Mesh Compression

Dissertation
der Fakultät für Informatik
der Eberhard-Karls-Universität zu Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Stefan Gumhold
aus Tübingen

Tübingen
2000

Tag der mündlichen Qualifikation: 19.Juli 2000
Dekan: Prof. Dr. Klaus-Jörn Lange
1. Berichterstatter: Prof. Dr.-Ing. Wolfgang Straßer
2. Berichterstatter: Prof. Jarek Rossignac

Zusammenfassung

Die Kompression von Netzen ist eine weitgefächerte Forschungsrichtung mit Anwendungen in den verschiedensten Bereichen, wie zum Beispiel im Bereich der Handhabung extrem großer Modelle, beim Austausch von dreidimensionalem Inhalt über das Internet, im elektronischen Handel, als anpassungsfähige Repräsentation für Volumendatensätze usw. In dieser Arbeit wird das Verfahren der Cut-Border Machine beschrieben. Die Cut-Border Machine kodiert Netze, indem ein Teilbereich durch das Netz wächst (region growing). Kodiert wird die Art und Weise, wie neue Netzelemente dem wachsenden Teilbereich einverleibt werden. Das Verfahren der Cut-Border Machine kann sowohl auf Dreiecksnetze als auch auf Tetraedernetze angewendet werden. Trotz der einfachen Struktur des Verfahrens kann eine sehr hohe Kompressionsrate erzielt werden. Im Falle von Tetraedernetzen erreicht die Cut-Border Machine die beste Kompressionsrate von allen bekannten Verfahren. Die einfache Struktur der Cut-Border Machine ermöglicht einerseits die Realisierung direkt in Hardware und ist auch als Implementierung in Software extrem schnell. Auf der anderen Seite erlaubt die Einfachheit eine theoretische Analyse des Algorithmus. Gezeigt werden konnte, dass für ebene Triangulierungen eine leicht modifizierte Version der Cut-Border Machine lineare Laufzeiten in der Zahl der Knoten erzielt und dass die komprimierte Darstellung nur linearen Speicherbedarf benötigt, d.h. nicht mehr als fünf Bits pro Knoten.

Neben der detaillierten Beschreibung der Cut-Border Machine mit mehreren Verbesserungen und Optimierungen, enthält die Arbeit eine Einführung zu Netzen und geeigneten Datenstrukturen und entwickelt mehrere Kodierungsverfahren, die im Bereich der Netzkompression Anwendung finden. Eine breite Übersicht verwandter Arbeiten gibt Einblick in des Forschungsgebiet. Weiterhin wird die Effizienz mehrerer in der Literatur beschriebener Verfahren verbessert. Insbesondere konnte die algorithmisch erzielte obere Schranke für die Kodierung ebener Triangulierungen bis auf zehn Prozent oberhalb der theoretischen unteren Schranke verbessert werden. Das ist bis jetzt das beste Resultat, das erzielt werden konnte.

Abstract

Mesh Compression is a broad research area with applications in a lot of different areas, such as the handling of very large models, the exchange of three dimensional content over the internet, electronic commerce, the flexible representation of volumetric data and so on. In this thesis the mesh compression method of the Cut-Border Machine is described. The Cut-Border Machine encodes meshes by growing a region through the mesh and encoding the way, in which the mesh elements are incorporated into the growing region. The Cut-Border Machine can be applied to triangular and tetrahedral meshes. Although the method is not too complicated, it achieves very good compression rates. In the tetrahedral case the Cut-Border Machine performs best among all known methods. The simple nature of the Cut-Border Machine allows on the one hand for a hardware implementation and performs also as software implementation extremely well. On the other hand the simplicity allows for a theoretical analysis of the Cut-Border Machine. It could be shown, that for planar triangulations a slightly modified version of the Cut-Border Machine runs in linear time in the number of vertices and that the compressed representation only consumes linear storage space, i.e. no more than five bits per vertex.

Besides the detailed description of the Cut-Border Machine with several improvements and optimizations, the thesis gives an introduction to meshes and appropriate data structures, develops several coding techniques useful for mesh compression and gives a broad overview of related work. Furthermore the author improves the encoding efficiency of several other compression techniques. In particular could the algorithmically achieved upper bound for the encoding of planar triangulations be improved to ten percent above the theoretical limit, what is the best known result up to now.

Contents

List of Figures

List of Tables

Preface

Mesh compression is young research area. It has been inspired by the 3D graphics hardware acceleration community. The foundation has been laid by the work of Michael Deering [Dee95] on triangle mesh compression. His goal was to reduce the data traffic between the main memory of a 3D visualization system and the graphics hardware accelerator. Before his innovative work the only way to reduce this data traffic had been the use of triangle strips, which allow to specify one triangle with only slightly more than the vertex data of one vertex, but still the vertex data of each vertex has to be transmitted twice. Deering could overcome this problem and reduce the vertex data repetition rate to only 6%. Other papers [AHMS94, BG96, ESV96] in the area also concentrated on the acceleration of the rendering of three dimensional surfaces. The work of Rossignac and Taubin [TR96] established triangle mesh compression as independent research area. This work was inspired by the MPEG forum. At the same time Hoppe published his work on the progressive encoding of triangle meshes [Hop96]. The so called Progressive Mesh representation allows to transmit a triangle meshes progressively over a connection with low bandwidth. In the year 1998 three very efficient methods had been developed to a publishable ripeness. The best compression rates achieves the method by Touma and Gotsam [TG98]. The fastest but still very efficient method – called the Cut-Border Machine – had been proposed by the author of this thesis together with Straßer [GS98]. Rossignac [Ros98] reported a very similar method but could bridge triangle mesh compression to the theory of graph encoding. He showed that his encoding consumes in the case of planar triangulations no more than four bits per vertex for the encoding of the neighbor relations. The four bits per vertex encoding was the best known at the time and could also be achieved in the graph encoding community by Chuang, Garg, He and Kao [CGHK98]. The theoretical lower limit of 3.245 bits per vertex to encoding a planar triangulation had already been established in 1962 by Tutte [Tut62], who had counted all different planar triangulations for a given number of vertices. The algorithmicly achieved upper bound of four bits per vertex could later on be improved by King and Rossignac [KR99] to 3.67 bits and finally by the author [Gum00] to 3.552 bits per vertex. The generalization of triangle mesh compression to polygonal mesh compression has only been started this year [IS00, KG00b].

The generalization from surface meshes to volume meshes has been performed basically at the same time by Szymczak and Rossignac [SR99] with their Grow & Fold method and by the author together with Guthe and Straßer [GGS99] with the generaliza-

tion of the Cut-Border Machine. There are only two more tetrahedral mesh compression approach up to now. The generalization of Hoppe's the progressive meshes to the progressive simplicial complexes [PH97], which has not been implemented for tetrahedral meshes yet and the Implant Sprays of Pajarola [PRS99]. The Implant Sprays is a progressive compression method for tetrahedral meshes with compression rates better than the Grow & Fold method. The author could significantly improve the compression rates of the Implant Sprays method. The single resolution method of the author's Cut-Border Machine still performs twice as good.

The thesis is structured in three parts. The first part gives a broad introduction to meshes, discusses related work and develops several encoding methods, which are often used in mesh compression. The second part describes the contributions of the author to triangle mesh compression and finally the last part constitutes the tetrahedral mesh compression work. Concluding remarks and future research directions are stated independently for triangle and tetrahedral mesh compression at the end of the corresponding parts.

Part I

Introduction

The first part not only familiarizes the reader with the area of mesh compression but also gives a complete overview of the research area and of related areas. Chapter 1 introduces the basic concepts about meshes and a data structure appropriate for our purposes. The problem treated in this thesis is discussed in chapter 2. A broad view of solutions presented in literature is compiled. In the last chapter 3 of the first part different coding techniques are introduced and applied to several coding problems, which arise very often in mesh compression.

Chapter 1

Basics on Meshes

This chapter introduces the concept of a mesh. Firstly, in section 1.1, the term mesh is defined based on a brief introduction of geometrical models. The next section 1.2 describes and motivates the usage of meshes. The performance of most compression methods can be estimated quite accurately with the help of relations between the numbers of different mesh elements and the average counts of mesh element neighbors. The most important relations are introduced in section 1.4. At the end of this chapter we introduce a simple and efficient data structure for triangle and tetrahedral meshes in section 1.5.

1.1 Mesh Concept

In the first part of this section we expose ideas on geometrical models similar to [Man88] with slightly different terminology. We first define a sensible notion of a solid in section 1.1.1, then we go over to express solids by their surface (section 1.1.2). In order to handle arbitrary solids we split the surface of the solid into smaller parts which are topologically equivalent to polygons consisting of edges and vertices. The incidence of the vertices, edges and polygons define the connectivity (section 1.1.3). Finally, we define the geometry of the mesh in section 1.1.4.

1.1.1 Point-Set Models

Most applications handle three-dimensional solid objects in the Euclidean space $\mathbb{R}^3$. In the most general form a solid is defined as follows.

Definition 1.1 (solid) *A* solid *is a bounded, closed subset of* $\mathbb{R}^3$.

The class of objects captured by this definition is by far too large. Also the restriction to rigid objects, that makes objects invariant under rigid transformations (i.e. rotations and translations) does not restrict the class of objects far enough.

Definition 1.2 (rigid object)
A rigid object *is an equivalence class of point sets of* $\mathbb{R}^3$ *spanned by the following relation* $\circ$*:* $\forall A, B \subset \mathbb{R}^3 : A \circ B \leftrightarrow A = R(B) \wedge R$ *is a rigid transformation.*

Our intuitive notion of solid does not allow isolated points and line segments. This requirement can be easily captured with the help of the closure operation $c : A \subset \mathbb{R}^3 \mapsto$ closure of A and the interior operation i:

Definition 1.3 (regular, r-set)

- *a set* $A \subset \mathbb{R}^3$ *is* regular*, iff it satisfies:* $A = c(i(A))$
- *a bounded regular set is termed an* r-set

By restricting ourselves to r-sets we ensure that all isolated points and curves are torn away and that the skin of the object is complete. This will also ensure that the solid can be reduced to its surface, which is used in computer graphics most often to render solids.

1.1.2 Surface-Based Models

The surface-based characterization of solids looks at the boundary of a solid object and composes it into a collection of faces, which are glued together such that they form a complete, closed skin around the object.

A surface can be seen as a two-dimensional subset of $\mathbb{R}^3$. Each surface point is surrounded by a ”two-dimensional” region of surface points. The inherent two-dimensionality of a surface means that we can study its properties through a two-dimensional model. The notion of a 2-manifold gives a more abstract notion of a surface.

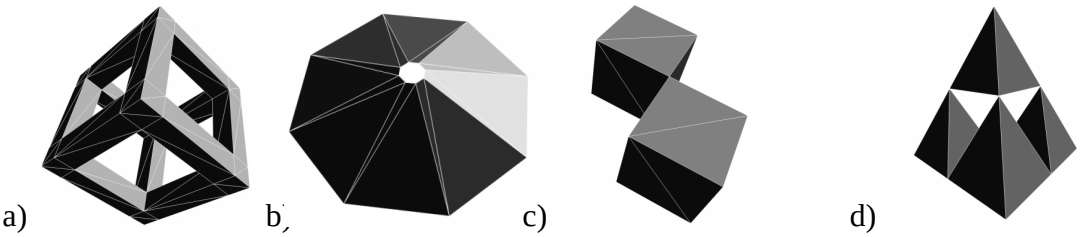


Figure 1.1: a) manifold surface mesh, b) manifold with border, c) non manifold because of edge with more than two incident faces, d) non manifold because of vertices with more than one connected face loop.

Definition 1.4 (2-manifold)
A 2-manifold *is a topological space, where every point has a neighborhood topologically equivalent to an open disk of* $\mathbb{R}^2$.

In figure 1.1 a) a manifold surface mesh is shown. In computer graphics it is quite common to handle also surfaces with boundaries as for example the lamp shade in figure 1.1 b). Thus one also allows points with a neighborhood topologically equivalent to a half disk and calls these surfaces *manifold with boundary*. But there are also quite common surface models, that are not manifold as the other two examples in figure 1.1 show. In c) the two cubes touch at a common edge, which contains points with neighborhood not equivalent to a disk nor a half disk and in d) the tetrahedra touch at points with non manifold neighborhood.

1.1.3 Connectivity

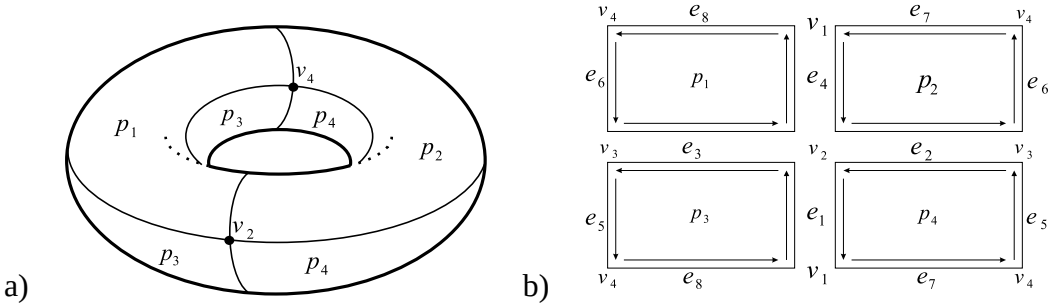


Figure 1.2: a) torus subdivided into four patches, b) planar embedding of patches with identified edges and vertices.

In order to analyse and represent complex surfaces, which represent solids, we subdivide the surfaces into polygonal patches enclosed by edges and vertices. Figure 1.2 a) shows the subdivision of the surface of a torus into four patches $p_1, \ldots, p_4$. Each patch can be embedded into the Euclidean plane resulting in four planar polygons as shown in figure 1.2 b). The embedding allows to map the Euclidean topology to the interior of each patch on the surface. The collection of polygons can represent the same topology as the surface if the edges and vertices of adjacent patches are identified. In figure 1.2 b) identified edges and vertices are labelled with the same specifier. The topology of the points on two identified edges is defined as follows. The points on the edges are parameterised over the interval $[0, 1]$, where zero corresponds to the vertex with smaller index and one to the vertex with larger index. The points on the identified edges with the same parameter value are identified and the neighborhood of the unified point is composed of the unions of half-disks with the same diameter in both adjacent patches. In this way the identified edges are treated as one edge. The topology around vertices is defined similarly. Here the neighborhood is composed of disks put together from several pies with the same radius of all incident patches.

We are now in the position to split the surface into two constitutes the connectivity and the geometry. The connectivity $\mathcal{C}$ defines the polygons, edges and vertices and their incidence relation. The geometry $\mathcal{G}$ on the other hand defines the mappings from the

polygons, edges and vertices to patches, possibly bent edges and vertices in the three dimensional Euclidean space. The pair $\mathcal{M} = (\mathcal{C}, \mathcal{G})$ defines a polygonal mesh and allows to represent solids via their surface. In this section we discuss the connectivity, which defines the incidence among polygons edges and vertices and which is independent of the geometric realisation.

Definition 1.5 (polygonal connectivity)

- *the polygonal connectivity is a quadruple $(V, E, F, \mathcal{I})$ of the set of vertices V, the set of edges E, the set of faces F and the incidence relation $\mathcal{I}$, such that*
- *each edge is incident to its two end vertices*
- *each face is incident to an ordered closed loop of edges $(e_1, e_2, \ldots, e_n)$ with $e_i \in E$, such that e_1 is incident to v_1 and v_2, $\forall i = 2 \ldots n-1 : e_i$ is incident to v_i and v_{i+1} and e_n is incident to v_n and v_1*
- *in the notation of the previous item the face is also incident to the vertices $v_1, \ldots, v_n$*
- *the incidence relation is reflexive*

The collection of all vertices, all edges and all faces are called the *mesh elements*. We next define the relation *adjacent*, which is defined on pairs of mesh elements of the same type.

Definition 1.6 (adjacent)

- *two faces are* adjacent, *iff there exists an edge incident to both of them*
- *two edges are* adjacent, *iff there exists a vertex incident to both*
- *two vertices are* adjacent, *iff there exists an edge incident to both*

Up to now we defined only terms for very local properties among the mesh elements. Now we move on to global properties.

Definition 1.7 (edge-connected) *A polygonal connectivity is* edge-connected, *iff each two faces are connected by a path of faces such that two successive faces in the path are adjacent.*

As the connectivity is used to define the topology of the mesh and the represented surface, one can define the following criterion for the surface to be manifold.

Definition 1.8 (potentially manifold) *A polygonal connectivity is* potentially manifold, *iff*

1. *each edge is incident to exactly two face*

2. *the non empty set of faces around each vertex form a closed cycle*

Definition 1.9 (potentially manifold with border) *A polygonal connectivity is* potentially manifold with border*, iff*

1. *each edge is incident to one or two faces*

2. *the non empty set of faces around each vertex forms an open or closed cycle*

A surface defined by a mesh is manifold, if the connectivity is potentially manifold and no patch has a self-intersection and the intersection of two different patches is either empty or equal to the identified edges and vertices. All the non-manifold meshes in figure 1.1 are not potentially manifold.

In order to determine whether a potentially manifold mesh can be embedded without self-intersections in three dimensional Euclidean space, the orientability plays the crucial role. The orientation of each face has been defined with the connectivity in the order of the edges and vertices. From the face orientation each incident edge inherits an orientation as illustrated in figure 1.2 b). With the inherit orientation of the edges, the orientability of a mesh can be defined.

Definition 1.10 (orientable) *A polygonal connectivity is* orientable *if the face orientations can be chosen in a way that for each two adjacent faces the common incident edges inherit different orientations from the different faces.*

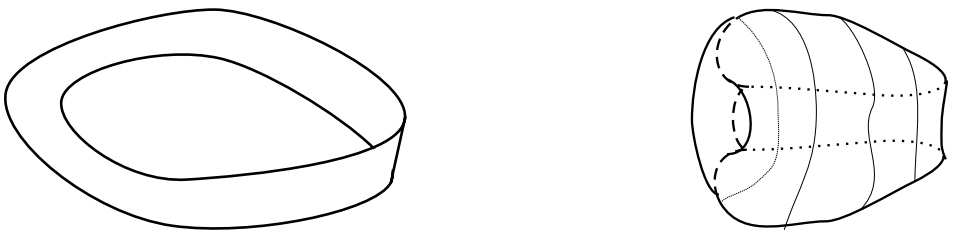


Figure 1.3: Two non orientable objects. On the left the well known Möbius strip and on the right the Klein bottle.

The orientation of a face in a polygonal mesh can be used to define the outside of a mesh or to calculate the surface normal. It is also important during the navigation through the mesh, which is essential for most connectivity compression techniques. The problem with non orientable meshes is that one cannot choose the orientation of the faces consistently. Thus surface normals can not be calculated consistently and no inside or outside relation makes sense. Further more it complicates the navigation in the mesh as one must know during the traversal between two adjacent faces, whether the orientation of the face changes. Figure 1.3 shows two examples of non orientable meshes. One can easily check their non orientability as one can move on the surface from one point always staying on the same side of the surface in a loop and arrive back at the same point but on the other side of the surface.

So far we restricted the definitions of a mesh to the two dimensional case. We want to describe also volumetric meshes and in particular tetrahedral meshes. The vertices are zero dimensional mesh elements, the edges one dimensional and the faces two dimensional. The embedding of a three dimensional mesh element is a subset of the Euclidean space with non zero volume. For this we define the topological polyhedron

Definition 1.11 (topological polyhedron)
A topological polyhedron *is a potentially manifold and edge-connected polygonal connectivity.*

Based on the definition of a topological polyhedron, we can define the polyhedral connectivity as a quintuple $(V, E, F, P, \mathcal{I})$ of vertices, edges, faces and polyhedra. Each polyhedron is incident to a set of oriented faces, that form a topological polyhedron. The local and global relations *adjacent, face-connected, manifold* and *manifold with border* are direct generalizations of the corresponding attributes in a polygonal connectivity. We do not want to define all these terms in detail, but want to mention that the roll of the face orientation is taken by the outside relation of the topological polyhedron. Please notice that in a pure polyhedral connectivity the border is always a closed polygonal connectivity and therefore the number of faces incident on an edge always larger than two. Polyhedral meshes embedded self-intersection free in the three dimensional Euclidean space are always orientable as polygonal meshes in the plane.

1.1.4 Geometry

It is now time to add some geometry to the connectivity. We want to describe this procedure only for the typical case of polygonal and polyhedral geometry in Euclidean space. Similarly, meshes with curved edges and surfaces could be defined.

Definition 1.12 (Euclidean polygonal/polyhedral geometry)
The Euclidean geometry $\mathcal{G}$ of a polygonal/polyhedral mesh $\mathcal{M} = (\mathcal{C}, \mathcal{G})$ is a mapping from the mesh elements in $\mathcal{C}$ to $\mathbb{R}^3$ with the following properties:

- *a vertex is mapped to a point in $\mathbb{R}^3$*
- *an edge is mapped to the line segment connecting the points of its incident vertices*
- *a face is mapped to the inside of the polygon formed by the line segments of the incident edges*
- *a topological polyhedron is mapped to the sub-volume of $\mathbb{R}^3$ enclosed by its incident faces*

Here arises a problem that also often arises in practice. In $\mathbb{R}^3$ the edges of a face often do not lay in a plane. Therefore the geometric representation of a face is not defined properly and also a sound two dimensional parameterization of the polygon is not easily

defined. In practice this is often ignored and the polygon is split into triangles for which a unique plane is given in Euclidean space.

Often further attributes like physical properties of the described surface/volume, the surface color, the surface normal or a parameterization of the surface are necessary. These attributes are typically stored as constant values at the vertices and interpolated along the edges and faces. Or higher order interpolation schemes are exploited by further attribute values given at the edges and or faces. But again the interpolation over arbitrary polygons or polyhedra is difficult. Therefore in practice one often simplifies the problem to the most simple types of mesh elements, the simplices. The k-dimensional simplex or for short k-simplex is formed by the convex hull of $k+1$ points of the Euclidean space. A 0-simplex is just a point, a 1-simplex a line segment, a 2-simplex is a triangle and the 3-simplex forms a tetrahedron. For simplices the linear and quadratic interpolation of vertex and edge attributes are simply defined via the barycentric coordinates.

In some applications the handling of mixed dimensional meshes is necessary. In this case singleton vertices, singleton one dimensional and in case of polyhedral meshes also singleton two dimensional meshes are allowed. As the handling of mixed dimensional polygonal/polyhedral meshes becomes very complication, one often gives up polygons and polyhedra and restricts oneself to *simplicial complexes*, which allow for singleton vertices and edges and non-manifold mesh elements. A simplicial complex is defined as follows.

Definition 1.13 (simplicial complex) *A k dimensional* simplicial complex *is a $(k+1)$-tuple $(S_0, \ldots, S_k)$, where S_i contains all i-simplices of the complex. The simplices fulfill the condition, that the intersection of two i-simplices is either empty or equal to a simplex of lower dimension.*

As a simplex and therefore a simplicial complex is only a geometric description, we have to define the connectivity of a simplicial complex, which is easily done by specifying the incidence relation among the simplices of different dimensions. A i-simplex is incident to a j-simplex with $i < j$ if the i-simplex forms a sub-simplex of the j-simplex.

1.2 Application Areas

There are several important application areas for meshes. One of the most important ones is in Finite Element simulations. Here a surface/volume is split into a polygonal/polyhedral mesh and attributed with physical quantities of the underlying material. The equations of motion are written in terms of the mesh elements and equation solvers are used to find solutions for different starting conditions. The flexible structure of a mesh allows to model arbitrary geometries. The Finite Element Method (FEM) has been successfully applied to simulate all types of materials including fluids and cloth. Therefore the FEM is widely used in all industrial branches. One common task in FEM is the generation of appropriate meshes from boundary data only. The mesh elements of

the produced meshes must fulfill certain quality criteria. More information on this topic can be found in [Geo91].

The second main application of meshes is the boundary representation of objects. Here polygonal and triangular meshes come into operation. The meshes are typically attributed with the surface normal, surface material information and a parameterization together with some textures specifying fine variations of the surface color, surface normal or of the surface offset in direction of the surface normal. Simple triangle meshes are very common because of their hardware accelerated rendering with all the mentioned attributes[1]. The boundary representation of objects is used in computer aided design, in virtual worlds, in the game industry, for terrain modeling and gains more and more importance in electronic commerce. New objects are often scanned with 3D scanners producing very fine and large meshes, which demand for compression.

Scientific visualization is also a broad application area for meshes. Not only the finite element meshes are directly visualized, but new surface meshes are generated to represent and visualize isosurfaces in volume datasets.

Finally, meshes are also used as algorithmic tool for spatial hashing and to build hierarchical structures for point location queries.

1.3 Triangular and Tetrahedral Meshes

A lot of algorithms that deal with meshes are restricted to the simple case of triangular or tetrahedral meshes. This is easily justified from the much simpler handling of triangles and tetrahedra in terms of intersection calculation, attribute interpolation, line up of physical equations, rendering and so on. But in real world data a lot of meshes are not triangular or tetrahedral.

For this reason a whole area of research has tackled the problem of efficiently subdivide a polygon, into triangles. A simple but efficient method searches for a intersection free diagonal in the polygon. This must always exist as the following lemma shows (the lemma is cited from [BE92]).

Lemma 1.14 *Every polygon with more than three sides has a diagonal.*

Proof: Let b be the vertex with minimum x-coordinate and ab and bc be its two incident edges. If ac is not cut by the polygon, then ac is a diagonal. Otherwise there must be at least one polygon vertex inside the triangle abc. Let d be the vertex inside abc furthest from the line through a and c. Now the segment bd cannot be cut by the polygon, since any edge intersecting bd must have one endpoint further from line ac. □

The triangulation algorithm cuts the polygon at the diagonal into two parts and recursively proceeds with the remaining two polygons until only triangles are left over.

[1] for material properties see [Pho75, BW86, DWS+88, Cla89, KB89], for color textures see [Hec83, Wil83, Cro84, Gla86, SKS96, H99], for surface normal textures see [Bli78, EJRW96, PAC97, Kug98] and for surface offset [GH99, GVSS00, LMH00]

The expensive operation is the intersection test of the diagonal segment and the polygon. In 1991 Chazelle [Cha91] came up with a solution to the polygon triangulation problem that can be computed in linear time in the number of edges in the polygon. Two other papers [KKT90, Sei91] give simpler solutions with only slightly worse running time. Held [Hel98] developed a very robust implementation, that always succeeds also on polygons with self-intersections and degenerated edges.

In the case of tetrahedral meshes the problem is more severe as there exist polyhedra without any tetrahedrization. Therefore new vertices – so called Steiner points – must be inserted to the polyhedron. A Solution that minimize the number of inserted Steiner points is described by Sapidis [SP91]. Another solution with application to collision detection is presented by Held [HKM96].

1.4 Basic Relations

For a polygonal connectivity $\mathcal{C} = (V, E, F, \mathcal{I})$ with $v = |V|$ vertices, $e = |E|$ edges and $f = |F|$ faces the following Euler equation holds

$$v - e + f = \chi \overset{(closed,manifold)}{=} 2(s - g), \tag{1.1}$$

where χ is the Euler characteristic. For a closed manifold connectivity the Euler characteristic depends on the number of edge-connected components s and the genus g of the mesh. The genus of a closed surface is the number of handles of the described solid. The surface of a cup has for example one handle, i.e. one hole in the circumscribed solid. A sphere has no handle, a torus has one handle and the surface of a solid eight has two handles. The mesh in figure 1.1 a) has genus five. The genus of a mesh can be derived from the number h_1 of closed curves that can be drawn on the mesh without dividing it into two or more separate pieces. The genus is just h_1 divided by two. A torus can for example be cut with two closed curves into a rectangle, which is still connected and no more closed curve can be drawn onto it without cutting it apart. Thus h_1 would be two and the genus is one.

As we will primarily deal with triangular meshes we can also consider the special characteristic of triangular meshes to derive a much simpler equation. For this we enumerate in a closed manifold triangle mesh all incidences between edges and triangles. In terms of edges there are $2e$ incidences as each edge is incident to two faces. In terms of triangles there are $3f$ incidences resulting in

$$2e = 3f.$$

Substitution of this relation in the Euler equation yields

$$2v - f \overset{(tgl,closed,manifold)}{=} 2(s - g), \tag{1.2}$$

and for edge-connected triangle meshes with genus one, we get

$$f \overset{(g=1,tgl,closed,manifold)}{=} 2v. \tag{1.3}$$

Although the number of conditions to this relation is large, it is a good approximate statement for meshes, that describe the surface of a solid. The Euler characteristic χ is typically small compared to the number of vertices and triangles. For completeness we want to incorporated the number of border edges b into the equations. As each border edge has only one incident face, the number of incidences between edges and face edge in terms of the edges must be corrected to $2e - b$ resulting in

$$2v - f - b \overset{(tgl,manifold)}{=} 2(s - g). \tag{1.4}$$

Often the compression rate of an encoding method depends on the average number of vertices adjacent to a vertex or on the average number of faces incident upon a vertex. We introduce the notation

Definition 1.15 (element-element order) *Let $\alpha, \beta \in \{v, e, f\}$, then we define the average α-β order (i.e. the vertex-vertex, vertex-edge, edge-vertex,...) as*

$$\bar{o}_{\alpha\to\beta} \overset{def}{=} \frac{\textit{total number of } \alpha\textit{-}\beta \textit{ incidences/adjacencies}}{\alpha}$$

Thus in a closed triangle mesh the average number of faces incident to a vertex – the average vertex-face order – is

$$\bar{o}_{v\to f} \overset{tgl}{=} 3f/v \overset{(g=1,closed,manifold)}{=} 6, \tag{1.5}$$

what is again a quite general applicable statement for triangle meshes with low genus, low non manifold spot count and low border fraction. Relations for tetrahedral meshes are derived in section 9.2.

1.5 Data Structure

A polygonal mesh has vertices, edges and faces as mesh elements. For navigation in the mesh the incidence and the adjacency relations are important. Figure 1.4 a) gathers the possible relations. The arrows between the sets of different mesh elements represent the incidence relations, whereas the self-pointing arrows illustrate the adjacency relations. In order to answer all possible incidence and adjacency questions, one needs to represent only a subset of all relations. This subset must connect all sets of equal and different mesh elements, i.e. there need not only be a path from each mesh element set to each other, but there also must be a path from each mesh element set back to the set in order to answer adjacency questions. In figure 1.4 a) this means that one can eliminate arrows as long as this condition is fulfilled. For example the relations $V \to E \to F \to V$ would suffice. It is often possible to represent some of the relations only partially by one or to representatives. For example, if the relations $F \to F \to V$ are known in a manifold mesh with border, the relation $V \to F$ can be stored with one face per vertex, as the remaining faces of each vertex can be determined through the adjacency relation of the

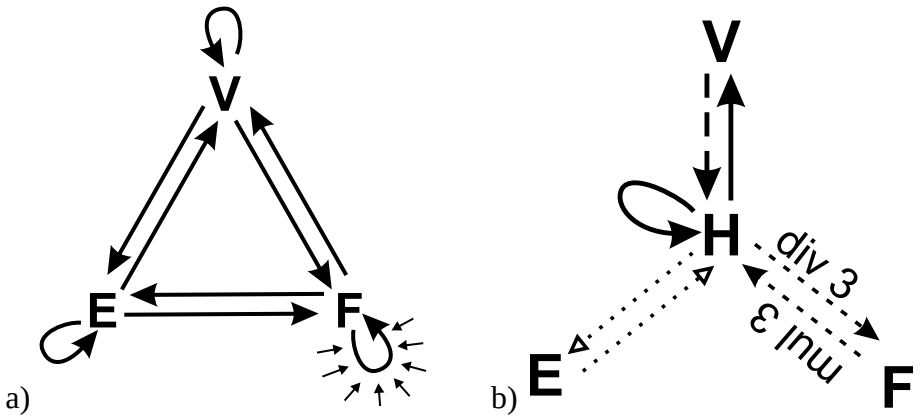


Figure 1.4: Relations between the mesh elements, a) polygonal mesh, b) polygonal mesh with halfedges

faces. It is clear that the enumeration of all faces incident on a vertex is more expensive than if the relation would have been stored explicitly. On the other hand the update of the explicit representation is more expensive. Ni [NB94] analyzes the different possible data structures.

Most single resolution mesh compression techniques are region growing methods. Here the most important relation is the face adjacency, i.e. one wants to know the adjacent faces of a currently processed face. For coordinate compression the incidence relation $F \rightarrow V$ is the second most important relation. Only in case of non manifold mesh compression the inverse relation $V \rightarrow F$ is necessary. The edges are seldom explicitly represented. The halfedge data structure [Man88] turned out to be very suitable for polygonal meshes [Ket98]. Other edge based data structures such as the winged edge data structure [Bau75] are discussed in [Wei85].

1.5.1 Halfedge Data Structure

For the halfedge data structure a new mesh element type is introduced – the *halfedge*. A halfedge represents one incidence between an edge and a face. In figure 1.5 a) the halfedges are illustrated with solid arrows. For each halfedge five pointers *next, prev, adjacent, face* and *vertex* are stored. The first three pointers represent halfedge adjacencies, where *prev* and *next* correspond to edge adjacent halfedges and *adjacent* to the face adjacent halfedge. For each halfedge the incident vertex and the incident face are stored. In figure 1.4 b) the mesh element relations in a mesh with halfedges are illustrated. The central role of the halfedges becomes clear. The relation $H \rightarrow H$ is represented by *next, prev* and *adjacent*, $H \rightarrow V$ by *vertex* and $H \rightarrow F$ by *face*. For each face one halfedge and if needed one halfedge per vertex is stored to represent the relations $F \rightarrow H$ and $V \rightarrow H$.

If the number of edges in a face does not change frequently, the halfedges can be aligned face after face in the order they appear in the faces. After the halfedges of each

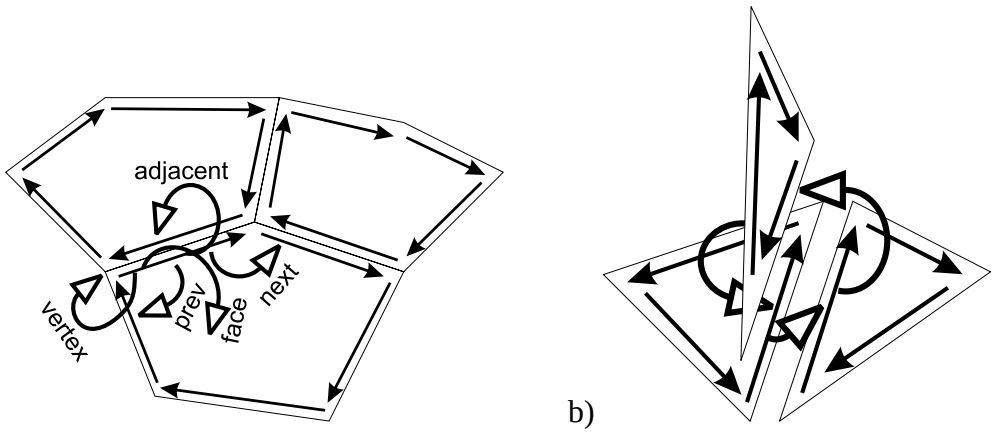


Figure 1.5: a) illustration of the halfedge data structure and b) the linkage of the *adjacent* pointer at non manifold edges

face a dummy halfedge is introduced storing the number of halfedges in the previous face. In this way the *prev* and *next* pointers can be saved.

