tabulated with each one successively describing whether the interval was split into the lower (0 bit) or upper (1 bit) half in order to bring the center inside the target interval. Below each line the subdivision interval after binary subdivision is shown, and the center dot is always inside the target interval. After the first symbol A has been encoded no subdivision is necessary as the interval center is still inside the target interval. The subdivision interval is zoomed with the target interval to the second line. In order to hit inside the target interval of the B symbol, the subdivision interval moves to the lower and then to the upper half of the lower half, resulting in two output bits. Accidentally, the new interval center specifies correctly the successive C and no further bit has to be encoded. But for the encoding of the next B the subdivision has to move to the upper half and four times to the lower half such that the interval shrinks by a factor of 32. In the end 15 bits are needed to encode the sequence instead of 17 bits for Huffman coding.

As the binary subdivision defines a point inside the target interval, the symbol sequence can be reconstructed by simply checking in which of the four blocks of the current interval the center of the binary subdivision points. Witten et al. [65] show how to implement arithmetic coding with integer arithmetic on an incremental basis. From the interval subdivision scheme it follows directly that arithmetic coding allows the encoding of each symbol σ_i, on average with $-\log_2 p_i$ bits and, therefore, the complete sequence with $-\sum_{i=1}^{a} np_i \cdot \log_2 p_i$ bits. This is the best that can be achieved without more knowledge about the sequence as it achieves entropy.

2 Mesh Decimation Techniques

2.1 Introduction

Mesh decimation describes a class of algorithms that transform a given polygonal mesh into another mesh with fewer faces, edges and vertices. The decimation procedure is usually controlled by user-defined quality criteria which prefer meshes that preserve specific properties of the original data as much as possible. Typical criteria include geometric distance (e.g. Hausdorff distance) or visual appearance (e.g. color difference, feature preservation) [6]. There are many applications for decimation algorithms. First, they obviously can be used to *adjust the complexity* of a geometric data set. This makes geometry processing a scalable task where different complex models can be used on

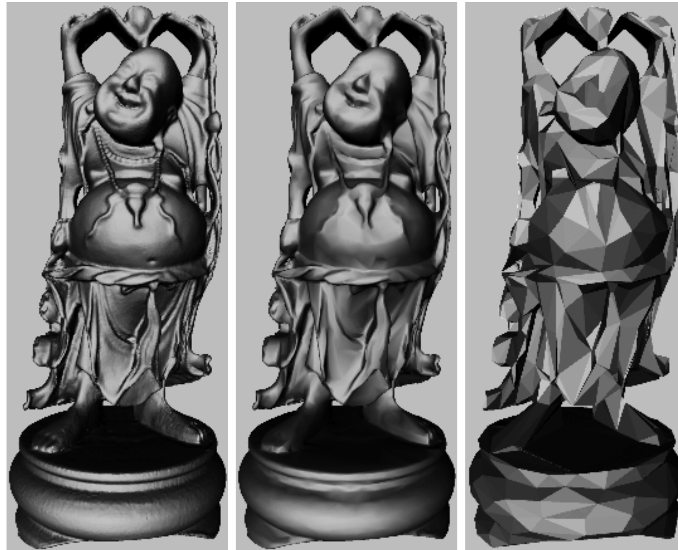


Fig. 2. Decimation of the Stanford Buddha model from 400K triangles to 40K triangles to 4K triangles.

computers with varying computing performance. Second, since many decimation schemes work iteratively, i.e. they decimate a mesh by removing one vertex at a time, they usually can be inverted. Running a decimation scheme backwards means reconstructing the original data from a decimated version by inserting more and more detailed information. This inverse decimation can be used for *progressive transmission* of geometry data [27]. Obviously, in order to make progressive transmission effective, we have to use decimation operators whose inverse can be encoded compactly (cf. Fig. 3).

The third class of applications exploits the implicit hierarchical information that emerges from mesh decimation. The decimated version of a polygonal mesh retains the global shape information while the finer levels of (less significant) detail have been removed. So in terms of multiresolution decompositions of geometric data, we can consider a mesh decimation scheme as a *decomposition operator*: applied to the original data, we obtain a low frequency component (the decimated mesh itself) and a high frequency component (the difference between original and decimated). The high frequency component can, for example, be represented simply as a log-file of the sequence of decimation steps. The *reconstruction operator* can then perform the inverse decimation steps in reverse order to recover the original data from its low frequency part [24,42].

2.2 Overview

There are several different conceptual approaches to mesh decimation. In principle we can think of the complexity reduction as a one-step operation or as an iterative procedure. The vertex positions of the decimated mesh can be obtained as a subset of the original set of vertex positions, as a set of weighted averages of original vertex positions, or by resampling the original piecewise linear surface. In the literature the different approaches are classified into

- Vertex clustering algorithms
- Incremental decimation algorithms
- Resampling algorithms.

The first class of algorithms is usually very efficient and robust. The computational complexity is typically linear in the number of vertices. However, the quality of the resulting meshes is not always satisfactory. *Incremental algorithms* in most cases lead to higher quality meshes. The iterative decimation procedure can take arbitrary user-defined criteria into account, according to how the next removal operation is chosen. However, their total computational complexity in the average case is $\mathcal{O}(n \log n)$, and can go up to $\mathcal{O}(n^2)$ in the worst case, especially when a global error threshold is to be respected. Finally, *resampling techniques* are the most general approach to mesh decimation. Here, new samples are more or less freely distributed over the original piecewise linear surface geometry. By connecting these samples, a completely new mesh is constructed.

The major motivation for resampling techniques is that they can force the decimated mesh to maintain a special connectivity structure, i.e. subdivision connectivity (or semi-regular connectivity). As a result they can be used in a straightforward manner to build multiresolution representations based on subdivision basis functions and their corresponding (pseudo-)wavelets [14].

The most serious disadvantage of resampling, however, is that *alias errors* can occur if the sampling pattern is not perfectly aligned to features in the original geometry. To avoid alias effects, many resampling schemes, to some degree, require manual pre-segmentation of the data for reliable feature detection [4].

The unstructured hierarchies that emerge from iterative decimation algorithms give rise to generalized multiresolution techniques for meshes with arbitrary connectivity. While simple basis functions (as in the semi-regular subdivision setting) are no longer available, we can still mimic the functionality of higher order decomposition and smooth reconstruction filters [40,24].

In the following sections we will explain the different approaches to mesh decimation in more detail. Usually there are many choices for the different ingredients and sub-procedures in each algorithm. We will also point out the advantages and disadvantages of each class.

2.3 Vertex Clustering

The basic idea of vertex clustering is quite simple: for a given approximation tolerance ε we partition the bounding space around the given object into cells with diameter smaller than ε. For each cell we compute a representative vertex position which we assign to all the vertices that fall into that cell. By this clustering step, original faces degenerate if two or three of their corners lie in the same cell and consequently are mapped to the same position. The decimated mesh is eventually obtained by removing all those degenerate faces [52].

The remaining faces correspond to those original triangles whose corners all lie in different cells. Stated otherwise: if $\mathbf{p}$ is the representative vertex for the vertices $\mathbf{p}_0, ..., \mathbf{p}_n$ in the cluster P and $\mathbf{q}$ is the representative for the vertices $\mathbf{q}_0 ..., \mathbf{q}_m$ in the cluster Q, then $\mathbf{p}$ and $\mathbf{q}$ are connected in the decimated mesh if and only if at least one pair of vertices $(\mathbf{p}_i, \mathbf{q}_j)$ was connected in the original mesh.

One immediately obvious drawback of vertex clustering is that the resulting mesh might no longer be 2-manifold even if the original mesh was. Topological changes occur when the part of a surface that collapses into a single point is not homeomorphic to a disc, i.e., when two different sheets of the surface pass through a single ε-cell. However, this disadvantage can also be considered as an advantage. Since the scheme is able to change the topology of the given model we can reduce the object complexity very effectively. Consider, for example, applying mesh decimation to a 3D-model of a sponge. Here, any decimation scheme that preserves the surface topology

cannot reduce the mesh complexity significantly since all the small holes have to be preserved.

The computational efficiency of vertex clustering is determined by the effort it takes to map the mesh vertices to clusters. For simple uniform spatial grids this can be achieved in linear time with small constants. Then for each cell a representative has to be found which might require fairly complicated computations, but the number of clusters is usually much smaller than the number of vertices.

Another apparently nice aspect of vertex clustering is that it automatically guarantees a global approximation tolerance by defining the clusters accordingly. However in practice it turns out that the actual approximation error of the decimated mesh is usually much smaller than the radius of the clusters. This indicates that for a given error threshold, vertex clustering algorithms do not achieve optimal complexity reduction. Consider, as an extreme example, a very fine planar mesh. Here decimation down to a single triangle without any approximation error would be possible. In contrast, the result of vertex clustering will always keep one vertex for every ε-cell.

Representative computation The way in which vertex clustering algorithms differ is mainly in how they compute the representative. Simply taking the center of each cell or the straight average of its members are obvious choices which, however, rarely lead to satisfying results.

A more reasonable choice is based on finding the optimal vertex position in the least squares sense. For this we exploit the fact that for sufficiently small ε the polygonal surface patch that lies within one ε-cell is expected to be piecewise flat, i.e., either the associated normal cone has a small opening angle (totally flat) or the patch can be split into a small number of sectors for which the normal cone has a small opening angle.

The optimal representative vertex position should have a minimum deviation from all the (regression) tangent planes that correspond to these sectors. If these approximate tangent planes do not intersect in a single point, we have to compute a solution in the least squares sense.

Consider one triangle T_i belonging to a specific cell, i.e., whose corner vertices lie in the same cell. The quadratic distance of an arbitrary point $\mathbf{x}$ from the supporting plane of that triangle can be computed by:

$$(\mathbf{n}_i^T \mathbf{x} - p_i)^2,$$

where $\mathbf{n}_i$ is the normal vector of T_i and p_i is the scalar product of $\mathbf{n}_i$ times one of T_i's corner vertices. The sum of the quadratic distances to all the triangle planes within one cell is given by

$$E(\mathbf{x}) \;=\; \sum_i (\mathbf{n}_i^T \mathbf{x} - p_i)^2. \tag{3}$$

The iso-contours of this error functional are ellipsoids and consequently, the resulting error measure is called *quadric error metric (QEM)* [48,18]. The

point position where the quadric error is minimized is given by the solution of

$$\left(\sum_i \mathbf{n}_i \, \mathbf{n}_i^T \right) \mathbf{x} \; = \; \left(\sum_i \mathbf{n}_i \, p_i \right) . \tag{4}$$

If the matrix has full rank, i.e. if the normal vectors of the patch do not lie in a plane, then the above equation could be solved directly. However, to avoid special case handling and to make the solution more robust, a pseudo-inverse based on a *singular value decomposition* should be used.

2.4 Incremental Mesh Decimation

Incremental algorithms remove one mesh vertex at a time. In each step, the best candidate for removal is determined based on user-specified criteria. Those criteria can be *binary* (i.e. removal is allowed or not) or *continuous* (i.e. rate the quality of the mesh after the removal between 0 and 1). Binary criteria usually refer to the global approximation tolerance or to other minimum requirements, e.g., minimum aspect ratio of triangles. Continuous criteria measure the *fairness* of the mesh in some sense; e.g., “round” triangles are better than thin ones, small normal jumps between neighboring triangles are better than large normal jumps.

Every time a removal is executed, the surface geometry in the vicinity changes. Therefore, the quality criteria have to be re-evaluated. During the iterative procedure, this re-evaluation is the computationally most expensive part. To preserve the order of the candidates, they are usually kept in a *heap data structure* with the best removal operation on top. Whenever removal candidates have to be re-evaluated, they are deleted from the heap and re-inserted with their new value. Through this, the complexity of the update-step increases only in the same way as $\mathcal{O}(\log n)$ for large meshes if the criteria evaluation itself has constant complexity.

Topological operations There are several different choices for the basic removal operation. The major design goal is to keep the operation as simple as possible. In particular this means that we do not want to remove large parts of the original mesh at once but rather remove a single vertex at a time. Strong decimation is then achieved by applying many simple decimation steps instead of a few complicated ones. If mesh consistency, i.e., topological correctness, matters, the decimation operator has to be an *Euler-operator* (derived from the Euler formula for graphs) [26].

The first operator one might think of *deletes one vertex* plus its adjacent triangles. For a vertex with valence k this leaves a k-sided hole. This hole can be fixed by any polygon triangulation algorithm [55]. Although there are several combinatorial degrees of freedom, the number of triangles will always be $k-2$. Hence the removal operation decreases the number of vertices by one, and the number of triangles by two.