In the special case of a pure triangle mesh the dummy halfedges are not needed as all faces have exactly three incident edges. Furthermore as described in [CKS98] the relations $F \rightarrow H$ and $H \rightarrow F$ can easily be computed by an integer division with three and a multiplication with three (see also figure 1.4 b)). Thus the typical halfedge data structure for triangle meshes only consumes the two pointers *adjacent* and *vertex* for each halfedge and if needed one pointer per vertex for the relation $V \rightarrow F$. As there are $3f$ halfedges in a triangle mesh, this sums up to

$$\mathcal{S}_{\mathrm{tgl,halfedge}} = 6f\ [+v] \text{ pointers.}$$

For non orientable meshes one additional flag is needed per *adjacent* pointer to specify whether the orientation changes when moving to the adjacent face. Also non manifold edges can be handled by the halfedge data structure. The *adjacent* pointers of the halfedges around a non manifold edge are linked in a closed cycle as shown in figure 1.5 b). If needed the relation $V \rightarrow F$ needs to be stored explicitly. An ordered set data structure suggest itself to store for each vertex the incident faces.

1.5.2 Computing Halfedge Adjacencies

The relation $F \rightarrow V$ is sufficient to define the connectivity as long as there are no singleton vertices or edges. Therefore most polygonal meshes are stored as a list of vertex indices with a invalid index of -1 as separator between successive faces. From this information one can instantly generate the halfedges in a way that the *prev* and *next* pointers are implicit in the halfedge order and one can also instantly derive the relations $F \leftrightarrow H \rightarrow V$. The only pointer that consumes some computing power is the *adjacent* pointer. All halfedges incident upon a certain edge must be linked together. For this the

genus5	2388
quader	1768
crocodile	1247
porsche	1973
helicopter	1871
monster	1502
bunny	1467
jaw	1434
average	1706

Table 1.1: The speed for calculating the *adjacent* pointer in a halfedge data structure in thousand triangles per second determined on a Pentium II 300.

vertex indices of the halfedges incident edge are available. The halfedges of one edge can be found by hashing over the set of two vertex representing the edge. A standard hash map is normally much slower than a simple approach with linked lists. For each vertex a linked list of hash entries is kept. A hash entry contains the larger vertex index of the hashed edge, the index of the halfedge and a pointer to the next linked list element. A hash element is searched by first determining the smaller vertex index of the edge and then linearly searching through the linked list of hash entries attached to the smaller vertex index. This algorithm consumes for each edge a hash entry and for each vertex a pointer to the first hash entry, altogether $3e + v$ pointers. If the maximum vertex order is limited by a constant, the running time is linear in the number of triangles. Table 1.1 gives the hashing speed for different triangle meshes in triangles per second.

1.5.3 Handling Mesh Attributes

The most important attribute of a mesh is the geometric representation of the vertices. In nearly all applications the geometry of a vertex is stored as a two, three or four dimensional point in the Euclidean space. The number format depends on the application but floating point values are most commonly used. The geometric representation of the edges, faces and polyhedra is normally not stored explicitly but is derived from the points of the vertices. The edges are mapped to the line segments between their end points, the faces to the polygon interiors described by the line segments of their edges and the polyhedra to the volume circumscribed by the polygons of their faces.

Further attributes such as surface normals, surface colors and texture coordinates, i.e. the surface parameterization, are often available at the vertices. The vertex attributes can simply be stored by extending the dimension of the point in order to include the normal, color and texture coordinates. It is also no problem to treat attributes given at the faces as the face indices are known. Interestingly is it neither a problem to handle attributes at the corner of faces – also called *corner attributes*. This is often necessary, if the represented surface has non differentiable creases. At vertices on the creases the surface

normal is not continuous and different normals might have to be stored for each face the vertex is incident to. But this is no problem as there is exactly one halfedge per vertex-face incidence and we can store the corner attributes within the halfedges.

The only problematic attributes in a halfedge data structure are edge attributes, which occur seldom in praxis. To handle edge attributes one adds the relation $H \rightarrow E$ to the halfedge data structure with one edge index per halfedge. This increases the connectivity storage space by one third and decreases performance slightly as the edge indices of adjacent halfedges must be kept consistent.

1.5.4 Tetrahedral Mesh Data Structure

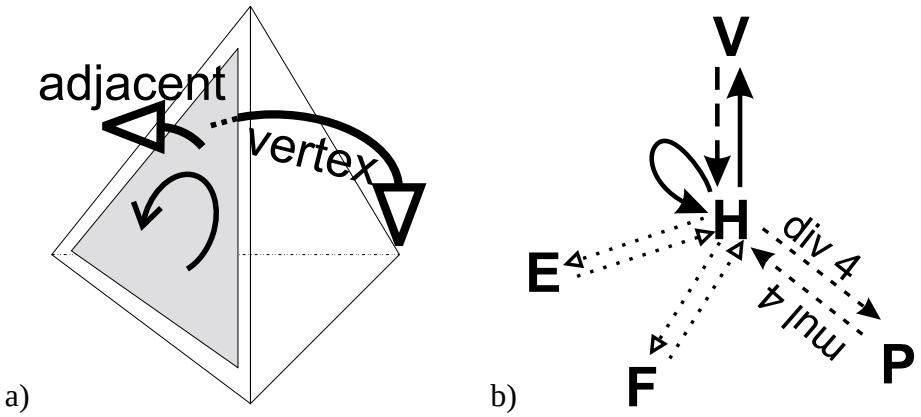


Figure 1.6: a)illustration of the pointers in a halfface data structure b) relations among the mesh elements

As mentioned in [SR99], the optimized data structure for triangle meshes can be easily generalized to tetrahedral meshes. We name this data structure the *halfface data structure*. Figure 1.6 a) shows a grey shaded halfface in a tetrahedron and the two explicitly represented pointers *adjacent* and *vertex*. The *adjacent* pointer points to the halfface incident to the same face in the adjacent tetrahedron. *vertex* points to the opposite vertex of the halfface in the tetrahedron incident to the halfface.

Figure 1.6 b) diagrams the relations between the mesh elements in a halfface data structure. H is the set of halffaces and P the set of polyhedra and in our case the set of tetrahedra. As in the case of the triangular halfedge data structure the halffaces are arranged tetrahedron by tetrahedron, such that the relations $H \leftrightarrow P$ can be easily calculated from the indices with *mul* 4 and *div* 4. Each halfface h has three neighboring halffaces in the same tetrahedron. The indices of them can easily be calculated: $\mathit{next}_1 = (h+1)\%4$, $\mathit{next}_2 = (h+2)\%4$ and $\mathit{next}_3 = (h+3)\%4$, where $\%$ is the modulo operator. With the next_i pointers, the relation $H \rightarrow V$ can be implemented through the *vertex* pointers of the adjacent halffaces. If t is the number of tetrahedra, the storage space for the specialized halfface data structure is

$$\mathcal{S}_{\mathrm{tetra,halfface}} = 8t\ [+v] \text{ pointers},$$

where the v pointers are only necessary, if the relation $V \rightarrow H \rightarrow P$ is used.

In the case of three dimensional polyhedral meshes, the problem of non manifold halffaces cannot arise, if self-intersections are not allowed. Neither can the meshes be non orientable.

Chapter 2

Related Work

In many of the applications described in section 1.2 the used meshes become either very large or come in a large collection. The output of 3D scanners are often meshes with a million triangles. In CAD modeling the meshes can become even larger and in terrain modeling arbitrarily huge meshes can be easily produced from satellite data. In the game industry, in electronic commerce and in 3D enhanced presentations a large data base with a lot of different models has to be handled. In all applications the need to shrink down the size of the meshes is obvious.

There are primarily three different approaches for reducing the size of a mesh: compression, simplification and remeshing. In the compression approach, as adapted in this thesis and discussed in section 2.1, the goal is to find an encoding of a mesh, that is as short as possible. The mesh connectivity is encoded without loss of any information. Most geometry compression schemes are based on a lossy quantization step somewhere in the geometry compression pipeline. Compression is especially useful for the efficient encoding of databases with a lot of small models, but also as encoding tool for simplification and remeshing approaches, which typically end up with a small mesh, that also has to be encoded efficiently. Large and regular models often contain more information than necessary or maybe even redundant information. Then it cannot be justified anymore that the connectivity of the mesh should be preserved and mesh simplification should be utilized as discussed in section 2.2. The most commonly adapted idea in mesh simplification is to simplify the mesh through a sequence of local operations that eliminate a small number of adjacent mesh elements. An important increment is the measurement of the approximation error of the simplified mesh. The minimization of the mesh element count for a given maximal approximation error is hard to solve optimal and good heuristics consume large computation times, too. The simplification and compression approach were recently re-unified with the idea of representing the mesh in terms of the inverse simplification process in compressed form. This is discussed in section 2.3. An also very interesting idea is remeshing. Here a second very regular mesh is generated that approximates the original mesh. The regularity of the approximation allows to store the new mesh much more efficiently. See section 2.4 for a brief discussion.

2.1 Mesh Compression

2.1.1 Acceleration of Rendering

As mentioned in the preface, mesh compression evolved from the aim to accelerate the rendering of triangle meshes. In the first subsection we therefore discuss representations for triangle meshes, that are used for the efficient transmission to a graphics accelerator. 3D-hardware support is primarily based on the rendering of triangles. Each triangle is specified by three vertices, where each vertex contains three coordinates, possibly the surface normal, material attributes and/or texture coordinates. The coordinates and normals are specified with floating point values, such that a vertex may contain data of up to 36 bytes[1]. Thus the transmission of a vertex is expensive and the simple approach of specifying each triangle by the data of its three vertices is wasteful as for an average triangle mesh each vertex must be transmitted six times (compare equation 1.5).

The introduction of triangle strips helped to save unnecessary transmission of vertices. Two successive triangles in a triangle strip join an edge. Therefore, from the second triangle on, the vertices of the previous triangle can be combined with only one new vertex to form the next triangle. As with each triangle at least one vertex is transmitted and as an average triangle mesh has twice as many triangles as vertices (see equation 1.3, the maximal gain is that each vertex has to be transmitted only about two times. Two kinds of triangle strips are commonly used – the *sequential* and the *generalized triangle strips*. In generalized triangle strips an additional bit is sent with each vertex to specify to which of the two free edges of the previous triangle the new vertex is attached. *Sequential strips* even drop this bit and impose the condition that the triangles are attached in an alternating fashion. OpenGL [NDW97] which developed to the commonly used standard graphics library allowed [Inc91] generalized triangle strips in earlier versions, but the current version is restricted to sequential strips. Therefore, the demands on the stripping algorithms increased. Non of the existing algorithms reaches the optimum that each vertex is transmitted only twice. The algorithm of Evans et al. [ESV96] produces strips such that each vertex is transmitted about 2.5 times. Xiang et al. [XHM99] describe a faster method with similar strip lengths. As nowadays graphics hardware accelerators still only support sequential triangle strips and triangle fans, Isenburg [Ise00] devised a method to encode a triangle mesh together with its stripification. There is no overhead for the encoding of the triangle strips. The additional structural information rather improves the encoding of the mesh connectivity.

Arkin et al. [AHMS94] examined the problem of testing whether a triangulation can be covered with one triangle strip. For generalized triangle strips this problem is NP-complete, but for sequential strips there exists a simple linear time algorithm. But no results or algorithms were given to cover a mesh with several strips.

To break the limit of sending each vertex at least twice, Deering [Dee95] suggests the use of an on-board vertex buffer of sixteen vertices. With this approach, which he

[1] where we assumed four bytes per floating point value and one byte per color component

calls *generalized mesh*, in theory only six percent of the vertices have to be transmitted twice. For connectivity encoding the generalized mesh consumes $15 + 0.25\mathrm{lb}v$ bits per vertex. Deering also proposes methods to quantize vertex locations, vertex colors and vertex normals. Chow [Cho97] shows how to build generalized meshes and refines the quantization of vertex locations to adapt the local resolution of the mesh. Bar-Yehuda et al. [BG96] examined different sized vertex buffers. They prove that a triangle mesh with n vertices can be rendered optimal, i.e. each vertex is transmitted only once, if a buffer for $12.72\sqrt{n}$ vertices is provided. They also show that this upper bound is tight and no algorithm can work with less than $1.649\sqrt{n}$ buffered vertices.

The *Cut-Border Machine* [GS98] as described in chapter 5 is based also on a very simple algorithmic scheme and is therefore suitable for hardware implementation. The software implementation allows to decompress one and a half million of triangles per second on a Pentium II with 300 MHz. No vertex data is repeated due to the use of a vertex buffer. It is not quite clear how large this buffer might grow, but experiments showed that always significantly less than $12.72\sqrt{n}$ vertices had to be buffered. By defining a fixed traverse order our approach minimizes the number of indices needed to reference vertices in the buffer, which results in an additional speed up for rendering. If these indices are Huffman-encoded, in the average only 3 bits per vertex are needed for references. A similar but even simpler approach, that does repeat some vertices, was proposed by Mitra [McC98].

2.1.2 Single Resolution Mesh Compression

Denny and Sohler [DS97] showed that for sufficiently large triangle meshes the connectivity can be encoded in a permutation of its vertices alone. This would make all work on connectivity coding useless. But there is a catch in it. The connectivity can be exploited to encode the vertex locations more efficiently. With a simple delta coding technique the connectivity information improves vertex locations encoding by about the amount of the storage space consumed by a permutation of the vertices, which grows with $O(v\mathrm{lb}v)$. The connectivity itself only consumes $O(v)$ bits, what justifies its encoding.

Single resolution mesh compression methods are important to encode large data bases of small objects, base meshes of progressive representations or for fast transmission of meshes over the internet. In practice a lot of meshes are non manifold, but typically only at a few spots. On the other hand most mesh compression methods are restricted to manifold meshes with border. As commonly used remedy to this grievance the non manifold meshes are cut apart at non manifold mesh elements by duplicating them. In order to avoid the replication of the attributes for the duplicated mesh elements, Gueziec et al. [GTLH98] describe a method to efficiently represent the reverse of the cutting process, such that the non manifold connectivity can be reconstructed before the geometry is mapped to the connectivity.

The first single resolution mesh compression method, that focuses on maximum compression has been the *Topological Surgery* by Taubin and Rossignac [TR98]. The mesh is first cut along a vertex spanning tree into a simple polygon. The simple polygon

is encoded as triangle spanning tree and with the help of the encoded vertex spanning tree the simple polygon can be glued together again. The encoding of both trees consumes approximately four bits per vertex and can be bound to consume no more than six bits per vertex for simple meshes.

For the encoding of the vertex locations Taubin proposes a predictive delta-coding scheme. In predictive delta-coding as first action the vertex coordinates are quantized according to the bounding box of the mesh to an appropriate number of bits. The number of bits should be chosen such that the edge of minimal length can be represented appropriately in the quantization grid. Sometimes the user has to specify the number of significant bits. To avoid the propagation of the quantization error one has to take care that for vertex location prediction the compression algorithm has only access to the quantized locations as also the decompression algorithm does.

After the quantization step the vertex locations are typically encoded in an order corresponding to the order in which the mesh elements are traversed during connectivity encoding. The first vertices are encoded without any delta-coding. Afterwards, each time a new vertex is inserted to the so far compressed mesh, the location l_{new} of the new vertex is predicted from the quantized vertex locations, of the so far compressed vertices. The difference Δl between the predicted location l_{pred} and the actual location l_{new} is encoded with any coding technique. Adaptive arithmetic coding (see section 3.2) normally produces the best results.

The different vertex location encoding approaches differ mainly in the way they calculate l_{pred} from the already compressed vertices. Taubin [TR98] uses a probably large but fixed number K of ancestors in the vertex spanning tree and predicts the new vertex location as a linear combination

$$l_{\mathrm{pred}} \stackrel{\mathrm{def}}{=} \sum_{i=1}^{K} \lambda_i l_i$$

The coefficients λ_i are chosen by least square minimization of the produced delta vectors. For twelve bit quantization the vertex locations of a typical CAD model compresses to about 14 bits per vertex.

The Cut-Border Machine [GS98], the *Edgebreaker* [Ros98] and the triangle mesh compression method of Touma and Gotsman [TG98] are region growing methods. The connectivity is traversed in a breadth-first like order starting with an initial triangle or with the mesh border. Encoded is how new operations are incorporated into the growing region. The Cut-Border Machine and its improvement [Gum99] are discussed in chapter 5 and 6. The Edgebreaker is described and improved for maximum compression in chapter 7. Thus we only briefly explain the method of Touma. In this method the addition of only two kinds of triangles to the growing region are explicitly encoded. Triangles that introduce new vertices and much more seldom triangles that split the border of the current region into two parts. This is possible because the vertex-triangle order of each newly introduced vertex is encoded and the number of triangles incident upon each vertex in the growing region is counted. If this number is only one less than the

vertex-face order, the triangle fan around the vertex can be closed without the need of an operation symbol. As most triangle meshes have a lot of vertices with six incident faces, a run-length encoding scheme of the degrees achieves connectivity compression to an average of only 2 bits per vertex. Touma also proposes a simple and very efficient vertex predictor. Each triangle that introduces a new vertex v_{new} is adjacent to a triangle $t = (v_1, v_2, v_3)$ of the growing region. The triangle t is extended to a parallelogram and the fourth vertex is the predicted location of the new vertex. If we assume that (v_1, v_2) form the interior edge of the parallelogram, the predicted locations can be computed very efficiently from

$$l_{\text{new}} \stackrel{\text{def}}{=} l_1 + l_2 - l_3.$$

In order to incorporate knowledge about the surface curvature, a crease angle at the edge (v_1, v_2) is estimated from the so far known crease angles of the other two edges incident to the triangle t. The vertex locations can be encoded to about 12 bits per vertex if quantized to ten bits.

Two further triangle mesh compression techniques were proposed by Isenburg [IS99a] and Bajaj [BPZ99a]. Li et al. [LK98a] were the first to come up with a connectivity encoding scheme for polygonal meshes. The scheme encodes the dual graph in a manner very similar to the Cut-Border Machine. Recently, the Edgebreaker has been generalized by Kronrod and Gotsman [KG00b] and in a different way by Isenburg [IS00] to polygonal meshes. Isenburg shows that the knowledge about the planarity and the convexity of the polygons improves the vertex location encoding. He also describes a very efficient coding scheme for a partitioning of the polygonal mesh into patches. Finally, we want to mention the work of Karni and Gotsman [KG00a] on the encoding of vertex locations. They used spectral methods on triangular meshes, that allow for very high vertex location compression rates and progressive transmission.

2.1.3 Graph Encoding Theory Related Results

The Edgebreaker of Rossignac [Ros98] bridged the endeavors in the area of mesh compression with the coding theory of planar triangulations. Rossignac can show that the Edgebreaker encoding does not consume more than four bits per vertex. The same results have been achieved by Itai et al. [IR82] and recently by Chuang et al. [CGHK98]. Already in 1962 Tutte [Tut62] had enumerated all planar triangulations and it turned out that at least 3.245 bits per vertex are needed to encoded an arbitrary planar triangulation. More bounds on the encoding of triangle meshes are discussed in chapter 4. The upper bound of four bits per vertex could be improved by King [KR99] to 3.667 bits per vertex and later on by the author [Gum00] to 3.552 bits per vertex. The latter result will be expatiated in chapter 7. In the same work the author could show an upper bound of no more than five bits per vertex for a slightly modified version of the Cut-Border Machine as described in section 6.4.

2.1.4 Tetrahedral Mesh Compression

In the area of tetrahedral mesh compression two single resolution methods have been proposed so far. The Grow & Fold method by Szymczak and Rossignac [SR99] is a generalization of the topological surgery method and is described in section 12.1. It allows to encode tetrahedral connectivity with slightly more than seven bits per tetrahedron. The Cut-Border Machine has also been generalized to the tetrahedral case [GGS99] and is described in detail in chapter 10. It only consumes two bits per tetrahedron for connectivity coding. Up to know the Cut-Border Machine is the only encoding technique for tetrahedral meshes that has compression schemes for vertex data and vertex attributes as described in chapter 11.

2.2 Mesh Simplification

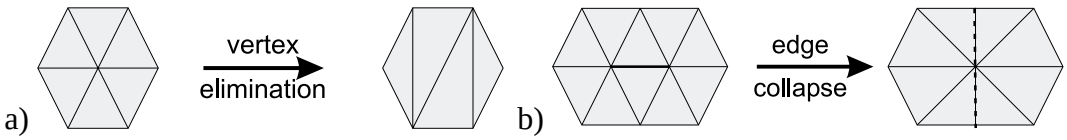


Figure 2.1: The most often used primitive simplification operations in mesh simplification: a) vertex elimination, b) edge collapse

A lot of mesh simplification algorithms are based on the successive application of local simplification operations such as the vertex elimination [SZL92, KLS96, CCMS97] and the edge collapse [HDD+93, Hop96, RR96, GH97, Gué99] as shown in figure 2.1. Gieng [GHJ+98] uses the more seldom triangle collapse operation. The successive application of simplification operations yields a sequence of meshes from the original mesh $\mathcal{M}_n$ to the coarsest mesh $\mathcal{M}_0$.

$$\mathcal{M}_n \overset{\text{edge collapse}}{\rightarrow} \mathcal{M}_{n-1} \overset{\text{edge collapse}}{\rightarrow} \cdots \mathcal{M}_1 \overset{\text{edge collapse}}{\rightarrow} \mathcal{M}_0$$

To ensure the quality of the simplified mesh an error measurement is defined. For all possible local simplification operations the produced error is determined by virtually performing the operation. The operations are entered into a priority queue sorted according to the produced error. As long as the error of the current mesh is smaller than the allowed error, the simplification operation, that produces the smallest error, is extracted from the queue and applied. New possible simplification operations will be possible in the surrounding of the performed operation. These will be virtually performed to determine the error produced by them and then the new operations are entered into the priority queue.

The methods differ primarily in the used error measurement, which significantly influences the performance of the simplification process. For a discussion of the different methods see [PS97].

The simplification approach has also been generalized to the tetrahedral case [ZCK97, CMPS97, SG98, THJW98]. All methods are based on edge collapse as other simplification operations are very difficult to implement for tetrahedral meshes.

2.3 Progressive Mesh Compression

The idea of progressive mesh compression is to encode a mesh through the inverse of the simplification process. The mesh in the coarsest resolution is encoded followed by a sequence of refinement operations, which are the inverse of the simplification operations. The inverse of the vertex elimination (compare figure 2.1 a)) is the vertex insertion and the inverse of the edge collapse operation (figure 2.1 b)) the vertex split operation. In the notation of above the decompression of a progressively compressed meshes can be described as a sequence of meshes

$$\mathcal{M}_0 \overset{\text{vertex split}}{\rightarrow} \mathcal{M}_1 \overset{\text{vertex split}}{\rightarrow} \cdots \mathcal{M}_{n-1} \overset{\text{vertex split}}{\rightarrow} \mathcal{M}_n.$$

This representation is ideal to stream meshes, as the mesh can be viewed in a coarse resolution before all vertex split operations have been transmitted. This feature is also known as *progressive transmission*.

Hoppe [Hop96] was the first to come up with a progressive representation – the *progressive mesh* representation –, which is based on vertex split operations. If viewed from right lo left, figure 2.1 b) shows a vertex split operation. The dashed edges on the right are split into new triangles. For update of the connectivity during a vertex split operation it is sufficient to specify the two edges, which have to be split. Thus the vertex split can be specified by the index of the split vertex and an index into an enumeration of all possible edge pairs from the edges incident to the split vertex. As there are about six edges incident upon a vertex (compare equation 1.5), the vertex split can be encoded with $\mathrm{lb}(5 \cdot 6) + \mathrm{lb}v$ bits, where v is the current number of vertices. The vertex data is compressed with a local prediction scheme and delta coding to about 21 bits per vertex. Some improvements over the progressive mesh representation have been proposed by Li [LK98b].

The major idea to get rid of the $\mathrm{lb}v$ bits per vertex was presented by Taubin in 1998 with the progressive forest split (PFS) representation [TGHL98]. Taubin generalizes the vertex split operation to the forest split operation. In the coarse mesh a forest of edges is specified. The refinement step splits all the edges at the same time. In this way the mesh is generated from a coarse simplified version by successively doubling the number of vertices. The forest split operation is encoded with one bit per edge specifying the split edges plus the encoding of a simple polygon that fills the cuts. In this way no vertex indices have to be encoded. The connectivity of a triangle mesh consumes in the PFS representation about eight bits per vertex. Taubin also proposes a derivative of an edge collapse simplification algorithm to build up the PFS representation. For the encoding of the vertices about 25 bits per vertex are consumed.

Let us extract the basic method – the *level split* method –, that allows to avoid the $\mathrm{lb}v$ bits per reversed simplification step: gather as many split operations as possible and encode their anchor mesh elements with flags and the additional local information, that is necessary to specify the refinement operations, in the same way as if no level split method would be used. Cohen-Or et al. [COLR99] applied the level split method to mesh simplification through vertex elimination. Figure 2.1 a) shows a vertex elimination operation. The inverse is the vertex insertion operation, which is fully determined by the set of triangles used for re-tiling the hole arising after the vertex elimination. Thus Cohen-Or proposes a coloring scheme to define a dense set of vertex insertion operations on the coarse mesh. Two schemes are presented a four coloring scheme and a two coloring scheme. The later is based on a special type of re-tiling after the vertex elimination. Very good compression rates are achieved. The connectivity is encoded with about six bits per vertex. The compression results for the vertex locations are not directly specified in the paper, but one can derive them from other measurements. With about 16 bits per vertex this method achieves very good compression rates for a twelve bit quantization.

Pajarola [PR00] applies the level split method directly to progressive meshes. The resulting method is called *Compressed Progressive Meshes*. He specifies the split vertices of a dense set of vertex split operations level by level with a flag. For each split vertex the two edges that have to be split (compare Progressive Meshes) are encoded as an index into an enumeration of all possible pairs of edges. The connectivity compresses to 7.2 bits per vertex. With the help of arithmetic coding this result could probably improved to six bits per vertex as with the method of Cohen-Or. For vertex location encoding Pajarola inverts a butterfly subdivision scheme. With ten bit quantization the vertex locations can be encoded with about 16 bits per vertex.

The only progressive compression method, which is not based on simple decimation operations is the one by Bajaj et al. [BPZ99b]. Here the triangle mesh is simplified by re-triangulation of long closed triangle strips. The method can also handle non manifold meshes and consumes about 10 bits per vertex for connectivity coding and 30 bits for vertex location coding.

Also in the area of tetrahedral mesh compression the level split method has been applied by Pajarola et al. [PRS99]. The method is called *Implant Sprays* and is described in section 12.2. Finally, the most general progressive method – the *Progressive Simplicial Complex* – has been proposed by Popovic et al. [PH97]. It can handle arbitrary simplicial complexes and is described in section 12.3.

2.4 Remeshing

Remeshing an arbitrary triangle mesh is a very difficult task. A simple approach as presented in [Tur92] does not always produce high quality results as sharp edges are not reproduced and neither any kind of error can be guaranteed. A more promising approach is the use of subdivision surfaces [CC78, DS78]. The idea is to approximate the original

mesh by subdividing a coarse mesh of the same topology, which serves as parameterization. During the subdivision process the inserted vertices are displaced such that the original mesh is approximated better. Eck et al. [EDD+95] were the first to follow that path. The major problem is to find a parameterization of the mesh, what has been only recently solved by Lee et al. [LSS+98]. Based on this work two new very similar representations for subdivision surfaces [GVSS00, LMH00] have been proposed. Both representation allow to encode surface detail as a displacement field along the surface normal. The difficult task is to convert an arbitrary surface into this representation, as the original mesh must be describable by a coarse base surface and an offset in direction of the surface normal. Both approaches use a simplification procedure based on edge collapse with some heuristics to avoid that the fine surface intersects the normal of the coarse surface several times. Then follows an optimization step that allows to improve the parameterization. Finally, the displacements along the normal are calculated by casting a dense set of rays from the subdivision surface in the direction of the surface normal. The resulting representation is extremely space efficient as only the coarse base mesh needs to be stored plus a one dimensional offset to the surface and not anymore the three coordinates for each mesh vertex.

Chapter 3

Coding Techniques

Most mesh compression schemes translate the uncompressed mesh into a sequence of symbols and indices. The efficient encoding of these two basic components is a chapter for itself – this chapter.

The first two sections 3.1 and 3.2 describe methods for the encoding of a sequence of symbols. All explanations assume the existence of an alphabet $\mathcal{A}$ over a set of symbols $\{\sigma_1, \ldots, \sigma_a\}$. Let $s = s_1, \ldots, s_n$ be the string that has to be encoded. The coding methods exploit the frequencies $\{\nu_1, \ldots, \nu_a\}$ of the symbols in the alphabet. If $\#_{\sigma_i}(s)$ is the number of symbols σ_i in the string s, the frequencies are given by

$$\nu_i \stackrel{\text{def}}{=} \#_{\sigma_i}(s)/n.$$

If no further knowledge about the symbols in the string is given, the optimal encoding, that can be achieved, consumes at least as many bits as the *binary entropy*, which is defined as

$$\epsilon(s) \stackrel{\text{def}}{=} -\sum_{i=1}^{a} \#_{\sigma_i}(s) \cdot \mathrm{lb}\nu_i. \tag{3.1}$$

The entropy limit is achieved up to a fractional of a percent by arithmetic coding as described in section 3.2. This section also describes several applications of arithmetic coding, that arise regularly in mesh compression.

Section 3.3 describes variable length coding schemes, which can be used to encode indices.

3.1 Huffman Coding

In 1952 Huffman [Huf52] devised a very efficient method to encode a string of symbols. Let us assume the settings given in the introduction to this chapter with the alphabet $\mathcal{A}$, the symbols σ_i and their frequencies ν_i. For each of the symbols a binary code is created, such that no code is a prefix of another code. The latter condition is essential for the unique parsing of the codes in linear time. The prefix condition is automatically

fulfilled, if the codes are generated from a binary tree with the symbols at the leaves. The path from the root of the tree to each leaf defines the bit code. Anytime one chooses the left child of a node the code is extended by a zero, in the right child by a one. It remains to specify how the tree is built. Huffman showed that the following method assigns optimal[1] prefix codes for each symbol.

- form the leaf nodes as the symbol-frequency pairs (σ_i, ν_i) and insert them into a priority queue sorted by increasing frequency
- as long as the queue contains more than one node, extract the first two nodes $(., \nu_i), (., \nu_j)$ with smallest frequencies, form a new node $(., \nu_i + \nu_j)$ with children $(., \nu_i)$ and $(., \nu_j)$ and insert the node into the queue.
- the last node in the queue is the binary tree describing the optimal prefix codes.

The same method of building the binary tree works also with the symbol counts $\#_{\sigma_i}(s)$.

Huffman coding in its simplest variant has several disadvantages. It makes all mesh compression techniques to two pass algorithms. In the first pass the string s is generated and the symbols are counted. Then the prefix codes are computed and encoded and finally in the second pass the symbols are encoded with the prefix codes. The second problem is that the prefix codes need to be encoded, what consumes additional storage space and can decrease encoding efficiency, if the number of different symbols a is not negligible compared to the total number of symbols n in the encoded string.

In case of the Cut-Border Machine for triangle meshes the frequencies of the different symbols do not vary strongly. Therefore fixed prefix codes can be generated by averaging the symbol frequencies over a representative set of triangle meshes.

Cormack [CH84] describes algorithms for the adaptive generation of Huffman codes. The basic idea is to start coding with some initial symbol counts, for example one or in case of the Cut-Border Machine the standard frequencies multiplied by some initial symbol count are used. During compression and decompression not only the symbol counts are updated after each encoding/decoding of a symbol, but also the binary tree defining the prefix codes. The update decreases coding performance slightly. The major disadvantage of Huffman coding is that it does not achieve the minimal coding costs given by the binary entropy.

3.2 Arithmetic Coding

The exciting fact about arithmetic coding is, that it approximately achieves the binary entropy and is therefore a near optimal encoding scheme in terms of space consumption. The idea goes back to the text book [Abr63] (see pages 61-62). The initial idea has been evolved to a coding scheme in [Pas76, Ris76] and finally became a practical method

[1] a set of optimal prefix codes achieves the minimum storage space consumption for the encoding of the string s

with the publication of [RL79]. A nice overview of arithmetic coding can be found in [WNC87]. We will briefly introduce the ideas behind arithmetic coding and describe afterwards several applications, which are important in the area of mesh compression.

In arithmetic coding the encoded symbols are not directly translated into bit codes. Each symbol is encoded into a sub-interval of the unit interval according to its frequency. With the notation from the introduction to this chapter, symbol σ_i is encoded by the sub-interval

$$\mathcal{I}(\sigma_i) \stackrel{\text{def}}{=} \left[\sum_{j=1}^{i-1} \nu_i, \sum_{j=1}^{i} \nu_i\right),$$

where the sum over all frequencies ν_k is one. The string, which has to be encoded, is a concatenation of symbols and will also be encoded by a sub-interval of the unit interval, which results from an interval subdivision and can be defined in a nice formal manner with the empty string ϵ and the concatenation operation "$\circ$"

$$\begin{aligned} \mathcal{I}(\epsilon) &\stackrel{\text{def}}{=} [0,1) \\ \mathcal{I}(s) = [A,B), \mathcal{I}(\sigma) = [a,b) &\Rightarrow \\ \mathcal{I}(s \circ \sigma) &\stackrel{\text{def}}{=} [A + a \cdot (B-A), B + b \cdot (B-A)). \end{aligned} \quad (3.2)$$

Now we are able to translate a string of symbols based on the symbol frequencies into a uniquely defined sub-interval $[A, B)$ of the unit interval. The target interval $[A, B)$ finally needs to be encoded with bits. This can be done through a binary fraction. On the one hand the binary fraction $.1101$ specifies the fraction $13/16$ but on the other hand, if we don't know how the binary fraction goes on after the four known digits, we can only safely tell that a binary fraction beginning with $.1101$ will lay within the interval $\mathcal{I}(.1101) \stackrel{\text{def}}{=} [.1101, .1110)$. The target interval $[a, b)$ can be uniquely specified through the shortest binary fraction f_{bin}, that satisfies the relation

$$\mathcal{I}(f_{\text{bin}}) \subseteq [A, B) = \mathcal{I}(s).$$

The typical encoding algorithm keeps a current interval and updates it according to the subdivision formula 3.2 for each newly encoded symbol. Decoding the string s is also very simple. One reads the binary fraction and re-does the interval subdivision by updating the current interval according to the currently decoded symbol. As the binary fraction is known, one can determine the next symbol by finding the sub-interval of the symbol, in which the binary fraction is completely contained.