Another decimation operator takes two adjacent vertices $\mathbf{p}$ and $\mathbf{q}$ and collapses the edge between them, i.e., both vertices are moved to the same new position $\mathbf{r}$ [27]. As a result, two adjacent triangles degenerate and can be removed from the mesh. In total this operator also removes one vertex and two triangles. The degrees of freedom in this *edge collapse* operator emerge from the freedom to choose the new position $\mathbf{r}$.

Both operators discussed so far are not unique. Either case involves some optimization to find the best local triangulation or the best vertex position. Conceptually this is not well-designed since it mixes the global optimization (which candidate is best according to the sorting criteria for the heap) with local optimization.

A possible way out is the so-called *half-edge collapse* operation: for an ordered pair $(\mathbf{p}, \mathbf{q})$ of adjacent vertices, $\mathbf{p}$ is moved to $\mathbf{q}$'s position [39]. This can be considered as a special case of edge collapsing where the new vertex position $\mathbf{r}$ coincides with $\mathbf{q}$. On the other hand, it can also be considered as a special case of vertex deletion where the triangulation of the k-sided hole is generated by connecting all neighboring vertices with vertex $\mathbf{q}$.

The half-edge collapse has no degrees of freedom. Notice that $(\mathbf{p} \to \mathbf{q})$ and $(\mathbf{q} \to \mathbf{p})$ are treated as independent removal operations both of which have to be evaluated and stored in the candidate heap. Since half-edge collapsing is a special case of the other two removal operations, one might expect an inferior quality of the decimated mesh. In fact, half-edge collapsing merely sub-samples the set of original vertices while the full edge collapse can act as a low-pass filter where new vertex positions are computed, e.g., by averaging original vertex positions. However, in practice this effect becomes noticeable only for extremely strong decimation where the exact location of individual vertices really matters.

The big advantage of half-edge collapsing is that for moderate decimation, the global optimization (i.e., candidate selection based on user specified criteria) is completely separated from the decimation operator which makes the design of mesh decimation schemes more orthogonal.

All the above removal operations preserve the mesh consistency and consequently the topology of the underlying surface. No holes in the original mesh can be closed, no handles can be eliminated completely. If a decimation scheme is also able to simplify the topology of the input model, we have to use non-Euler removal operators. The most common operator in this class is the *vertex contraction* where two vertices $\mathbf{p}$ and $\mathbf{q}$ can be contracted into one new vertex $\mathbf{r}$ even if they are not connected by an edge [18,56]. This operation reduces the number of vertices by one but it does keep the number of triangles constant. The implementation of mesh decimation based on vertex contraction requires flexible data structures that are able to represent non-manifold meshes since the surface patch around vertex $\mathbf{r}$ after the contraction may no longer be homeomorphic to a (half-)disc.

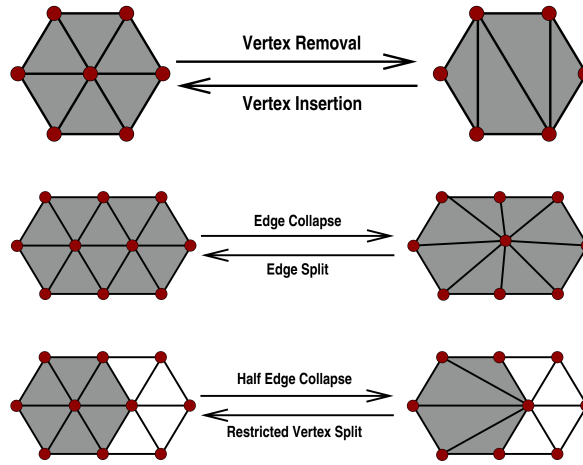


Fig. 3. Euler-operations for incremental mesh decimation and their inverses: vertex removal, full edge collapse, and half-edge collapse.

Distance measures Guaranteeing an approximation tolerance during decimation is the most important requirement for most applications. Usually an upper bound ε is prescribed and the decimation scheme looks for the mesh with the least number of triangles that stays within ε of the original mesh. However, exactly computing the geometric distance between two polygonal mesh models is computationally expensive [38,7], and hence conservative approximations are used that can be evaluated quickly.

The generic situation during mesh decimation is that each triangle T_i in the decimated mesh is associated with a sub-patch S_i of the original mesh. Distance measures have to be computed between each triangle T_i and either the vertices or faces of S_i. Depending on the application, we have to take the maximum distance or we can average the distance over the patch.

The simplest technique is error accumulation [55]. For example, each edge collapse operation modifies the adjacent triangles T_i by shifting one of their corner vertices from $\mathbf{p}$ or $\mathbf{q}$ to $\mathbf{r}$. Hence the distance of $\mathbf{r}$ to T_i is an upper bound for the approximation error introduced in this step. Error accumulation means that we store an error value for each triangle and simply add the new error contribution for every decimation step. The error accumulation can be done based on scalar distance values or on distance vectors. Vector addition takes into account the effect that approximation error estimates in opposite directions may cancel each other.

Another distance measure assigns distance values to the vertices $\mathbf{p}_j$ of the decimated mesh. It is based on estimating the squared average of the distances of $\mathbf{p}_j$ from all the supporting planes of triangles in the patches S_i

which are associated with the triangles T_i surrounding $\mathbf{p}_j$. This is, in fact, what the quadric error metric does [18].

Initially we compute the error quadric E_j for each original vertex $\mathbf{p}_j$ according to (3) by summing over all triangles which are directly adjacent to $\mathbf{p}_j$. Since we are interested in the *average* squared distance, E_j has to be normalized by dividing through the valence of $\mathbf{p}_j$. Then, whenever the edge between two vertices $\mathbf{p}$ and $\mathbf{q}$ is collapsed, the error quadric for the new vertex $\mathbf{r}$ is found by $E_r = (E_p + E_q)/2$.

The quadric error metric is evaluated by computing $E_j(\mathbf{p}_j)$. Hence when collapsing $\mathbf{p}$ and $\mathbf{q}$ into $\mathbf{r}$, the optimal position for $\mathbf{r}$ is given by the solution of (4). Notice that due to the averaging step the quadric error metric neither gives a strict upper nor a strict lower bound on the true geometric error.

Finally, the most expensive but also the sharpest distance error estimate is the *Hausdorff distance* [38]. This distance measure is defined to be the maximum minimum distance; i.e., if we have two sets A and B, then $H(A, B)$ is found by computing the minimum distance $d(\mathbf{p}, B)$ for each point $\mathbf{p} \in A$ and then taking the maximum of those values. Note that in general, $H(A, B) \neq H(B, A)$ and hence the *symmetric Hausdorff distance* is the maximum of both values.

If we assume that the vertices of the original mesh represent sample points measured on some original geometry, then the faces have been generated by some triangulation pre-process and should be considered as piecewise linear approximations to the original shape. From this point of view, the correct error estimate for the decimated mesh would be the one-sided Hausdorff distance $H(A, B)$ from the original sample points A to the decimated mesh B.

To efficiently compute the Hausdorff distance we have to keep track of the assignment of original vertices to the triangles of the decimated mesh. Whenever an edge collapse operation is performed, the removed vertices $\mathbf{p}$ and $\mathbf{q}$ (or $\mathbf{p}$ alone in the case of a half-edge collapse) are assigned to the nearest triangle in a local vicinity. In addition, since the edge collapse changes the shape of the adjacent triangles, the data points that previously have been assigned to these triangles must be re-distributed. Consequently, every triangle T_i of the decimated mesh at any time maintains a list of original vertices belonging to the currently associated patch S_i. The Hausdorff distance is then evaluated by finding the most distant point in this list.

A special technique for exact distance computation is suggested in [9], where two offset surfaces to the original mesh are computed to bound the space where the decimated mesh should remain.

Fairness criteria The distance measures can be used to decide which removal operation among the candidates is legal and which is not (because it violates the global error threshold ε). In an incremental mesh decimation scheme we have to provide an additional criterion which ranks all the legal

removal operations. This criterion determines the ordering of the candidates in the heap.

One straightforward solution is to use the distance measure for the ordering as well. This implies that in the next step the decimation algorithm will always remove that vertex that increases the approximation error the least. While this is a reasonable heuristic in general, we can use other criteria to optimize the resulting mesh for special application-dependent requirements.

For example, we might prefer triangle meshes with faces that are as close as possible to equilateral. In this case we can measure the quality of a vertex removal operation, e.g., by the *longest edge to inner circle radius ratio* of the triangles after the removal.

If we prefer visually smooth meshes, we can use the maximum or average normal jump between adjacent triangles after the removal as a sorting criterion. Other criteria might include color deviation or texture distortion if the input data does not consist of pure geometry but also has color and texture attributes attached [8,10,19,29,49].

All these different criteria for sorting vertex removal operations are called *fairness criteria* since they rate the quality of the mesh beyond the mere approximation tolerance. If we keep the fairness criterion separate from the other modules in an implementation of incremental mesh decimation, we can adapt the algorithm to arbitrary user requirements by simply exchanging that one procedure. This gives rise to a flexible tool-box for building custom tailored mesh decimation schemes [39].

2.5 Resampling

While vertex clustering and incremental decimation distill the set of original vertices to obtain the vertices of the decimated mesh, resampling techniques distribute new surface samples on the faces of the input mesh. These new samples are triangulated to form the decimated mesh while all original vertices are removed. The advantage of resampling techniques is that the strategy to place the new samples can adapt the local vertex density to user specified requirements. A disadvantage is that alias errors can occur in the vicinity of sharp features (as is the case with any discrete sampling scheme).

Classical resampling schemes run over all faces of the given mesh and place a new sample with a probability that is proportional to the area of the face or to some curvature dependent measure [63]. Since the number of triangles is approximately twice the number of vertices in the original mesh and since we want to reduce the mesh complexity, the resampling is very likely to place at most one new sample in each original face.

The new samples can be integrated into the given mesh by 1-to-3 splits of the corresponding triangles. This leads to a meta-mesh that contains the original vertices as well as the new samples. In a second step we can now eliminate all original vertices by vertex deletion or edge collapsing such that

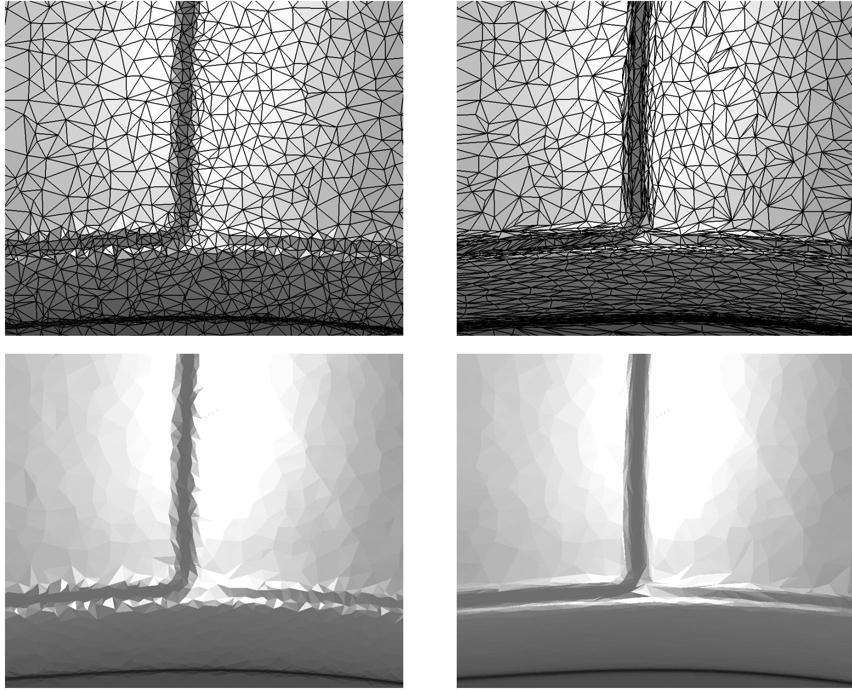


Fig. 4. Different fairness criteria can optimize the decimated mesh with respect to the shape of the individual triangles (left) or with respect to the overall smoothness of surface (right).

we end up with a decimated mesh that consists of the new samples only (which lie on the original surface).

The quality of the resampled mesh can be improved if the new samples are not placed independently from each other. Samples in neighboring or nearby faces of the original mesh should be distributed evenly. One way to achieve this is to shift the samples within the surface like particles that follow attraction and repulsion forces. Obviously, this re-distribution of the sample points has to be done before the samples are topologically inserted into the mesh.

Another motivation for resampling techniques is to increase the regularity of the mesh connectivity. In a regular triangle mesh, every vertex has exactly valence 6 but it is well-known that we cannot find a globally regular mesh for surfaces that are not homeomorphic to (a part of) the torus. Increasing the regularity therefore means reducing the number of mesh vertices with valence $\neq 6$.