At first glance this approach seems to involve arbitrary precision arithmetic, but there is a nice and simple method to perform arithmetic coding with integer arithmetic alone. For this the integer values are interpreted as representing binary fractions of fixed length, often 32 or 64 bits. The encoder stores the current interval as integer values. The important observation is that if the highest bit of the lower and of the upper bound of the current interval are equal, they cannot change any more as the encoded sub-interval only can shrink. Thus one can shift all calculations one bit to the left and send the highest bit to a binary stream, that represents the binary fraction.

Although the arithmetic coding technique is quite sophisticated and the underlying algorithms not too simple, Langdon[Lan84] describes how to implement the encoding and decoding algorithms in hardware. Before we come to some applications, let us state again the fact, that arithmetic coding achieves the binary entropy of the encoded string up to less than a percent as described in [WNC87]. Thus it is valid to say that a symbol consumes a fractional amount of bits. Let ν_i be again the frequency of the symbol σ_i and $\mathcal{S}_\sigma(\nu_i)$ the fractional amount of bits, the symbol σ_i consumes in arithmetic coding. Then the following equation is true up to a negligible deviation

$$\mathcal{S}_\sigma(\nu_i) \approx -\mathrm{lb}\nu_i. \tag{3.3}$$

3.2.1 Index Coding

The first application of arithmetic coding is the encoding of indices. The precondition is that we know the range of each encoded index at the moment before it is encoded or decoded. Suppose the index i falls into the range $1, .., n$. Then the unit interval of the arithmetic coder is subdivided into n equal sized sub-intervals and the i-th sub-interval is encoded. As the frequency of each possible index value is $1/n$, the encoding of an index consumes (see equation 3.3)

$$\mathcal{S}_{\mathrm{index}}(n) \approx \mathrm{lb}n \tag{3.4}$$

3.2.2 Flag Encoding

A frequently arising situation is that we have to encode a flag, i.e. a one bit value, for a whole sequence of elements. Suppose there are n flags to be encoded and the frequency of the flag to be true is ν_1. Then the average consumed amount of bits per flag is

$$\mathcal{S}_{\mathrm{flag}}(\nu_1) = -\nu_1\mathrm{lb}\nu_1 - (1-\nu_1)\mathrm{lb}(1-\nu_1) \tag{3.5}$$

Figure 3.1 illustrates equation 3.5. If the flag is exactly in half of the case one ($\nu_1 = 0.5$), nothing can be saved and one bit per flag is consumed. But if the frequency ν_1 goes to 0 or 1 a significant improvement can be achieved with arithmetic coding.

3.2.3 Adaptive Arithmetic Coding

The adaptive arithmetic coding exploits the same idea as adaptive Huffman coding (see section 3.1). During encoding and decoding the symbol counts are incremented and the frequencies of the symbols updated. This update process is quite expensive. Hester et al.[HH85] describe an efficient variant of a self-organizing linear search.

In geometry coding one has to encode vertex locations, what is often done by first quantizing the coordinates to 12 or 16 bit indices. For the use of adaptive arithmetic coding one faces the problem of handling $65,536$ symbols and update their frequencies all the time. To avoid this, the indices are sub-divided into packages of 4 bits, such that

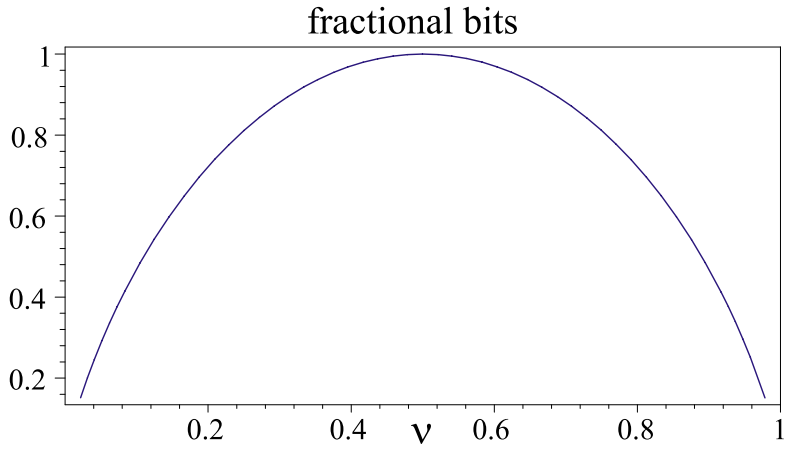


Figure 3.1: The diagram shows the fractional storage space $\mathcal{S}_{\text{flag}}$ for a bit in terms of the frequency ν_1 of the flag being one.

only four times sixteen different symbols need to be handled. The encoding efficiency does decrease only slightly with this practical approach.

3.2.4 Sparse Flag Coding

In mesh coding a common problem is the efficient encoding of the locations of the appearances of a very rare symbol ρ in the symbol string s of length n, i.e. the frequency ν_ρ is much smaller than one. Two not very efficient methods are often used. Firstly, one can encode the locations of ρ with a flag consuming n bits. The obviously better approach to this idea is the use of the flag encoding of subsection 3.2.2. Per symbol ρ this approach would consume the storage space for one true valued flag plus the storage space for $(1-\nu_\rho)n/\nu_\rho n$ false valued flags (compare equation 3.5)

$$\mathcal{S}_{\text{sparseFlag}} \approx -\text{lb}\nu_\rho - \frac{1-\nu_\rho}{\nu_\rho}\text{lb}\,(1-\nu_\rho) = -\text{lb}\left[\nu_\rho\,(1-\nu_\rho)^{\frac{1}{\nu_\rho}-1}\right]. \tag{3.6}$$

The second approach is to encode the locations with indices into the string, consuming even with arithmetic coding $\text{lb}n$ bits per appearance of ρ. This is neither very efficient because one additionally encodes a permutation of the ρ symbols as the order of the indices could be permuted arbitrarily. As there are $k\text{lb}k$ different permutations on k symbols, the index encoding would waste $\text{lb}k$ bits per index. These overhead of bits can be saved by sorting the indices, encoding the first one and afterwards for each further index only the difference to the previous index. If one applies adaptive arithmetic coding to the differences, the $\text{lb}k$ bits can be saved. This is difficult to proof theoretically but was shown by measurements on randomly distributed indices. Thus with $k = \nu_\rho n$ the index encoding of a sparse flag would consume for each symbol ρ

$$\mathcal{S}_{\text{sparseIndex}} \approx (\text{lb}n - \text{lb}(\nu_\rho n)) = -\text{lb}\nu_\rho. \tag{3.7}$$

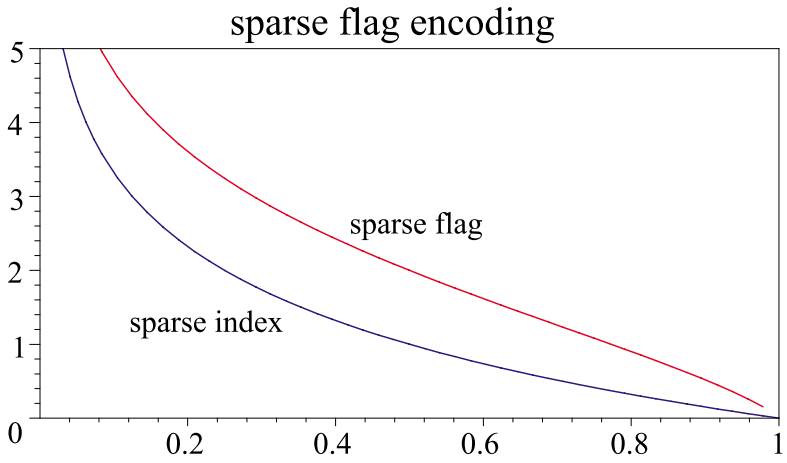


Figure 3.2: The diagram compares the sparse flag coding (equation 3.6) with the sparse index coding (equation 3.7).

igure 3.2 compares the two approaches. The difference index coding is always superior, but it is more difficult to implement and slower in performance.

3.3 Variable Length Coding

In this section we describe variable length coding schemes for signed indices. The zero case is special because -0 is the same as $+0$. In our applications of variable length coding we do not need the zero index. Therefore we exclude the zero case and assume that it never arises. For the Cut-Border Machine we do neither need the ± 1 cases and therefore we restrict the discussion in this section to signed indices with absolute values larger than one. But we could have chosen any other minimal absolute index value.

Figure 3.3 illustrates three different simple variable length coding schemes for the signed indices starting with ± 2. All three schemes begin with one bit for the sign of the index. Coding scheme a) encodes a bit with two bits – the bit of the index and an additional control bit specifying, whether further bits follow. In scheme a) indices ± 2

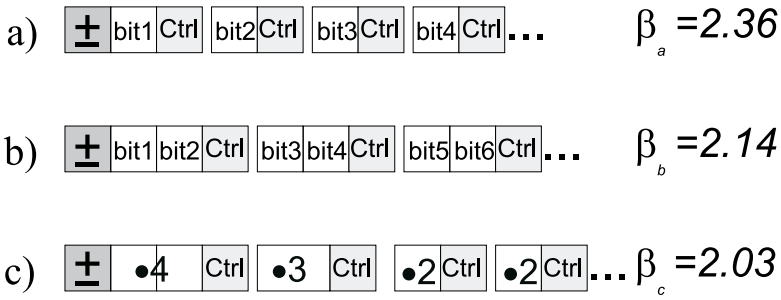


Figure 3.3: Three different variable length coding schemes for signed indices.

and ± 3 are encoded with one sign bit, one index bit and one control bit. The indices $\pm 4 \ldots \pm 9$ are encoded with five bits and so on. Scheme b) packs the index bits in three bit bundles of two index- and one control-bit. Finally, the third scheme c) mixes both approaches and simple arithmetic coding. The first three-bit bundle specifies the two lowest significant bits of the absolute value of the index minus two or equivalently the remainder of the index minus two when divided by four. The second bundle encodes the remainder of a fourth of the by two decremented index divided by three. Using arithmetic coding this bundle can be encoded with $\mathrm{lb}3 + 1 \approx 2.585$ bits. The by two decremented index divided by twelve is encoded with two-bit bundles as in scheme a).

For all of the three schemes in figure 3.3 the storage space $\mathcal{I}_{\alpha \in \{a,b,c\}}$ for encoding an index i obeys the relation

$$\forall i \geq 2 : \mathcal{I}_\alpha(i) \leq \beta_\alpha \mathrm{lb}(i+1) + 1, \tag{3.8}$$

with the different values for β_α as given on the right of figure 3.3. It is somehow arbitrary that we wrote the term on the right side of equation 3.8 in terms of $\mathrm{lb}(i+1)$. Actually, one would have assumed no plus one but rather a minus. The plus one was chosen in accordance to the application of the equation in the case of the Cut-Border Machine, where the $i+1$ corresponds to the number of encoded edges. The fact that index 2 implies, that already three edges have been encoded, corresponds to some extra savings, which allow us to keep the β in equation 3.8 smaller.

Let us justify the validity of relation 3.8 exemplary for scheme c). The problematic indices are the ones, which force the usage of a new bundle. In scheme c) these are the indices $\pm 2, \pm 6, \pm 14, \pm 26, \ldots, 12 \cdot 2^k + 2, \ldots$. The first bundle consumes together with the sign four bits, the second bundle $\mathrm{lb}3 + 1$ bits and each following bundle further two bits. Thus for the indices ± 2 one must check $4 \leq \beta_c \mathrm{lb}3 + 1$, for the indices ± 6 check $4 + \mathrm{lb}3 + 1 \leq \beta_c \mathrm{lb}7 + 1$ and for the remaining problematic indices relation 3.8 is valid, iff

$$\forall k \geq 0 : 4 + \mathrm{lb}3 + 1 + 2(k+1) \leq \beta_c \mathrm{lb}\left(12 \cdot 2^k + 3\right) + 1. \tag{3.9}$$

Solving the equal case of this relation for k yields no real solution and the relation holds true for $k = 0$. Therefore, it must hold true for all values of k. Similar arguments show the validity of relation 3.8 for the variable length coding schemes a) and b).

The minimal value for β can be achieved by an arithmetic variable length coding scheme. Again the first bit is used for the sign. To each absolute value of the indices a sub-interval of the unit interval is assigned, the length of which corresponds to the frequency ν_i of the encoded index. With equation 3.3 we can relate the frequencies to the number of consumed bits $b_i = -\mathrm{lb}\nu_i$. From relation 3.8 we assume that $b_i = \beta_{\min} \mathrm{lb}(i+1)$. As all frequencies of the different indices must sum up to one this yields a condition for $\beta_{\min}$

$$1 = \sum_{i \geq 2} 2^{-\beta_{\min} \mathrm{lb}(i+1)} = \sum_{i \geq 2} \frac{1}{(i+1)^{\beta_{\min}}}. \tag{3.10}$$

This equation is hard to solve for $\beta_{\min}$, but with the integral $\int \frac{1}{x^{\beta_{\min}}}$ we could proof the following relation

$$1.589 < \beta_{\min} < 1.59. \tag{3.11}$$

An arithmetic coder that achieves $\beta_{\min}$ requires arbitrary precision arithmetic and therefore is not able to encode and decode symbols in constant time. We did not find a simple coding scheme to improve on $\beta_c = 2.03$ but there probably is one. For the remainder we will stick to β_c.

Part II

Triangle Mesh Compression

In this part we describe the manifold triangle mesh compression technique of the Cut-Border Machine and the Edgebreaker in detail. As also upper bounds for the methods will be presented, we start in chapter 4 with a discussion on lower and upper bounds of the encoding of triangle meshes.

Chapter 5 describes the Cut-Border Machine in its original version with the goal to provide compression and decompression algorithms for real-time applications. The second chapter on the Cut-Border Machine elaborates several improvements to the encoding scheme including better connectivity compression rates and provable linear bounds on the running time and storage space consumption in the case of planar triangulations or meshes with low Euler characteristic.

The chapter 7 on the Edgebreaker scheme reflects the work of the author to get closer to the theoretical lower bound of a 3.245 bits per vertex encoding of planar triangulations.

Finally, this part closes with a chapter on concluding remarks and directions for future work.

Chapter 4

Bounds on Triangle Mesh Compression

In this chapter we describe theoretical bounds for the encoding of triangle mesh connectivity. For planar triangulations there is a tight theoretical lower bound for connectivity encoding given by the number of different triangulations.

4.1 Planar Triangulations

Tutte counts in [Tut62] the number of different planar triangulation with a fixed number of border edges. Figure 4.1 shows a sample planar triangulation with four border edges. Tutte only counts triangulations with different connectivity, i.e. the location of the vertices v_0 to v_5 is arbitrary. Tutte does not account for all kinds of symmetries of different triangulations, but just fixes the border vertices v_0 to v_3. A different triangulation of the one in figure 4.1 is for example generated if the border vertices are renamed from (v_0, v_1, v_2, v_3) to (v_1, v_2, v_3, v_0). A renaming into (v_2, v_3, v_0, v_1) results in the same triangulation, as the connectivity defines the same incidence relations.

Tutte calculates an asymptotic formula for the number of different planar triangulations Ψ_v with three border edges in dependence on the number of vertices v in the planar triangulation

$$\Psi_v \sim \frac{1}{16}\sqrt{\frac{3}{2\pi}}v^{-\frac{5}{2}}\left(\frac{256}{27}\right)^{v+1}. \tag{4.1}$$

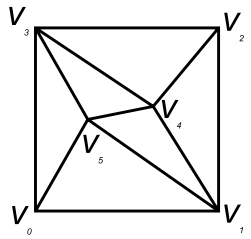


Figure 4.1: An example of a planar triangulation with four border edges.

The special case of three border edges can also be interpreted as a closed triangle mesh with the topology of a sphere. In order to determine the minimum number of bits needed to encode a closed triangle mesh we just have to take the binary logarithm of the count given in equation 4.1. Skipping terms of the order of $\mathrm{lb}v$ results in a lower bound for connectivity encoding of closed manifold triangle meshes $\mathcal{L}_v^{\text{planar}}$ in bits per vertex

$$\mathcal{L}_v^{\text{planar}} = \mathrm{lb}\left(\frac{256}{27}\right) \approx 3.2451125. \tag{4.2}$$

And using the formula 1.2 the lower bound can be converted into bits per triangle

$$\mathcal{L}_t^{\text{planar}} = \frac{1}{2}\mathrm{lb}\left(\frac{256}{27}\right) \approx 1.62255625. \tag{4.3}$$

We can conclude with the following theorem

Theorem 4.1 *To encode a sufficiently large planar triangulation or a closed manifold triangle mesh with t triangles at least $1.6225562 \cdot t$ bits are needed for sufficiently large t.*

4.2 Planar Triangulations with Holes

More border edges and more border loops do not change the assymptotic behaviour of the number of different triangle meshes significantly. For each edge-connected component the number of border loops is bound by the number of triangles in the component.

Lemma 4.2 *The number of border edges in an edge-connected closed manifold triangle mesh with border is limited by the number of triangles plus two.*

Proof: Take an arbitrary triangle of the mesh. For this triangle the lemma is true. Then we rebuild the mesh by adding the remaining triangles at the border of the sofar rebuild mesh. This is possible as the mesh is edge connected. Each addition of a triangle changes at least one border edge into an inner edge and generates no more than two new border edges. Thus no more than one border edge is added per triangle. In this way the lemma is true during the whole rebuilding process and therefore also holds for the mesh itself. □

The holes of an edge-connected manifold triangle mesh can be easily triangulated with consistent connectivity by a triangle strip as shown in figure 4.2, where it is not important, whether this triangulation will produce self-intersections or not. A hole with b border edges can be triangulated with $b - 2$ triangles. Thus one can extend each encoding scheme for planar triangulations easily to planar triangulations with holes by triangulating the holes, encoding the resulting planar triangulation and finally encoding, which of the triangles were dummy triangles with one bit per triangle. Together with lemma 4.2 the following theorem holds:

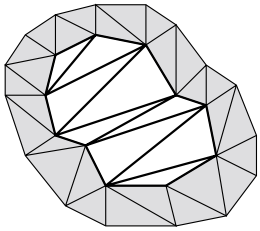


Figure 4.2: Any orientable hole in a triangle mesh can be triangulated with a triangle strip as drawn in bold style.

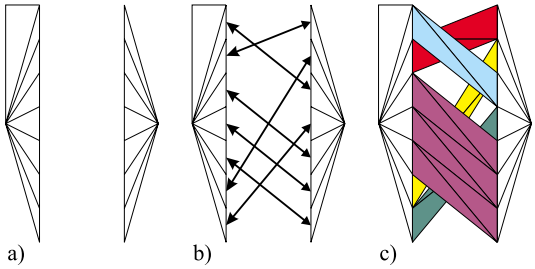


Figure 4.3: Representation of a permutation through a non planar triangle mesh.
a) domain and range of permutation, b) mapping and c) the resulting triangle mesh.

Theorem 4.3 *An encoding scheme for planar triangulations that consumes $\mathcal{S}$ bits per triangle allows to encode planar triangle meshes with holes with no more than $2\mathcal{S}+2$ bits per triangle.*

Theorem 4.3 also gives an upper bound for the number of different triangulations with holes. With the lower bound for planar triangulations in equation 4.3 we know that there are no more than $2^{2t(\mathcal{L}_t^{\text{planar}}+1)}$ different planar triangulations with holes.

4.3 Non Planar Triangle Meshes

In the case of non planar triangle meshes the situation becomes worse. Figure 4.3 demonstrates how to represent a permutation of $n = 7$ elements through a triangle mesh. In the first step in figure 4.3 a) the domain and the range of the permutation is constructed with two triangle fans. The domain fan is marked with an addition triangle such that the fans contain $2n+1$ triangles. In the second step the adjacencies of edges in the two different fans are defined according to the permutation. And finally each adjacency is represented by two triangles. The first triangle is always adjacent to the domain triangle fan and the second one is attached to the range fan and the edge of the first triangle which is incident on the upper vertex of the domain fan edge. This can be

consistently done by choosing a fixed orientation. Thus for each domain element the element in the range can be easily found following the two triangles representing the adjacency.

We can conclude that a permutation of n elements can be represented through a manifold triangle mesh with border consisting of $t = 4n + 1$ triangles. As there are $n!$ different permutations of n elements, from the Stirling's formula follows, that non planar manifold triangles meshes with border consume $\Omega(t \log t)$ bits. The same holds true for closed manifold triangle meshes of higher genus. Just imaging to blow up the example in figure 4.3 in the third dimension. The standard representation for the connectivity of a triangle mesh given by the relation $F \rightarrow V$ consumes $3t\mathrm{lb}v$ bits. As $v \leq 3t$ the connectivity of any triangle mesh can be encoded in $O(t \log t)$ bits. Finally, we have the same assymptotic bounds for non manifold triangle meshes.

Theorem 4.4

- *encoding of closed manifold triangle meshes with t triangles consumes $\Theta(t \log t)$ bits*
- *encoding of manifold triangle meshes with border consisting of t triangles consumes $\Theta(t \log t)$ bits*
- *encoding of non manifold triangle meshes with t triangles consumes $\Theta(t \log t)$ bits*

Chapter 5

The Cut-Border Machine

We introduce the Cut-Border Machine connectivity encoding by comparison with generalized triangle strips. The latter approach utilizes a vertex buffer of only two vertices but in turn has to encode each vertex twice. Thus the first idea is to simply increase the size of the vertex buffer to avoid all vertex repetitions. As in the case of triangle strips, the Cut-Border Machine encodes triangle by triangle. Thus at any time during the encoding the triangle mesh is split into two parts, the *inner part* consisting of all encoded triangles and the *outer part* formed by all the still to be encoded triangles (see figure 5.1). The set of edges between the inner and the outer part is called the *cut-border*. All the vertices contained in the cut-border have been encoded already, as at least one triangle of the inner part is incident to them, and they are still needed to specify a not yet encoded triangle from the outer part. Thus the Cut-Border Machine buffers all the vertices on the cut-border. In order to avoid a strong fragmentation of the cut-border, the triangle from the outer part, which is encoded next, is chosen incident to one of the cut-border edges. This specific edge is called the *gate*. On the cut-border the choice for the next gate can become quite broad. The second idea of the Cut-Border Machine is to fix the choice of the next gate, i.e. the *traversal order*, in order to avoid the need for any

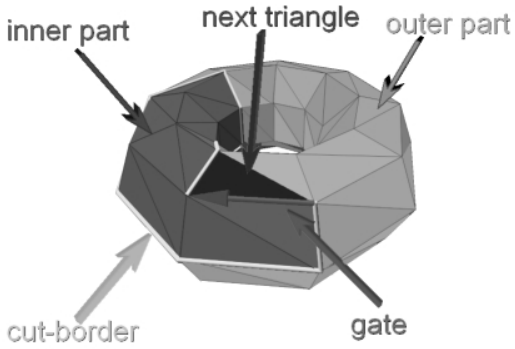


Figure 5.1: Snapshot during compression of a toroidal triangle mesh with the cut-border machine.

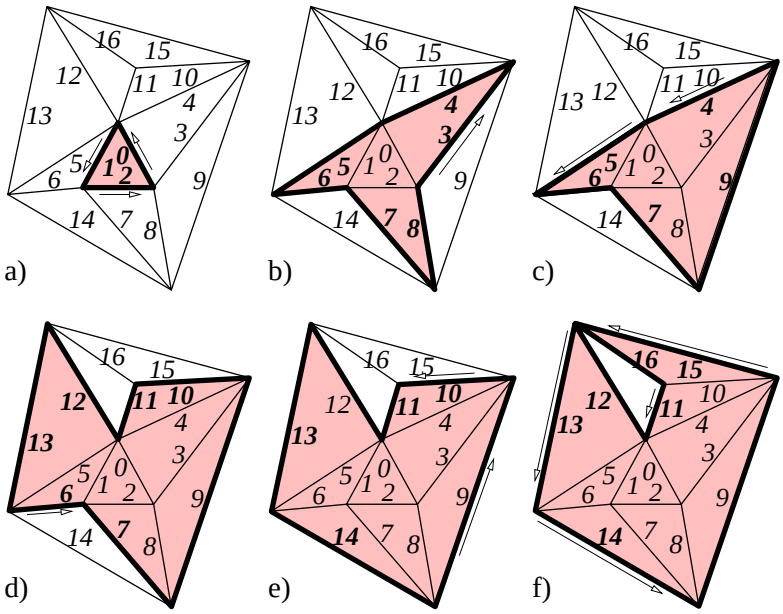


Figure 5.2: The shown sample triangle mesh is encoded in a breadth-first order. The different cut-border operations are illustrated.

additional bits. Thus the Cut-Border Machine can encode the connectivity of a triangle mesh by specifying for each triangle how it is incorporated at the gate into the inner part. We also call these different ways of adding a triangle the *cut-border operations*. The Cut-Border Machine starts with an arbitrary triangle as initial inner part and an arbitrary initial gate incident to this triangle. The vertices are encoded in the order in which they appear for the first time in the encoding process.

In the following we describe the Cut-Border Machine compression technique in more detail. In order to find the next triangle incident on the gate during the compression of a triangle mesh, we use the half edge data structure described in section 1.5. In section 5.1 we gather the different cut-border operations and describe the compressed representation of a triangle mesh. Details about the implementation are given in section 5.2. After some measurements the traversal order, which defines the choice of the gate after each cut-border operation, is optimized in section 5.2.3. We close the first chapter on the cut-border machine with some extensions in section 5.3.

5.1 Cut-Border Operations & Compressed Representation

Figure 5.2 illustrates the encoding of a sample triangle mesh, where all except one cut-border operation arise. The triangle mesh is always built from an initial triangle

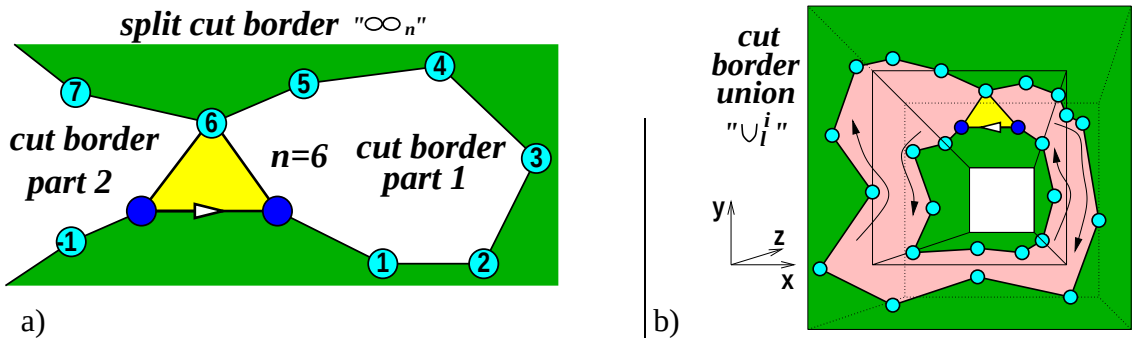


Figure 5.3: The *"split cut-border"*-/*"cut-border union"*-operation needs one/two indices to specify the third vertex together with which the current cut-border edge forms the next triangle. The vertices of the gate are shaded dark and the newly attached triangle light.

consisting of the first three vertices. The initial cut-border operation is not encoded but will be denoted with the symbol "Δ". Between figure 5.2 a) and 5.2 b) all of the three initial cut-border edges $0, 1$ and 2 become the gate in increasing order of their indices and to each edge the adjacent triangle is added to the inner part. Each operation introduces a new vertex and two new edges to the cut-border. Let us call this cut-border operation *"new vertex"* and abbreviate it with the symbol "$*$". The new cut-border edges are enumerated in the order they are added to the cut-border, what causes a breadth-first traverse order.

Between figure 5.2 b) and 5.2 c) the triangle of the outer part, which is incident to gate 3, is added to the inner part. This time no new vertex is inserted, but edge 3 forms a triangle with the preceding cut-border edge. This operation will be called *"connect backward"* and is represented by the symbol "$\leftarrow$".

Moving on to figure 5.2 d), two *"new vertex"*-operations arise at the cut-border edges 4 and 5. At the cut-border edge 6 the mirror image of the *"connect backward"*-operation is applied to connect the gate to the subsequent edge on the cut-border. Naturally, this operation is called *"connect forward"* and is abbreviated with "$\rightarrow$". No triangle is added to cut-border edge 9 as it is part of the mesh border. This fact has to be encoded, too, and is called *"border"*-operation ("$\_$").

A more complex operation arises at cut-border edge 10 in figure 5.2 e). The adjacent triangle in the outer part is neither formed with the preceding nor with the subsequent cut-border vertex, but with a vertex further apart. The result is that the cut-border splits into two loops. In figure 5.2 f) the first loop is formed by the edges $11, 12$ and 16 and the second loop by $13, 14$ and 15. This operation will be called *"split cut-border"* ("∞_i"), which takes the index i to specify the third vertex relative to the current cut-border edge. Figure 5.3 a) shows another *"split cut-border"*-operation. The relative indices are written into the cut-border vertices. The *"split cut-border"*-operation has two consequences. Firstly, the cut-border cannot be represented anymore by a simple linked list, but a list of linked lists is necessary. And secondly, the choice of the next

cut-border loop to be processed after a *"split cut-border"*-operation yields a new degree of freedom for the traverse order. To minimize the number of cut-border loops the cut-border loop with fewer vertices is chosen.

Another operation arises in figure 5.2 f) at cut-border edge 11. The incident triangle closes the triangle mesh and the current cut-border loop is removed. This operation is called *"close cut-border"* and is denoted by "∇". As the size of the current cut-border loop is known during compression and decompression, the *"close cut-border"*-operation can also be encoded with *"connect forward"* or *"connect backward"* and the different symbol is only introduced for didactic reasons. On the other hand if there really is a hole in form of a triangle, the hole is encoded with three *"border"*-operations.

Finally, there exists a somewhat inverse operation to the *"split cut-border"*-operation – the *"cut-border union"*-operation. An example is visualized in figure 5.3 b). The figure shows in a perspective view a cube with a quadratic hole. The so far compressed inner part consists of the two dark shaded regions. There are two cut-border loops, which are connected by the new light triangle, which is attached to the gate (dark vertices). Therefore, this operation is called *"cut-border union"* or for short "$\bigcup_l^i$". Two indices are needed to specify the second cut-border loop l and the index i of the vertex within the second cut-border loop. The vertices in a cut-border loop are enumerated according to the cut-border edges. Therefore, the vertex at the beginning of the cut-border edge with the smallest index in the cut-border loop l is labeled zero, the vertex at the second smallest cut-border edge is labeled one and so forth.

It can be shown that the number of *"cut-border union"*-operations is exactly the genus of the compressed triangle mesh. Seen from a different angle, the operations "∇", "$\rightarrow/\leftarrow$", "∞_i" and "$\bigcup_l^i$" provide the possibility to connect the current cut-border edge to any possible vertex in the cut-border, whereas the operations "Δ" and "$*$" utilize new vertices.

The encoding of the sequence of cut-border operations uniquely defines the connectivity of a triangle mesh. The connectivity of the sample mesh in figure 5.2 can be encoded by the following sequence of operations:

$$***\leftarrow**\rightarrow\_\,\infty_2\rightarrow\_\_\_$$

The symbols for the different operations can be encoded with Huffman Codes to achieve good compression rates. Therefore, the mesh connectivity is sequentially stored in a *bit stream*.

The geometry and material data must be supplied additionally. For each vertex this data can include the vertex position, the surface normal at the vertex and the texture coordinates or some material information. We will refer to all this data with the term *vertex data*. The material of the surface can also be given for each triangle. Similarly, data can be supplied for the inner edges and the border edges of the mesh. We will collect the different kinds of data in the terms *triangle data*, the *inner edge data* and the *border edge data*. Thus for each type of mesh element, data can be supplied with the connectivity of the mesh. We refer to the collection of all additional data with the term *mesh data*.

op.:	**vertex**	**inner edge**	**border edge**	**triangle**
Δ	3	0	0	1
$*$	1	1	0	1
$\rightarrow/\leftarrow$	0	2	0	1
$\_$	0	0	1	0
∞_i	0	1	0	1
∇	0	3	0	1
$\bigcup_l^i$	0	1	0	1

Table 5.1: The table shows for each cut-border operation, which mesh elements are newly introduced to the inner part. Inner edges are introduced after both of their triangles have been incorporated to the inner part in order to distinguish them from border edges.

Depending on the application there exist two approaches to combine the connectivity and the mesh data of a compressed triangle mesh. If an application has access to sufficient memory for the complete mesh data, the bit stream for the connectivity can be stored separately. For each type of mesh element the specific data is stored in an array. While the triangle mesh is traversed a vertex, triangle, inner edge and border edge index is incremented after each operation, such that the current mesh elements can be found in the corresponding arrays with the suitable indices. Table 5.1 shows the increments for each index after the different operations. For example after a *"connect forward"*-operation the inner edge index is incremented by two and the triangle index by one. The advantage of this representation is that the mesh data can be processed without traversing the compressed connectivity representation, for example to apply transformations to the coordinates and normal vectors.

If the compressed triangle mesh is passed to the graphics accelerator or if the mesh data is encoded with variable length values, no random access to the vertex data is possible. Then the mesh data is inserted into the bit stream of the mesh connectivity. After each operation symbol in the stream, the corresponding mesh data is encoded to the stream appropriately. For example after a *"split cut border"*-symbol the mesh data for one inner edge and one triangle is transmitted (see table 5.1). If we only assume vertex and triangle data to be present and denote the vertex data for the i^{th} vertex with v_i and the triangle data for triangle j with t_j, the extended bit stream representation of the mesh in figure 5.2 would be:

$$v_0 v_1 v_2 t_0 * v_3 t_1 * v_4 t_2 * v_5 t_3 \leftarrow t_4 * v_6 t_5 * v_7 t_6 \rightarrow t_7 \_ \infty_2 t_8 \rightarrow t_9 \_ \_ \_$$

Remember that the initial triangle is implicitly stored without symbol and introduces the vertices v_0, v_1, v_2 and the triangle t_0. If the triangle mesh consists of several unconnected components, the compressed bit stream representation consists of the concatenation of the bit streams of the different components.