One important class of meshes are those with *semi-regular* connectivity. In this case the mesh is made out of large triangular patches with regular

connectivity. In between these patches we have a small number of isolated extraordinary vertices with valence $\neq 6$.

There are several algorithms to convert an arbitrary input mesh into one with semi-regular connectivity by resampling [14,25,44,41,16,17]. The generic structure of such algorithms is the following: first, a small set of base points is distributed over the surface. These base points can be generated by a random process or they can emerge from an incremental decimation scheme. Those base points are then connected by geodesic curves. The collection of all geodesics between base points splits the input mesh into a set of triangular patches.

For each patch, a parameterization over a unit triangle is computed. To minimize distortion, *harmonic parameterizations* are preferred [15]. One way to think of harmonic parameterizations is to consider the mesh as a mass-spring system: each edge of the original mesh is replaced by a spring with the remainder of the length proportional to the actual length of the edge. To find the parameter values for each vertex of the triangular patch we fix the boundary vertices of that patch on the boundary of the unit triangle. Spring-energy minimization will find an equilibrium state that corresponds to a planar triangulation of the unit triangle. By this, we assign a parameter value to each vertex of the triangular patch.

The actual resampling is now performed by evaluating the parameterization on a uniform grid within the unit triangle. The resulting samples on the surface can be connected trivially to a regular patch. The collection of all these regular patches provides a piecewise regular resample of the original mesh. This process is sometimes called *remeshing*.

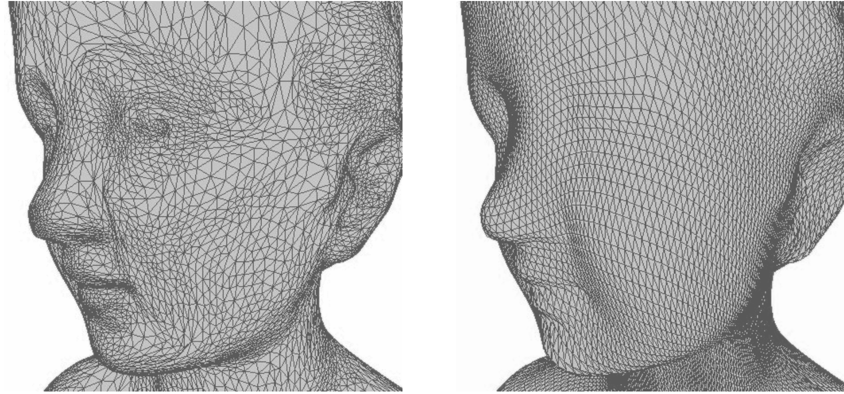


Fig. 5. Resampling techniques are able to impose a semi-regular structure on the resulting mesh.

3 Connectivity Coding

In this section we describe how the face vertex incidence table of a mesh can be encoded efficiently. If the table is encoded by a list of vertex indices, the connectivity of a triangle mesh with f triangles and v vertices consumes $3f\lceil\log_2 v\rceil \approx 6v\lceil\log_2 v\rceil$ bits, i.e. $6\lceil\log_2 v\rceil$ bits per vertex (bpv). For a mesh with $10,000$ vertices this results in $6 \times 14 = 84$ bpv, which is of the order of the geometry data.

The connectivity graph of typical polygonal meshes is well behaved in the sense that the maximum number of faces around a vertex as well as the maximum number of vertices in a face is limited by a small constant and by the graph being nearly planar. Furthermore, most applications do not rely on a special order of vertices or faces but only need to know the structure of the connectivity graph. This means that no damage is caused if a connectivity compression scheme rearranges the vertices and faces into the order in which it encounters the mesh elements during some deterministic traversal of the connectivity graph. For planar graphs Tutte [64] enumerated all the different structures that a connectivity graph can assume, showing that in the case of triangular graphs, the encoding consumes at least $\log_2 \frac{256}{27} \approx 3.245$ bpv, i.e. the entropy of the connectivity graph is 3.245 bpv. On the other hand, this raises the hope that there is a coding scheme consuming a constant number of bits per vertex, at least for planar graphs. Turan [62] was the first to report such a coding scheme. As meshes of genus zero, i.e. topologically equivalent to a sphere, are planar graphs and as the genus of most encountered meshes is typically small, we can build on this result.

The first attempt to code non-planar triangular meshes was made by Deering [13] and improved by Chow [5], who accelerated the data transfer from the CPU to a special purpose graphics accelerator that implemented the decoding algorithm. Later, Taubin and Rossignac [59] aimed for maximum compression of triangular meshes with a combined method for connectivity and geometry coding. The mesh connectivity coding part is similar to Turan's method but works for arbitrary triangle meshes. In the case when a mesh is non-planar and also non-manifold, the mesh can be cut at non-manifold regions by duplicating some of the mesh elements. Rossignac and Cardoze [54] attempt to minimize the number of mesh element replications. Gueziec and Taubin [21] show how to encode the non-manifold part of the incidence relations explicitly.

In this section we focus on recent connectivity coding methods for manifold meshes that grow a region over the mesh and incrementally encode the mesh elements and their incidence relation to the growing region. The methods can be categorized as face based, edge based and vertex based methods according to the type of mesh element playing the dominant role in the compression scheme. The Cut-Border Machine proposed by Gumhold and Strasser [22] and Rossignac's Edge Breaker [53] are face based coding schemes, the Face Fixer of Isenburg and Snoeyink [32] is edge based and the triangle

mesh compression technique of Touma and Gotsman [61] is vertex based. Before we discuss these methods in detail we describe a common framework.

3.1 Basics of Connectivity Coding

The *growing region* is the collection of faces that have been *processed* by the connectivity coding method. The remaining faces are said to be *untouched*. An edge or vertex is called *processed*, iff all incident faces have been processed. Vertices and edges not yet processed are called *active*, iff at least one incident face has been processed. The collection of all active edges and vertices forms the *cut-border* that separates the processed faces from those not yet processed. Similar to untouched faces, non-processed vertices and edges are said to be untouched. During the growing process only untouched faces incident to the cut-border are processed. The active edge incident to the next treated face is called the *gate*. The gate is an oriented edge and points to the *pivot* vertex, whose neighborhood is encoded next. The untouched face incident to the gate is called the *current face*. The *growing operations* are described by *command symbols* and define the type of the current face, the incidence relation to the cut-border and/or further attributes of non-active edges and vertices incident to the current face. To begin coding a mesh, the growing region is

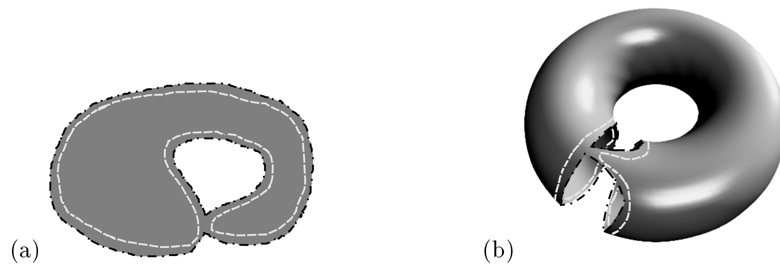


Fig. 6. Special situations during region growing a) split b) merge.

initialized to a face whose surrounding edges become the initial cut-border. In the case of a mesh with border one can assign a virtual face to each border loop. These virtual border faces can also be used to initialize the cut-border. Figure 6 shows two special situations with which any of the coding methods must deal. The *split* situation in Figure 6 (a) arises when the cut-border grows into itself. The current cut-border loop is split into two loops. The light dashed line shows the cut-border before the split and the black dash-dotted line the cut-border after the split. Coding proceeds with one of the loops, which finally closes with an *end* situation, where only one last face or edge is left in the current loop that vanishes after the last element has been encoded. The cut-border loops produced by splits are handled with a *loop stack* that stores a loop with pivot and gate location in each entry. After each

split situation one of the resulting cut-border loops is pushed onto the stack together with its gate and pivot locations. This loop is popped from the loop stack after the end situation that terminates coding of the other loop.

The *merge* situation as depicted in Figure 6 (b) arises once per handle. Here two loops of the cut-border grow into each other and are merged to one loop. The basic algorithmic scheme for all region growing coding methods proceeds as follows

- build a data structure with efficient access to incidences and adjacencies
- reconstruct mesh border loops
- initialize the cut-border to a border loop or to a face and define a pivot and gate location
- repeat
  - analyze incidences at the gate and determine next growing operation
  - in case of split, push one loop on the stack and proceed with other
  - in case of end, pop loop from stack or terminate if stack is empty
  - in case of merge, extract second loop from stack and merge it to current loop
  - in all cases, add new mesh elements, update incidences and choose new gate location and, if necessary, a new pivot vertex.

3.2 Connectivity Coding Methods

Face-Based Coding techniques define growing operations that, on the one hand, specify the size of the face incorporated at the gate, and on the other hand, the incidence relation between the face and the cut-border. The Cut-Border Machine [22] and the Edge Breaker [53] methods each describe a set of growing operations for the special case of pure triangular meshes. As all faces are triangles, only the different incidences of the faces need to be specified, as illustrated in Figure 7. We unify the notations and basic ideas of both methods but also explain the main differences.

Both methods in the preprocessing stage determine all border loops of the model and initialize the cut-border to one of the border loops. If the model has no single border loop, the cut-border is initialized to the edge loop of an arbitrary triangle. The pivot vertex is chosen arbitrarily on the initial cut-border loop and the gate is set to the edge pointing to the pivot vertex. During the actual coding phase, the incidence relation of the current face with the cut-border is determined by a mesh data structure of choice that allows for constant time query of the incidences. Each incidence is encoded by a symbol together with parameters that give all information needed by the decoding process.

Figure 7 illustrates the different growing operations. The legend at the end defines the appearance of untouched, active and processed faces and vertices and of the gate and cut-border before and after the operation. The pivot vertex is always the one to which the gate points.

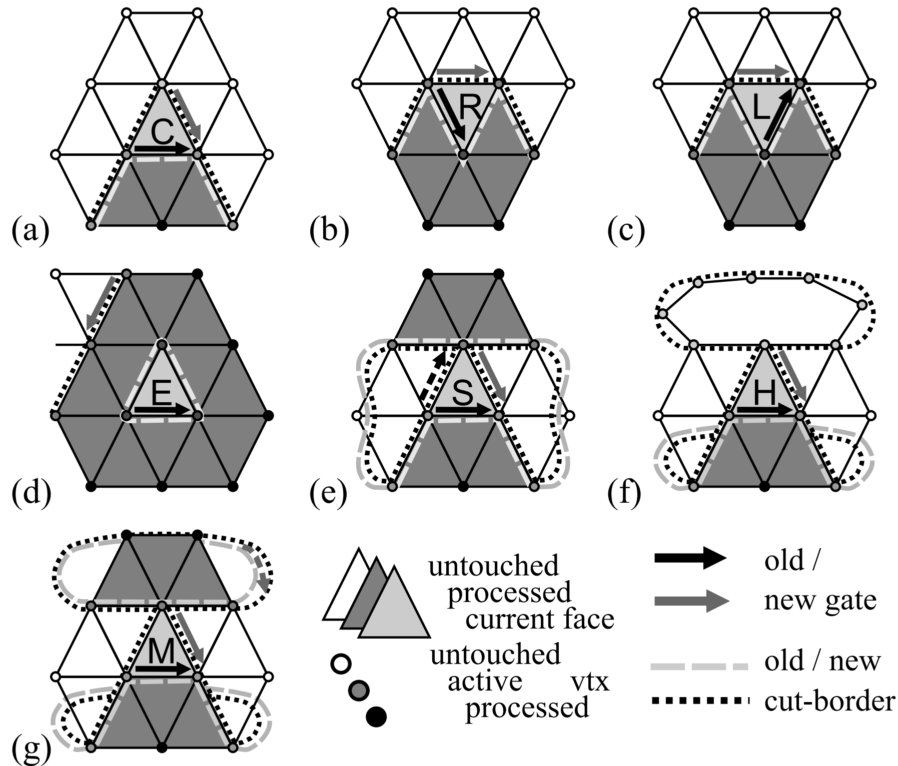


Fig. 7. Face-based update operations for the triangular case.

(a) The ***center* operation *C*** adds a new triangle to the growing region that is incident only to the old gate. A new active vertex is introduced. After the center operation the new gate is chosen such that it points to the pivot vertex.

(b) In the ***right* operation *R*** the current face is incident to the gate *and* the next edge on the cut-border. The neighborhood of the pivot vertex is closed and a new pivot vertex with a new gate is chosen on the cut-border.