5.2 Implementation

All algorithms, which process the compressed representation, are based on the implementation of the data structure for the cut-border as introduced in the next paragraph. This data structure implements the rules, which define the traverse order. All other algorithms such as the compression and decompression algorithms presented later on in this section use this implementation. Further algorithms such as homogeneous transformations of the mesh geometry would also use the cut-border data structure to iterate through the compressed representation

The data structures and algorithms in this section are given in a C++-like syntax. For readability and brevity indentation and additional keywords replaced parentheses.

5.2.1 Cut-Border Data Structure

Data Structure 1 cut border

```
struct Loop
    int         rootElement, nrEdges, nrVertices;
struct Element
    int         prev, next;
    Data        data;
    bool        isEdgeBegin;
struct CutBorder
    Loop        *parts, *loop;
    Element     *elements, *element;
    Element     *emptyElements;

                CutBorder(int maxLoops, int maxElems);
    bool        atEnd();
    void        traverseStep(Data &v0, Data &v1);

    void        initial(Data v0, Data v1, Data v2);
    void        newVertex(Data v);
    Data        connectForward/Backward();
    void        border();
    Data        splitCutBorder(int i);
    Data        cutBorderUnion(int i, int l);

    bool        findAndUpdate(Data v, int i, int l);
```

The data structure for the cut-border is a list of doubly linked lists storing the vertex data of the buffered vertices. All elements in the doubly linked lists of the different

loops are stored within one homogeneous buffer named *elements*. The maximum number of vertices in the buffer during the compression or decompression defines its size. The maximum buffer size is known once the triangle mesh is compressed and can be transmitted in front of the compressed representation. For the first compression of the mesh the maximum number of vertices can be estimated by $10\sqrt{v}$ (see section 5.2.5), where v is the number of vertices in the triangle mesh. With this approach a simple and efficient memory management as described in [Mey97] is feasible. Only the pointer *emptyElements* is needed, which points to the first of the empty elements in the buffer. Any time a new element is needed, it is taken from the empty elements and the deleted elements are put back to the empty elements. On the one hand the memory management avoids dynamic storage allocation which is not available on graphics boards and on the other hand it speeds up the algorithms by a factor of two if no memory caches influence the performance.

The different loops can be managed with an array *loops* with enough space for the maximum number of loops which are created while the mesh is traversed. Again this number must be estimated for the first compression and can be transmitted in front of the compressed representation. Thus the constructor for the cut-border data structure takes the maximum number of loops and the maximum number of cut-border elements.

loop and *element* point to the current loop and the current element within the current loop respectively. Each loop stores the index of its root element, the number of edges and the number of vertices. These numbers may differ as each loop is not simply a closed polygon. Any time a *"border"*-operation arises, one cut-border edge is eliminated but the incident cut-border vertices can only be removed if they are incident to two removed edges. Therefore, each cut-border element stores in addition to the indices of the previous and next element and the vertex data, a flag which denotes whether the edge beginning at this cut-border element belongs to the cut-border or not.

The cut-border data structure provides methods to steer the traversal via a bit stream or with the help of a triangle mesh. The methods *atEnd*`()` and *traverseStep*`(`v_0`,` v_1`)` are used to form the main loop. The method *traverseStep*`(&`v_0`, &`v_1`)` steps to the next edge in the cut-border data structure and returns the vertex data of the two vertices forming this edge. If no more edges are available, *atEnd*`()` becomes true.

During decompression the operations are read from the bit stream and the cut-border data structure is updated with the corresponding methods *initial*, *newVertex*, *connectForward/Backward*, *border*, *splitCutBorder* and *cutBorderUnion*. For compression additionally the method *findAndUpdate* is needed to localize a vertex within the cut-border data structure. The loop and vertex indices are returned and can be used to deduce the current building operation. If the vertex has been found by the *findAndUpdate*-method, it is connected with the gate.

5.2.2 Compression Algorithm

Besides the cut-border we need two further data structures for the compression algorithm — a random access mesh with access to the third vertex of an edge and a permu-

Algorithm 1 compression

Input:	*RAM*	...	random access representation
Output:	*bitStream*	...	compressed representation
	perm	...	permutation of the vertices

```
vertexIdx = 3;
RAM.chooseTriangle(v0, v1, v2);
perm.map((v0.idx, 0), (v1.idx, 1), (v2.idx, 2));
bitStream << v0 << v1 << v2;
cutBorder.initial(v0, v1, v2);
while not cutBorder.atEnd() do
    cutBorder.traversalStep(v0, v1);
    v2 = RAM.getVertexData(v1.idx, v0.idx);
    if v2.isUndefined() then
        bitStream << "_";
        cutBorder.border();
    else
        if not perm.isMapped(v2.idx) then
            bitStream << "*" << v2;
            cutBorder.newVertex(v2);
            perm.map((v2.idx, vertexIdx++));
        else
            cutBorder.findAndUpdate(v2, i, l);
            if p > 0 then bitStream << "⋃_l^i";
            else if i == ±1 then bitStream << "→/←";
            else bitStream << "∞_i";
```

tation. The random access mesh provides two methods:

1. the *chooseTriangle* (v_0, v_1, v_2) -method returns the vertex data v_0, v_1, v_2 of the three vertices in an initial triangle

2. the *getVertexData* (i_0, i_1) , which takes the vertex indices i_0 and i_1 of a halfedge $h = (i_0, i_1)$ and returns the vertex data of the third vertex of the triangle containing the oriented halfedge h

The random access mesh can be easily implemented with a halfedge data structure as described in section 1.5.1. The permutation is used to build a bijection between the vertex indices in the random access representation and the vertex indices in the compressed representation. It allows to map an index of the first kind to an index of the second kind and to determine whether a certain vertex index in the random access representation has been mapped.

Given a random access triangle mesh, the compression algorithm computes the mentioned permutation and the compressed representation of the mesh, which is sent to a bit stream. The current vertex index of the compressed representation is enumerated in the index *vertexIdx*. After the initial triangle has been processed, the cut-border data structure is used to iterate through the triangle mesh. In each step the vertex data v_0 and v_1 of the gate is determined. From the vertex indices the vertex data of the third vertex in the triangle incident to the gate is looked up in the random access triangle mesh. If no triangle is found, a *"border"*-operation is sent to the bit stream. Otherwise it is determined whether the new vertex has already been mapped, i.e. sent to the cut-border. If not, a *"new vertex"*-operation is sent to the bit stream and the vertex index is mapped to the current index in the compressed representation. If the third vertex of the new triangle is contained in the cut-border, the *findAndUpdate*-method is used to determine the loop index and the vertex index within that cut-border loop. If the loop index is greater than zero, a *"cut-border union"*-operation is written. Otherwise a *"connect forward/backward"*-operation, *"split cut-border"*-operation or a *"cut-border union"*-operation is written dependent on the loop and vertex indices.

5.2.3 Decompression Algorithm

The decompression algorithm reads the compressed representation from an input bit stream and enumerates all triangles. The triangles are processed with the subroutine *handle* (v_0,v_1,v_2) , which for example renders the triangles. As in the compression algorithm, firstly, the initial triangle is processed and then the mesh is re-built with the help of the cut-border methods *atEnd* and *traversalStep*. In each step the next operation is read from the bit stream and the corresponding method of the cut-border data structure is called such that the third vertex of the new triangle is determined in order to send it to the subroutine *handle*. Performance & Cut-Border Traverse Order In this section we analyze our software implementation of the compression and decompression algorithm.

Algorithm 2 decompression

Input: *bitStream* . . . compressed representation
Output: *handle* . . . processes triangles

```
bitStream >> v0 >> v1 >> v2 ;
handle ( v0, v1, v2 ) ;
cutBorder.initial ( v0, v1, v2 ) ;
while not cutBorder.atEnd () do
    cutBorder.traversalStep ( v0, v1 ) ;
    bitStream >> operation ;
    switch (operation)
        case "→/←":
            handle ( v1, v0,
                cutBorder.connectForward/Backward () ) ;
        case "∞i":
            handle ( v1, v0, cutBorder.splitCutBorder ( i ) ) ;
        case "⋃_l^i":
            handle ( v1, v0, cutBorder.cutBorderUnion ( i,l ) ) ;
        case "*":
            bitStream >> v2 ;
            cutBorder.newVertex ( v2 ) ;
            handle ( v1, v0, v2 ) ;
        case "_":
            cutBorder.border () ;
```

triangle mesh				compr	decom	storage
name	**t**	**v**	**|bd|**	$k\Delta/s$	$k\Delta/s$	$bits/t$
genus5	144	64	0	386	782	4.23±5.7%
vase	180	97	12	511	796	2.15±6.0%
club	515	262	6	541	857	2.09±3.5%
surface	2449	1340	213	490	790	1.87±0.8%
spock	3525	1779	30	496	820	1.97±0.7%
face	24118	12530	940	430	791	1.81±0.3%
jaw	77692	38918	148	332	809	1.62±0.5%
head	391098	196386	1865	321	796	1.71±0.1%

Table 5.2: The basic characteristics of the models, the compression and decompression speed on an O2 R10000 175MHz and the storage needs per triangle.

Firstly, we introduce the test set of models. Then we examine the influence of the traverse order on the compression ratio and the size of the cut-border and come up with Huffman codes, which are independent of the compressed mesh. And finally we gather the important results on the performance of the presented algorithms.

5.2.4 The Models

The measurements were performed on the models shown in figure 5.4. All models are edge-connected manifold meshes and differ in their size. From top left: genus5, vase, club, surface, spock, jaw, face, head. The detail of the head model is hidden in the small interior structures. Therefore, we present in figure 5.4 a view into the inside of the head.

The basic characteristics of the models are shown in the left half of table 5.2. Here the number of triangles t, the number of vertices v and the number of border edges $|\text{bd}|$ are tabulated for each model.

5.2.5 Traverse Order and Cut-Border Size

In section 5.1 we defined a simple breadth-first traversal order. The degrees of freedom in the traversal order are the choice of the initial triangle and the next gate after each cut-border operation. To study the influence of the initial triangle we measured the storage needs for the compressed connectivity of each model several times with randomly chosen initial triangles. Then we computed for each model the average value and the standard deviation as tabulated on the very right of table 5.2. The influence of the initial triangle vanishes with increasing size of the model and is still less than ten percent for the smallest models. With the same measurements the fluctuation of the cut-border size was determined as shown in table 5.4. Here the fluctuation is higher and reaches up to twenty percent for the jaw and the club models.

There are a large number of enumeration strategies for the choice of the gate. For

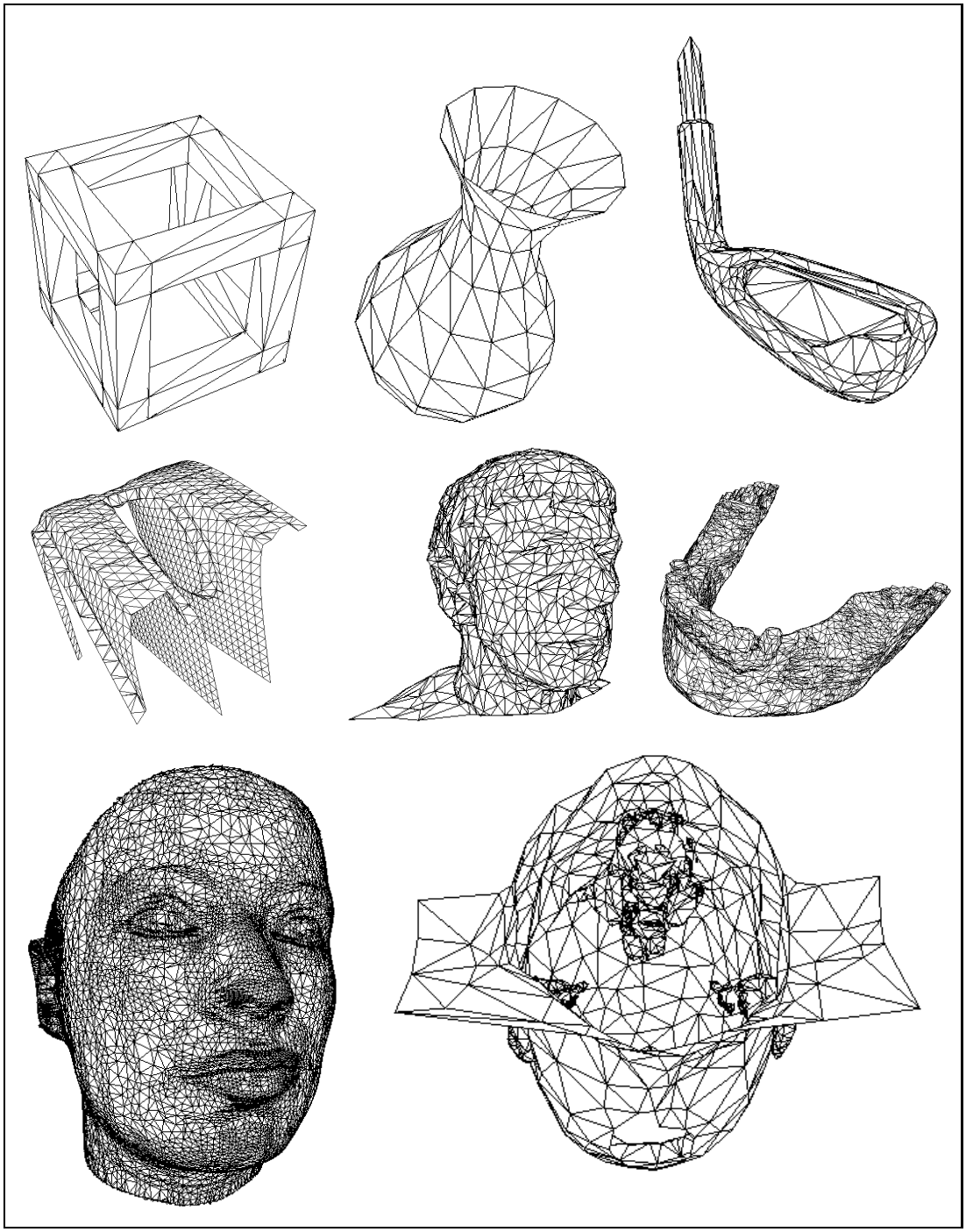
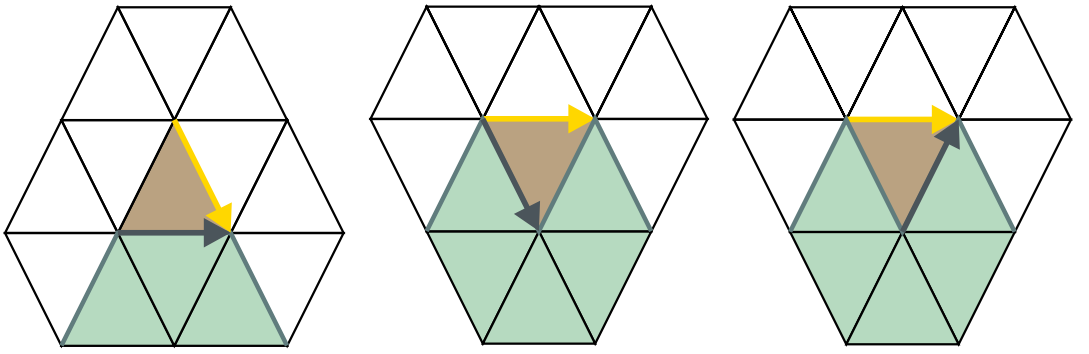
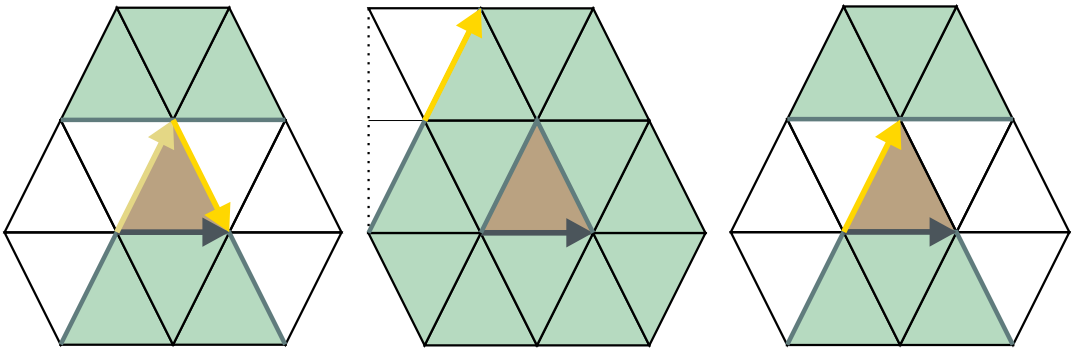


Figure 5.4: The models used to analyze the compression and decompression algorithms.



a) *"new vertex"*: $\star$ b) *"connect forward"*: $\rightarrow$ c) *"connect backward"*: $\leftarrow$



d) *"split cut-border"*: ∞_i e) *"close"*: $\nabla \stackrel{\text{def}}{=} \rightarrow$ e) *"cut-border union"*: $\bigcup_l^i$

Figure 5.5: The choice of the gate after each of the cut-border operations. The current gate is shaded dark, the next triangle also dark and the next gate(s) light.

performance reasons and the simplicity of the implementation, we favored the enumeration strategies, which can be implicitly handled with the cut-border data structure introduced in section 5.2. Therefore a newly introduced cut-border edge may either be delayed until all present edges have become gate or the new edge will become the next gate. These two strategies apply to the *"connect forward/backward"*-operations and correspond to attaching the next highest and the next smallest edge index respectively to the new edge. In the case of a *"new vertex"*-operation two new edges are introduced to the cut-border. In this case three possible strategies are feasible. Either the first/second new edge is chosen as gate next or both edges are delayed. The *"split cut-border"*- and *"cut-border union"*-operations arise much more rarely and therefore were excluded from the analysis of the traversal strategy. Thus we were left with twelve strategies, three choices for the *"new vertex"*-operation and for each *"connect"*-operation two choices. Luckily, it turned out that the strategy, where the new edge is processed next after both *"connect"*-operations and where the second edge is processed next after a *"new vertex"*-operation, is superior over all others. This strategy achieved best compression ratios, if we used standard Huffman coding, and kept the cut-border smallest for all models.

Figure 5.5 illustrates the choice of the gate after each cut-border operation. The

operation	code	bits
$*$	0	1
$\rightarrow$	10	2
∞_2	110	3
$\leftarrow$, ∞_3, _	11100...11110	5
$\infty_{-3\ldots-2}$, $\infty_{4\ldots8}$	11111000...11111110	8
$\infty_{-4\ldots-7}$, $\infty_{9\ldots18}$,	111111110000...	
∞_i, $\bigcup_l^i$	111111111111	12

Table 5.3: Fixed bit codes for real-time encoding of the cut-border operations.

name	$loop_{\max}$	$vert_{\max}$	*prop*
genus5	3.21±12.7%	32.75±15.4%	5.35
vase	2.30±24.2%	22.56±10.2%	2.99
club	3.11±11.9%	44.24±21.0%	4.45
surface	3.10± 9.7%	83.16±12.1%	3.10
spock	3.24±13.2%	120.10± 4.5%	3.23
face	3.40±15.6%	329.08±14.5%	4.22
jaw	4.76±10.7%	564.42±19.7%	4.55
head	9.00±11.1%	1255.20± 8.6%	3.56

Table 5.4: The maximum number of loops and the maximum number of buffered vertices needed for mesh traversal. The last column gives the quantity $prop = (vert_{\max} + 6 \cdot s_{vert})/\sqrt{n}$.

current gate is the bold dark arrow. The next triangle is darkly shaded and the new gate or in case of the *"cut-border split"*-operation the two new gates are drawn as light arrows. After the *"cut-border split"*-operation one gate is pushed together with its loop onto a stack and popped back after a *"close"*-operation.

The relative frequencies of the different cut-border operations are very similar for common triangle meshes. The *"new vertex"*-operation arises with a probability of 50% and the *"connect forward"*-operation with 45%. This can be explained from the fact that the best traversal strategy favors to cycle around the current vertex and close the neighborhood of the current vertex with a *"connect forward"*-operation. Thus we came up with the fixed Huffman codes in table 5.3, which are independent of the encoded triangle mesh and allow for a single pass encoding. *"Cut-border split"*-operations with index larger than 18 and less than -7 are encoded with the symbol "∞_i" followed by a 16 bit index. Similarly, the *"cut-border union"*-operations are encoded with the symbol "$\bigcup_l^i$" followed by a 4 bit loop index and a 16 bit vertex index. The choice of the gate in the smaller cut-border loop after a *"cut-border split"*-operation limits the maximum number of cut-border loops to the binary logarithm of the maximum number of cut-border edges in one loop. Therefore, four bits are sufficient for 16 bit vertex indices.

Table 5.4 shows for each model the maximum number of loops and the maximum

number of buffered vertices needed for mesh traversal. The values are averaged over several random choices of the initial triangle. As the values fluctuate significantly we add three standard deviations to the values such that 99.73% of the values are smaller than our estimation if we suppose a normal distribution. The maximum number of cut-border loops is comparably small and can safely be estimated by 100 for the first compression of a triangle mesh. To show that the maximum number of buffered vertices increases with $\sqrt{v}$ we divide the measured values plus three standard deviations by $\sqrt{v}$ and get values between three and six independent of the size of the model. Thus a save estimation for the size of the vertex buffer in a first compression of a triangle mesh is $10\sqrt{v}$.

5.2.6 Performance

The last column of table 5.2 shows that our approach allows compression of the connectivity of a triangle mesh to two bits per triangle[1] and less. The theoretical lower limit is 1.5 bits per triangle, which is achieved with uniform triangle meshes. To understand this fact let us neglect the *"split cut-border"*- and *"cut-border union"*-operations. Each *"new vertex"*-operation introduces one vertex and one triangle, whereas each *"connect"*-operation only introduces a triangle to the mesh. To arrive at a mesh with twice as many triangles as vertices, equally many *"new vertex"*- and *"connect"*-operations must appear. The Huffman code for the *"new vertex"*-operation consumes one bit and the *"connect"*-operations are encoded with two and three bits as still other operations must be encoded. If both *"connect"*-operations are equally likely, we get a compression to 1.75 bits per triangle. If on the other hand one *"connect"*-operation is completely negligible a compression to 1.5 bits is feasible. The optimal traversal strategy found in the previous section avoids *"connect backward"*-operation and therefore allows for better Huffman-encoding than the other strategies.

Table 5.2 also shows the compression and decompression speed in thousands of triangles per second measured on a 175MHz SGI/O2 R10000. The decompression algorithm clearly performs in linear time in the number of triangles with about 800,000 triangles per second. But the performance of the compression algorithm seems to decrease with increasing n. Actually, this impression is caused by the 1 MB data cache of the O2 which cannot keep the complete random access representation of the larger models, whereas the small cut-border data structure nicely fits into the cache during decompression. On machines without data cache the performance of the compression algorithm is also independent of n. The compression algorithm is approximately half as fast as the decompression algorithm. About 40% of the compression time is consumed by the random access representation of the triangle mesh in order to find the third vertex of the current vertex. The other ten percent are used to determine the loop and vertex index within the cut-border.

[1]The genus5 model consumes more storage as its genus forces five *"cut-border union"*-operations and the model is relatively small.

If our compression scheme is used to increase the bandwidth of transmission, storage or rendering, we can easily compute the break-even point of the bandwidth. The total time consumed by our compression scheme is the sum of the times spent for compression, transmission and decompression. The total time must be compared to the transmission time of the uncompressed mesh. Let us assume for the uncompressed representation an index size of 2 bytes, such that each triangle is encoded in 6 bytes. If we further use the estimation that the triangle mesh contains twice as many triangles as vertices, the break-even point computes[2] to a bandwidth of 12MBit/sec. Thus the compression scheme can be used to improve transmission of triangle meshes over standard 10MBit Ethernet lines. As our approach allows us to compress and decompress the triangle mesh incrementally, the triangle mesh can also be compressed and decompressed in parallel to the transmission process. Then even the transmission over a 100MBit Ethernet line could be improved.

5.3 Extensions

In this section we describe how to extend our method on non orientable triangle meshes. Additionally, we show how to encode attributes which are attached to vertex-triangle pairs.

5.3.1 Non Orientable Triangle Meshes

As we restricted ourselves to manifold triangle meshes, the neighborhood of each vertex must still be orientable even for non orientable meshes. From this follows that each cut-border loop must be orientable at any time: a cut-border part is a closed loop of adjacent edges. The orientation of one edge is passed on to an adjacent edge through the consistent orientation of the neighborhood of their common vertex. Therefore, only the *"split cut-border"*- and *"cut-border union"*-operations need to be extended as they introduce or eliminate cut-border parts. Both operations connect the current cut-border edge to a third vertex in the cut-border, which is either in the same or in another cut-border loop. The only thing, which can be different in a non orientable triangle mesh, is that the orientation of the cut-border around this third vertex is in the opposite direction as in the orientable case. Therefore, only one additional bit is needed for each "∞_i"- and "$\bigcup_l^i$"-operation, which encodes whether the orientation around the third vertex is different. During compression the value of the additional bit can be checked from the neighborhood of the third vertex. Previously a *"split cut-border"*-operation produced a new cut-border loop. In the new case with different orientations around the third vertex, the orientation of one of the new loops must be reversed and the loops are concatenated again as illustrated in figure 5.6. In a *"cut-border union"*-operation the cut-border loop containing the third vertex is concatenated to the current cut-border loop and in the

[2] with a compression rate of 400,000 triangles per second and a decompression rate of 800,000 triangles per second

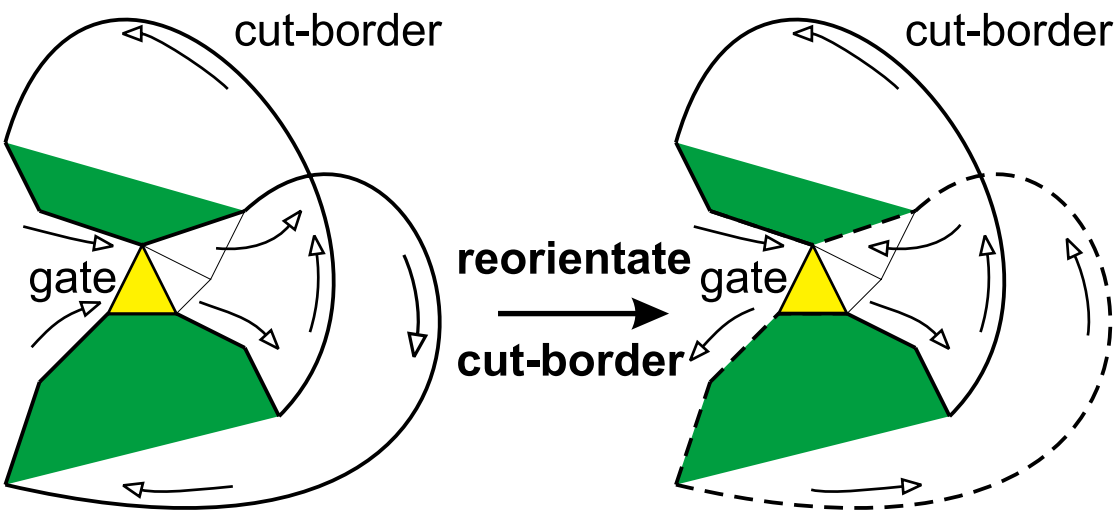


Figure 5.6: After some *"split cut-border"*-operations of a non orientable manifold half of the cut-border (drawn in dashed style) must be re-oriented and no new loop is generated.

new case the orientation of the cut-border loop with the third vertex is reversed before concatenation.

5.3.2 Corner Attributes

A lot of triangle meshes contain sharp creases that force the attachment of certain vertex attributes to triangle corners. See for example the genus5 model in figure 5.4, which contains a lot of creases. For each vertex on a crease exist two or more different normal vectors, which must be attached to the same vertex, which is contained in different triangles. Thus for models with creases it must be possible to store several different vertex normal vectors for different corners. Similarly, discontinuities in the color attribute force storage of several RGBA values within the corners. A simple solution to support corner attributes is to encode these attributes with every appearance of a triangle corner. This implies that the same corner attribute for one vertex may be replicated several times. On the other hand if we duplicated these vertices, which lay on creases, the vertex coordinates would be replicated.

With a small overhead we can do better and encode each vertex location and each corner attribute exactly once. Let us denote the collection of all vertex specific data as for example its coordinates with v and the different collections of the corner data with $v^{t_1}, v^{t_2}, \ldots$. As an example let us describe the encoding of v^t in the case of creases as illustrated in figure 5.7. The neighborhood of each vertex is split by the creases into several regions. Within each region there is exactly one corner attribute valid for the vertex and all triangles in this region. On the right side of figure 5.7 a cut-border vertex is shown during compression or decompression. We see that at any time it is sufficient to

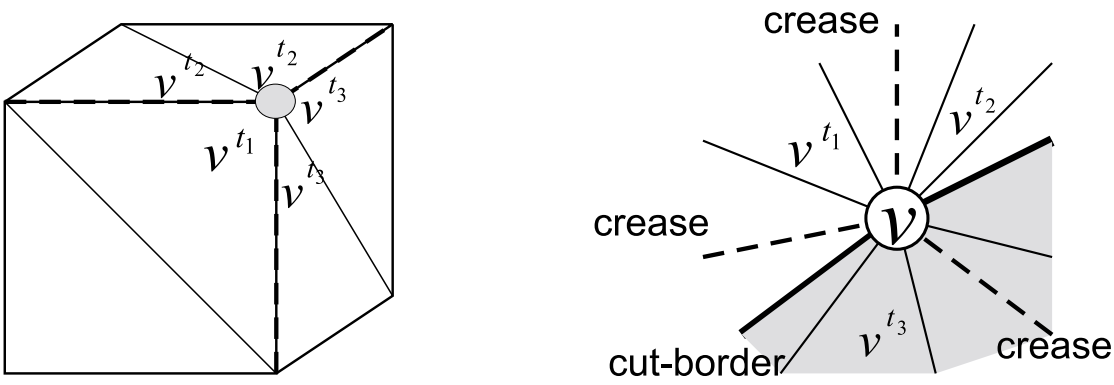


Figure 5.7: Creases divide the neighborhood of a vertex into regions. Each region contains the triangles with one corner attribute.

store besides the vertex data v two corner attributes $v^{t_{\text{left}}}$ and $v^{t_{\text{right}}}$ for each vertex within the cut-border. When a new triangle is added to the cut-border, the corner attributes of a vertex can only change, if the vertex is part of the gate and if the gate is a crease. If one of the corner attributes for example $v^{t_{\text{left}}}$ changes after an operation which adds a triangle, there are two possible cases. Either a new corner attribute is transmitted over the bit stream or the new attribute is copied from $v^{t_{\text{right}}}$.

To encode when a new corner attribute has to be transmitted we transmit one or two control bits after each operation, which adds a triangle to the gate. Two control bits are needed only for the *"connect"*-operations as the new triangle contains two cut-border edges. The control bits encode whether the affected cut-border edges are creases. Afterwards, for each cut-border vertex on a cut-border edge, which is a crease, we transmit one further bit which encodes whether a new corner attribute is transmitted or the attribute should be copied from the other corner attribute stored in the cut-border. If we denote the total number of inner edges with e and the total number of crease edges with e_c this approach results in an overhead of less than $e + 2e_c$ bits if encoded with the arithmetic flag encoding of section 3.2.2.

Chapter 6

Optimized Cut-Border Machine

In this section we describe four optimizations of the Cut-Border Machine. First we apply arithmetic coding to the Cut-Border Machine to allow for a more adaptive symbol encoding. We also describe how to use information from the inner part to improve the compression rates (see subsection 6.1). As the encoding of the mesh border is rather inefficient with the standard cut-border machine, subsection 6.2 describes how to decrease the consumed storage space for border operations and how to include non manifold borders. The third optimization addresses the running time of the compression and decompression algorithms. In subsection 6.3 a data structure is described, which allows to detect and perform all cut-border operations except of the *"cut-border union"*-operation in constant time during compression and decompression, respectively. This implies a linear running time in the number of triangles[1] for families of triangle meshes with a genus limited by a constant. Finally, we describe a modified version of the Cut-Border Machine with variable length encoding of the split indices in subsection 6.4 and prove that this scheme does not consume more than five bits per vertex for planar triangulations.

6.1 Arithmetic Coding

For maximum storage space compression arithmetic coding suggests itself. Firstly, we describe a very simple approach and show afterwards how to use conditional probabilities to improved the encoding further.

6.1.1 Adaptive Frequencies

The encoding with fixed Huffman Codes in table 5.3 is only optimal for certain relative frequencies of the symbols, i.e. when the *"new vertex"*-symbol arises in fifty percent of all cases, the *"connect forward"* in twenty five percent, ∞_2 in 12.5% and so on. But the *"connect forward"*-symbol arises in more than forty percent of all cases. The relative

[1] actually in the number of triangles plus the number of border edges

frequencies also differ slightly for different triangle meshes. If no further information is known, the relative frequencies are equivalent to the probabilities of the symbols.

We used adaptive arithmetic coding as described in section 3.2. As initial symbol count we used 32 and distributed the counts among the most frequent symbols according to the frequencies of the symbols in a set of standard meshes. In order to introduce new symbols an additional dummy symbol is kept with a constant counter of one. In table 6.1 on page 67 column "Arith." shows the compression results for this simple approach with adaptive frequencies. The resulting storage space is near to the entropy of the symbols.

6.1.2 Conditional Probabilities

Adaptive frequencies are only optimal if no further information besides the relative frequencies is given. But during compression with the cut-border machine we can use some information of the inner part - the so far decompressed mesh. The gate is incident upon two cut-border vertices. We want the connectivity compression scheme to be independent of the vertex locations and as simple as possible such that we can still achieve very fast compression and decompression. Therefore, we use only the order of the vertex onto which the oriented gate points (see figure 5.1), which we call the *end vertex*. The order of the end vertex is the number of incident triangles, which have already been encoded. This can easily be determined with a simple counter for each vertex.

The higher the order of the end vertex is, the higher is also the probability of a *"connect forward"*-operation and the lower is the probability of the *"new vertex"*-operation. To include this knowledge into the arithmetic coder, we make the probabilities of the symbols dependent on the order of the end vertex. For each order i and each symbol s we keep a counter $c_{s,i}$. With a counter c for the total number of symbols we can compute the probabilities $P(s \cap i) = c_{s,i}/c$. By summation over all symbols we can also compute the probabilities $P(i) = \sum_s P(s \cap i)$.

Before encoding the next symbol, the cut-border machine determines the order i of the end vertex. The arithmetic coder subdivides the current interval according to the conditional probabilities $P(s|i) = P(s \cap i)/P(i)$. Afterwards c and the counter of the encoded symbol for end vertex order i is incremented. In practice we unify all counters with $i > 8$ into one counter. Additionally, it turns out that the frequencies of all symbols besides the *"new vertex"* and the *"connect forward"* symbols are too small to allow efficient adaptive frequencies for the probabilities $P(s \cap i)$. Therefore, we only use for the *"new vertex"* and the *"connect forward"* symbols the conditional frequencies, where as we use for the remaining symbols only one counter which represents $P(s)$.

Table 6.1 compares the storage space consumption of the new approach with the IBM compression scheme [TR98] in column "IBM", the method proposed by Touma [TG98] in column "Touma" and with the original Cut-Border Machine from chapter 5 in column "fixed". In column "Arith." the arithmetic coder with adaptive frequencies is used and in column "Conditional" also the conditional probabilities are exploited. We can see that adaptive frequencies reduce the storage space consumed by the compressed

Model	Vertices	IBM $\frac{bits}{vtx}$	Touma $\frac{bits}{vtx}$	fixed $\frac{bits}{vtx}$	fixed $\frac{K\triangle}{sec}$	Arith. $\frac{bits}{vtx}$	Conditional $\frac{bits}{vtx}$	Conditional $\frac{K\triangle}{sec}$
blob	8036	4.8	2.4	3.1	613	2.5	1.9	176
tricerotops	2832	4.3	2.2	3.3	678	2.8	2.5	181
eight	766	3.8	0.6	3.3	710	2.8	1.2	179
shape	2562	2.2	0.2	3.0	706	2.2	0.3	198
beethoven	2655	4.8	2.4	3.6	664	2.9	2.7	169
engine	2164	3.8	1.2	4.3	653	3.0	2.3	158
dumptruck	11738	3.4	0.8	3.2	627	2.5	1.4	182
cow	3066	4.6	2.0	3.6	680	2.8	2.5	173
average		3.8	1.4	3.4	666	2.7	1.9	177

Table 6.1: Comparison of the storage space consumed by the connectivity in bits per vertex for the different approaches and the compression speed measured on a Pentium with 300MHz.

representation of the Cut-Border Machine with fixed codes to about 75%. The conditional probabilities improve especially the meshes with high regularity as for example the shape, the engine and the eight models. For the fixed and the conditional encoding also the compression speed in thousands of triangles per second are shown in table 6.1. Primarily the arithmetic coding and not the computation of the vertex orders decreases the compression speed by a factor of about 3.5. But even the monster model could be compressed in a quarter of a second.

We have to admit that our approach only beats the compression rates of Touma in case of the blob model. Compared to the IBM algorithm on the other hand only about half the storage space is consumed.

6.2 Optimized Border Encoding

The standard cut-border machine encodes each border edge of the mesh by a border symbol. In most triangular meshes the amount of border edges is rather small compared to the total number of encoded symbols. If the ratio of border edges increases, the standard approach of border encoding is rather inefficient. Two solutions have been proposed. Touma [TG98] connects the border edges of each border loop with a triangle fan to a dummy vertex and thus eliminates all border edges. Rossignac [Ros98] proposes to initialize the cut-border to the border loops and for this transmit the border loops at the beginning of the compressed representation, which can be done very space efficiently in $L \cdot \mathrm{lb}b$ bits, where L is the number of border loops and b the maximum number of border edges in one loop.

We present a different approach, which does not need any preprocessing and naturally allows for encoding of non manifold borders. During compression the Cut-Border Machine stores for each cut-border edge a flag, which tells whether this edge is an

encoded border edge or not. Any time a *"border"*-operation has to be encoded, the cut-border machine checks whether the next edge on the cut-border is marked as border edge. In this case the border operation is encoded with a *"connect forward"*-symbol. Similarly, the *"connect backward"*-symbol can be used if the previous edge is marked as border. As the *"connect forward"*-symbol is much more frequent than the *"border"*-symbol this approach reduces the consumed storage space from five bits per border edge to about two bits.

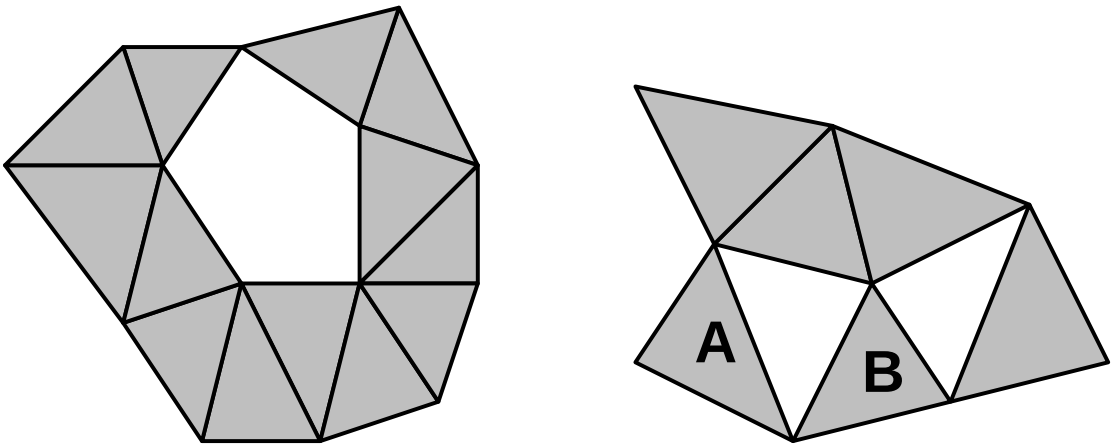


Figure 6.1: Non manifold borders. On the left: edge-connected component and on the right: vertex-connected components.

In order to allow for non manifold borders, we only have to modify the Cut-Border Machine slightly. Figure 6.1 shows on the left a triangle mesh with non manifold border, which is edge-connected. This kind of non manifold borders are handled by the Cut-Border Machine without any modifications. The non manifold connections are just encoded with connect and union operations.

The example on the right of figure 6.1 shows non manifold borders of a triangle mesh with components which are only connected through vertices. So far the Cut-Border Machine did cut the different components apart by doubling the non manifold vertices. Therefore the vertex data of these vertices also was doubled and some proximity information was lost. To avoid these drawbacks, two modifications of the Cut-Border Machine are required. Firstly, the Cut-Border Machine keeps the border edges of all processed edge-connected components. In this way non manifold connections between edge-connected components can be encoded with union symbols. The remaining problem arises at the initial triangles of each edge-connected component. Recall the triangle mesh on the right of figure 6.1. Let us assume that the Cut-Border Machine encodes the triangle labeled **A** first, followed by triangle **B**. Each triangle constitutes a complete edge-connected component of the mesh and is encoded with one bit telling that a further edge-connected component follows, an implicit initial triangle operation and a sequence of three border operations. The initial triangle of component **B** has to be connected to a vertex of component **A**. As each of the three vertices of the initial

triangle can be connected to another edge-connected component, we introduce three new symbols: initial triangle with one non manifold vertex denoted by the symbol Δ_1, initial triangle with two non manifold vertices Δ_2 and correspondingly Δ_3. For each of the non manifold vertices a loop index and a vertex index is encoded as in the case of a union operation. The first vertex of the initial triangle is always chosen, such that the non manifold vertices of the initial triangle are at the end.

		border			total $\frac{bits}{tgl}$		
Model	Vertices	total	encoded	percent	fixed	border	conditional
beethoven	2655	274	12	4%	3.6	3.4	2.7
helicopter	1972	743	140	19%	4.3	3.5	3.1
monster	25118	272	37	14%	3.3	3.3	2.2

Table 6.2: Improvement with the optimized border encoding.

The optimized border encoding reduces the consumed storage space of the original Cut-Border Machine especially for models with high border fraction as shown in table 6.2. The table shows the total number of border edges in column "total", the number of border edges explicitly encoded with the border symbol in column "encoded" and the resulting percentage. In the column "fixed" the storage space consumed without the border optimization is tabulated. Column "border" gives the storage space only with the border optimization and column "conditional" the storage space for border optimization and conditional probabilities. In case of the helicopter model even 0.8-bits per vertex can be saved just by applying the border optimization.

6.3 Linear Time Cut-Border Data Structure

So far the data structure for the cut-border has been implemented as linked lists and the *"cut-border split"*-operation could neither be encoded nor decoded in linear time yielding a worst case running time of at least $O(n \ln n)$. The advantage of the linked list data structure has been that the gate could be updated after each operation arbitrarily. In this section we have to give up the strategy that always the smaller cut-border loop is encoded next. But this was only important to keep the number of cut-border loops small. In terms of compression rates we only loose in the encoding of the rare *"cut-border union"*-operations as we need the same number of bits to encode the loop index as for the vertex index. We present a new data structure, which ensures constant time detection and updates after each operation except of the *"cut-border union"*-operation. For the *"cut-border union"*-operation this cannot be possible as the minimal storage space consumption for non planar meshes is of the order $O(v \ln v)$ (compare section 4.3) and therefore the running time must at least be evenly bad.

Figure 6.2 shows the new data structure. It consists of a vertex stack, a loop stack and two markers. The vertex stack is extended by the markers p_s and p_e defining the current loop. On the loop stack marker pairs of loops, which have been pushed during

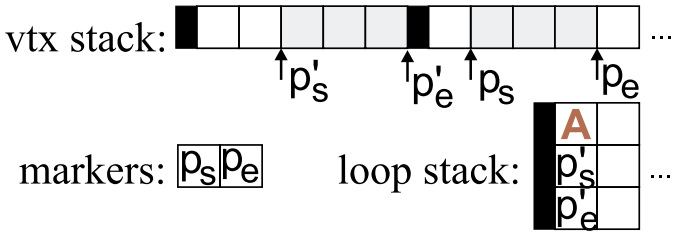


Figure 6.2: Cut-border data structure for constant time updates consists of two stacks and two pointers.

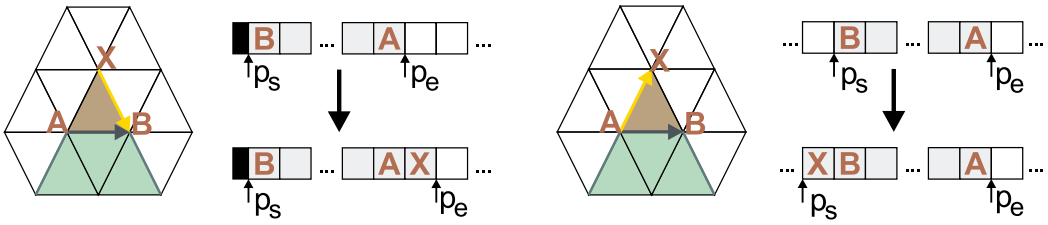
a split operation, are stored together with the split vertex. The current gate is always the edge between the vertex before the p_e marker and the vertex after the p_s marker. In figure 6.2 the different loops on the vertex stack are visually separated with black blocks, just for the convenience of the reader.

Figure 6.3 shows how the data structure is updated after each of the five operations with constant update time.

new vertex: There are two possible updates of the cut-border data structure in case of a *"new vertex"*-operation. The first update in figure 6.3 a) is always possible. The new vertex X is just appended to the vertex stack and the p_e marker incremented. With this update the new gate is implicitly chosen to be XB as in the optimal traversal strategy (see figure 5.5). The only reason for the second update after the *"new vertex"*-operation is that the vertex stack might become fragmented with the exclusive use of the first update, as the frequent *"connect forward"*-operations delete cut-border vertices at the beginning of the current loop, i.e. at the p_s marker. With alternating *"new vertex"*- and *"connect forward"*-operations the current loop moves to the right of the vertex stack. This movement is no severe problem, as the number of vertices limits the maximum size of the vertex stack because the *"new vertex"*-operation is the only one, which increases the vertex stack. But if the size of the cut-border data structure should be kept small, one can also use the second update in figure 6.3 b). Here the empty places at the beginning of the current loop are filled with the new vertices. This implicitly forces the new gate to the other free edge of the newly added triangle and therefore might cause worse compression rates. The second update should only be used if the cut-border data structure needs to be kept as small as possible. If used, it is applied whenever the place before the p_s marker is empty.

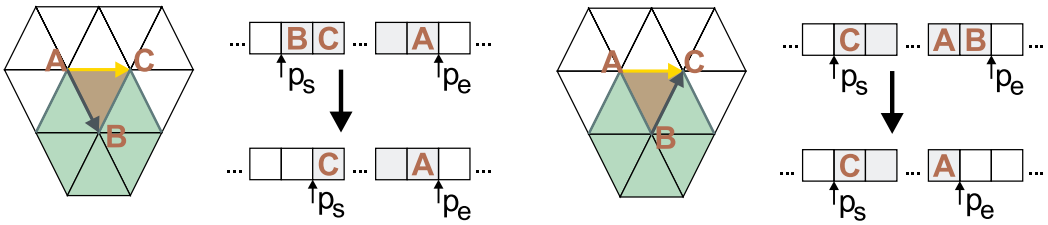
connect forward: During the *"connect forward"*-operation (figure 6.2 c) the vertex B is removed from the beginning of the current loop by moving the p_s marker one position to the right. The empty vertex location can be filled during the next new vertex operation if the second update is used. The gate is implicitly chosen to be AC as in the optimal traversal strategy.

connect backward: The *"connect backward"*-operation (figure 6.2 d) just pops one vertex from the stack, moves p_e one position to the left and implicitly chooses the gate to be AC.



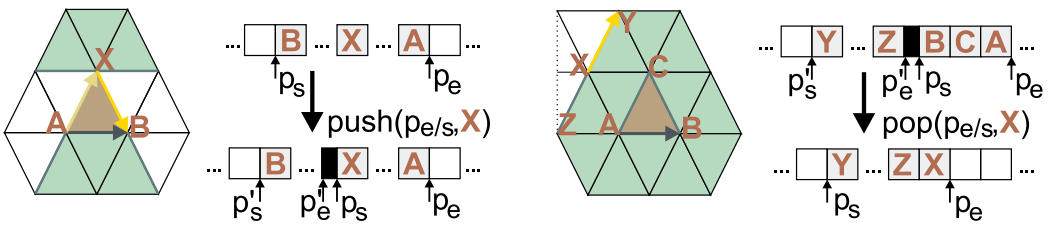
a) *: append

b) *: insert



c) →: delete previous

d) ←: delete next



e) ∞: push loop

f) ∇: pop loop

Figure 6.3: Update of the optimized data structure after the five cut-border operations with constant update time.

split cut-border: During a *"split cut-border"*-operation (figure 6.2 e) the current loop is split into two loops by setting p_s to p'_s and introducing two new markers p'_e and p_s before the split vertex X. The primed markers together with the split vertex X are pushed onto the loop stack such that the pushed loop can be restored after the right loop has been encoded. In order to find X in constant time we store with each mesh vertex a pointer to the corresponding cut-border vertex. This works as no operation except of the *"cut-border union"*-operation invalidates any of the pointers, i.e. as long as a cut-border vertex exists, it is at the same vertex stack location. The gate location of the right loop after the split operation is AX and the pushed gate location of the other loop is XB.

close: The *"close"*-operation (figure 6.2 f) eliminates the current loop of three vertex indices from the vertex stack. If the loop stack is empty, the triangle mesh has been completely encoded / decoded. Otherwise the top loop on the loop stack is popped together with the corresponding split vertex, which is inserted after the popped p_e marker and the marker is moved once to the right. Please notice, that the X vertex will be on the same vertex stack location as during the encoding of the removed loop and therefore the pointer from the mesh vertex to the cut-border vertex needs no update.

Finally, the computation of the split indices can be performed by simple pointer arithmetic in constant time[2]. The *"cut-border union"*-operation can be detected in constant time as the location of the union vertex on the stack is stored with the mesh vertex and can be compared to the current p_s and p_e. The update of the *"cut-border union"*-operation basically adds the loop containing the third vertex to the current loop and is performed in three steps. Suppose the current loop consists of the vertices $BCDEFGA$ with gate AB and the third vertex X of the next triangle is in the loop $IJXKLMNH$ with gate HI. Then the update is performed as follows:

- The loop $IJXKLMNH$ is temporarily stored in a buffer. All vertices on the vertex stack between the last vertex H of the loop $IJXKLMNH$ and the first vertex B of the current loop are moved $8 = length(IJXKLMNH)$ spaces to the left. All of the pointers stored with the mesh vertices, which correspond to one of the moved vertices, are decremented by 8.

- The freed 8 places on the vertex stack before the current loop are filled with $KLMNHIJX$. The corresponding pointers in the mesh vertices are updated accordingly.

- Finally, a duplicate of X is pushed onto the vertex stack.

The resulting current loop is $KLMNHIJXBCDEFGAX$ with gate XK, what corresponds to the unified cut-border loops. The duplication of the vertex X forces the extension of the pointers in the mesh vertices to lists of pointers. As this extension is

[2]Here we assume that the number of vertices in the mesh can be represented by the pointer/integer format of the used computer. This must be the case as we need to store the mesh itself somehow.

only needed if *"cut-border union"*-operations arise, i.e. if the genus of the encoded mesh is larger than zero, we can state the following theorem:

Theorem 6.1 *Planar triangulations without holes and triangle meshes with genus zero can be encoded in linear time in the number of triangles with the Cut-Border Machine.*

6.4 Linear Space Limit for Planar Triangulations

The results in this section are valid for planar triangulations as introduced in section 4.1 as well as for triangle meshes of genus zero. In both cases no *"cut-border union"*-operation can arise. The main problem in achieving a linear storage space for the cut-border operation symbols is the encoding of the split indices in linear space. There are two important ideas to tackle this problem. Firstly, the indices are encoded with variable length, such that an index i is encoded with no more than a constant number times the binary logarithm of i. The second idea is that split indices defining vertices on the current cut-border loop before the gate are encoded with negative indices. In this way all split operations, which cut away small parts of the current loop also consume few bits.

6.4.1 Upper Bound on Index Coding

Theorem 6.2 *For any planar triangulation or triangle mesh of genus zero with v vertices the indices of the split operations in the cut-border representation can be encoded with less than $\alpha = 1.87v$ bits.*

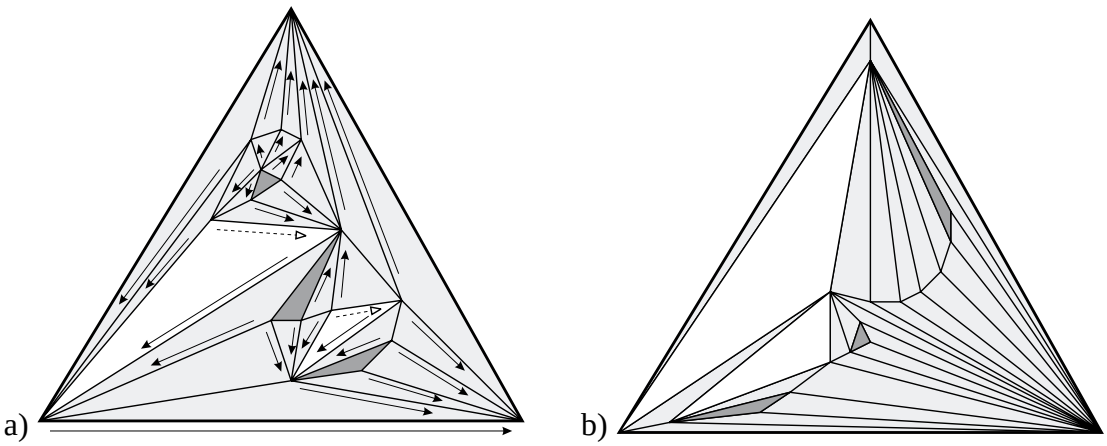


Figure 6.4: a) Example for a planar triangulation with three border vertices. The initial gate is the bottom boundary edge. The gate(s) in each triangle are depicted with an arrow. b) Planar triangulation for rearranged cut-border symbols.

Proof: The proof is performed for planar triangulations only but is in principle the same for triangle meshes of genus zero as they become planar triangulations with three border edges if one triangle is removed. We do also avoid all border symbols by initializing the cut-border with the border loop of the planar triangulation. For this only the length of the border loop is transmitted in advance.

For the coding of the indices of the ∞-symbols we want to use the coding scheme of section 3.3 figure 3.3 c). The index coding does not depend on the order of the encoded symbols. Therefore we can rearrange the symbols in the following manner. Let us take the example in figure 6.4 a). The cut-border is initialized to the three border edges of the triangulation and the initial gate is the bottom boundary edge. The gate after each operation is shown as arrow in the newly added triangle. Split triangles are shaded white and contain two new gates. The pushed gate is shown as dashed arrow. Loop closing triangles are shaded dark and contain no further gate. The cut-border string is

$$****\rightarrow****\rightarrow*\infty_{-4}*\rightarrow**\infty_{3}\leftarrow\nabla\rightarrow\leftarrow\nabla**\rightarrow*\rightarrow\rightarrow\rightarrow\rightarrow\nabla.$$

Then we rearrange it by extracting all $*$-symbols to the front:

$$**********\rightarrow\rightarrow\infty_{-4}\rightarrow\infty_{3}\leftarrow\nabla\rightarrow\leftarrow\nabla\rightarrow\rightarrow\rightarrow\rightarrow\nabla.$$

By going through the cut-border symbols in reverse order, we can keep track of the lengths of the cut-border loops and determine new indices for the *"split cut-border"*-operations. At a *"close"*-operation the current length is pushed on a stack and a new length of three is generated. At a *"connect forward"*- and a *"connect backward"*-operation the current loop length is increased by one. At each *"split cut-border"*-operation a new split index is determined for the operation symbol. The absolute value of the new index is the smaller length among the current length and the latest pushed length minus one. It is positive if the current length is smaller, negative otherwise. The split operation is reversed by popping one length, decrementing it by one and adding it to the current length. Finally, each new vertex operation decreases the current length by one, such that if the beginning of the rearranged cut-border symbol sequence is reached, exactly one length remains, which is equal to the length of the border loop of the planar triangulation. Thus the cut-border sequence with the new split indices is

$$**********\rightarrow\rightarrow\infty_{-7}\rightarrow\infty_{3}\leftarrow\nabla\rightarrow\leftarrow\nabla\rightarrow\rightarrow\rightarrow\rightarrow\nabla.$$

By construction of the new split indices the resulting sequence is a valid traversal description for the cut-border machine. Furthermore the absolute values of the new indices of each split symbols can only be greater or equal to the original indices as during the reverse tracking of the lengths the missing *"new vertex"*-operations can only increase the tracked lengths.

There actually exists a planar triangulation producing the rearranged sequence as shown in figure 6.4 b), but this is not important for this proof and requires in general further mathematical techniques. It is though important that we can now restrict our considerations to the situation when at the beginning of the encoding one cut-border loop

with all vertices is built, which is recursively split. Let $\mathcal{L}(v)$ be the maximal storage space consumed by the indices of *"split cut-border"*-operations arising during the encoding of a cut-border loop with v vertices. Then all possible split operations performed on the loop yield the recursive formula for $\mathcal{L}(v)$

$$\mathcal{L}(v) = \max_{i=2}^{i=\left\lceil \frac{v}{2} \right\rceil - 1} \left\{ \mathcal{I}_c(i) + \mathcal{L}(i+1) + \mathcal{L}(v-i) \right\}. \tag{6.1}$$

$\mathcal{L}(v)$ is the maximum storage space of all possible split operations. The absolute value of the split index i runs from two to $\left\lceil \frac{v}{2} \right\rceil - 1$ and can be encoded with $\mathcal{I}(i)$ bits. Additionally, the maximum storage space of the two remaining loops $\mathcal{L}(i+1)$ and $\mathcal{L}(v-i)$ have to be included. These are the same for positive and negative split indices. A loop with three or four vertices cannot be split further, for the split of a loop with five vertices there is only one possibility and a loop with six vertices can be split with ∞_2 or ∞_{-2}. Therefore, it holds

$$\mathcal{L}(3) = \mathcal{L}(4) = \mathcal{L}(5) = 0 \qquad\qquad \mathcal{L}(6) = 1. \tag{6.2}$$

To proof the theorem we have to show the validity of the relation $\mathcal{L}(v) \leq \alpha \cdot v$ for all v. It is obviously true for $v < 7$ if $\alpha \geq 1$. As equation 6.1 contains the storage space for the index i, we have to prove for the remaining values of v an even stronger upper bound

$$\mathcal{U}(v) \stackrel{\text{def}}{=} \alpha \cdot v - (\beta \mathrm{lb} v + \gamma), \tag{6.3}$$

i.e. we want to prove the relation

$$\forall v \geq 7 : \mathcal{L}(v) \leq \mathcal{U}(v). \tag{6.4}$$

To abbreviate the proof we just guess the values of α, β and γ to be $1.87, \beta_c = 2.03$ and 4.92. The values were chosen in a way that $\mathcal{U}(6) = 1$. But we still have to consider all special cases where i assumes the values $2, 3$ or 4 and the loop length v the values $3, 4$ or 5.

Cut-border loops of length six to nine can be split into two loops of length less than six. For the $6, 7, 8$ and 9-vertex loops we introduce special coding schemes for the indices of split operations performed on these loops. This is possible, as the cut-border machine knows the length of the current loop during encoding and decoding. For a split operation on a loop with $l \geq 6$ vertices, there are $l - 4$ different split indices. Using arithmetic coding the different indices can be encoded with $\mathrm{lb}(l-4)$ bits. A seven-vertex loop can be split in three different ways (∞_2, ∞_{-2} and ∞_3). The three cases can be encoded with $\mathrm{lb}3 < 1.585$ bits. The resulting loops are of length less or equal five and therefore $\mathcal{L}(7) < 1.585$ as no further indices need to be encoded. The eight-vertex loop has four possible splits, which can be encoded in two bits. As the eight-vertex loop can be split into a three-vertex and a six-vertex loop, an additional bit might be needed. Thus $\mathcal{L}(8) = 2 + \mathcal{L}(6) = 3$. Finally, the nine-vertex loop split index consumes $\mathrm{lb}5$ bits

and we get $\mathcal{L}(9) = \text{lb}5 + \mathcal{L}(7) < 3.91$. Gathering these cases we state

$$\begin{aligned} \mathcal{L}(7) < 1.585 &< \mathcal{U}(7) > 2.4 \\ \mathcal{L}(8) = 3 &< \mathcal{U}(8) > 3.8 \\ \mathcal{L}(9) < 3.91 &< \mathcal{U}(9) > 5.3 \end{aligned}$$

Next we consider the inductive step for the cases, where i is less or equal four in equation 6.1 and show

$$\forall i \in \{2,3,4\} : \mathcal{I}_c(i) + \mathcal{L}(i+1) + \mathcal{L}(v-i) \leq \mathcal{U}(v) \tag{6.5}$$

In the following we apply equations 3.8, 6.2 and 6.3 as denoted above the less or equal signs

$$\begin{aligned} \forall i \in \{2,3,4\} : & \\ & \mathcal{I}_c(i) + \mathcal{L}(i+1) + \mathcal{L}(v-i) \\ \overset{(3.8,6.2)}{\leq} & \beta_c \text{lb}(i+1) + 1 + \mathcal{L}(v-i) \\ \overset{(6.4,6.3)}{\leq} & \beta_c \text{lb}(i+1) + 1 + \alpha \cdot (v-i) - (\beta_c \text{lb}(v-i) + \gamma) \\ \overset{(6.3)}{=} & \mathcal{U}(v) - i\alpha + \beta_c \text{lb}(i+1) + 1 + \beta_c \text{lb}\frac{v}{v-i}. \end{aligned}$$

From the last expression we learn that equation 6.5 holds true, iff anything in the last expression besides $\mathcal{U}(v)$ is less or equal zero

$$\begin{aligned} (6.5) & \iff \\ \forall i \in \{3,4,5\} : i\alpha &\geq \beta_c \text{lb}(i+1) + 1 + \beta_c \text{lb}\frac{v}{v-i} \\ & \iff \\ \forall i \in \{2,3,4\} : v &\geq i/\left(1 - (i+1)/2^{\frac{i\alpha-1}{\beta_c}}\right) \\ & \impliedby \\ v &> 9. \end{aligned}$$

The last step was performed by plugging in the values for α, β_c and i and proves equation 6.5.

With all the preliminaries we can finally attack equation 6.4 and prove it by induction. That is we validate equation 6.4 for v under the assumption that equation 6.4 holds true for all $v' < v$. We start with equation 6.1:

$$\begin{aligned} \mathcal{L}(v) &= \max_{i=2}^{i=\lceil \frac{v}{2} \rceil - 1} \{\mathcal{I}_c(i) + \mathcal{L}(i+1) + \mathcal{L}(v-i)\} \\ &\overset{(6.5)}{\leq} \max \left\{ \mathcal{U}(v), \max_{i=5}^{i=\lceil \frac{v}{2} \rceil - 1} \{\mathcal{I}_c(i) + \mathcal{L}(i+1) + \mathcal{L}(v-i)\} \right\} \end{aligned}$$

If $\mathcal{U}(v)$ is the maximum, equation 6.4 holds true and therefore we do neglect the outer maximum in what follows. We can now plug in the inductive assumption:

$$\begin{aligned}\mathcal{L}(v) &\overset{(6.4)}{\leq} \max_{i=5}^{i=\lceil \frac{v}{2} \rceil - 1} \{\mathcal{I}_c(i) + \mathcal{U}(i+1) + \mathcal{U}(v-i)\} \\ &\overset{(3.8,6.3)}{\leq} \mathcal{U}(v) + 1 + \alpha - \gamma + \beta_c \max_{i=5}^{i=\lceil \frac{v}{2} \rceil - 1} \left\{ \mathrm{lb} \frac{v}{v-i} \right\}.\end{aligned}$$

We skipped some simple algebra in the second step. The term inside the logarithmic function evaluates always to a value greater one and less than two, because i is always less than $v/2$. Therefore the logarithmic expression is always less than one and we get

$$\mathcal{L}(v) \leq \mathcal{U}(v) + 1 + \alpha - \gamma + \beta_c \leq \mathcal{U}(v),$$

what proves equation 6.4 and together with equation 6.2 the theorem.

□

For a better variable length index coding scheme with $\beta = \beta_{\min}$ theorem 6.2 can be improved to 1.54 bits per vertex with only one change in the proof: the special coding must also be applied to ten-vertex loops.

6.4.2 Coding of Operation Symbols

Now that we know how to encode the split indices we describe the encoding of the operation symbols, which allows for an encoding of planar triangulations with less than five bits per vertex. There are $t = 2v - b - 2$ triangles in a planar triangulation with b border edges and the same number of symbols to be encoded. The *"new vertex"*-operation is encoded with one bit. The total number of $v - b$ `*`-symbols contribute less than v bits to the overall storage costs. There are two further constraints, which can be exploited.

1. If the current cut-border loop has three vertices only the *"new vertex"*- and the *"close"*-operations are possible.

2. After a new vertex operation no *"connect backward"*-operation may follow.

As the *"close"*-operation may only arise in the situation of the first constraint, it can be encoded with one bit, i.e. if the current cut-border loop contains only three vertices, one bit encodes whether a *"close"*- or a *"new vertex"*-operation follows.

To fully exploit both constraints, we define the constant τ as the number of bits, which are consumed for encoding a triangle not introduced by a new vertex operation. With this definition the operation symbols without the split indices are encoded with less than $(\tau + 1)v$ bits. Table 6.3 shows all possible cases of subsequent symbols, which might arise at the current gate, and the number of bits, that may be consumed by each case in the column "bound". Here we included the observation that each ∞-operation

forces one ∇-operation, which can be encoded with one bit. Thus each ∞-operation may consume $2\tau - 1$ bits.

From the maximum number of bits b in the column "bound", the frequency ν of each case can be computed from 2^{-b} (compare equation 3.3 on page 34). The frequencies of all cases in table 6.3 must sum up to one. This condition yields an equation for τ and τ computes to 2, which is rather an accident but simplifies the encoding of symbols. In table 6.3 the cases are arranged in rows, such that each row contains all cases with the same number of bits, which is denoted in the first column of table 6.3. For each bit number starting with two bits, there are exactly two cases. Thus each case is encoded in two parts. First the row r is encoded with $r - 1$ one bits followed by a zero bit and then one bit selects the column. Together with theorems 6.1 and 6.2 we can conclude the whole discussion with the following theorem.

Theorem 6.3 *Planar triangulations and closed triangle meshes of genus zero with v vertices can be encoded with the cut-border encoding scheme in linear time in v with less than $4.92v$ bits.*

If a better variable length index coding scheme is found an upper bound of 4.54 bits per vertex can be shown.

As the coding of the symbols without the split indices consumes only 3 bits per vertex and an optimal coding consumes at least 3.245 bits, it is worth while to check, whether the split indices can be encoded even better. The best coding scheme we can imaging, which does not allow for a linear time coding algorithm, exploits the knowledge of the current loop length: we use the loop storage space $\mathcal{L}(3 \ldots 6)$ from equation 6.2. For the loop of length $v > 6$ we assume $\mathcal{L}(v)$ is the same for all possible split operations. We distinguish the split operation with index $i = 2 \ldots v - 3$, such that we do not need to handle a sign. Finally, we calculate $\mathcal{L}(v)$ recursively from the arithmetic coding equation, which sets the sum of the frequencies of all possible split operations

bits	case	bound	case	bound
2	$\leftarrow$	τ	$\rightarrow$	τ
3	∞	$2\tau - 1$	$*\rightarrow$	$\tau + 1$
4	$*\infty$	2τ	$**\rightarrow$	$\tau + 2$
5	$**\infty$	$2\tau + 1$	$***\rightarrow$	$\tau + 3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 6.3: Assignment of bit consumption for all possible combinations of subsequent symbols.

for one loop length v to one:

$$\forall v > 6 : 1 = \sum_{i=2}^{v-3} 2^{-(a-\mathcal{L}(i+1)-\mathcal{L}(v-i))}.$$

We calculated the fraction $\mathcal{L}(v)/v$ for $v = 3 \ldots 100$ and the resulting plot converges to 1.15 and crosses 1. Thus it seems to be impossible to encode a planar triangulation with less than four bits per vertex with the Cut-Border Machine encoding scheme.