(c) In the ***left* operation *L*** the current face is incident to the gate and the previous edge on the cut-border. The neighborhood of the start vertex of the gate is closed and the pivot vertex is preserved.

(d) In the ***end* operation *E*** all edges of the current face are incident to the cut-border. After the end operation the current cut-border loop vanishes. Coding terminates, if the loop stack is empty, or continues with the top loop, gate and pivot vertex from the stack.

(e) The ***split* operation S_i** handles the split situation in Figure 6 (a), where the cut-border grows into itself such that the current triangle touches the so-called *split vertex*, which is the i-th vertex after the pivot vertex on the current cut-border loop. If i is negative, it defines the $(|i| + 1)$-th vertex

before the pivot vertex. i is called the *split index*. The left resulting loop is pushed onto the loop stack together with the left edge of the current face as gate and the split vertex as pivot vertex. Coding goes on with the right loop at the gate, pointing to the previous pivot vertex.

(f) The ***hole* operation** $\boldsymbol{H}_l$ merges the current cut-border with a border loop. The index l specifies the length of the loop, which needs to be known by the decoder, and is encoded in addition to the operation symbol H. Before the hole operation the border loop is rotated such that the operation always connects to the first vertex of the border loop.

(g) The ***merge* operation** $\boldsymbol{M}_{s,i}$ encodes the merge situation in Figure 6 (b) and merges the current cut-border loop at the *merge vertex* with a second loop somewhere inside the loop stack. The index s specifies the location of the second loop in the loop stack and i is the location of the merge vertex in the second loop relative to its pivot vertex.

The Cut-Border Machine coding method encodes and decodes the connectivity in exactly the same way and exploits the knowledge about the length of the current cut-border loop. It defines an alphabet of the symbols C,R,L,H,M and for each index $i = \pm 2, \pm 3, \ldots$ a symbol S_i. The end symbol is not needed as it can only arise when the current cut-border loop is of length three. In this case no split operation can arise and the left, right and end operations lead to the same result, such that the end operation can be encoded through an L or R. The indices necessary for the hole and merge operations are encoded in addition to the symbols. The Cut-Border Machine permutes the vertices of the mesh in the order in which the vertices are introduced by C operations. The triangles are permuted in the order in which they are introduced by any of the symbols. In the decoding stage all encoded operations are simply replayed and the connectivity is built up triangle by triangle.

The Edge Breaker method does not encode the split indices. This is possible if the end operation is explicitly coded with an E symbol. If we assume that there are no merge or hole operations, the split indices can be reconstructed in the following way. The symbols following a split operation describe the right loop, which is terminated with an end operation. As there can be further splits inside the right loop, one has to jump over all nested pairs of S and E symbols when looking for the E symbol of the current S. An always positive split index can be reconstructed from the length of the right loop minus one. In order to determine the length of the right loop we examine the influence of the different operations on the loop length. L and R decrement the current loop length by one, C and S increment it by one and E decrements it by three. Thus we just have to count the number of each type of symbol after the current S up to and including the corresponding E and the split index computes to $i = \#_R + \#_L - \#_C - \#_S + 3\#_E - 1$.

A simple approach [31] to decoding the operation symbols without the split indices, which also handles merge and hole operations, is to decode in reverse order starting with the final E symbol at the end of the symbol se-

quence. It is possible to reverse all the operations in Figure 7. The untouched triangles become the processed triangles and vice versa. The old gate and cut-border become the new gate and cut-border and vice versa. Processed and untouched vertices exchange their meaning and active vertices are still active. Let us now describe the reverse decoding process and examine what needs to be known to decode the inverse operations.

The inverse E operation decodes a face that defines a new cut-border loop, gate and pivot location. The current cut-border loop and gate (if any) are pushed onto the loop stack. The indices of the three newly introduced active vertices of the decoded face are not finalized yet and are set to dummy indices. The inverse R and L operations introduce one more active vertex. The inverse S operation pops a cut-border loop from the stack and merges it with a triangle connecting the two gates. No split index is necessary to encode the split operation! The two active vertices that represent the split vertex in each loop are identified. This is the reason why newly introduced active vertices are not final and have dummy indices. The dummy indices can be finalized after the neighborhood of the vertex has been closed, i.e. whenever a vertex changes its state from active to processed. This happens after an inverse C operation, after which the vertex whose neighborhood closes receives the next higher final vertex index. In this way the vertices are enumerated in reverse order compared to the Cut-Border Machine encoding. Therefore, also the vertices are decoded in reverse order.

The inverse hole operation H_l splits a mesh border loop off the current cut-border loop by connecting the gate to a second edge on the current loop. The location of the second edge can be determined from the length of the border loop, such that the Edge Breaker scheme encodes H_l in the same way as the Cut-Border Machine. Finally, the inverse merge operation $M_{s,i}$ splits the current loop into two and inserts one loop into the stack at location s exactly where it was during encoding. The gate and pivot location of the loop on the stack is given by i, but for reverse decoding of the merge operation also, the length l of the loop that is put on the stack needs to be known and the symbol M and the indices s, i and l are encoded for each merge operation. Figure 8 illustrates the face based coding and the reverse decoding on a simple example. The cut-border is initialized to the border loop (a). Two C operations introduce the first two interior vertices (b) and (c). Then the neighborhood of the bottom right vertex is closed with a R operation (d). The vertex becomes processed. Two further R operations (e,f) and two C operations (g,h) follow. The S operation (i) splits the cut-border into two loops. The Cut-Border Machine encodes the symbol S_2. One loop together with the location of its gate (drawn dashed) is pushed onto the stack. Figure 8 (i') illustrates the reverse decoding, which we will return to later. An E operation closes the first loop (j). The loop pushed earlier is popped from the stack together with its gate location. A C (k) and two R (l,m) operations follow

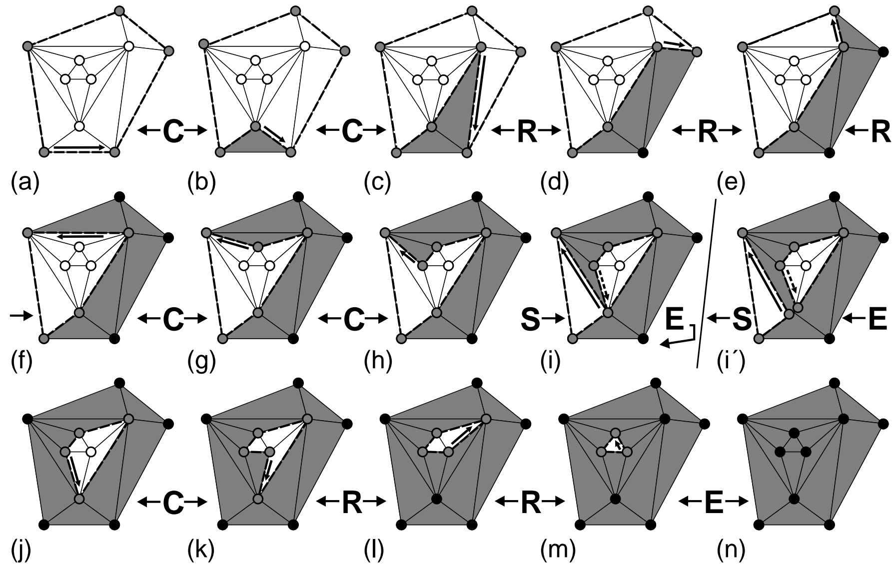


Fig. 8. Example of face-based connectivity coding.

before a second E operation closes the last cut-border loop and terminates the encoding process (n). The resulting code string is CCRRRCCSECRRE.

The Cut-Border Machine decoding simply replays Figure 8 (a–n). The reverse Edge Breaker decoding starts with the inverse E operation (m). The black vertices and gray triangles are the not-yet decoded mesh elements. The following two inverse R operations (l,k) decode two triangles and introduce two more active vertices with dummy indices. The C operation (j) completely decodes the neighborhood of the vertex in the middle, and its vertex index is finalized. Figure 8 (i') illustrates how the E operation introduces a second cut-border loop, while the active one is pushed onto the loop stack. At this point we do not know that one of the active vertices in the new cut-border loop is the same as the bottom-most vertex in the pushed loop, illustrated by duplicating the vertex. The following inverse S operation (h) unites the two copies of the same vertex. The remaining sequence of C and R operations (g–a) decodes the remaining triangles. The cut-border loop left at the end of the decoding process defines the mesh border.

The operation symbol string of simple meshes is free of H and M operations. Closed triangle meshes satisfy $f = 2(v - \chi)$ and meshes with border satisfy $f \leq 2(v - \chi)$. Thus in the case of a simple mesh, $v \geq f/2 + 2$. As the center operation is the only operation that introduces new vertices, at least every second operation must be a C. If the C operation is encoded with a one bit code and the four remaining operations each with a three bit code, a simple mesh is encoded with v bits for the C operations and $3(f - v) \leq 3(v - 4)$

bits for the other operations which results in less than four bits per vertex on the average.

Gotsman and Kronrod [43] generalize the Edge Breaker to arbitrary polygonal meshes. The case of pure quadrilateral meshes can be encoded with 3.5 bpv and the case of quadrilateral meshes with a minority of triangles in 4 bits per face.

Edge-Based Coding defines two types of update operations. The first type attaches faces and other edge loops to the current cut-border at the gate. The adjacency of the new face or border loop is only defined at the gate. The adjacencies of the other newly introduced active edges are specified separately with *gluing operations*. Each gluing operation identifies two active edges and therefore specifies the adjacency of their incident faces. Isenburg and Snoeyink proposed a method for polygonal meshes [32] called Face Fixer. The following operations are introduced:

The *face* operation F_l attaches a face with l edges to the current cut-border loop. Similarly, the *hole* operation H_l creates a border loop with l edges and is handled exactly as in the case of face based coding. The gluing operations of Face Fixer resemble corresponding face based operations. The *right* operation R identifies the gate edge with the next edge on the cut-border. In the triangular case the face based R operation is equivalent to a F_3 followed by a R operation. The *left* operation L identifies the gate with the previous edge on the cut-border. The *split* operation S identifies the gate with another edge on the cut-border splitting the current loop into two loops. Decoding is performed in reverse, which allows us to avoid any additional parameters for the split operation without further computations. The *merge* operation $\mathrm{M}_{s,i,l}$ handling the situation of Figure 6 (b) connects the gate with an edge on a different cut-border loop merging the two loops. For reverse decoding the location s of the second loop on the loop stack, the location i of the gate in the second loop and the length l of the second loop i is encoded. Figure 9 illustrates the Face Fixer encoding and decoding on a polygonal mesh with border. To encode the mesh, the cut-border is initialized to the border loop of the mesh (a). The gate edge is marked inside the growing region to better visualize the reverse decoding. The first F_3 operation (b) adds a triangle to the growing region. The gate cycles around the vertex to which it points. Next, a quadrilateral is incident to the gate. In (c) the incidence of the left quadrilateral edge to the cut-border is not specified yet, as illustrated by the cut open edge. The L operation (d) specifies this incidence, closing the edge. Further operations are applied introducing faces and gluing edges together until only one cut open edge is left in (n), which is closed with the final E operation (o) yielding the code string $\mathrm{F}_3\mathrm{F}_4\mathrm{L}\mathrm{F}_3\mathrm{F}_4\mathrm{L}\mathrm{F}_3\mathrm{L}\mathrm{F}_5\mathrm{L}\mathrm{L}\mathrm{F}_4\mathrm{L}\mathrm{L}\mathrm{F}_3\mathrm{L}\mathrm{E}$.

Reverse decoding starts without any knowledge in (o). The inverse E operation (n) generates a cut open edge with a gate location. The inverse L operation (m) adds another open edge before the inverse F_3 operation (l) generates a triangle. Two L operations (k,j) and a F_4 operation (i) attach a

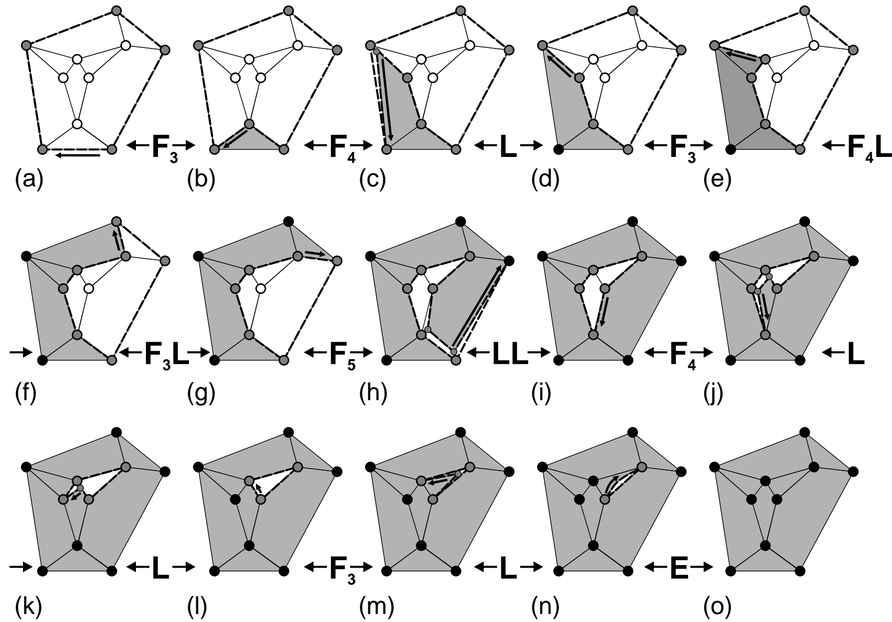


Fig. 9. Example of edge-based connectivity coding.

quadrilateral to one edge of the triangle. Another two L operations (h) and a F_5 operation (g) attach a pentagon to two of the previous cut-border edges. Decoding continues until only the initial border loop remains.

Vertex-Based Coding encodes the vertex valences in order to exploit this information later on. Vertex-based coding was developed by Touma and Gotsman [61] for the special case of triangular meshes. In the beginning all border loops are closed with a triangle fan around an additional dummy vertex; see Figure 10 (o) for clarification. For each newly introduced vertex its valence, i.e. the number of incident edges in the final mesh, is encoded. During encoding and decoding one keeps track of the number of untouched edges for each vertex on the cut-border not contained in the growing region. These edges are called *free edges*. Anytime an active vertex has no more free edges, its neighborhood can be closed with the last triangle. This happens every time when, in face based coding, a L,R or E operation is performed. Thus the information about the vertex valences can be exploited to avoid the explicit encoding of all L,R and E operations.

We are left with three mesh growing operations. The *add* operation A_i introduces a vertex of valence i. It is used at the beginning to specify the three vertices of the triangle initializing the cut-border and in each situation that corresponds to a C operation of Edge Breaker. After an add operation the gate is chosen as in face based coding such that it cycles around the *pivot* vertex and closes its neighborhood first. Anytime a R, L or E operation arises

in face based coding, at least one of the affected vertices on the cut-border has no more free edges. In this case the neighborhood of this vertex can be closed without encoding any operation symbol. After the neighborhood of the current pivot vertex has been closed, the new pivot vertex is chosen as the next vertex along the cut-border. For the *split* operation S_i the split index i is encoded such that the encoding and decoding process is done in forward direction. The trick with reverse decoding is not applicable as no E operations are encoded. No H operation is needed as all the border loops have been tessellated. The dummy vertices are marked as holes with a negative sign or their location in the code string is encoded. The *merge* operation $M_{s,i}$ takes two integer parameters specifying the location of the second cut-border loop in the stack and an index into the free edges of this loop.

In [1] Alliez and Desbrun describe a strategy to reduce the number of S operations in the valence-based scheme. They propose to choose the next pivot vertex based on the information about the number of free edges on the entire cut-border. They choose the vertex with the smallest number of free edges as these vertices are less likely to produce an S operation in their neighborhood. If there is more than one potential pivot vertex they compute a mean number of free edges considering also the left and right neighbors along the cut-border. This strategy for pivot vertex selection reduces the number of S operations by about 70%. Figure 10 illustrates the vertex-based coding on an example. The tessellation of the border loop with a dummy vertex is done in advance but only shown in (o). This increases the valence of the border vertices by one. Coding starts with an arbitrary triangle and the

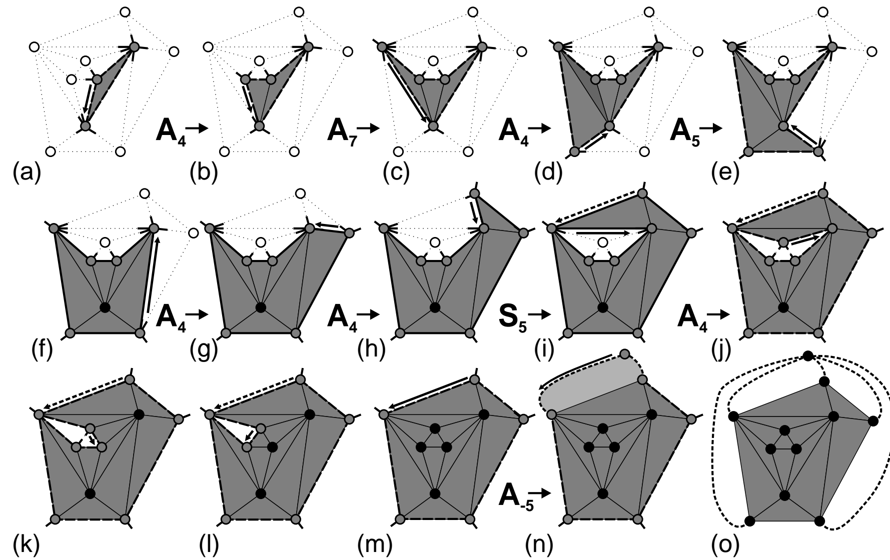


Fig. 10. Example of vertex based connectivity coding.

specification of the valences of the three incident vertices, in the examples 4, 6 and 7. The free edges for each vertex on the cut-border are illustrated as bold short arms in the figure. From (a) to (b) an A_4 operation introduces a vertex of valence 4. Three further A operations (c–e) specify vertices of valence 7, 4 and 5, where one free edge points to the dummy vertex of the border loop. In (e) the pivot vertex had no more free edges. Thus we can immediately close the neighborhood of the vertex. Two A operations (g,h) follow, introducing vertices with valence 4. The S_5 symbol (i) denotes a split operation and the number of free edges, in this case 5, skipped along the current cut-border loop, i.e. five of the bold short arms have to be skipped until the free edge is found to which the pivot vertex is connected. The outer cut-border loop together with its gate location is pushed onto a stack. An A_4 operation (j) specifies the last vertex of valence 4. The vertex pointed to by the gate has no more free edges and its neighborhood can be closed. In (k) the neighborhood of the next vertex without any more free edges is closed, and from (l) to (m) the current cut-border loop vanishes by closing the neighborhoods of the last three loop vertices. The previously pushed cut-border loop is retrieved and an A_{-5} operation introduces the dummy vertex of valence 5 that represents the border loop of the mesh. Till (o) the neighborhoods of the remaining vertices on the cut-border are closed. The complete code representing the connectivity of the mesh is $A_4A_5A_7A_4A_7A_5A_4A_4S_5A_4A_{-5}$. The decoding process replays the coding process and builds up the target data structure for the mesh connectivity.

3.3 Comparison and Discussion

All of the region-growing connectivity coding methods described in Subsection 3.2 may be implemented very efficiently. For simple meshes the coding and decoding time is linear in the number of mesh elements, as each growing operation can be implemented in constant time. The merge operation is an exception, but it does not appear in simple meshes and is rare in other models. Gumhold and Straßer [22] report compression speeds of half a million triangles per second on a SGI O2/R10000 with 175MHz.

Gumhold and Straßer [22] describe an update operation that reorients half of a border loop in order to handle non orientable meshes. This operation can easily be added to all other methods. Hole loops are encoded differently. In reverse decoding methods such as the Edge Breaker or Face Fixer, hole loops are incorporated into the current cut-border loop when the cut-border touches the hole for the first time. A symbol with the hole loop length as parameter is used for encoding. The vertex based method adds dummy vertices to close the hole loops. The valence of the dummy vertex is encoded with a minus sign, which is equivalent to encoding a symbol and the length of the hole loop.

Tutte showed that for planar triangulations, and therefore for simple triangular meshes, at least 3.245 bits per vertex (bpv) are necessary on the

average (over all possible mesh connectivities) for connectivity coding. We already showed that the Edge Breaker coding scheme encodes simple meshes with no more than 4 bpv. This result was improved to 3.67 bpv by King and Rossignac [36] and to 3.585 bpv by Gumhold [23]. For real-world 3D mesh connectivities, better compression ratios may be achieved (since the symbols in the resulting sequences have further dependencies). Many models exhibit regularity in the form of a large fraction of valence six vertices. Szymczak and King [58] showed that triangle meshes with more than 85% valence six vertices can be encoded in 1.85 bpv. Alliez and Desbrun [1] show that the entropy of the vertex valences corresponds to the lower bound of 3.245 bpv derived by Tutte. Thus for meshes with a negligible number of S operations, an arithmetic coder combined with a valence coder can generate a code whose length is very close to the theoretical lower bound. Regular triangle meshes with a large fraction of valence 6 vertices can be compressed with 2 bpv and less using valence coding. King et al. [37] generalize the Edge Breaker coding scheme for pure quadrilateral connectivities and prove that in the worst case no more than 2.75 bpv are consumed. Kronrod and Gotsman [43] generalize the Edge Breaker coding to polygonal meshes and give a worst case coding for quadrilateral meshes including a few triangles with less than 4 bits per face. The Face Fixer encodes general polygonal meshes to $2 - 3$ bpv on average.

3.4 Progressive Connectivity Coding

Progressive representations as introduced in the previous section represent models starting from a coarse base model – the so-called *base mesh* – as a sequence of vertex split or vertex insertion operations. The connectivity of the base mesh can be encoded with any of the coding techniques described in Section 3.2. Thus work on the progressive coding of meshes concentrates in the connectivity section on the efficient coding of vertex split and vertex insertion operations. The vertex split operation is defined by a vertex in the current connectivity – the *split vertex* – and the two incident *split edges* that are extended to the two triangles incident to the edge collapsed during mesh decimation. The vertex insertion is defined by the faces that were used during decimation to re-tessellate the hole resulting from the vertex removal.

Hoppe's [27] *progressive meshes* codes an arbitrary simplification process based on edge collapse but consumes $\log_2 v$ bits to encode the index of each split vertex plus an average of 5 bits to encode the two split edges. In [28] Hoppe shows how to reorder the vertex split operations such that the indices of the split vertices can be encoded with an average of 6 bpv.

Taubin et al. [60] propose to simplify models with a combination of several edge collapses at the same time such that the inverse operation splits a complete forest of edges and vertices. This *forest split* operation can be encoded with about 12 bpv. Pajarola and Rossignac [51] group vertex-split operations into batches and specify split vertices by marking each vertex of the current mesh using one bit, leading to a total of about 7 bpv. Cohen-Or,

Levin, and Remez [11] propose a progressive representation based on removal of independent sets of mesh vertices. They encode groups of vertex insertions by coloring the face patches at the coarse level with four or even two different colors only, resulting in 6 bpv. Finally, Alliez and Desbrun [2] describe a similar vertex insertion representation that primarily encodes the valences of the inserted vertices. If pure topological simplification criteria are used, the resulting code consumes only about 4 bpv, which is close to that achievable by single resolution methods.

4 Compression of Geometric Data

4.1 Introduction

The previous sections dealt with the efficient coding of the connectivity component of a 3D mesh, which has a discrete nature, essentially that of a graph. The following sections will deal with the second component of the 3D mesh — its geometry. The geometry manifests in 3D coordinate data — three real values per vertex. The first issue that must be addressed is the precision of this data. Typical 3D polygonal data sets are generated by 3D scanning devices, having finite precision, or as a result of a modeling process with an interactive software tool. In both cases, the 24-bit precision that the data is given in is usually far more than the actual information content of the data, and the higher-precision bits are just “noise”. A standard technique employed at one point or another in compressing numerical data is *quantization*. This means that the total number of possibilities of the data vectors, say n, is reduced significantly to a representative set of, say, $k \ll n$ vectors. This reduces the range of the input data set to a manageable size before the domain-specific coding technique is applied. Quantization, however, is irreversible, meaning that it will be impossible to recover the original data set once quantized, even after decoding. Techniques such as this are called *lossy* techniques, because some of the original data is lost. The more aggressive quantization is performed, the more loss is incurred, and the more compact the data becomes. An important question is how to optimize this tradeoff in order to minimize loss and also data code length.