Chapter 7

Edgebreaker

In this section we describe the Edgebreaker connectivity encoding scheme, which was developed by Rossignac [Ros98] independent of the Cut-Border Machine, and is quite similar to the Cut-Border Machine encoding. Therefore we only briefly illustrate the differences between the Edgebreaker and the Cut-Border Machine in section 7.1. As the decoding of the encoded Edgebreaker strings is not trivial and quite different to the Cut-Border Machine decoding, section 7.2 describes a simple and fast reverse decoding technique developed by Isenburg [IS99b]. Finally, we improve in section 7.3 the Edgebreaker encoding in case of planar triangulations to come closer to the theoretical limit of 3.245 bits per vertex.

7.1 From the Cut-Border Machine to the Edgebreaker

cut-border		edgebreaker	
name	symb.	name	symb.
new vertex	*		C
connect forward	$\rightarrow$	right	R
connect backward	$\leftarrow$	left	L
split cut-border	∞	split	S
close cut-border	∇	end	E
cut-border union	$\bigcup$	merge	M, M'

Table 7.1: Translations between the cut-border and the Edgebreaker symbols.

The edgebreaker encoding scheme is very similar to the cut-border machine and differs in the following three aspects:

- The initial cut-border for compression is initialized to the set of boundary loops of the encoded mesh. This avoids all *"border"*-operations.

- The split indices are not encoded but instead the *"close"*-operation is encoded with a unique symbol.
- There are two different operation symbols for the *"cut-border union"*-operation. The M operation encodes the union of the current cut-border loop with a mesh boundary loop and the M' operation represents the union of the current loop with a different cut-border loop.

Table 7.1 gives a complete list of translations between the cut-border symbols and the Edgebreaker symbols. Thus the cut-border string for the planar triangulation in figure 6.4 a) would transform to the following edgebreaker string:

$$\begin{aligned}&****\rightarrow****\rightarrow*\infty_{-4}*\rightarrow**\infty_{3}\leftarrow\nabla\rightarrow\leftarrow\nabla**\rightarrow*\rightarrow\rightarrow\rightarrow\rightarrow\nabla \quad\mapsto\\ &CCCCRCCCCRCSCRCCSLERLECCRCRRRRE\end{aligned}$$

By encoding the C-symbol with one bit and all other symbols except the M symbols with three bits, the Edgebreaker scheme allows to encode any planar triangulation with no more than four bits per vertex[1]. In [KR99] the upper bound for the storage space is improved to 3.67 bits per vertex.

7.2 Spirale Reversi Edgebreaker Decoding

The decoding of the Edgebreaker strings is not obvious in a forward traversal of the string as the split operations cannot be performed without further computations. Rossignac proposed two different methods for the decoding with a lookahead method in [Ros98] and a two pass decoding in [JR99]. But the simplest method has been developed by Isenburg in [IS99b]. The Edgebreaker string is interpreted in reverse order. For this the tailing E-symbol is deleted from the string, an S-symbol is added to the beginning and the string is reversed. The string representation of the sample in figure 6.4 a) is mapped to

$$\begin{aligned}&CCCCRCCCCRCSCRCCSLERLECCRCRRRRE \quad\mapsto\\ &RRRRCRCCELRELSCCRCSCRCCCCRCCCCS.\end{aligned}$$

The tailing S-symbol marks the end of the string representation.

The decoding algorithm knows, that the encoding ended with an E-operation. Therefore, it recreates the triangle encoded by the tailing E-operation and initializes the cut-border to one single loop surrounding this triangle. The gate location is chosen arbitrarily and dummy vertex indices are used for all newly introduced vertices.

Then the decoding algorithm iterates through the symbols of the string representation and performs all encoded operations in an inverse fashion. This is clear for the C-, R- and L-operations: in figure 5.5 on page 59 interpret the white triangles as the so far decoded triangles, the dark triangle as the currently decoded triangle, the light arrow

[1]Each symbol introduces one triangle. There are no more than twice as many triangles as vertices. Each vertex corresponds to exactly one C symbol. This sums up to $v + 3v = 4v$ bits.

as the current gate location and the dark arrow as the new gate location after the next triangle has been decoded. After each C-operation the neighborhood of the new vertex is completely decoded and a new final vertex index is assigned to this vertex. In this way the vertices are assigned indices in the reverse order in which they have been encoded.

For planar triangulation coding it only remains to explain the decoding of the E- and S-operations[2]. Each time an E-symbol is encountered, the current loop is pushed onto a stack together with the current gate location and a new loop is generated with a single triangle and an arbitrary gate location. When an S-symbol is found, one loop together with its gate location is popped from the stack and is merged together with the current loop at the current gate location inserting a triangle as depicted in figure 5.5 d). In figure 5.5 d) the left light arrow represents the gate of the popped loop in the decoding algorithm and the pushed loop in the encoding algorithm. During the merging the two dummy vertices of the different loops, where the two gates touch, are identified in all incident triangles. The new gate location is set according to the dark arrow in figure 5.5 d). If an S-symbol is found, when the stack of loops is empty, this S-symbol is the marker of the end of the string and decoding is complete. This decoding algorithm can easily be implemented in linear time and we can state the following theorem.

Theorem 7.1 *The connectivity of planar triangulations with v vertices can be encoded with a unique string of length $2v$ over five different symbols in linear time in v, from which the original connectivity can be decoded also in time linear in v.*

7.3 Towards Optimal Planar Triangulation Coding

The optimal encoding of planar triangulations is strongly correlated to the optimal encoding of triangle meshes with low genus. But in practical applications the triangle meshes are quite regular such that compression results can be achieved that lay below the theoretical limit as shown in section 6.1. But in the coding theory of planar graphs the optimal encoding of planar triangulations is very important. In the next subsection we describe an improved encoding of the edgebreaker strings that exploits constraints in a forward traversal of the edgebreaker strings and in the second subsection in this section we exploit constraints in a reverse traversal.

7.3.1 3.557 Bits per Vertex Encoding of Edgebreaker String

In this subsection we use two constraints of the edgebreaker strings to improve the $4v$ bit encoding to $\mathrm{lb}3 + 2 \approx 3.586$ and then to 3.557 bits per vertex.

The first constraint is that after a C-symbol neither an L- nor an E-symbol may follow, as otherwise the two successive symbols would encode the same triangle twice. We can use this constraint in the following manner. First we notice that the C-symbols constitute half of all the symbols and therefore should be encoded with one bit or in an

[2]For the decoding of the M and M'-operations please refer to the original work [IS99b]

arithmetic setting with a frequency of $\frac{1}{2}$. Next we assume that all other symbols may consume the same number of bits τ and therefore correspond to the same frequency $\nu_\tau = 2^{-\tau} \in [0,1]$. Table 7.2 shows the different possible cases (compare table 6.3 for the Cut-Border Machine cases), when the first constraint is exploited. If we use arithmetic coding, the frequencies for the different cases must sum up to one, what results in the following equation:

$$1 = 4\nu_\tau + 2\nu_\tau \sum_{i \geq 1} \frac{1}{2^i} = 6\nu_\tau. \tag{7.1}$$

From this ν_τ computes to $\frac{1}{6}$ and τ to $\mathrm{lb}6 < 2.585$. As there are v C-symbols and v symbols of other type and the C-symbols consume 1 bit and the others τ bit, we end up with less than 3.585 bits per vertex. Coding and decoding of the symbols is also very simple. The unit interval is subdivided into 6 equal sized sub-intervals assigned to the cases R, L, S, E, C^+R, C^+S. In the C^+-cases the number of C-symbols is encoded with the same number of one bits followed by a zero bit.

The second constraint, which has not been considered yet, makes use of the knowledge of the length of the cut-border. The observation is that during encoding the current cut-border loop is at least of length three. Thus two successive C symbols increase the length to at least five and a following R will reduce the length to not less than four, what prohibits a following E symbol, as this can only appear if the current cut-border loop has exactly length three. To take this constraint into account, we introduce the concept of the *conditional unity*. Let us introduce this concept with the example of the first constraint. For the first symbol there are the five possibilities C, R, L, S or E. But after a C-symbol has been encoded, only three possibilities are left (C, R and S). Thus under the condition of a preceding C-symbol the unity is split into the frequencies for the symbols C, R and S. We can re-formulate the said as follows

$$1 \;=\; 4\nu_\tau + \frac{1}{2}\mathbf{1}_C \tag{7.2}$$

case	bits	freq.	case	bits	freq.
L	τ	$\nu\tau$	E	τ	$\nu\tau$
R	τ	$\nu\tau$	S	τ	$\nu\tau$
CR	$\tau+1$	$\frac{1}{2}\nu\tau$	CS	$\tau+1$	$\frac{1}{2}\nu\tau$
CCR	$\tau+2$	$\frac{1}{4}\nu\tau$	CCS	$\tau+2$	$\frac{1}{4}\nu\tau$
$CCCR$	$\tau+3$	$\frac{1}{8}\nu\tau$	$CCCS$	$\tau+3$	$\frac{1}{8}\nu\tau$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		

Table 7.2: Different cases of possible the edgebreaker sequences considering the constraint, that no E- nor L-symbol may follow upon C.

$$\mathbf{1}_C \quad = \quad 2\nu_\tau + \frac{1}{2}\mathbf{1}_C. \tag{7.3}$$

In these equations $\mathbf{1}_C$ denotes the conditional unity for the condition that a C symbol is preceding. Solving the system of equations yields the same result $\nu_\tau = \frac{1}{6}$. With the concept of the conditional unity all equations look just like a partitioning of the unit interval.

With all this preliminaries we can attack the second constraint. Here we not only want to account for preceding C-symbols but also for the minimal length of the current cut-border loop. If the minimal length is for example known to be at least four, the conditional unity is denoted by $\mathbf{1}_{4,C}$ if a C-symbol is preceding and $\mathbf{1}_4$ otherwise. For each condition we just have to enumerate all possible succeeding symbols and the resulting post-conditions and can easily write down the corresponding equation as shown in table 7.3. The first column contains the known minimal length of the current cut-border loop. The second column tells whether a C-symbol is preceding. The third column enumerates all symbols which can appear under the precondition in the same order they are accounted for in the equations in the last column. Let us explain the equation for the conditional unity if no C is preceding and the minimal loop length is six. Then the symbols R, L, S or C may follow, not E as the current loop is too long. The symbols R and L each yield the post-condition of no preceding C and a minimal loop length of

$cond.$		$follow$	$equation$
3		$RLSEC$	$1 = 4\nu_\tau + \frac{1}{2}\mathbf{1}_{4,C}$
4	C	RSC	$\mathbf{1}_{4,C} = 2\nu_\tau + \frac{1}{2}\mathbf{1}_{5,C}$
5	C	RSC	$\mathbf{1}_{5,C} = (\mathbf{1}_4 + 1)\,\nu_\tau + \frac{1}{2}\mathbf{1}_{6,C}$
6	C	RSC	$\mathbf{1}_{6,C} = (\mathbf{1}_5 + 1)\,\nu_\tau + \frac{1}{2}\mathbf{1}_{7,C}$
$\vdots$	$\vdots$	$\vdots$	$\vdots \quad \vdots \qquad \vdots$
i	C	RSC	$\mathbf{1}_{i,C} = (\mathbf{1}_{i-1} + 1)\,\nu_\tau + \frac{1}{2}\mathbf{1}_{i+1,C}$
$\vdots$	$\vdots$	$\vdots$	$\vdots \quad \vdots \qquad \vdots$
4		$RLSC$	$\mathbf{1}_4 = 3\nu_\tau + \frac{1}{2}\mathbf{1}_{5,C}$
5		$RLSC$	$\mathbf{1}_5 = (2 \cdot \mathbf{1}_4 + 1)\,\nu_\tau + \frac{1}{2}\mathbf{1}_{6,C}$
6		$RLSC$	$\mathbf{1}_6 = (2 \cdot \mathbf{1}_5 + 1)\,\nu_\tau + \frac{1}{2}\mathbf{1}_{7,C}$
$\vdots$		$\vdots$	$\vdots \quad \vdots \qquad \vdots$
i		$RLSC$	$\mathbf{1}_i = (2 \cdot \mathbf{1}_{i-1} + 1)\,\nu_\tau + \frac{1}{2}\mathbf{1}_{i+1,C}$
$\vdots$		$\vdots$	$\vdots \quad \vdots \qquad \vdots$

Table 7.3: Conditional unities including first and second constraint.

$l_{\max}$	τ	ν_τ	$\mathbf{1}_{4,C}$	$\mathbf{1}_{5,C}$	$\mathbf{1}_{6,C}$
5	2.56256	.169275	.645798	.614494	
6	2.55779	.169846	.641316	.603290	.591385
7	2.55677	.169955	.640358	.600895	.586446
9	2.55651	.169986	.640111	.600277	.585172
$l_{\max}$	$\mathbf{1}_{7,C}$	$\mathbf{1}_{8,C}$	$\mathbf{1}_{9,C}$	$\mathbf{1}_4$	$\mathbf{1}_5$
5				.815073	
6				.811152	.741053
7	.581919			.810313	.738612
9	.579478	.5773922	.576739	.810098	.737983
$l_{\max}$	$\mathbf{1}_6$	$\mathbf{1}_7$	$\mathbf{1}_8$		
5					
6					
7	.711977				
9	.710618	.700273	.696429		

Table 7.4: Results for different restrictions $l_{\max}$ for the precondition on the current loop length.

five as both of them eliminate one vertex from the cut-border. This is represented by the term $2 \cdot \mathbf{1}_5 \cdot \nu_\tau$. After a split symbol nothing about the length of the current cut-border loop is known, which is accounted for by the term ν_τ. Finally, a new vertex operation C will increase the loop length by one and therefore the right side of the equation also contains the term $\frac{1}{2}\mathbf{1}_{7,C}$, where the $\frac{1}{2}$ represents the frequency of the C-symbol.

We solved the set of equations with a computer algebra program for different restrictions $l_{\max}$ of the minimal loop length. If we for example restrict the minimal loop length to six, we replace in the equation for $\mathbf{1}_{6,C}$ the $\mathbf{1}_{7,C}$ on the right side with $\mathbf{1}_{6,C}$. This is valid as if the loop length is at least of length seven then it is also longer than six. Table 7.4 gives the results for different values of $l_{\max}$ and also the values for the conditional unities, which allow to build an arithmetic coder. τ converges very fast to 2.557 and we conclude this section with the following theorem.

Theorem 7.2 *With the encoding scheme described in this section a planar triangulation with v vertices can be encoded and decoded in linear time to less than $3.557v$ bits.*

7.3.2 Using More Constraints for 3.552 Bits per Vertex Encoding

If we apply the techniques of the previous section to the reverse decoding, we can consider more constraints. During reverse decoding each E operation starts a cut-border loop with three vertices. Each new vertex operation C decreases the current cut-border loop by one vertex and the R and L operations increase the loop by one. A C operation

Table 7.5: Conditional unities accounting for the constraints induced by two successive loops.

$l_{\max}$	$ll_{\max}$	τ	ν_τ
7	4	2.55197	0.17052
11	5	2.55122	0.17061
17	6	2.55102	0.17063

Table 7.6: Results for different restrictions $l_{\max}$ for the precondition on the current loop length and $ll_{\max}$ for the precondition on the length of the previous and the current loop.

is never allowed, when the current cut-border loop is of length three or if the previous operation was an L operation. Thus in order to keep track of the current cut-border loop length we define two types of conditional unities. For all $i \geq 3$: $\mathbf{1}'_i$ is the unity under the condition that the current cut-border loop is of length i and $\mathbf{1}'_{i,L}$ is the unity under the additional condition that the previous symbol was L. Finally, we define $\mathbf{1}'_L$ to be the unity when nothing about the loop length is known except that L has been the previous symbol. Using these unities to built a system of equations similar to the one in table 7.3 we can achieve again a value of $\tau = 2.557$.

After a split operation during reverse decoding the last two cut-border loops are merged and we do not know anything about the loop length with the so far described approach. But it is actually feasible to keep track of the lengths of two successive cut-border loops as long as they are short and we define the conditional unities $\mathbf{1}'_{j,i}$ and $\mathbf{1}'_{j,i,L}$ for all $i, j \geq 3$. The index i represents the loop length of the current cut-border loop and j of the previous loop. With the new unities a sub-sequence of $EESCCC$ can be correctly excluded as the first two E operations would create two loops of length three each, the split concatenates these two loops to one loop of length five and the three new vertex operations C would reduce the loop length to two what is not possible.

Table 7.5 gives the different kinds of equations parametrized over the loop lengths of the current loop with length i and the previous loop with length j. In order to calculate the different conditional unities and the value for ν_τ we restricted the maximal loop length for $\mathbf{1}'_i$ to $i \leq l_{\max}$ and the loop lengths for $\mathbf{1}'_{j,i}$ to $i \leq ll_{\max}$. The resulting values for ν_τ and τ are shown in table 7.6.

Theorem 7.3 *With the encoding scheme described in this section a planar triangulation with v vertices can be encoded and decoded in linear time to less than $3.552v$ bits.*

Chapter 8

Conclusion & Directions for Future Work

Connectivity compression for triangle meshes is not yet perfect, but probably not far from perfect. A lot of methods are available, that exploit the regularity of meshes to encode the connectivity to even better than the theoretical limit would allow for arbitrary irregular planar meshes. An interesting question is an appropriate measurement for the regularity of the mesh connectivity and how the regularity can be exploited optimal.

Newer efforts have generalized the best triangle connectivity encoding methods for polygonal meshes and also achieve convincing results. Here might be some more work to be done. Also do most methods not support non manifold meshes, which are quite common. Although the solution of Gueziec [GTLH98] can be combined with most encoding schemes, it is not optimal, as all non manifold spots are completely cut apart. It would be interesting to examine the minimal number of cuts needed to produce manifold connectivity and encode only these cuts. The Cut-Border Machine can be generalized quite easily to non manifold meshes by adding cut-border operations, that handle the non manifold spots in a mesh. The minimization of the number of performed cuts would be steered by the traversal order. One could also introduce markers of non manifold spots onto which the non manifold cut-border operations could reference later on in order to minimize the size of the to be encoded vertex indices.

Very interesting in terms of practical relevance of mesh connectivity encoding is the work of Isenburg [Ise00, IS00], who began to encode other incidence relations, like triangle strips or a partitioning of the mesh, in an interwoven fashion with the mesh connectivity at small or no additional cost.

In the progressive coding of mesh connectivity the level split method has established itself as most efficient approach. For the most important simplification operations exist very compact representations. An interesting question is, how much information is included in the different levels of detail in terms of connectivity and whether the current methods are optimal in a similar sense as the single resolution connectivity encoding methods are optimal for the special case of planar triangulations.

In the research field of the optimal encoding of planar triangulations new results

could be achieved in the last years. The theoretical lower bound of a 3.245 bit per vertex encoding could be achieved up to ten percent by the author as illustrated in section 7.3. It is not clear whether the researchers are motivated to make an effort to get closer to the theoretical limit. But if this was so, the Edgebreaker encoding would probably be the best method to promote the research in this direction. Firstly, one could examine, whether the constraints of the Edgebreaker strings described in section 7.3 are complete or if further constraints can be found. The completeness of a set of constraints can be checked by building for each constraint string a unique planar triangulation. The fact that each planar triangulation corresponds to one unique constraint string would prove the equivalence of the set of planar triangulations and the set of constraint strings. An induction over the length of the string suggest itself to build the planar triangulations from the constraint strings. A basic ingredient to this induction would be the fact, that for any planar triangulation with bent edges exists a planar triangulation with straight edges. If a complete set of constraints is found, one has to explore for each constraint, whether it is exploited in an optimal fashion. Clearly, this is only a vague research plan, that could end in several dead ends. There might be no complete set of constraints, what is rather improbable. The proof of completeness could be too difficult or even more frustrating some of the constraints might be too complex to be exploited optimal, what is most probable of all dead ends. But there is also the chance, that a different encoding scheme can achieve the theoretical lower bound.

The encoding of the vertex locations is not satisfactory yet. The quantization in most methods leads to visual artifacts in the meshes. Predictive delta-coding seems to be exhausted. Current and still unpublished work on vector quantization applied to vertex location encoding suggest a slight improvement over predictive delta-coding. Also the progressive schemes should lead to significantly better results compared to the single resolution methods, but still do not do. The work of Karni [KG00a] allows to perform a frequency analysis on the full connectivity of a mesh. No sampling locations of the surface have to be encoded explicitly. Thus this method is very convincing. In future work the method must be made hierarchical, such that also the connectivity can be refined progressively. The other approach of avoiding the explicit encoding of sample locations is to remesh a given mesh with a new mesh of subdivision connectivity [GVSS00, LMH00], that automatically defines new and different sampling locations. The compression rates for vertex locations are very high but the new approach has problems in the modeling of sharp creases. We propose a different strategy of transforming mesh connectivity into subdivision connectivity. The idea is to change the connectivity of a given mesh only slightly by some local operations as for example edge flips in order to produce a connectivity, that allows at least one inverse subdivision step. By applying this inverse subdivision the mesh is transformed to a coarser level. The same coarsening process is repeated until only a very small and efficiently to be encoded mesh remains. The mesh can now be compactly represented through the coarse mesh and an efficient encoding of the performed edge flips on each level. This method would be very efficient on regular meshes, that are near to subdivision connectivity, but would also easily adapt

to meshes with sharp creases.

A very exciting avenue of research will open up with the development of new 3D-scanners, that allow the acquisition of moving objects. A lot of problems must be solved: How can the scanned surface meshes of different time frames be brought together? What data structure can we generalize to meshes with connectivity, that can change in time? Can we simplify these time dependent or better said morphing objects? And finally, how do we compress the representation of the morphing object?

Part III

Tetrahedral Mesh Compression

Chapter 9

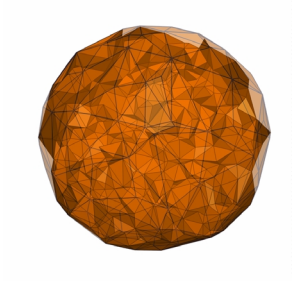
Introduction to Tetrahedral Meshes

Tetrahedral meshes have been around in finite element simulations on volumetric domains for a long time. With the growing need of visualization for the simulation data, tetrahedral meshes established themselves also in volume visualization. There are several beautiful properties of tetrahedral meshes which make them the natural choice for volume data representation. The flexibility of a tetrahedral mesh is ideally suited for an irregular sampling and for multi-resolution analysis. The convex nature of a single tetrahedron allows for a simple visibility sorting algorithm[Wil92], which is essential in volume visualization.

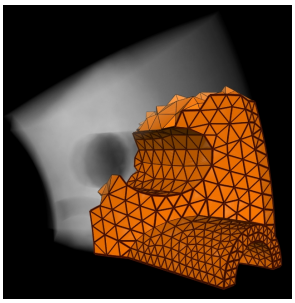
In most application areas of tetrahedral meshes some data is attached to the mesh elements. The data can be attached to the vertices, edges, the faces, the border faces or the tetrahedra. A density might be attached to the vertices, the intensity of a flow to the edges or material identifiers to the tetrahedra. The tetrahedral mesh serves several different purposes. It can be used to store nearest neighbors, to subdivide a volume into convex primitives or to sample and, by the use of barycentric coordinates, to parameterize the domain of a function with volumetric domain. The function can be scalar, a vector field or even a tensor field as for example the stress tensor of an inhomogeneous material. The Cut-Border Machine compression algorithm can be extended in a natural way to support the compression of all three types of data functions defined on all different types of mesh elements.

9.1 Basic Definitions and Notations

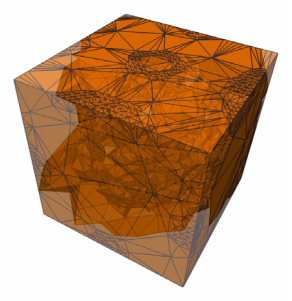
We us the notation of section 1.4 and denote the number of tetrahedra with t and the number of border faces with b . Figure 9.1 shows six typical tetrahedral meshes, which we used for our measurements. They differ in their sizes and their origin. The “Random” mesh was generated by delaunay tetrahedralization of a cloud of randomly distributed points. In order to show that the interior of this mesh is more complex than the surface, we blended a cut through the mesh with its surface. The “Proto” mesh is a quite regular tetrahedralization of an object with non trivial boundary. The “Bubble” is the output



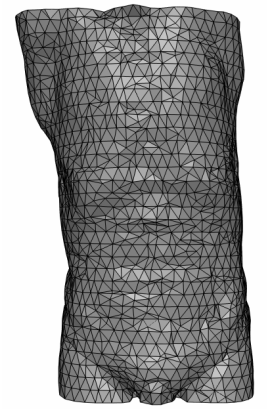
a) Random



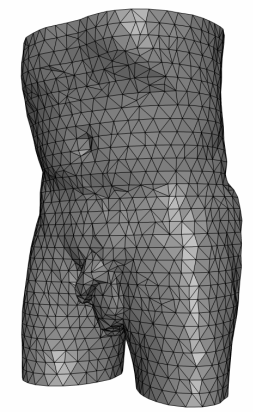
b) Proto



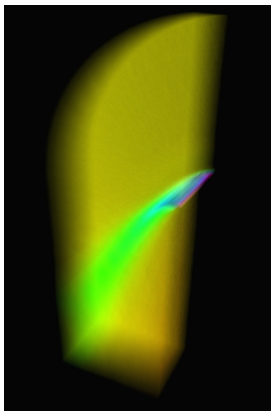
c) Bubble



d) Torso I



e) Torso II



f) Blunt Fin

Figure 9.1: Typical tetrahedral meshes. The transparent meshes were rendered with the method of projected tetrahedra [ST90]. The “Blunt Fin”-mesh was rendered with false colors.

of a simplification algorithm applied to a spherical symmetric scalar function. Again the blending technique shows part of the interior. The “Torso” meshes are regularly tetrahedralized real world meshes and the “Blunt Fin” is originally a curvy linear grid. Table 9.1 shows the basic quantities of the different meshes and the average vertex-edge and edge-tetrahedron order (see section 1.4.

We will denote the total amount of bits consumed by a tetrahedral mesh with $\mathcal{S}$, where we use a right subscript to express a special representation type. $\mathcal{S}_{\text{std}}$ denotes for example the standard representation with a list of vertex coordinate triples, a list of vertex index quadruples representing the relation $T \rightarrow V$ from the set T of tetrahedra to the vertices and additional lists for the attached data. We split the storage space $\mathcal{S}$ into the bits $\mathcal{L}$ consumed by the locations of the vertices, $\mathcal{C}$ consumed by the connectivity and $\mathcal{D}$ consumed by the data attached to the mesh elements. If no data is present only the mesh consisting of connectivity and vertex locations has to be encoded in $\mathcal{G}$ bits. For reasonable representations we get:

$$\mathcal{S} \leq \mathcal{G} + \mathcal{D} \qquad \mathcal{G} \leq \mathcal{C} + \mathcal{L}$$

The combined representation of two and more components of the tetrahedral mesh can

mesh	v	v: e: f: t	$\frac{v_b}{v}$	b	$\bar{o}_{v\to e}$	$\bar{o}_{e\to t}$
Random	2000	1:7.39:12.67:6.29	0.101	400	14.77	5.11
Proto	2896	1:5.94: 9.41:4.47	0.477	2760	11.89	4.51
Bubble	5715	1:6.89:11.63:5.74	0.150	1710	13.78	5.00
Torso I	11140	1:6.55:10.91:5.35	0.197	4380	13.10	4.90
Torso II	15164	1:6.61:11.04:5.43	0.180	5454	13.22	4.93
Blunt Fin	40960	1:5.74: 9.32:4.58	0.165	13516	11.48	4.78
average		1:6.52:10.83:5.31	0.212		13.04	4.87

Table 9.1: Basic quantities of the measured tetrahedral meshes.

be more efficient since better predictions might improve delta coding or just because the coding mechanism can combine some fractional bits.

9.2 Basic Equations and Approximations

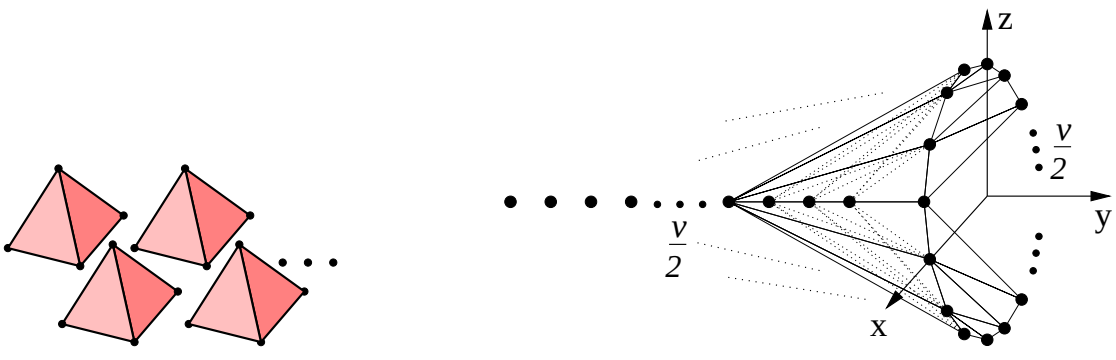
The basic equation for a tetrahedral mesh is also the Euler equation as in the case of polygonal meshes (compare equation 1.1)

$$v - e + f - t = \chi, \tag{9.1}$$

where χ is the Euler characteristic of the mesh and in most cases negligibly small. If we count the halffaces (see section 1.5.4) once for each tetrahedron and once for each face we get a second equation including the number of border faces b

$$f = 2t + \frac{b}{2}. \tag{9.2}$$

In the case of triangle meshes the corresponding equations are sufficient to determine the average vertex-face order and the number of triangles per vertex in a mesh with small



a) b)

Figure 9.2: tetrahedral meshes with a) minimum and b) maximum vertex-tetrahedron order $\frac{4t}{v}$.

Euler characteristic and few boundary edges, but not in the tetrahedral case as Figure 9.2 illustrates. The vertex-tetrahedron order might vary between one as in Figure 9.2 a) and v for the mesh in b)[1]. Thus for the number of tetrahedra in an arbitrary tetrahedral mesh we only know

$$\frac{v}{4} \leq t \in \Omega(v^2). \tag{9.3}$$

Non of the tetrahedral meshes of Figure 9.2 are used to sample volumetric functions for volume visualization or finite element analysis. The tetrahedral meshes of interest normally have a limited vertex-edge order, a small border portion and low Euler characteristics of the mesh and of the border mesh, respectively. Therefore, we express the fraction between t and v in terms of the average number of edges around a vertex $\bar{o}_{v\to e} = \frac{2e}{v}$, the number of border vertices v_b, χ and χ_b the Euler characteristic of the border. If we assume a manifold border mesh, $3b = 2e_b$ holds true, if e_b is the number of border edges. Notice that the border mesh must be closed as it describes the surface of a volume. With equation 1.2 for the border triange mesh and equations 9.1 and 9.2 we get

$$\frac{t}{v} = \frac{\bar{o}_{v\to e}}{2} - 1 - \frac{v_b}{v} + \frac{\chi + \chi_b}{v}. \tag{9.4}$$

To find a basic approximation for the relation between t and v in a typical tetrahedral mesh with small border portion and low Euler characteristics we are left with the estimation of $\bar{o}_{v\to e}$ for a regular tetrahedral mesh. Unfortunately, the Euclidean space can not be tetrahedralized with equilateral tetrahedra. But the fraction of 4π over the steradian occupied by an equilateral tetrahedron yields[2] with 11.64 a good approximation of the average vertex-edge order. The tetrahedralization of a cubic grid yields $\bar{o}_{v\to e} \overset{v\to\infty}{\longrightarrow} 12$ for an $1:5$ zoning[3] and $\bar{o}_{v\to e} \overset{v\to\infty}{\longrightarrow} 14$ for an $1:6$ zoning. Considering this and the measured average vertex-edge orders in Table 9.1, we assume in the following an average vertex-edge order of thirteen. For tetrahedral meshes with small Euler characteristic and border portion we get in agreement with Table 9.1

$$v : e : f : t \approx 1 : 6.5 : 11 : 5.5. \tag{9.5}$$

Let us use this approximation to estimate the storage consumption of a tetrahedral mesh in the standard representation, where each vertex is given by three 32bit floating point coordinates and each tetrahedron by four vertex indices:

$$\mathcal{L}_{\text{std}} = 96v, \qquad \mathcal{C}_{\text{std}} = 4t \cdot \text{lb}v \overset{v\approx 10^5}{=} 374v. \tag{9.6}$$

For a typically sized tetrahedral mesh with a hundred thousand vertices the connectivity consumes about four times more storage space than the vertex coordinates. With the Cut-Border Machine compression technique the storage space for the connectivity can be reduced to about eleven bits per vertex. This reduces the overall storage space of the tetrahedral meshes to a quarter without loosing any information.

[1] The mesh is even one of the delaunay tetrahedrizations of the shown set of points.

[2] We applied the Euler equation for spherical triangle meshes.

[3] Each cube is split into five tetrahedra.

Chapter 10

Generalization of the Cut-Border Machine

In this section we generalize the Cut-Border Machine compression method to the tetrahedral case. We first give a brief overview of the changes in section 10.1. After that we describe the different cut-border operations (section 10.2) and the compressed representation (section 10.3). The best traversal strategy we found is proposed in section 10.4. In section 10.5 we introduce an improvement for the mesh border encoding, which is similar to the improved border coding for the triangular Cut-Border Machine as introduced in section 6.2. In the triangular case the Cut-Border Machine is very simple to implement and also extremely fast. The generalization to the tetrahedral case requires a more sophisticated data structure for non manifold triangle meshes, which is described in section 10.6.

10.1 From Triangular to Tetrahedral Cut-Border Machine Compression

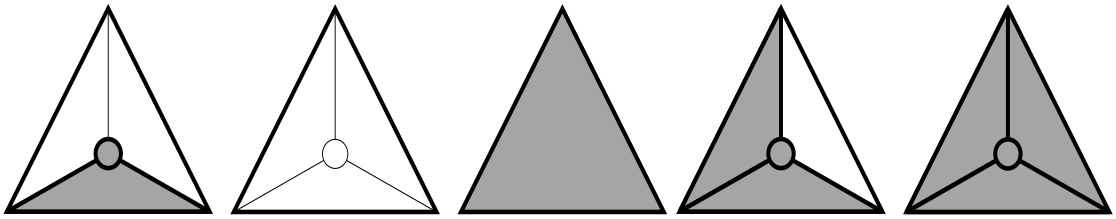
Similar to the triangular case, the uncompressed tetrahedral mesh is transformed into a halfface data structure.

The inner and the outer part consist of a set of tetrahedra. The cut-border is the triangular surface between the inner and the outer part and the gate is a triangle of the cut-border. For each face-connected component of the mesh the traversal begins with an arbitrary tetrahedron and successively adds outer part tetrahedra, which are incident upon the gate, to the inner part. The different cut-border operations are described in the next section. The cut-border may become the surface of an arbitrary face-connected tetrahedral mesh and therefore contain non manifold vertices and edges. In section 10.6 we describe an appropriate data structure. We assume that the tetrahedral mesh is embedded in three dimensional space and that the tetrahedra do not penetrate each other.