The 3D coordinate vectors associated with each vertex of a mesh are not independent, hence collectively contain much less information than the sum of the information content of each individual coordinate vector. This is obviously true for smooth meshes, where there exists a strong correlation between the values of the coordinates of vertices neighboring in the mesh. As in most coding methods, this correlation may be exploited to minimize redundancy. A simple rule-of-thumb (but not so easy to exploit - as we shall see later), is that the coordinates of a vertex in a smooth mesh are very close to the simple average of the coordinates of its immediately neighboring vertices. This is an example of a *prediction rule*, where the coordinates of a set of vertices are used to predict the coordinates of a vertex v not in the set. In

this manner, if the code already enables the decoder to know the coordinate values of the set of vertices, the prediction rule may be applied to compute the as-yet unknown coordinates of v. Since this is only a (sometimes crude) estimate of the true value, it cannot suffice to decode the value satisfactorily, and a *prediction error* must be present in order to recover the correct value. The prediction error is just the difference between the true value and the predicted value, and is stored in the code. This type of coding is known as *predictive coding*. The fundamental assumption behind predictive coding is that the prediction errors are sharply concentrated about the origin, hence possess a much smaller entropy that the original data set.

Since the geometric coordinate data is associated with the mesh vertices, which are connected to each other as dictated by the connectivity information, this connectivity information plays an important role in the compression of the geometry, specifically in the prediction. It indicates which vertices should be used to predict another, and in which order. At the decoder two possibilities exist: decode the connectivity fully before starting to decode the geometry, or decode the geometry in lock-step with the connectivity. The advantage of the former is that the entire connectivity structure is available when decoding the geometry, hence better predictions can possibly be made. The advantage of the latter is a more efficient (in terms of run-time) decoding algorithm.

4.2 Quantization

Real numbers are traditionally represented in floating point format, i.e. with a mantissa and exponent. A 32 bit representation can distinguish between 2^{32} different values, which, in many cases, is far more than is needed for a given application, and the same amount of information may be represented in fewer bits. In many cases, the extra bits are "wasted", containing essentially random values, which ironically, some users tend to believe have an information content which must be preserved at all costs.

Quantization is a subject that has been studied extensively. We shall elaborate here only on those topics relevant to 3D mesh coding. The simplest form of quantization is *uniform quantization*, where the domain of interest is discretized onto a uniformly-spaced multidimensional grid structure. Given a set of data vectors, each one of these is then "snapped" to one of the grid points, usually the closest one. Figure 11a shows how a set of 20 two-dimensional vectors is quantized to a uniform 4x4 grid, hence requiring 4 bits per vector. Note that since there are 20 input data vectors, but only 16 quantized vectors, the mapping will not be bijective. In fact, as can be seen, only 11 quantization values are used, so many of these bits are wasted. However, the fact that there are only 16 distinct output vectors means that each may be represented in 4 bits, instead of the high precision of the input vectors. Albeit, *quantization error* has occurred in the transition, as, without additional information, the originals cannot be recovered. Assuming n input

vectors, $V = \{v_1, \ldots, v_n\}$, k quantization (vector) values $Q = \{q_1, \ldots, q_k\}$, and a mapping $m : V \to Q$, the quantization error is usually measured using the l_2 norm:

$$Err(v, m) = \sum_{i=1}^{n} \left\| v_i - m(v_i) \right\|^2. \quad (5)$$

For uniform quantization, the size of the region to be quantized must be specified. Obviously, the smaller it is, the less quantization error will be incurred for a given number of quantization levels. It would be wasteful to quantize regions of the data domain in which there are no input data vectors. The standard way to do this is to enclose the input data vectors in a multidimensional *bounding cube*, which is as small as possible. For example, in three dimensions, denote the length of each edge of the cube by l, originating at the point (x_0, y_0, z_0). Then the x dimension of the cube is sampled at the k locations $\{x_0, x_0 + \frac{l}{(k-1)}, x_0 + \frac{2l}{(k-1)}, \cdots, x_0 + l\}$ as are the y and z dimensions. Usually k is taken to be a power of two and then each quantized coordinate may be expressed in k bits. Using a bounding cube is to be preferred over a bounding *box*, where the quantization interval in each dimension can be different, depending on the spread of the coordinate, as then each *quantization cell* is a cube, even though it might seem wasteful, since many of these cubes will be void of data points. For uniform quantization, the mapping m independently associates with each coordinate the quantization value closest to it. Once each dimension has been quantized to k^3 values, each coordinate of the vertex geometry vector may be represented as an integer in the range $\{0, \ldots, k-1\}$. In order to recover the original quantized points, the code must contain the parameters of the bounding box. Uniform quantization is simple to understand and implement, but is not optimal, since it takes into account only the length of the interval in which the data points lie, but not the distribution of the points within the interval. It would seem more reasonable to position more quantization vectors in the regions where more of the data points lie, and hence reduce the value of the quantization error in (5). This is possible, and is known as *non-uniform* quantization. The simplest way to do this is on each dimension separately, i.e. solve three independent one-dimensional optimization problems. The result is three sets of quantization values: $\{x_0, x_1, x_2, \ldots, x_{k-1}\}$, $\{y_0, y_1, y_2, \ldots, y_{k-1}\}$, $\{z_0, z_1, z_2, \cdots, z_{k-1}\}$. The quantization vectors are the *cartesian product* of these three sets, namely the set of $K = k^3$ 3D vectors $\{(x_{i_1}, y_{i_2}, z_{i_3}) : 0 \leq i_1, i_2, i_3 \leq k-1\}$.

An even better way to reduce quantization error is to take into account the *joint* distribution of the data vectors, namely, the correlation between the three coordinates. The set of techniques which deals with this is known collectively as *vector quantization* (or VQ), and in general, good solutions are computationally expensive. In its most general form, vector quantization may be formulated as an optimization problem related to (5):

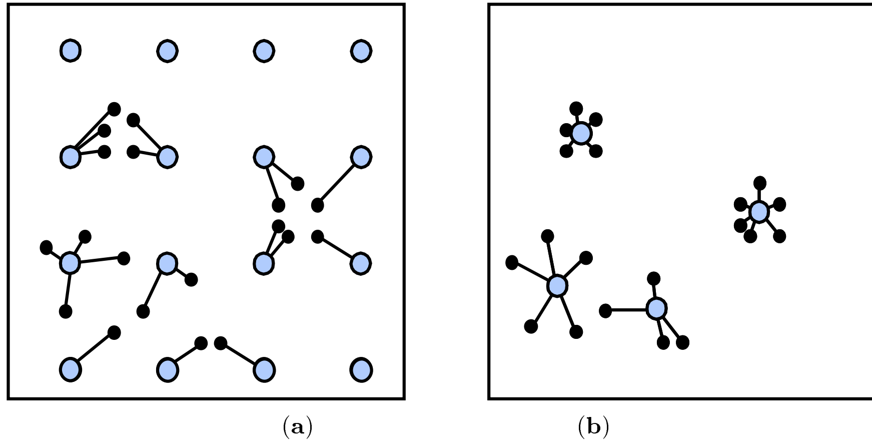


Fig. 11. Quantization of a 20 point data set (small black circles) in two dimensions. (a) Uniform quantization to 16 levels (large gray circles). Mapping is denoted by lines. Note that only 11 of the 16 quantization levels are used.
(b) Non-uniform quantization. Four levels suffice to achieve a quantization error comparable to (a).

$$(m, v) = arg\ min\ Err(v, m), \tag{6}$$

so that both the representative set Q and the mapping m are unknowns. The simplest solution for this problem is the classical Lloyd's algorithm (sometimes also known as the LLB algorithm). This algorithm maintains the n input vectors in K dynamic disjoint sets (or *clusters*), where all vectors in a set are mapped to the same representative. As the algorithm proceeds, both the sets and their representatives evolve until they converge to stable values. Figure 11 (b) shows the result of non-uniform quantization on the input vector set of Figure 11 (a). Four carefully chosen quantization vectors (2 bits) suffice to quantize the input set with comparable quantization error. A description of many VQ algorithms may be found in the classic book by Gersho and Grey [20].

When quantizing 3D mesh geometry, artifacts may occur if care is not exercised. One of the more common artifacts is the appearance of what seem to be "cracks" in the geometry. This is because in some models distinct vertices may have almost identical geometries, meaning that for all practical purposes they are located at the same position in space, possibly joining separate components of the mesh. This is very frequent in models where entire boundaries coincide. Since quantization in effect "moves" vertices in space, some vertices may move differently from others, and vertices which previously coincided may no longer do so, forming "cracks" or "overlaps"; see Figure 12 for an example. To prevent this problem, care should be exercised to move all (almost) coincident vertices together. This will usually happen automatically

if simple uniform quantization is employed on a common bounding box for the entire model.

One way to guarantee that coincident vertices are quantized to the same values is to perform a preprocessing stage in which these vertices are identified, and then treated together during quantization.

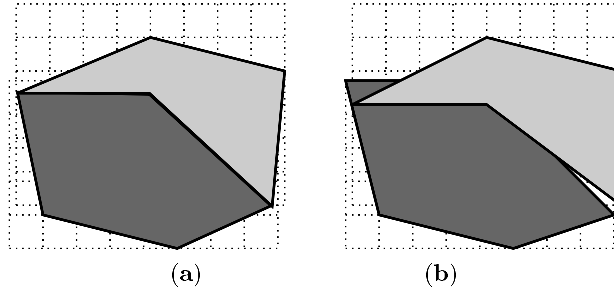


Fig. 12. "Cracks" formed by quantizing two model components using separate uniform quantization grids. (a) Before quantization (b) After quantization.

4.3 Prediction Methods

Predictive coding is the standard entropy-reduction method for coding geometry. It is based on the observation that there is some correlation between the geometry of a vertex and that of its neighborhood, so if a specific prediction rule is used, it suffices to code just the *prediction error*, namely the difference between the actual vertex geometry value and its predicted value. Hopefully, the entropy of the prediction error population will be much less than that of the original vector population, and significant savings can result.

Practical prediction methods are *local* and *causal*, meaning that the geometry of a vertex is predicted from a small number of neighboring vertices and in an order dictated by another process, usually the connectivity coding process. For example, using a vertex tree to code the connectivity, it is possible to predict the geometry of a vertex using the geometries of all its ancestors in the tree. Note that it is not possible to use the geometries of its descendants, since these will not necessarily be available at the decoder at the time when they are needed. The standard way to predict the geometry of a vertex is as a linear combination of the geometry of a small number of its ancestors in the tree. This is commonly called *linear predictive coding* (LPC), and is widely used in audio compression. The simplest linear prediction method is $p(v_i) = v_{i-1}$, meaning that the geometry of a vertex is predicted to be identical to that of its immediate ancestor, based on the underlying assumption that the geometry function is constant and, in effect, the difference between the two will be coded. A more sophisticated method is to

predict, based on the two previous ancestors, assuming the *first derivative* is constant: $v_i - v_{i-1} = v_{i-1} - v_{i-2}$, or, in other words, $p(v_i) = 2v_{i-1} - v_{i-2}$. Higher order predictors may be used, where predicting based on the previous k values involves assumptions on the first $k-1$ derivatives of the geometry function. For $k = 3$, the assumption of a uniform second derivative leads to the prediction rule: $p(v_i) = 3v_{i-1} - 3v_{i-2} + v_{i-3}$. In general the sum of the prediction coefficients will be one. These coefficients are data independent, and, theoretically, at least, given a mesh with vertices $\{v_i : i = 1,..,n\}$ in a prediction order, and integer k, it is possible to compute *optimal* prediction coefficients $a_1, \ldots, a_k$, such that the average prediction error is minimal:

$$(a_1, \ldots, a_k) = arg\ min \sum_{i=k+1}^{n} \left\| v_i - \sum_{j=1}^{k} a_j v_{i-j} \right\|^2 .$$

This can be solved using the linear least squares minimization technique, which involves computing the *singular value decomposition* (SVD) of a $k \times k$ matrix. These optimal coefficients and the first k values of the sequence must then be stored as part of the code. In practice, however, the additional benefit from optimal coefficients is small, and $k = 3$ is usually sufficient.