As in the triangular case the traversal order highly influences the distribution of the *"connect"*-operations with different offsets. Section 10.4 describes the best heuristic

we could find.

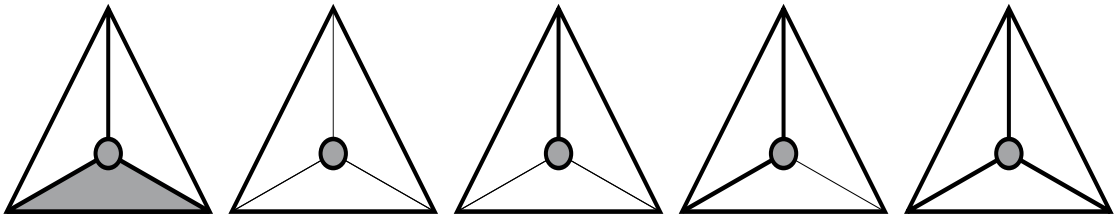
10.2 Cut-Border Operations and Situations



a) *"flip"* b) *"new vertex"* c) *"border"* d) *"top"* e) *"close"*

Figure 10.1: The different manifold cut-border situations. The newly encoded tetrahedron is viewed from top with the gate at the bottom. Additionally to the gate triangle the bold drawn edges and the dark shaded triangles are part of the cut-border before the cut-border operation.

There are three possibilities for the fourth vertex of a newly added tetrahedron at the gate: the gate is a border triangle of the tetrahedral mesh, the gate forms a tetrahedron with a new vertex or the gate is connected through a tetrahedron to another cut-border vertex. The corresponding cut-border operations will again be called *"border"*, *"new vertex"* and *"connect"* and are abbreviated with the symbols $\Delta, *$ and ∞_i.



a) *"nm flip"* b) *"join"* c) *"join, 1 nm"* d) *"join, 2 nm"* e) *"join, 3 nm"*

Figure 10.2: The different types of non manifold cut-border situations.

Although only three different types of cut-border operations exist, we distinguish ten different situations which describe the surrounding of the cut-border around the gate for the different cut-border operations. All the situations are illustrated in figures 10.1 and 10.2. The diagrams show the newly added tetrahedron from above with the gate at the bottom. Besides the gate at the bottom of the tetrahedron the bold drawn edges and the dark shaded faces also belong to the cut-border before the cut-border operation is performed. Figure 10.1 shows the situations which do not introduce non manifold vertices or edges. For the *"border"*- and the *"new vertex"*-operation only one situation exists which is depicted in figure 10.1 c) and b), respectively. The *"connect"* operation comes along with a whole variety of situations. The most frequent of these is the *"flip"*

operation illustrated in figure 10.1 a). Here the newly added tetrahedron connects the gate to an adjacent triangle of the cut-border. The common edge of these two cut-border triangles is kind of flipped if the two former cut-border triangles are replaced by the two new cut-border triangles introduced by the new tetrahedron. The *"top"* and the *"close"* operations are very similar to the *"flip"* operation. The only difference is that not only two faces of the newly added tetrahedron are part of the cut-border but three of them in the case of the *"top"* operation or even all in the case of the *"close"* operation. The *"close"*-operation eliminates or closes an edge-connected component of the cut-border triangle mesh (figure 10.1 e)).

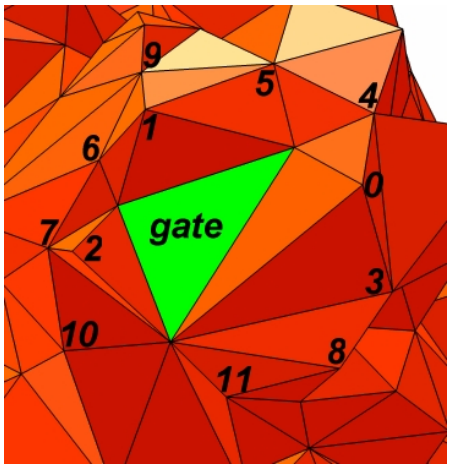
As mentioned earlier, the cut-border can be a non manifold triangle mesh. Figure 10.2 portrays all types of situations which introduce a non manifold vertex or edge. In figure 10.2 a) the non manifold counterpart of the *"flip"* situation is shown. Here the free edge of the *"flip"* situation is touched by the cut-border and therefore already belongs to the cut-border. The touched edge becomes non manifold after application of the *"connect"* operation. The *"join"* situation in figure 10.2 b) is the non manifold counterpart of the *"new vertex"* operation. The fourth vertex of the newly added tetrahedron is part of a region of the cut-border triangle mesh which is further apart from the gate. This vertex becomes non manifold. Finally, in the *"join"* situations depicted in figures 10.2 c), d) and e) not only the fourth vertex of the newly added tetrahedron belongs to the cut-border but also one two or all three free edges of the *"join"* situation. Thus one, two or three non manifold edges are introduced.

The situations depicted in figures 10.1 and 10.2 constitute all possible situations, which can be easily verified by considering a newly added tetrahedron: the three triangles of the tetrahedron which are unequal to the gate may all be part of the cut-border or not be part. The same holds true for the fourth vertex and the three edges not incident to the gate. All of these seven mesh elements might be present in the cut-border or not. The presence of one of the three triangles implies the presence of the fourth vertex and the two incident edges. If we take such implications into account each possible assignment of presence to the three triangles, three edges and the fourth vertex yields exactly one of the discussed situations. Thus each face-connected component of the tetrahedral mesh can be compressed without any vertex repetitions. Only if two components of the tetrahedral mesh are exclusively connected through edge-adjacency and vertex-adjacency the involved non manifold vertices are repeated. In a simple way the *"border"*-operation allows for the encoding of all possible border surfaces of tetrahedral meshes including non manifold borders.

The *"connect"* operation takes one index as parameter, which specifies the fourth vertex in the cut-border. The fourth vertex is with high probability near to the gate. We can exploit this fact for a more efficient encoding by mapping near fourth vertices to small connect indices. This is achieved by a breadth-first traversal through the triangles of the cut-border starting at the gate as shown in the illustration of algorithm 1. The enumeration is not uniquely defined before one edge of the gate is specified at which the enumeration with the zero connect index will begin. This edge will be called the

Algorithm 1 *Vertex Enumeration*

```
fifo.pushback(gate.zeroEdge())
fifo.pushback(gate.oneEdge())
fifo.pushback(gate.twoEdge())
while not fifo.empty do
  edge = fifo.popfront()
  tgl = edge.rightTriangle()
  if not marked(tgl) then
    mark(tgl)
    vtx = tgl.oppositeVtx(edge)
    if not marked(vtx) then
      mark(vtx)
      enumerate(vtx)
    fifo.pushback(tgl.nextEdge(edge))
    fifo.pushback(tgl.prevEdge(edge))
```



zero edge and is specified by the traversal strategy (see section 10.4). Algorithm 1 gives pseudo code for the vertex enumeration. The algorithm is similar to the cut-border traversal in the case of a triangle mesh. In a *fifo* these edges of the cut-border are stored which are adjacent both to a visited triangle and to a not visited triangle at the same time. The zero edge is firstly placed into the *fifo*. Triangles are visited by extracting the next edge from the *fifo* and addressing the adjacent triangle which has not been visited yet. If the third vertex of the newly visited triangle is reached the first time, the next available connect index is assigned to it. In this way the vertices obtain the indices illustrated in the figure of algorithm 1.

The *"flip"* situation can arise for the operations ∞_0, ∞_1 and ∞_2, the *"top"* situation for ∞_0 and ∞_1 and *"close"* only for ∞_0. The different *"join"* situations correspond to *"connect"* operations with larger index and are less frequent. The traversal strategy described in section 10.4 optimizes the choice of the zero edge in a way that most *"flip"* and *"top"* situations are encoded with ∞_0.

10.3 Compressed Representation

In the triangular case the *"new vertex"*-operation $*$ is performed in about half the cases and is most frequent. In the tetrahedral case the relative frequency of $*$ is only about $\frac{1}{5.5}$, whereas the connect operations with small index are most frequent. For optimal encoding of the operation symbols we use arithmetic coding since the relative frequencies are unequal to 2^{-k} and therefore Huffman-coding is not appropriate.

The connectivity of the tetrahedral mesh is given by the sequence of cut-border operations. As each operation adds one tetrahedron or specifies one border face, $t + b$ operations are encoded. The binary entropy defined in equation 3.1 gives a good lower

bound

$$\mathcal{C}_{\text{CB}} \stackrel{\text{def}}{=} \mathcal{E}\left(n, \nu_{\Delta}, \nu_{*}, \nu_{\infty_0}, \nu_{\infty_1}, \ldots\right) < \mathcal{C}_{\text{CB}}^{\text{adapt}} \tag{10.1}$$

for the storage space $\mathcal{C}_{\text{CB}}^{\text{adapt}}$ consumed by our arithmetic coder with adaptive relative frequencies, which are initialized to the average values given in the last row of Table 10.1. Table 10.2 shows that our arithmetic coder almost achieves the optimum.

The vertex coordinates and the data at the vertices, edges, faces and tetrahedra are incorporated in the arithmetic coding stream with separate coding models. Each time a cut-border operation produces a new mesh element, the corresponding data is added to the stream. The representation of a 1:6 zoning of a cube with vertex data $v_0, v_1, \ldots, v_7$ and tetrahedral data $t_0, t_1, \ldots, t_5$ might look as follows:

$$\begin{aligned}
&t_0 x_0 y_0 z_0 v_0 x_1 y_1 z_1 v_1 x_2 y_2 z_2 v_2 x_3 y_3 z_3 v_3 \Delta\Delta \\
&*t_1 x_4 y_4 z_4 v_4 \Delta * t_2 x_5 y_5 z_5 v_5 \Delta \\
&*t_3 x_6 y_6 z_6 v_6 \infty_0 t_4 \Delta\Delta * t_5 x_7 y_7 z_7 v_7 \Delta\Delta\Delta\Delta\Delta\Delta.
\end{aligned}$$

10.4 Traversal Order

The traversal strategy chooses after each cut-border operation the next gate and zero edge. The aim is to favorite a small number of different kinds of operations. To avoid most connect operations with large indices it turned out that a good strategy is to stay at one cut-border vertex until all adjacent tetrahedra have been visited. The cut-border vertices are processed in a fifo order. For the choice of the zero edge and the order in which the triangles around a cut-border vertex are added, we tried two heuristics that favorite the ∞_0-operation. The first one cycles around edges and tries to close up with a ∞_0-operation by setting the zero edge of the gate to the edge around which the Cut-Border Machine cycles. The second strategy defines the zero edge of each cut-border triangle at the time when the triangle is created. The zero edge is set to the edge which is shared by the gate and the new triangle. In case of a new vertex operation it is obvious that with this choice the zero edge is the edge with the smallest angle in the outer part. This still holds true to some extent for the other operations. The first heuristic increased the frequency of the ∞_0-operation to 45% and the second heuristic even to 60%. Thus we chose the second strategy, which is documented in Table 10.1. The first column shows for each mesh the total number $t + b$ of encoded operations. In the following columns the relative frequencies of the different cut-border symbols are shown. ∞_0 is with 60% the most frequent operation, followed by $*$, ∞_1 and ∞_2. With the border optimization described in the next section the frequency of the border symbol became negligibly small. The last column shows the fraction of the non manifold situations in Figure 10.2 which arose during compression. This number is important for the optimal running time of the compression and decompression algorithms as the non manifold operations consume more computing power.

10.5 Mesh Border Encoding

In order to allow for a non manifold mesh border, we explicitly encode the border operations. The border symbol can be avoided when an edge-adjacent triangle of the gate has already been encoded as border triangle. In this case the corresponding connect symbol can be used. This optimization helped to decrease the additional amount of storage for the mesh border to one bit per border triangle as tabulated in Table 10.2. The same optimization improves the border encoding in the triangular case of the Cut-Border Machine.

10.6 Cut-Border Data Structure

Data Structure 1 *Cut-Border*

```
CutBorder
  CutBorderTriangle   triangles[]
  Fifo<CutBorderVertex> vertices
  TriangleIndex       gate
CutBorderTriangle
  VertexIndex         vertexIndices[3]
  TriangleIndex       adjacentTriangles[3]
  TetraIndex          innerTetra
  Boolean             meshBorder
  Integer             zeroEdge
CutBorderVertex
  VertexIndex         meshVertexIndex
  Set<TriangleIndex>  adjacentTriangles
```

Data structure 1 shows the cut-border data structure. Three relations between the cut-border vertices and the cut-border triangles are stored: for each triangle the three incident vertices and three edge-adjacent triangles; for each vertex all incident triangles.

mesh	$t+b$	ν_Δ	ν_*	ν_{∞_0}	ν_{∞_1}	ν_{∞_2}	$\nu_{\infty_{i>2}}$	ν_{nm}
Random	12971	0.001	0.154	0.519	0.118	0.108	0.101	0.116
Proto	15695	0.001	0.184	0.631	0.073	0.067	0.044	0.046
Bubble	34526	0.001	0.165	0.549	0.106	0.091	0.088	0.109
Torso I	64028	0.002	0.174	0.607	0.080	0.072	0.064	0.069
Torso II	87788	0.001	0.173	0.603	0.083	0.075	0.065	0.069
Blunt Fin	200910	0.000	0.204	0.707	0.045	0.044	0.000	0.000
average		0.001	0.176	0.602	0.084	0.076	0.060	0.068

Table 10.1: Total number of encoded operations; relative frequencies of cut-border operations; relative frequency of non manifold situations.

The latter relation is stored in a set data structure which allows insertion and elimination of elements and the intersection of two sets. This relation allows for the handling of non manifold vertices and edges. For each cut-border triangle the incident tetrahedron of the inner part is stored in order to find the new tetrahedron if the triangle becomes the gate. The `meshBorder`-flag tells us when the cut-border triangle has already been encoded as border triangle of the mesh and therefore does not have to be visited again. With the help of this flag the optimized border encoding is realized. As the traversal order introduced in section 10.4 defines the zero edge for each triangle at creation time, an index between zero and two is stored for each cut-border triangle defining the zero edge. The cut-border vertices are organized in a fifo as demanded by the traversal strategy chosen in section 10.4.

We generate for each vertex of the tetrahedral mesh a field which stores the index of the cut-border vertex and initialize it before compression to minus one. In this way we can not only map a tetrahedral mesh vertex index to a cut-border vertex index but do also know which of the tetrahedral mesh vertices are part of the cut-border.

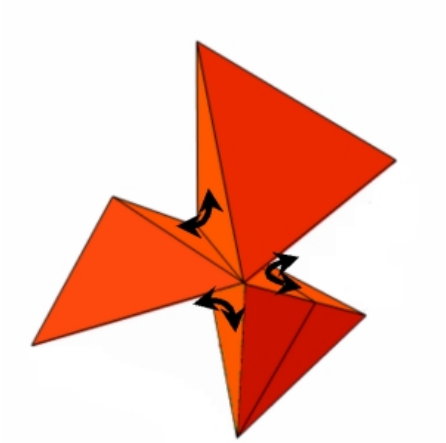


Figure 10.3: Face adjacency of cut-border triangles around non manifold edge.

Let us describe why it is sufficient to keep for each triangle only three edge-adjacent neighbors even at non manifold edges. At any time the cut-border describes the surface of a tetrahedral mesh. Thus the faces around a non manifold edge divide the space into regions alternately belonging to the inner and the outer part. These regions around a non manifold edge are called *inner/outer regions*. The faces bounding the same outer region can be set to be edge-adjacent as illustrated in Figure 10.3. This definition correctly reflects the proximity needed in enumerating the vertices relative to the gate. Faces of different outer regions can not be connected through a tetrahedron without intersecting an inner region.

Finally, we describe the updates of the cut-border data structure for the different situations depicted in figures 10.1 and 10.2. During the *"connect"* operation of a manifold *"flip"* situation (see figure 10.1 a)) the two present triangles in the cut-border

are replaced with two new ones where the common edge is flipped. The vertices and face-adjacent triangles of the two new triangles can be easily determined from the old triangles. For each new triangle the zero edge is set to the edge, which is incident to the gate. The `innerTetra` index of the newly added triangles is set to the newly added tetrahedron, as in all other situations of all operations. Finally, the old triangles are removed from the triangle sets of the vertices and the new triangles are added.

The first step during the update of the *"new vertex"* operation is to create a new cut-border vertex for the fourth vertex of the newly added tetrahedron and store its vertex index of the tetrahedral mesh in the corresponding field. Conversely, the index of the new cut-border vertex is stored within the corresponding field of the tetrahedral mesh vertex. Next the gate triangle is removed and three new triangles are inserted. Again their zero edges are set to the edges incident to the gate. The *"border"* operation just sets the border flag of the gate triangle. For the border optimization the border flags of the three edge-adjacent cut-border triangles are checked and if one of them is set, the operation is encoded with the corresponding *"connect"* operation. The *"top"* situation is similar to the *"flip"* situation except that three triangles are removed and only one is added. As last manifold situation the *"close"* operation eliminates all involved triangles and these vertices for which the set of adjacent triangles becomes empty. If a cut-border vertex is removed the index stored with the corresponding tetrahedral mesh vertex is set to minus one again.

In order to distinguish between manifold and non manifold situations we have to clear up how to decide whether an edge of the newly added tetrahedron belongs to the cut-border or not. The question is trivially answered positively if an incident triangle of the newly added tetrahedron already belongs to the cut-border. Otherwise the answer can be determined by intersecting the set of adjacent triangles of the incident vertices of the edge in question. If the intersection is empty no cut-border triangle contains the edge and therefore the edge cannot belong to the cut-border. The intersection test must be performed for all edges of the non manifold situations in figure 10.2 which are not incident to a cut-border triangle. In case of the *"nm flip"* situation this is one edge and in case of the four *"join"* situations these are three edges. Only if the non manifold edges are detected, the face-adjacencies can be updated according to figure 10.3. And this is the only difference in the update process between the *"nm flip"* and *"flip"* situations and between the four different *"join"* situations and the *"new vertex"* operation.

The *"nm flip"* operation is distinguished from the *"flip"* situation by checking if the edge connecting the two newly added triangles belongs to the cut-border or not. This check can be done after the update performed for the *"flip"* situation, such that the face-adjacencies of the two new triangles can be corrected if necessary. This is only possible if we assume that the vertex coordinates are known and given in three dimensional space. For more general tetrahedral meshes the neighbors of the newly added triangles must be explicitly encoded. This can be done with few bits and as the non manifold situations are much less frequent as the manifold situations, the total storage space won't increase significantly for typical meshes.

mesh	$\frac{\mathcal{C}_{\mathrm{CB}}^{\mathrm{adapt}}}{v}$	$\frac{\mathcal{C}_{\mathrm{CB}}}{v}$	$\frac{\mathcal{C}_{\mathrm{CB}}}{t}$	$\frac{\mathcal{C}_{\mathrm{CB},\Delta}}{b}$	$\left(\frac{t}{sec}\right)_{\mathcal{C}}$	$\frac{\mathcal{L}_{\mathrm{CB}}^{16\mathrm{bit}}}{v}$	$\left(\frac{t}{sec}\right)_{\mathcal{G}}$
Random	15.12	15.02	2.39	1.37	84831	34.40	73866
Proto	9.55	9.48	2.12	0.90	93603	30.86	74259
Bubble	13.52	13.43	2.34	1.11	85774	30.09	74146
Torso I	11.02	10.99	2.05	1.29	92508	30.41	76749
Torso II	11.15	11.14	2.05	1.20	92574	29.64	76992
Blunt Fin	6.00	5.99	1.31	0.54	98587	26.36	78493
average	11.06	11.01	2.04	1.07	91313	30.29	75751

Table 10.2: Cut-Border Machine: consumed storage for connectivity, border and quantized vertex coordinates. Running time for connectivity alone and together with vertex coordinates in tetrahedra per second on a Pentium II 350MHz.

The family of *"join"* situations is detected whenever the three triangles of the newly added tetrahedron, which are not equal to the gate, are not part of the cut-border but the fourth vertex is part of the cut-border. The latter condition is checked with the help of the cut-border index field attached to the tetrahedral mesh vertices. The update of the *"join"* situations is the same as in the case of the *"new vertex"* operation accept that the three newly added triangles must also be inserted to the triangle set of the fourth vertex. Finally, the three potential non manifold edges are checked for their presence in the cut-border and the face-adjacencies of the corresponding triangles are corrected if necessary as in the case of the *"nm flip"* situation.

10.7 Results

Table 10.2 illustrates different aspects of the consumed storage space and running time for the Cut-Border Machine. The first column shows the storage space consumed by our arithmetic coder for the connectivity. The second and third columns tabulate the binary entropy of the cut-border operations in bits per vertex and bits per tetrahedra. Comparison of the first two columns shows that our arithmetic coder is near the optimum. The Cut-Border Machine consumes on average about two bits per tetrahedron, even for the randomly generated mesh which forces more connect operations with a high index. $\mathcal{C}_{\mathrm{CB},\Delta}$ is the binary entropy of the sequence of cut-border operations, that were used to encode the border faces. The fourth column of Table 10.2 shows that the border could be encoded with about one bit per triangle. As the best triangle mesh compression methods consume also about one bit per triangle, the initializing of the Cut-Border Machine with the border of the tetrahedral mesh would not improve our border encoding described in 10.5. The fifth column of Table 10.2 documents the compression speed in tetrahedra per second for connectivity alone. The decompression speed is approximately the same. The speed does not depend on the size but more on the frequency of non manifold operations (compare the last column of Table 10.1). The last but one column contains the storage space consumed by the vertex coordinates, if compressed with the technique described in section 11.1. Finally, the last column shows that the vertex compression

doesn’t decrease the compression speed significantly.

Chapter 11

Encoding Mesh Attributes

In this section we describe simple methods for the compression of vertex locations (subsection 11.1) and vertex attributes (subsection 11.2).

11.1 Vertex Locations

In a first step we quantize each vertex coordinate to 16 bits according to the diagonal of the bounding box of all vertices. Thus the compression is lossy and for some applications not appropriate. All the meshes we received came in ASCII format with six to eight valid digits which is equivalent to 19-26 bits. We loose some information in the quantization step and the shape of small tetrahedra changed slightly, but no tetrahedron changed its orientation.

To encode the 16 bit coordinates arithmetically it turned out to be economical to split each coordinate into four packages of four bits. For each package we use a different set of adaptive frequencies for the arithmetic coder. This strategy dramatically reduced the storage space consumed by the arithmetic coder and increased the compression speed.

The next step in coordinate compression is delta coding. We encode the vertex coordinates during the compression of the connectivity. After each new vertex operation the difference vector from the center of the gate triangle to the new vertex is encoded. Thus we use the proximity information given by the tetrahedralization of the vertices. We can estimate the number of bits saved through delta coding with the following simple argument. Suppose the vertices are uniformly distributed. Then there are approximately $\sqrt[3]{v}$ vertices per coordinate axis and it should be possible to save $\mathrm{lb}\sqrt[3]{v}$ bits per coordinate. Thus the storage space consumed per vertex can be estimated with $48 - \mathrm{lb}v$ bits, which is about three bits above the actually achieved storage space.

A final improvement of two bits less storage space per vertex could be achieved by rotating the coordinate system such that the z-axis is the normal of the gate and the x-axis parallel to the zero edge. Quantization is done after changing to the new coordinate system. To avoid accumulation of rounding errors it is very important that during compression the center of the gate is computed with the same quantized coordinates

which are available to the decompression algorithm. The change of the coordinate system saved two bits in the x- and y-axes. The final storage space consumed per vertex by the coordinates is tabulated in Table 10.2 in the column labeled $\frac{\mathcal{L}_{\text{CB}}^{16\text{bit}}}{v}$.

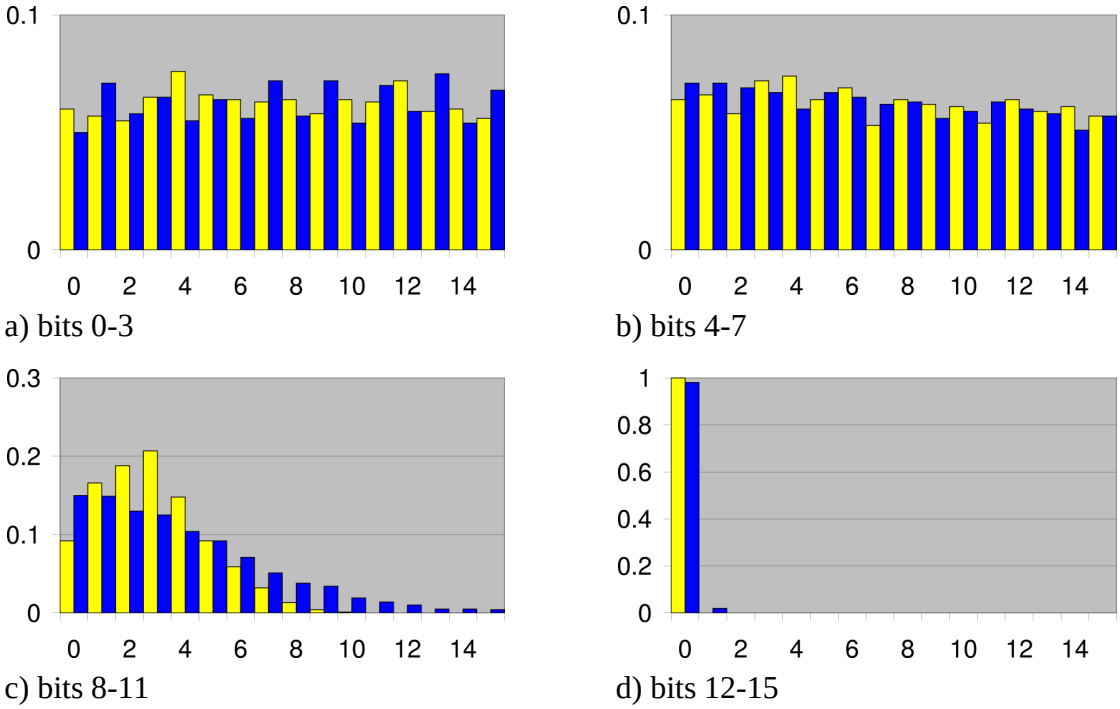


Figure 11.1: Distribution of coordinates.

Figure 11.1 shows the relative frequencies of the 16 different values of each 4 bit package in the case of the randomly generated mesh (see figure 9.1). The yellow bars represent the z-coordinate and the blue bars the x- and y-coordinates. The x- and y-coordinates were united as their distributions do not differ at all. The lower eight bits are distributed quite uniformly, whereas the higher four bits are nearly exclusively zero. The bits $8-11$ are especially interesting. The x- and y-coordinates frequencies have a Gaussian fall off, whereas the z-coordinate frequencies increase to a maximum at the value 3 and then drop of much faster than the x- and y-frequencies.

This encouraged us to predict the z-coordinate, which is the height of the new tetrahedron in the new vertex operation, from the height of the tetrahedron of the inner part which is adjacent to the gate. But the distribution of the z-coordinate frequencies was even smoothed out and the compression became worse. We also tried to predict the x- and y-coordinates from the interior adjacent tetrahedron but with a similar failure. All these tests were also performed on the more regular meshes of figure 9.1 with no success. The prediction of the tetrahedron height, which is proportional to its volume, should help in meshes where the sampling density changes significantly. But we still have to conclude that tetrahedral meshes are too irregular to predict vertex coordinates much better than with the proximity information of the connectivity alone.

11.2 Scalar and Vector Valued Attributes

The last section showed that tetrahedral meshes are not regular enough for a good prediction of vertex coordinates. Therefore, we propose to encode data given for the mesh elements in a different way. In this section we restrict ourselves to a scalar data function attached to the vertices. The approach can be extended in a natural way to vertex data by coordinate wise application. The vertex data is transmitted with each new vertex operation after the vertex coordinates. We propose delta encoding for the data function after an appropriate quantization. This time we can additionally use the vertex coordinates to predict the function value at the new vertex.

Let us denote the scalar data function with f and the location of the new vertex with $\vec{v}_n$. A linear approximation f_{lin} of the function f is of the form

$$f_{lin}(\vec{v}) = \vec{f_{lin}}^{\,T} \cdot \vec{v} + f_{lin}(\vec{0}). \tag{11.1}$$

Thus the linear approximation must be known at four locations in order to determine the unknowns $\vec{f_{lin}}$ and $f_{lin}(\vec{0})$. In a new vertex operation the new tetrahedron is always adjacent to a tetrahedron of the inner part, where the function f is already known. We can use the corner vertices of this tetrahedron $\vec{v}_0, \vec{v}_1, \vec{v}_2$ and $\vec{v}_3$ and the corresponding values of the data function $f(\vec{v}_i)$ to determine the unknowns of the linear approximation. The linear system of equations is

$$f(\vec{v}_i) = \vec{f_{lin}}^{\,T} \cdot \vec{v}_i + f_{lin}(\vec{0}), \quad i \in \{0,1,2,3\}.$$

If this linear system is solved and the values are plugged into equation 11.1 with $v = v_n$, we get as linear prediction at the location $\vec{v}_n$

$$\begin{aligned} f_{lin}(\vec{v_n}) &= \Delta\vec{f}^{T} \cdot \Delta V^{-1} \cdot (\vec{v}_n - \vec{v}_0) + f(\vec{v}_0), \text{ with} \\ \Delta\vec{f} &\stackrel{\text{def}}{=} \begin{pmatrix} f(\vec{v}_1) - f(\vec{v}_0) \\ f(\vec{v}_2) - f(\vec{v}_0) \\ f(\vec{v}_3) - f(\vec{v}_0) \end{pmatrix} \text{ and} \\ \Delta V &\stackrel{\text{def}}{=} \begin{pmatrix} (\vec{v}_1 - \vec{v}_0) & (\vec{v}_2 - \vec{v}_0) & (\vec{v}_3 - \vec{v}_0) \end{pmatrix}. \end{aligned}$$

The matrix ΔV can be inverted, iff the tetrahedron $(\vec{v}_0, \vec{v}_1, \vec{v}_2, \vec{v}_3)$ is not degenerated.

Chapter 12

Other Compression Methods

12.1 Grow & Fold Compression

The Grow & Fold method was developed by Szymczak and Rossignac [SR99]. It is very similar to the Topological Surgery triangle mesh compression technique. The tetrahedral mesh is encoded in two phases. In the first phase (grow) a tetrahedral spanning tree is encoded with three bits per tetrahedron. And in the second phase (fold) it is encoded how the tetrahedral spanning tree has to be folded together in order to obtain the original tetrahedral mesh.

12.1.1 Growing the Spanning Tree

For the explanation of the growth of the spanning tree, the notion of a halfface as introduced in section 1.5.4 is helpful. The halfface can also be interpreted as a pair of a tetrahedron and one of its four faces.

The spanning tree is grown starting with a halfface, which is incident to the border of the tetrahedral mesh, in the following frequently used recursive manner. The halfface is pushed onto a stack. As long as the stack is not empty the top halfface is popped from the stack. The tetrahedron of the popped halfface is marked visited. For each of the three remaining[1] faces in the current tetrahedron it is encoded with one bit, whether there is a not visited face-adjacent tetrahedron or not. If there is one the corresponding halfface is pushed onto the stack. This encoding strategy of the spanning tree corresponds to a depth-first traversal and consumes $3t$ bits.

Decompression works in the same way. The first tetrahedron introduces four vertices and all other tetrahedra one further vertex. Thus the decoded spanning tree contains $t+3$ different vertices, which are labeled in the order they are created.

Comparing this approach with the Cut-Border Machine corresponds to applying all new vertex operations at the beginning.

[1] unequal to the face in the halfface

12.1.2 Folding the Spanning Tree

The spanning tree is the result of cutting the tetrahedral mesh along the *cut-faces*. Each cut-face is split into two faces one for each of its two *exterior* halffaces. In the second phase the cut halffaces must be distinguished from border halffaces and the originally face adjacent halffaces must be brought together again. This is done with two operations: the more frequent *fold*- and the rare *glue*-operation. The fold-operation unifies two edge adjacent halffaces in the spanning tree, i.e. the so far folded tetrahedral mesh. And the glue-operation unifies two arbitrary halffaces but in turn for its generality consumes two indices to specify the halffaces in the spanning tree.

The operations are encoded in the *folding string*, which consists of two parts. In the first part for each of the $2t+1$ exterior halffaces[2] a two bit code is encoded in altogether $4t+2$ bits. A code of 0 tells that the halfface is either at the mesh border or will be glued to its originally face-adjacent halfface. The glued pairs of halffaces are specified at the end of the folding string with pairs of indices into the 0 labeled exterior halffaces, what consumes $2g\mathrm{lb}\lceil 2g+b\rceil$ bits, where g is the number of glue operations and b the number of border faces of the tetrahedral mesh. The fold operations are completely encodable by specifying for both participating halffaces the common edge or *folding edge*. The remaining cases 1, 2 and 3 of the two bit code for each exterior halfface are sufficient to specify the folding edge.

During compression the halffaces are visited in the same order as their bits are encoded in the spanning tree encoding. An exterior halfface pair is classified as participant in a fold operation, if it is edge-adjacent to its folding partner via an inner edge of the original tetrahedral mesh and if the two participants are the only halffaces incident upon this edge[3]. If this is the case, the corresponding folding codes for both partners are set. If not, the halfface is delayed until it can be folded with its partner by an update in their neighborhood caused by another folding operation. If no folding operation is possible anymore, a glue operation is applied to the first delayed halfface. Again the changed neighborhood of the glue operation is checked for possible folding operations. In this way the number of glue operations is minimized.

The decompression algorithm first unifies all glue faces and then foldes edge-adjacent folding partners. Folding is only allowed if the edge adjacent halffaces both specify the folding edge. Again folding is delayed until this condition is fulfilled. This can be implemented in linear running time by traversing all exterior halffaces once and checking for delayed folding operations in the neighborhood after each performed folding operation.

[2] the initial tetrahedron introduces three exterior halffaces and each addition of a tetrahedron removes changes one halfface to an interior halfface and adds three exterior halffaces

[3] if there would be more than two, during decompression the partners could not be determined anymore

12.1.3 Results

The Grow & Fold method allows to encode a single resolution tetrahedral mesh with $7t + 2g\mathrm{lb}\lceil 2g + b\rceil + 2$ bits. As the glue operations are very seldom typical tetrahedral meshes can be encoded with slightly more than $7t$ bits. Compression runs in $O(t\mathrm{lb}t)$ time as the glue indices must be determined and the decompression algorithm is linear in t.