A form of the scalar LPC which better captures the spatial form of the vertex geometries relies on the triangle structure of the mesh, so that a vertex geometry may be predicted as a linear combination of other vertices in its preceding neighborhood. A good predictor of this kind is the so-called *parallelogram* predictor [61], which relies on the empirical observation that two adjacent triangles in a triangle mesh tend to form a parallelogram, hence the fourth vertex in such a structure may be predicted from the other three: $v_4 = v_3 + v_2 - v_1$; see Figure 13. Note that this implies that the four vertices are co-planar, which is usually not true, but relatively close to reality, especially for smooth meshes. Continuing this line of thought, it should be

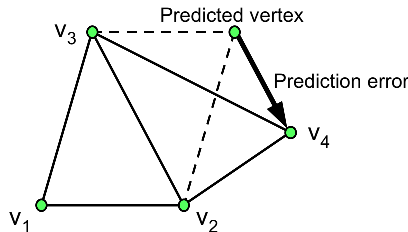
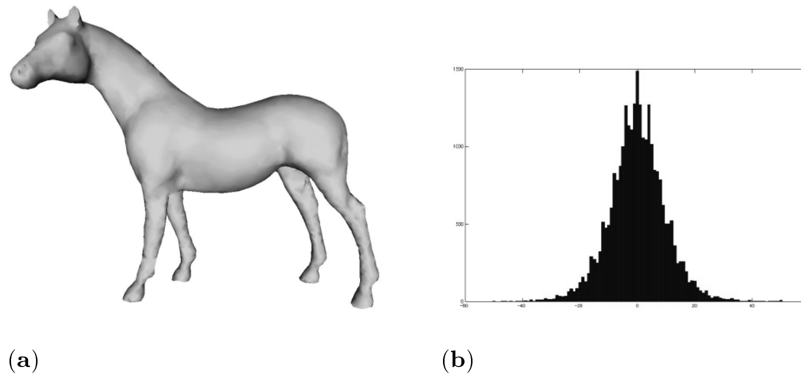


Fig. 13. Parallelogram predictor.

possible to find other local patterns in the mesh geometry which adjacent triangles satisfy, e.g. triangles forming a fan. If no prior information exists on the patterns in a mesh, it is possible to search for them as described by

Lee and Ko [45]. If a few such patterns dominate, it is possible to use them all as predictors, provided that the identifier of the predictor used at each vertex is specified in the code. The simplest way to do this is to insert an index into the predictor set per each new vertex, or, assuming the predictors appear in long runs, run-length encoding might be better. Figure 14 shows a 3D model and the distribution of the coordinate prediction errors obtained by applying the parallelogram rule. There the geometry has been quantized to 10 bits/coordinate, hence 30 bits/vertex are required to specify the mesh geometry without any coding. The entropy of the prediction error distribution is 15.15 bits/vertex, which is a significant saving realizable through Huffman or arithmetic coding.



(**a**) (**b**)

Fig. 14. Predictive mesh geometry coding. (a) 3D triangle mesh quantized to 10 bits/coordinate. (b) Distribution of parallelogram prediction errors of the coordinates with entropy of 5.05 bits/coordinate (or 15.15 bits/vertex).

4.4 Spectral Methods

This section describes a different approach to coding mesh geometry, which bears some similarity to transform coding used for image and other signal compression.

Imagine we were to predict the geometry of each vertex to be the simple average of *all* its neighbors in the connectivity graph and code just the prediction error. Order the vertices in a vector fashion as three column vectors x, y and z, each containing one of the three vertex coordinates. The prediction error ε is also given by three column vectors $\varepsilon_x, \varepsilon_y, \varepsilon_z$. In this formulation, the decoder would have to solve the following three $n \times n$ systems of linear

equations in order to recover the mesh geometry:

$$\begin{aligned} Dx - Nx &= \varepsilon_x \\ Dy - Ny &= \varepsilon_y \\ Dz - Nz &= \varepsilon_z, \end{aligned} \tag{7}$$

where N is the *adjacency matrix* of the mesh connectivity:

$$N_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are neighbors,} \\ 0 & \text{otherwise,} \end{cases}$$

and D the diagonal matrix $D = diag(d_1, .., d_n)$ where d_i is the degree (valence) of the i-th vertex. If the system of equations is singular (which happens if the mesh is closed), we require that the solutions x, y, z have minimal norm, or add various other constraints (such as "anchor points" in the mesh which pin down some of the vertices).

Since the treatment of each of the coordinates is identical, we restrict the discussion, without loss of generality, to the x coordinate. The equation is $Lx = \varepsilon$, where L is a matrix derived from the connectivity graph: $L = D - N$. Solving this system for x would cost $\mathcal{O}(n^3)$ operations, which is not practical for large meshes. On the other hand, it might be possible to approximate x well in far fewer operations. Let Ψ be the matrix whose rows are the eigenvectors of L, i.e. $\Psi L = E\Psi$, where E is a diagonal matrix of $L's$ eigenvalues. Multiply each side of the equation by $\Psi : \Psi Lx = \Psi\varepsilon$, obtaining

$$E\Psi x = \Psi\varepsilon.$$

This means that Ψx and $\Psi\varepsilon$ are related through a simple relationship and solving for Ψx given $\Psi\varepsilon$ is very easy (as opposed to solving for x given ε). This indicates that transforming x by Ψ is useful in (and, in a sense, equivalent to) solving (7). Denote Ψx by X. Now if X contains a large number of negligible entries, these may be taken as zero, and only the non-zero entries used as the code instead of x or ε_x. The decoder, receiving the compact X, can now compute

$$x = \Psi^T X$$

which, in practice, requires $\mathcal{O}(kn)$ operations, where k is the number of non-zero coefficients of X.

This approach has its roots in *spectral graph theory*. The matrix L is the well-known graph *Laplacian* operator, and the diagonal entries of E the graph *spectrum*. The matrix Ψ represents the *spectral* basis associated with the connectivity graph, and the vector X the *spectral transform* or *spectral coefficients* of x. Similarly to Fourier theory, the "frequency" associated with each spectral basis vector is the eigenvalue of the eigenvector. L has at least one vanishing eigenvalue, corresponding to the "DC" component of the coordinates. Eigenvectors with larger eigenvalues represent basis vectors that

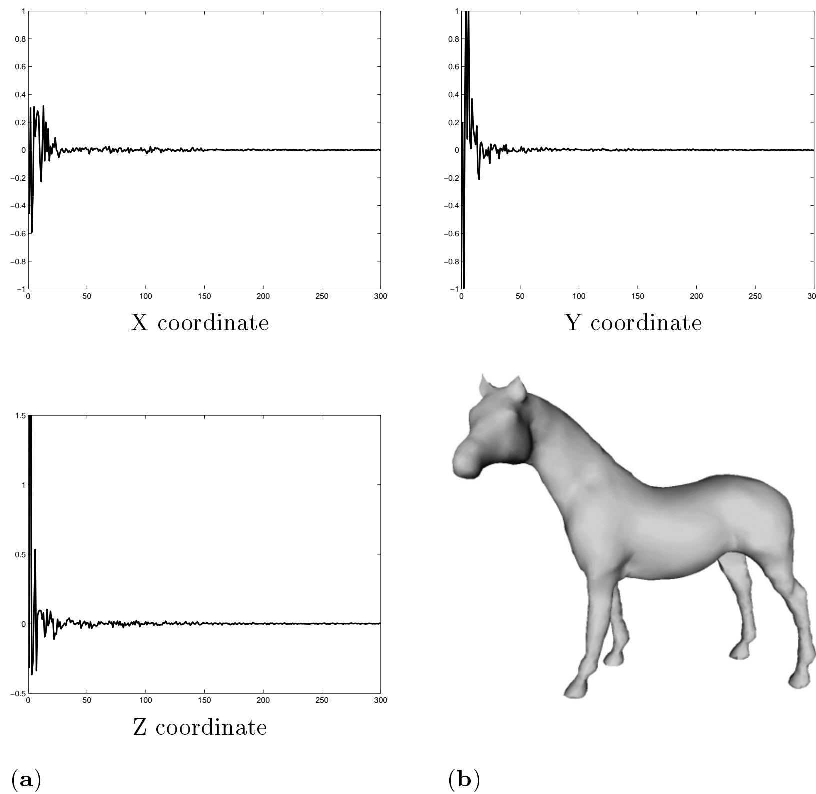


Fig. 15. Spectral mesh geometry coding. (a) The spectral coefficients of the 3 coordinate vectors of the mesh of Fig 14a. Note the significant decay. (b) Mesh reconstructed using only the first 25% of the coefficients (4.8 bits/vertex).

capture the detail in the geometry. The bulk of the spectrum of smooth geometries is concentrated on the basis vectors with the small eigenvalues. Figure 15 shows the decay of the spectral coefficients of each of the 3 coordinate vectors of the mesh of Figure 14. Entropy coding of the coefficients quantized to 14 bits/coefficient results in a code containing 15.5 bits per vertex, comparable to coding using predictive methods. However, if we are willing to compromise slightly on mesh quality, entropy coding of the first 25% of the spectral coefficients yields a code of length 4.8 bits per vertex, which may be decoded to the mesh of Figure 15 (b). More details and examples may be found in [33].

In practice, spectral mesh coding requires computation of the Laplacian eigenbasis at both the encoder and decoder, requiring $\mathcal{O}(n^3)$ computation at both ends. Beyond this, the computation becomes unstable for large n, as the conditioning of the linear system depends on the distance between adjacent eigenvalues, and this tends to zero as n increases. Thus, spectral mesh coding

in its simple form is not practical for large meshes. Since L is symmetric and sparse, it is possible to use the relatively efficient Lanczos methods (see e.g. the LASO2 package [57] to compute the first few eigenvectors). Each will cost approximately $\mathcal{O}(n^{1.4})$, which is, nonetheless, still quite expensive.

One way to reduce the complexity is to partition the mesh into a number of small submeshes of more manageable proportions. For k submeshes, this will reduce the complexity to $\mathcal{O}(k(n/k)^3) = \mathcal{O}(n^3/k^2)$, but will introduce artifacts in the submesh boundaries, or, in other words, reduce the efficiency of the code in these regions.

A particularly efficient mesh partitioning algorithm is implemented in the MeTiS software package [35], available on the Web. MeTiS partitions a given graph into parts such that the partition is *balanced*, i.e. each part contains approximately the same number of vertices, and the *edge-cut* is minimal, namely, that the total number of edges straddling the parts is as small as possible. A reasonable choice for the submesh size is approximately 500 vertices. Note that the partitioning information must either be incorporated into the mesh code, or the partitioner must be run at the decoder too, once the mesh connectivity is decoded, in order to reconstruct this information. This does not impose significant overhead, as MeTiS runs in time and space linear in the size of the mesh.

Conventional geometry coding methods first quantize the geometry before coding it in a lossless manner. This is possible because most of the prediction methods applied later in the process may be formulated to operate on integers. Since spectral methods do not involve prediction, rather the computations of real coefficients, there is no point in quantizing the geometry so early in the process. Instead, the spectral coefficients are quantized before transmission to a fixed number of bits.

It turns out that more precision is required for the spectral coefficients than would have been required on the original geometry in order to achieve similar distortion. If 10 bits/coordinate were used for the geometry, 14-16 bits/coefficient are needed for the spectral coefficients. Here too, uniform quantization is the easiest.

The spectral method described in the previous section can, in fact, be used in a progressive manner. This can be achieved by transmitting the spectral coefficients one at a time, and building up the mesh progressively as the coefficients are received by adding in the appropriate basis function weighted by the coefficient. Each such update to the model would require $\mathcal{O}(n)$ operations.

The key to effective spectral coding is to make sure the data does not contain too many high frequencies. This is the case for relatively smooth models, such as those generated by 3D scanners. However, there exist classes of 3D models which are far from smooth. For example, models used in CAD/CAM applications, describing mechanical parts and such, contain sharp edges and

corners. These manifest in high frequency content, and spectral coding is not very efficient.

Spectral methods as described above are not very practical, mainly because the eigenbasis of the connectivity mesh (or submeshes) must be computed both at the encoder and at the decoder. The encoder is less of a problem, as it can be run offline; however, decoding is usually an online process which is required to be very fast. Since the mesh connectivity is different for different meshes, there is no easy way to precompute the eigenvectors at the decoder. For this reason, attempts have been made to use a fixed connectivity structure for defining the spectral basis, and map any other mesh connectivity encountered onto this. The simplest connectivity structure for a triangle mesh is the *regular triangle* graph, where each vertex has degree six. The eigenvectors of this graph are identical to those of the regular *grid* graph (where each vertex has degree four) - the two-dimensional Fourier basis - which is used in image coding. If the size of the mesh is a power of two, spectral coefficients on this basis can even be computed using the famous Fast Fourier Transform (FFT) algorithm. The main problem, of course, is - given a mesh with some connectivity graph C - to find a mapping $m : V \to V$ of the vertices of C to the vertices of the regular triangle graph R, such that the neighborhood relationships are preserved as much as possible. The precise formulation of the problem is as follows:

$$m = argmin \sum_{(i,j) \in E(C)} |dist_R(m(i), m(j)) - dist_C(i,j)|$$

where $dist_C(i,j)$ is the edge distance between vertices i and j in the graph C, and $E(C)$ is the edge set of C.

This is, in general, a difficult problem, and we refer the reader to the paper by Karni and Gotsman [34] for details on how to find an approximate solution. It turns out that a reasonable mapping may be obtained such that the resulting spectral compression using the regular basis yields compression ratios not significantly worse than the optimal basis.

References

1. P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3D meshes. Computer Graphics Forum **20**, 2001, 480–489.
2. P. Alliez and M. Desbrun. Progressive compression for lossless transmission of triangle meshes. SIGGRAPH 2001, 195–202.
3. B. G. Baumgart. A polyhedron representation for computer vision. Proc. of the Nat. Comp. Conf., 1975, 589–596.
4. M. Botsch and L. Kobbelt. Resampling feature and blend regions in polygonal meshes for surface anti-aliasing. Computer Graphics Forum, 2001, C402–C410.
5. M. Chow. Optimized geometry compression for realtime rendering. Proc. IEEE Vis., 1997, 347–354.

6. P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. Computers & Graphics, 1998, 37–54.
7. P. Cignoni, C. Montani, and R. Scopigno. Metro: Measuring error on simplified surfaces. Computer Graphics Forum, 1998, 167–174.
8. P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini. Preserving attribute values on simplified meshes by re–sampling detail textures. The Visual Computer, 1999, 519–539.
9. J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. SIGGRAPH 1996, 119–128.
10. J. Cohen, M. Olano, and D. Manocha. Appearance preserving simplification. SIGGRAPH 1998, 115–122.
11. D. Cohen-Or, D. Levin, and O. Remez. Progressive compression of arbitrary triangle meshes. Proc. IEEE Vis., 1999, 67–72.
12. G. V. Cormack and R. N. Horspool. Algorithms for adaptive Huffman codes. Inform. Proc. Letters **18**, 1984, 159–165.
13. M. Deering. Geometry compression. Proc. SIGGRAPH 1995, 13–20.
14. M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, W. Stuetzle. Multiresolution analysis of arbitrary meshes. SIGGRAPH 1995, 173–182.
15. M. Floater. Parameterization and smooth approximation of surface triangulations. Comput. Aided Geom. Design **14**, 1997, 231–250.
16. M. S. Floater and K. Hormann. Parameterization of triangulations and unorganized points. This volume.
17. M. S. Floater, K. Hormann, and M. Reimers. Parameterization of manifold triangulations. *Approximation Theory X: Abstract and Classical Analysis*, C. K. Chui, L. L. Schumaker, and J. Stöckler (eds.), Vanderbilt University Press, Nashville, 2002, 197–209.
18. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. SIGGRAPH 1997, 209–216.
19. M. Garland and P. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. Proc. IEEE Vis., 1998, 264–270.
20. A. Gersho and R. Grey. *Vector quantization and signal compression.* Kluwer, Boston, 1992.
21. A. Gueziec, G. Taubin, F. Lazarus, and W. Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. IEEE Visualization, 1998, 383–390.
22. S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. SIGGRAPH 1998, 133–140.
23. S. Gumhold. New bounds on the encoding of planar triangulations. Tech. Rep. WSI-2000-1, Univ. of Tübingen, 2000.
24. I. Guskov, W. Sweldens, and P. Schröder. Multiresolution signal processing for meshes. SIGGRAPH 1999, 325–334.
25. I. Guskov, K. Vidimce, W. Sweldens, and P. Schröder. Normal meshes. SIGGRAPH 2000, 95–102.
26. H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. SIGGRAPH 1993, 19–26.
27. H. Hoppe. Progressive meshes. SIGGRAPH 1996, 99–108.
28. H. Hoppe. Efficient implementation of progressive meshes. Computers and Graphics **22.1**, 1998, 27–36.
29. H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. Proc. IEEE Vis., 1999, 59–66.

30. D. A. Huffman. A method for the construction of minimum-redundancy codes. Proc. Inst. Radio Eng., 1952, 1098–1101.
31. M. Isenburg and J. Snoeyink. Spirale reversi: reverse decoding of the Edge-Breaker encoding. 12th Can. Conf. on Comp. Geom., 2000, 247–256.
32. M. Isenburg and J. Snoeyink. Face Fixer: Compressing polygon meshes with properties. SIGGRAPH 2000, 263–270.
33. Z. Karni and C. Gotsman. Spectral coding of mesh geometry. SIGGRAPH 2000, 279–286.
34. Z. Karni and C. Gotsman. 3D mesh compression using fixed spectral bases. Proc. Graph. Interf., 2001, 1–8.
35. G. Karypis and V. Kumar. MeTiS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Version 4.0, Univ. of Minnesota, Dept. of Computer Science, 1998. Available at `www-users.cs.umn.edu/ karypis/metis/metis.html`.
36. D. King and J. Rossignac. Guaranteed 3.67v bit encoding of planar triangle graphs. 11th Can. Conf. on Comp. Geom., 1999, 146–149.
37. D. King, J. Rossignac, and A. Szymczak. Connectivity compression for irregular quadrilateral meshes. Tech. Rep. TR–99–36,GVU, Georgia Tech, 1999.
38. R. Klein, G. Liebich, and W. Straßer. Mesh reduction with error control. Proc. IEEE Vis., 1996, 311–318.
39. L. Kobbelt, S. Campagna, and H.-P. Seidel. A general framework for mesh decimation. Proc. Graph. Interf., 1998, 43–50.
40. L. Kobbelt, S. Campagna, J. Vorsatz, and H.-P. Seidel. Interactive multiresolution modeling on arbitrary meshes. SIGGRAPH 1998, 105–114.
41. L. Kobbelt, J. Vorsatz, U. Labsik, and H.-P. Seidel. Shrink wrapping approach to remeshing polygonal surfaces. Computer Graphics Forum, 1999, 119–130.
42. L. Kobbelt, J. Vorsatz, and H.-P. Seidel. Multiresolution hierarchies on unstructured triangle meshes. Computational Geometry **14**, 1999, 5–24.
43. B. Kronrod and C. Gotsman. Efficient coding of non-triangular meshes. Proc. 8-th Pacific Graphics, 2000, 235–242.
44. A. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. SIGGRAPH 1998, 95–104.
45. E. Lee and H. Ko. Vertex data compression for triangular meshes. Proc. Pacific Graphics, 2000, 225–234.
46. P. Lindstrom and G. Turk. Fast and memory efficient polygonal simplification. Proc. IEEE Vis., 1998, 279–286.
47. P. Lindstrom and G. Turk. Evaluation of memoryless simplification. Proc. IEEE Trans. on Vis. and Comp. Graph., 1999, 98–115.
48. P. Lindstrom. Out–Of–Core simplification of large polygonal models. SIGGRAPH 2000, 259–262.
49. P. Lindstrom and G. Turk. Image-Driven Simplification. ACM Trans. on Graph., 2000, 204–241.
50. M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.
51. R. Pajarola and J. Rossignac. Compressed Progressive Meshes. IEEE Trans. on Vis. and Comp. Graph. **6.1**, 2000, 79–93.
52. J. Rossignac and P. Borrel. Multi–resolution 3D approximation for rendering complex scenes. 2nd Conf. on Geom. Model. in Comp. Graph., 1993, 453–465.
53. J. Rossignac. EdgeBreaker: Connectivity compression for triangle meshes. IEEE Trans. on Vis. and Comp. Graphics, 1999, 47–61.

54. J. Rossignac and D. Cardoze. *Matchmaker: Manifold Breps for non-manifold r-sets*. Tech. Rep. GIT-GVU-99-03 GVU Center, Georgia Inst. of Tech., 1998.
55. W. Schroeder, J. Zarge, and W. Lorensen. Decimation of triangle meshes. SIGGRAPH 1992, 65–70.
56. W. Schroeder. A topology modifying progressive decimation algorithm. Proc. IEEE Vis., 1997, 205–212.
57. D. S. Scott. LASO2 Documentation. Technical Report, Computer Science Dept., University of Texas at Austin, 1980.
58. A. Szymczak, D. King, and J. Rossignac. An EdgeBreaker-based efficient compression scheme for regular meshes. Preprint, 2000.
59. G. Taubin and J. Rossignac. Geometric compression through topological surgery. ACM Trans. on Graphics **17.2**, 1998, 84–115.
60. G. Taubin, A. Gueziec, W. Horn, and F. Lazarus. Progressive forest split compression. SIGGRAPH 1998, 123–132.
61. C. Touma and C. Gotsman. Triangle mesh compression. Proc. Graph. Interf., 1998, 26–34.
62. G. Turan. Succinct representations of graphs. Discrete Appl. Math. **8**, 1984, 289–294.
63. G. Turk. Re-tiling polygonal surfaces. SIGGRAPH 1992, 55–64.
64. W. Tutte. A census of planar triangulations. Can. Journ. of Math. **14**, 1962, 21–38.
65. I. H. Witten, R. M. Neal and J. G. Cleary. Arithmetic coding for data compression. Comm. of the ACM **30**(6), 1987, 520–540.

Exercises

Exercise 1. **Connectivity Coding.**

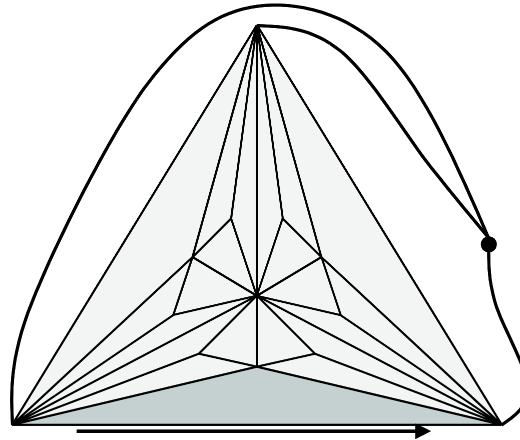


Fig. 16. Sample mesh for connectivity coding.

(a) Run manually the face-based, edge-based and vertex-based connectivity coding algorithms described in Subsection 3.2 on the mesh in Figure 16 and generate the code strings. For your convenience fill the next encoded triangle and draw the new gate location after each operation. In case of face based and edge based coding initialize the cut-border to the mesh border and start with the marked gate location. For the valence-based coding use the added dummy vertex, start the code with the two vertices of the marked gate edge and perform an add operation to encode the third vertex of the marked triangle first. In all three schemes make sure to cycle the gate around the pivot vertex pointed to by the gate. After a split operation proceed with the gate on the right side when the gate is at the bottom.

(b) Compute the entropy of the code strings generated in (a). Which scheme performs best?

(c) Why is the entropy not a good performance measure for small meshes?

Exercise 2. **Lower Bound for Valence-Based Coding.**

The valence-based encoding scheme proposed by Alliez and Desbrun [1] avoids many of the split operations. In this exercise we follow Alliez and Desbrun's ideas and compute the maximum entropy of the vertex valences of a triangular mesh with spherical topology for a fixed number of vertices v. This is a lower bound for valence-based coding with a context free backend.

In case of a negligible number of split operations and an arithmetic coding backend one can achieve worst case results close to Tutte's lower bound.

To maximize the entropy of a code string it should contain as many different symbols as possible and the frequency of each symbol should be about the same. In our case the alphabet consists of the different valences $3, 4, 5, 6, \ldots$. Independent of the total number of symbols in the code string we want to choose the probabilities $p_i, i = 3, 4, \ldots$ in a way that maximizes

$$E = -v \sum_{i \geq 3} p_i \log_2 p_i$$

under the constraint, and that the probabilities sum to one: $\sum_{i \geq 3} p_i = 1$. Besides this constraint we need to make sure that the valence distribution is valid and obeys the Euler equation $v + f = e + 2$. A closed triangular mesh satisfies $2e = 3f$, as each edge is incident to two faces and each face to three edges.

(a) Show that the average valence of a triangular mesh of spherical topology is $\sum_{i \geq 3} p_i \cdot i = 6 - \frac{12}{v} \approx 6$. This is the second constraint under which we want to maximize the entropy.

(b) Show, that $p_i = \alpha \cdot e^{-\beta \cdot i}$ maximizes $\sum_{i \geq 3} p_i \log_2 \frac{1}{p_i}$ under the constraints $\sum_{i \geq 3} p_i = 1$ and $\sum_{i \geq 3} p_i \cdot i = 6$. *Hint:* Use the method of Lagrange Multipliers.

(c) With the equalities

$$\sum_{i \geq 3} e^{-\beta \cdot i} = \frac{e^{-2\beta}}{e^{\beta} - 1}$$

and

$$\sum_{i \geq 3} i \cdot e^{-\beta \cdot i} = \frac{e^{-2\beta} \left(-3e^{2\beta} + 5e^{\beta} - 2\right)}{\left(e^{\beta} - 1\right)\left(-e^{2\beta} + 2e^{\beta} - 1\right)}$$

show that $\alpha = \frac{16}{27}$ and $\beta = \log \frac{4}{3}$ yield a unique maximum and that the entropy in this case is $E = -\log_2 \alpha + \frac{6\beta}{\log 2} = \log_2 \frac{256}{27} \approx 3.245$. This is exactly the lower bound derived by Tutte by enumeration of all possible triangulations.