The $7t$ bits can be improved by exploiting the fact that the spanning tree is encoded with t ones and $2t$ zeros with arithmetic coding to $6.75t$ bits.

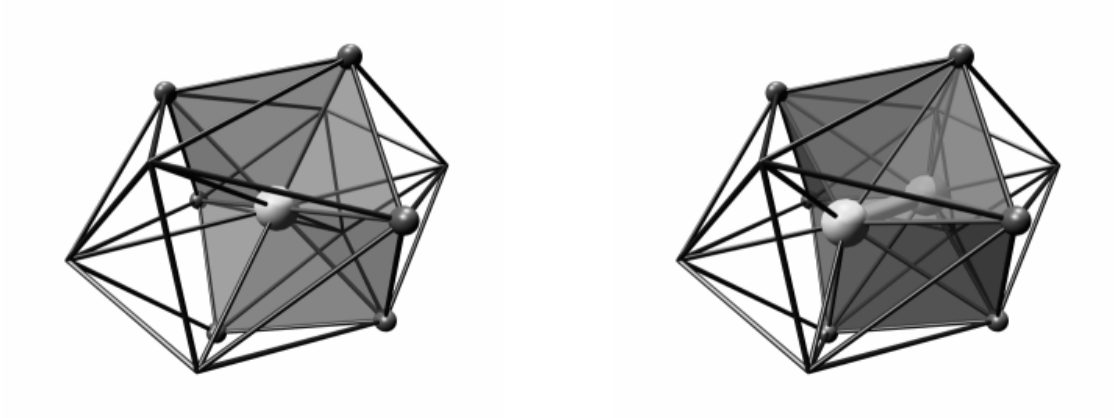
The most interesting question of theoretical content is whether closed tetrahedral meshes with manifold surface can always be encoded without glue operations and therefore with $6.75t$ bits.

12.2 Implant Sprays

The tetrahedral mesh compression method called Implant Sprays was developed by Pajarola and Rossignac [PRS99]. It is the generalization of the Compressed Progressive Meshes [PR00] and therefore a level split method (compare to section 2.3). The compression method is coupled to a simplification algorithm, which is based on the favorite edge collapse operation. The tetrahedral mesh is simplified level by level. For each level a maximal set of independent edge collapse operations is requested from the simplification algorithm, which supplies them in the order of decreasing error produced by the edge collapse according to an appropriate error measurement. In this way the simplifying compression algorithm collapses level by level. The information needed for the inverse vertex split operations is encoded. The coarsest level is then encoded with the Cut-Border Machine or the Grow & Fold method. The information needed by the decompression algorithm can be divided into two parts. The first part for each level is the identification of the to be split vertices. In the second part for each split vertex the incident triangles which have to be split into tetrahedra are specified. These triangles are called the *skirt*. Figure 12.1 shows a vertex split operation. The skirt triangles are shaded transparently in figure 12.1 a). During the split each of the skirt faces is split into a tetrahedron. Thus in terms of encoding, it remains to explain how to specify the split vertices and how to encode the skirts.

12.2.1 Split Vertices Specification

The split vertices are encoded with a bit stream. For each vertex in the current level one bit specifies, whether the vertex is a split vertex or not. If we assume that at each level each k^{th} vertex is split, the encoding of the split vertices consumes k bits per split vertex. As the base mesh is small one can assume that the split vertices specification accounts for kv bits.



a) b)

Figure 12.1: Vertex split operation.

12.2.2 Skirt Encoding

The problem of skirt encoding is best understood if one has a look at the triangle mesh formed by the not on the split vertex incident faces of the incident tetrahedra. In a regular case this triangle mesh looks like a sphere with the split vertex as center. Pajarola calls this surface the *orbital surface*. The intersection of the skirt with the orbital surface is a closed edge path on the orbital surface. Thus the brute force method to encode the skirt optimal is to enumerate all possible closed paths on the orbital surface and then encode the index of the actual path. As the average number of neighboring vertices of a split vertex is about 14 this approach is not feasible. Pajarola suggest to specify the path by the first vertex and then track the path along the orbital surface. The first path vertex is encoded as an index to a canonically ordered list of the vertices in the orbital surface, which consumes about $\mathrm{lb}14 < 3.9$ bits. Each edge can then be selected from the edges incident to the previous path vertex. As each vertex has about six neighbors in a triangle mesh, an edge can be specified by about $\mathrm{lb}6 < 3$ bits. As an edge in a tetrahedral mesh has in average about 5 neighboring tetrahedra, the path and therefore the skirt can be encoded with $4 + 5 \cdot 3 = 19$ bits.

12.2.3 Implementation & Results

For the selection of a maximal independent set of edge collapses it is important to know, when two edge collapses are independent and what edge collapses are valid. Two edge collapses are independent if their sets of tetrahedra, which are incident to one or two vertices of the edge, are disjunct. Valid is an edge collapse of edge $e = (v_1, v_2)$ if the following conditions are fulfilled:

- If a vertex w is incident to v_1 and v_2, the triangle (w, v_1, v_2) must be a triangle of the tetrahedral mesh.

- If two vertices w_1 and w_2 form the triangles (v_1, w_1, w_2) and (v_2, w_1, w_2), the quadruple (v_1, v_2, w_1, w_2) must be a tetrahedron in the tetrahedral mesh.

The precondition of the first condition is fulfilled if two tetrahedra not incident to the collapsed edge form a roof upon an incident tetrahedron. The collapse would collapse all three tetrahedra in one face, but the skirt decoding recovers only one tetrahedron from this face and would produce a different tetrahedral mesh. Thus this case must be prohibited. The second precondition arises if a large virtual tetrahedron incident upon the collapsed edge contains a further vertex and therefore actually consists of three tetrahedra. Again the edge collapse will collapse all three tetrahedra into one face, what cannot be recovered by the decompression algorithm.

The restrictions on the selection of edge collapses cause a relatively large k of 12 to 16, what means that on each level only each $12-16^{\text{th}}$ vertex is split. This results in an overall storage space consumption of about $19+14=33$ bits per vertex. For an average of about six tetrahedra per vertex this would result in about 5.5 bits per vertex, what is only twice as much as the Cut-Border Machine consumes for the single resolution encoding. The measurements of Pajarola confirm this estimations.

12.2.4 Improved Implant Sprays Encoding

It is not too difficult to improve the results of Pajarola. First of all we can simply apply arithmetic coding to the bit stream that defines the split vertices. Let there be v vertices on the current level and v/k split vertices. With the sparse flag encoding desribed in section 3.2.4 the split vertex bits can be encoded with $-\text{lb}1/k = \text{lb}k$ bits. For $k=16$ the split vertices can be encoded with 4 bits per vertex and not with 16.

On can also improve the skirt encoding with a little bit more effort. As in the original encoding we want to encode the skirt by the closed cycle of edges on the orbital surface. We want an encoding scheme that consumes a fixed number of τ bits per encoded edge. As each encoded edge corresponds to one skirt face and as each skirt face generates exactly one tetrahedron, we can derive at once that our coding scheme will use – for compressed models with sufficiently small base tetrahedral mesh – τ bits per tetrahedron for skirt encoding. The idea is to encode the cycle length and use this length to encode the last two or three edges in the edge cycle more efficiently in order to compensate the additionally consumed storage space for the length encoding and the encoding of the first edge.

The length l of a cycle is at least three. We use an arithmetic coding scheme that consumes

$$\mathcal{S}_{\text{length}}(l) \stackrel{\text{def}}{=} a \cdot l + b \text{ bits},$$

where a and b are constants, which will be determined later on. As all frequencies of the lengths must sum up to one, a and b must fulfill the equation

$$1 = \sum_{l>2} 2^{-\mathcal{S}_{\text{length}}(l)} = 2^{b-3a} \big/ \left(1-2^{-a}\right). \tag{12.1}$$

The second condition for a and b will distribute the consumed bits equally among the cycles with different length.

Let us now describe how we encode the edges of the cycle. The first edge is chosen according to the strategy described below and is encoded directly by an index ranging from 1 to the number of edges in the orbital surface. Then the cycle is built edge by edge starting from the first edge in a canonically chosen direction as also Pajarola does. The last two edges are encoded differently. As they must close the cycle, there are only a few possibilities. Figure 12.2 shows in the left two cases the typical closing of a cycle. For

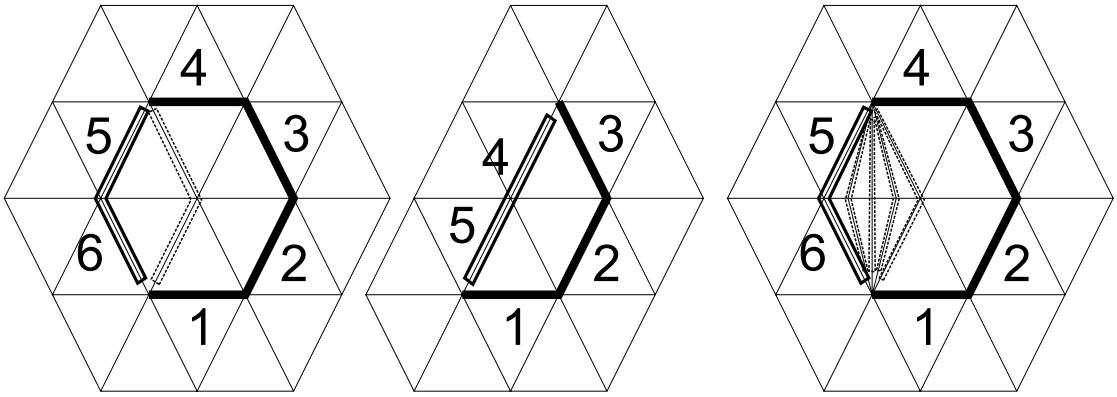


Figure 12.2: Different closings of cycles. The possible cases for the last two edges are surrounded with dashed lines.

the very left cycle there are two possibilities, which can be encoded with one bit. In case of the middle cycle the one and only possibility does not consume any bit. But there are also cases with more than two possibilities as shown in the right cycle of figure 12.2. To avoid these cases, we choose the first encoded cycle edge by minimizing the number of closing cases. For this we look at each edge in the cycle and determine the number of connections via two edges between the endings of the previous two edges, which will be the closing edges, if this edge is chosen as first encoded edge. In this way we can also exploit the choice of the first edge and the cost for encoding the last two edges decreases to about one bit.

Let us analyze this encoding for the average case. We assume an average number of $\bar{o}_{v\to v} = 14$ vertex-vertex neighbors. From the Euler equation for closed surfaces applied to the orbital surface, we now that there are $\bar{e} = 3(\bar{o}_{v\to v} - 2) = 42$ edges in the orbital surface. Thus we can select the first cycle edge among all edges with an index consuming

$$\mathcal{S}_{\text{edge}} \stackrel{\text{def}}{=} \text{lb}\bar{e} \approx 5.4 \text{ bits.}$$

The fractional bit consumption can actually be achieved by the use of arithmetic coding. As the average number of vertex-vertex neighbors on the orbital surface is about six, there are five possibilities for each further edge, what can be encoded in

$$\mathcal{S}_{\to\text{edge}} \stackrel{\text{def}}{=} \text{lb}5 \approx 2.322 \text{ bits.}$$

Including the length encoding of the cycle and the bit for the last two edges, we end up with the storage space consumption for a complete cycle of length l

$$\mathcal{S}_{\text{cycle}}(l) \stackrel{\text{def}}{=} \mathcal{S}_{\text{length}}(l) + \mathcal{S}_{\text{edge}} + (l-3) \cdot \mathcal{S}_{\rightarrow\text{edge}} + 1.$$

If we plug in the definition for the length encoding and divide by the cycle length we just get the number of bits consumed per edge, i.e. per split face, i.e. per tetrahedron, what is just τ

$$\tau = \mathcal{S}_{\text{cycle}}(l)/l = a + \mathcal{S}_{\rightarrow\text{edge}} + \frac{b + \mathcal{S}_{\text{edge}} - 3\mathcal{S}_{\rightarrow\text{edge}} + 1}{l}.$$

In order to distribute the consumed bits equally among the edges in cycles of different length, we want this term in the previous equation, that depends on l, to vanish. Thus we set $b = 3\mathcal{S}_{\rightarrow\text{edge}} - \mathcal{S}_{\text{edge}} - 1$. Substitution of b in equation 12.1 yields a and τ. We can summarize by plugging in $\bar{o}_{v\rightarrow v} = 14$

$$\bar{o}_{v\rightarrow v} = 14 \rightarrow a = 0.45, b = 0.55 \quad \Longrightarrow \quad \forall l > 2 : \tau < 2.8 \text{ bits}.$$

With an average number of six tetrahedra per vertex in a typical tetrahedral mesh, we can conclude that the encoding of the split vertices consumes $3.9/6 \approx 0.65$ bits per tetrahedron. Together with the skirt encoding this sums up to 3.45 bits per tetrahedron which is quite an improvement over the 5.7 bits achieved by Pajarola.

12.3 Progressive Simplicial Complexes

Popovic and Hoppe [PH97] realized that a lot of triangle meshes are non manifold. Further more models can be simplified to a much coarser level with a smaller approximation error, if one allows edge collapse operations that produce non manifold spots or even singleton edges and vertices. Therefore they generalized the edge collapse operation and the progressive mesh representation to arbitrary simplicial complexes in a very elegant way. The compressed representation of the simplicial complex is called *Progressive Simplicial Complex* (PSC). As tetrahedral meshes are also simplicial complexes, the method also works for them, although Popovic did not implement this case.

The PSC representation is very similar to the progressive mesh representation. It is built during the simplification of the simplicial complex, which is based on vertex unification operations, i.e. two arbitrary vertices in the mesh may be collapsed. The inverse opration of each vertex unification the so called *generalized vertex split* operation is recorded. Thus PSC representation contains the mesh in the coarsest resolution and a sequence of generalized vertex split operations in reverse order of the corresponding vertex unification operations. Each split operation is given by a vertex index specifying the to be split vertex and a split code out of $\{1, 2, 3, 4\}$ for all incident simplices of the split vertex. Figure 12.3 shows the meaning of the different split codes for the simplices

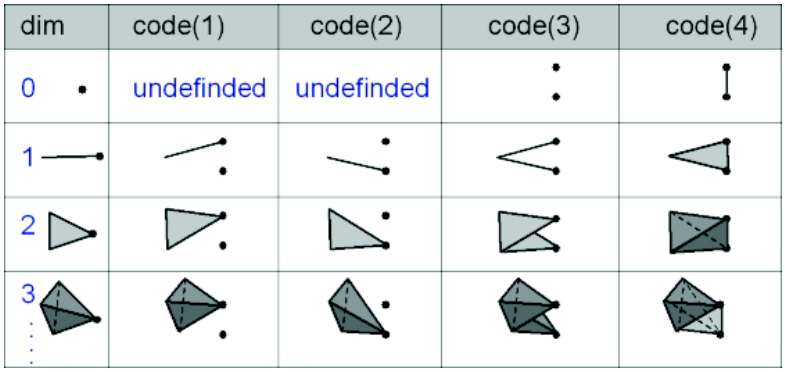
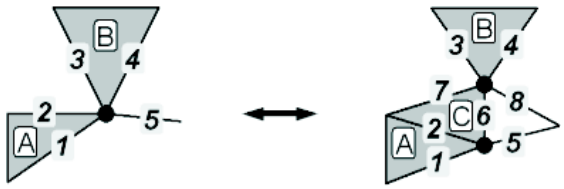


Figure 12.3: The different split codes for simplices of different dimensions.

of different dimensions. The 0-simplex is just the split vertex. There are only two possibilities: either the vertex splits into two separate vertices or an edge connects the two split vertices. For all other dimensional simplices there are four possibilities. After the generalized vertex split operation the simplex will either be incident to the first resulting vertex (code 1), to the second (code 2) or to both (code 3). The fourth code specifies the case, when the simplex is split into two simplices, that span a simplex of higher dimension. For example if a triangle (2-simplex) is split with split code 4, it will be split into two triangles incident to each of the resulting vertices and the space between them is connected by a tetrahedron.



simplex	●	1	2	3	4	5	A	B
code	4	1	4	2	2	3	1	2

Figure 12.4: An example of a generalized vertex split operation together with the split codes for all simplices.

Figure 12.4 shows a quite general example of a generalized vertex split operation and the codes for all simplices.

In the case of triangle meshes the average consumption for the split codes can be estimated from the average vertex split operation, where the split vertex has eight neighbors, what results from an edge collapse of two order six vertices (see figure 12.5). Then there are eight edges and eight triangles incident to the split vertex. Case 4 may not appear for triangles and therefore the split codes of the average vertex split consumes $1 + 8 \cdot 2 + 8 \cdot \mathrm{lb}3 \approx 30$ bits, what is exactly what Popovic reports.

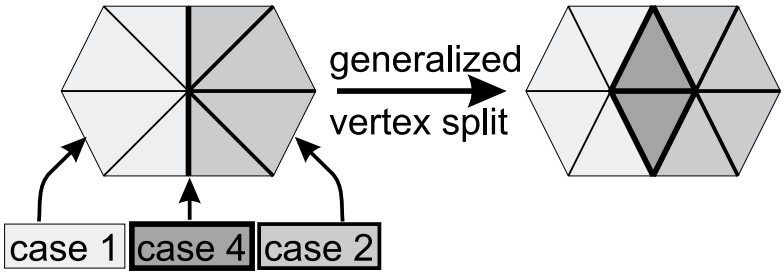


Figure 12.5: The average vertex split operation in case of triangle meshes.

There are two types of dependencies among the split codes of different dimensions. Firstly, if a simplex has split code $c \in \{1, 2\}$, all simplices of higher dimension, which are incident to this simplex, have the same split code c. And secondly, if a simplex has split code 3, none of the incident simplices of higher dimension have split code 4. Let us apply these constraints to the coding of the average generalized vertex split operation shown in figure 12.5. There are three edges with code 1 (fine edges), three with code 2 (middle thick edges) and two with code 4 (thick edges). All triangles are incident to at least one edge with code 1 (light shaded triangles) or 2 (middle dark shaded triangles) and therefore only one code is allowed and need not be encoded. Thus the average vertex split consumes altogether $1 + 8 \cdot 2 = 17$ bits. This is slightly more than the 14 bits reported by Popovic but is still very close to the measurements. With the help of a non-uniform probability model and arithmetic coding, the encoding can be improved to about 8 bits per split vertex.

Now we can generalize these ideas and estimate the storage space consumed by PSC when applied to tetrahedral meshes. Figure 12.1 shows a vertex split operation for a split vertex with twelve neighbors. Again the split code for the split vertex can be encoded with one bit. Then the split codes for the twelve edges incident to the split vertex need to be encoded with two bits each. The six edges incident upon the shaded split faces receive the split code 4 and all other edges to the front split code 1 and the ones to the back split code 2. Thus for all faces incident to the split vertex, which are not among the split faces, there is only one possible split code, which therefore need not be encoded. Only the split codes for the six split faces have to be encode with two bits each. Finally, all tetrahedra are incident to at least one edge with split code 1 or 2. Thus nothing has to be encoded for the tetrahedra. Altogether the storage space consumption for the average case of a generalized vertex split operation sums up to $1+12\cdot2+6\cdot2 = 37$ bits or 43 bits if we use an average of sixteen neighbors per vertex. It is not quite clear, how much a non-uniform probability model could improve upon this. A reduction to the half, i.e. 20 bits, is probably a too good estimation as already the improved skirt encoding proposed in section 12.2 consumes 15 bits per vertex split. But 25 bits per vertex seems to be reasonable.

As the encoding of the indices of the split vertices consumes a logarithmic amount of bits in terms of the total number of vertices, the level split approach should also be

applied to progressive simplicial complexes. In a level split approach again a maximally independent set of vertex unification operations would be applied during the simplification process and the progressive representation would encode the indices of the split vertices level by level with one bit per vertex on the current level. Similar coding results would be achieved as in case of the implant sprays and the split vertices could be encoded with about five bits per vertex. Altogether the level split version of PSC would consume 30 bits per vertex or $5 - 6$ bits per tetrahedron for encoding the connectivity of a non manifold tetrahedral mesh.

Chapter 13

Conclusion & Directions for Future Work

The area of tetrahedral mesh compression is only about two years old. Therefore only a few methods have been proposed yet. All of them are generalizations of triangle mesh compression techniques. On the one hand there is a huge amount of triangle mesh compression techniques which all could be generalized to the tetrahedral case but on the other hand the existing tetrahedral mesh compression approaches already generalize the best triangle mesh compression methods. Further more the implementation of tetrahedral mesh compression techniques is significantly more complicated and therefore the threshold for the generalization of triangle mesh compression techniques is quite high. But it is probable that we will see a variety of tetrahedral methods in future.

The important point in tetrahedral mesh compression is that there are about six times as many tetrahedra than there are vertices in a typical mesh. Thus the connectivity consumes in a non compressed standard representation about six times more storage space than the vertex coordinates. For this reason it is very important to encode the connectivity of a tetrahedral mesh as efficient as possible. At the moment there are methods for single resolution tetrahedral mesh compression [GGS99, SR99], for progressive compression [PRS99] and for non manifold compression [PH97]. The best single resolution consumes about 2 bits per tetrahedron, the progressive method in the improved version about 4 bits[1] per tetrahedron and the improved[2] PSC method encodes non manifold tetrahedral meshes with about 6 bits per tetrahedron. The two bits per tetrahedron are for typical meshes equivalent to about twelve bits per vertex, what corresponds to a reduction rate of $30:1$ over the standard representation in case of the connectivity.

For the geometry, i.e. the vertex locations and attributes, things look much worse. Although the corresponding storage space is comparably small in the standard representation, it is three times larger in the compressed representation of the Cut-Border Machine. The Cut-Border Machine quantizes the coordinates to fifteen bits and applies

[1] see improvements in section 12.2

[2] see discussion for the tetrahedral case in section 12.3

delta coding. But no simple strategy could be found based on the knowledge of triangle mesh geometry coding techniques that efficiently encodes the quantized coordinates. One is typically left with $30 - 36$ of the 48 bits per vertex. This can be explained from the fact that the tetrahedral mesh does not describe a surface or hyper-surface of a higher dimensional space as triangles meshes do. Therefore all three coordinates define sampling locations, which can be chosen quite arbitrarily and the choice strongly depends on the generation algorithm. In case of Delaunay tetrahedralized random point clouds for example only the 36 bits per vertex could be achieved. But there might be better encoding schemes for the point locations, the encoding of which has not been addressed by any other approach than the Cut-Border Machine. In case of the vertex and tetrahedral attributes much better results should be achievable, as they normally represent quite smooth functions defined in the three dimensional space. With the knowledge of the vertex – i.e. the sampling – locations one can develop the function into Taylor or other polygon basis functions and perform higher order predictions.

The theoretical aspect of tetrahedral mesh compression is not explored yet in any detail. The major question, whether a tetrahedral mesh can be encoded in linear space and time in terms of the number of tetrahedra, cannot be answered yet and will be a hot area of research in the future.

Future work will also concentrate on the application of compressed tetrahedral meshes. Important questions are how to render tetrahedral meshes from a compressed representation and how to integrate compressed representations into Finite Element solvers in order to allow simulations on larger data sets. For Finite Element other cell types than tetrahedra are important, such as hexahedra or octahedra. Mixed polyhedra meshes are also very common. Therefore a generalization of the tetrahedral mesh compression techniques to the general polyhedral case is important. The most suitable method for this task seems to be the Face Fixer proposed by Isenburg [IS00].

Bibliography

[Abr63] N. Abramson. *Information Theory and Coding*. McGraw-Hill, New York, 1963.

[AHMS94] Arkin, Held, Mitchell, and Skiena. Hamiltonian triangulations for fast rendering. In *ESA: Annual European Symposium on Algorithms*, 1994.

[Bau75] Bruce G. Baumgart. A Polyhedron Representation for Computer Vision. In *Proceedings of the National Computer Conference*, pages 589–596, 1975.

[BE92] Marshall Bern and David Eppstein. Mesh generation and optimal triangulation. Technical report, Xerox PARC, March 1992. CSL-92-1. Also appeared in "Computing in Euclidean Geometry", F. K. Hwang and D.-Z. Du, eds., World Scientific, 1992.

[BG96] Reuven Bar-Yehuda and Craig Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15(2):141–152, April 1996. ISSN 0730-0301.

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12(3), pages 286–292, August 1978.

[BPZ99a] Bajaj, Pascucci, and Zhuang. Single resolution compression of arbitrary triangular meshes with properties. In *DCC: Data Compression Conference*. IEEE Computer Society TCC, 1999.

[BPZ99b] C. Bajaj, V. Pascucci, and G. Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In B.Hamann D.Ebert, M.Gross, editor, *Proceedings of the Visualization '99 Conference*, pages 307–316, San Francisco, CA, October 1999. IEEE Computer Society Press.

[BW86] Gary Bishop and David M. Weimer. Fast Phong shading. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 103–106, August 1986.

[CC78] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, September 1978.

[CCMS97] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, 13(5):228–246, 1997. ISSN 0178-2789.

[CGHK98] R. C. Chuang, A. Garg, X. He, and M. Kao. Compact encodings of planar graphs via canonical orderings and multiple parentheses. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 118–129, 1998.

[CH84] Gordon V. Cormack and R. Nigel Horspool. Algorithms for adaptive Huffman codes. *Information Processing Letters*, 18(3):159–165, March 1984.

[Cha91] B. Chazelle. Triangulating a Simple Polygon in Linear Time. *Discrete and Computational Geometry*, 6:485–524, November 1991.

[Cho97] Mike M. Chow. Optimized geometry compression for real-time rendering. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization ́97*, pages 346–354. IEEE, November 1997.

[CKS98] S. Campagna, L. Kobbelt, and H.-P. Seidel. Directed edges–a scalable representation for triangle meshes. *Journal of Graphics Tools*, 3(4):1–12, 1998.

[Cla89] U. Claussen. Real time phong shading. In D. Grimsdale and A. Kaufman, editors, *Fifth Eurographics Workshop on Graphics Hardware*, 1989.

[CMPS97] Paolo Cignoni, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Multiresolution representation and visualization of volume data. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):352–369, October–December 1997. ISSN 1077-2626.

[COLR99] Daniel Cohen-Or, David Levin, and Offir Remez. Progressive compression of arbitrary triangular meshes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings of the 1999 IEEE Conference on Visualization (VIS-99)*, pages 67–72, N.Y., October 25–29 1999. ACM Press.

[Cro84] Franklin C. Crow. Summed-area tables for texture mapping. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 207–212, July 1984.

[Dee95] Michael F. Deering. Geometry compression. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 13–20. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[DS78] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10:356–360, September 1978.

[DS97] M. Denny and C. Sohler. Encoding a triangulation as a permutation of its point set. In *Proceedings of the 9th Canadian Conference on Computational Geometry*, pages 39–43, August 1997. held in Ontario, August 11-14.

[DWS+88] Michael F. Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 21–30, August 1988.

[EDD+95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 173–182. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[EJRW96] I. Ernst, D. Jackèl, H. Rüsseler, and O. Wittig. Hardware-supported bump mapping. *Computers & Graphics*, 20(4):515–521, July 1996.

[ESV96] Francine Evans, Steven S. Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96*. IEEE, October 1996. ISBN 0-89791-864-9.

[Geo91] P. L. George. *Automatic Mesh Generation*. John Wiley & Sons, New York, NY, 1991. excellent coverage of current state-of-the-art in automatic meshing algorithms for finite element methods.

[GGS99] Stefan Gumhold, Stefan Guthe, and Wolfgang Straßer. Tetrahedral mesh compression with the cut-border machine. In B.Hamann D.Ebert, M.Gross, editor, *Proceedings of the Visualization '99 Conference*, pages 51–58, San Francisco, CA, October 1999. IEEE Computer Society Press.

[GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 209–216. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[GH99] Stefan Gumhold and Tobias Hüttner. Multiresolution rendering with displacement mapping. In Steven Molnar and Bengt-Olaf Schneider, editors, *1999 EUROGRAPHICS / SIGGRAPH Workshop on Graphics Hardware*, pages 55–66, New York City, NY, August 1999. ACM SIGGRAPH / Eurographics, ACM Press.

[GHJ+98] Tran S. Gieng, Bernd Hamann, Kenneth I. Joy, Gregory L. Schussman, and Issac J. Trotts. Constructing hierarchies for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):145–161, April 1998.

[Gla86] Andrew Glassner. Adaptive precision in texture mapping. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 297–306, August 1986.

[GS98] Stefan Gumhold and Wolfgang Straßer. Real time compression of triangle mesh connectivity. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 133–140. ACM SIGGRAPH, Addison Wesley, July 1998.

[GTLH98] Andre Gueziec, Gabriel Taubin, Francis Lazarus, and William Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. In Scott Grisson, Janet McAndless, Omar Ahmad, Christopher Stapleton, Adele Newton, Celia Pearce, Ryan Ulyate, and Rick Parent, editors, *Conference abstracts and applications: SIGGRAPH 98, July 14–21, 1998, Orlando, FL*, Computer Graphics, pages 245–245, New York, NY 10036, USA, 1998. ACM Press.

[Gué99] André Guéziec. Locally toleranced surface simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2), April 1999.

[Gum99] Stefan Gumhold. Improved cut-border machine for triangle mesh compression. In *Erlangen Workshop '99 on Vision, Modeling and Visualization*, Erlangen, Germany, November 1999. IEEE Signal Processing Society.

[Gum00] Stefan Gumhold. New bounds on the encoding of planar triangulations. Technical Report WSI–2000–1, Wilhelm-Schickard-Institut für Informatik, University of Tübingen, Germany, January 2000.

[GVSS00] I. Guskov, K. Vidimce, W. Sweldens, and P. Schröder. Normal meshes. to appear in Siggraph 2000, July 2000.

[HÖ99] T. Hüttner. Fast footprint mipmapping. In *to appear in Proc. of Eurographics/SIGGRAPH workshop on graphics hardware 1999*, 1999.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 19–26, August 1993.

[Hec83] Paul S. Heckbert. Texture mapping polygons in perspective. TM 13, NYIT Computer Graphics Lab, April 1983.

[Hel98] M. Held. Efficient and reliable triangulation of polygons. In Franz-Erich Wolter and Nicholas M. Patrikalakis, editors, *Proceedings of the Conference on Computer Graphics International 1998 (CGI-98)*, pages 633–645, Los Alamitos, California, June 22–26 1998. IEEE Computer Society.

[HH85] James H. Hester and Daniel S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys*, 17(3):295–311, September 1985.

[HKM96] Martin Held, James T. Klosowski, and Joseph S. B. Mitchell. Collision detection for fly-throughs in virtual environments. In *Proceedings of the Twelfth Annual Symposium On Computational Geometry (ISG '96)*, pages V13–V14, New York, May 1996. ACM Press.

[Hop96] Hugues Hoppe. Progressive meshes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Inst. Radio Eng.*, pages 1098–1101, September 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.

[Inc91] Silicon Graphics Inc. GL programming guide. 1991.

[IR82] Alon Itai and Michael Rodeh. Representation of graphs. *Acta Informatica*, 17(2):215–219, June 1982.

[IS99a] Martin Isenburg and Jack Snoeyink. Mesh collapse compression. In *Proceedings of the Conference on Computational Geometry (SCG '99)*, pages 419–420, New York, N.Y., June 13–16 1999. ACM Press.

[IS99b] Martin Isenburg and Jack Snoeyink. Spirale reversi: Reverse decoding of the edgebreaker encoding. Technical Report TR-99-08, Department of Computer Science, University of British Columbia, October 4 1999. Mon, 04 Oct 1999 17:52:00 GMT.

[IS00] M. Isenburg and J. Snoeyink. Face fixer: Compressing polygon meshes with properties. to appear in Siggraph 2000, July 2000.

[Ise00] M. Isenburg. Triangle strip compression. In *Proceedings Graphics Interface 2000*, pages 197–204. Morgan Kaufmann Publishers, May15–17 2000.

[JR99] A. Szymczak J. Rossignac. Wrap & zip: Linear decoding of planar triangle graphs. Technical Report GIT-GVU-99-08, Georgia Institute of Technology, August 1999.

[KB89] A. A. M. Kuijk and E. H. Blake. Faster phong shading via angular interpolation. *Computer Graphics Forum*, 8(4):315–324, December 1989.

[Ket98] Kettner. Designing a data structure for polyhedral surfaces. In *COMPGEOM: Annual ACM Symposium on Computational Geometry*, 1998.

[KG00a] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. to appear in Siggraph 2000, July 2000.

[KG00b] B. Kronrod and C. Gotsman. Efficient coding of non-triangular meshes. preprint, 2000.

[KKT90] D. G. Kirkpatrick, M. M. Klawe, and R. E. Tarjan. Polygon triangulation in $o(n \log \log n)$ time with simple data-structures. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 6th Annual Symposium on Computational Geometry (SCG '90)*, pages 34–43, Berkeley, CA, June 1990. ACM Press.

[KLS96] Reinhard Klein, Gunther Liebich, and Wolfgang Straßer. Mesh reduction with error control. In *IEEE Visualization '96*. IEEE, October 1996. ISBN 0-89791-864-9.

[KR99] Davis King and Jarek Rossignac. Guaranteed 3.67V bit encoding of planar triangle graphs. In *Proceedings of 11th Canadian Conference on Computational Geometry*, pages 146–149, 1999.

[Kug98] Anders Kugler. Imem: An intelligent memory for bump- and reflection-shading. In *Proceedings of Eurographics/SIGGRAPH Hardware Workshop '98*, pages 113–122. ACM SIGGRAPH, August 1998.

[Lan84] Glen G. Langdon, Jr. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, March 1984.

[LK98a] J. Li and C.-C. Kuo. A dual graph approach to 3d triangle mesh compression. In *IEEE International Conference on Image Processing*, Chicago, 1998.

[LK98b] J. Li and C.-C. Kuo. Progressive coding of 3d graphics models. In *Proceedings of the IEEE, Special Issue on Multimedia Signal Processing*, volume 86(6), pages 1052–1063, June 1998.

[LMH00] A. Lee, H. Moreton, and H. Hoppe. Displaced subdivision surfaces. to appear in Siggraph 2000, July 2000.

[LSS+98] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. *Computer Graphics*, 32(Annual Conference Series):95–104, August 1998.

[Man88] M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Md, 1988.

[McC98] Tulika Mitra and Tzi cker Chiueh. A breadth-first approach to efficient mesh traversal. In Stephen N. Spencer, editor, *Proceedings of the Eurographics / Siggraph Workshop on Graphics Hardware (EUROGRAPHICS-98)*, pages 31–38, New York, August 31–September 1 1998. ACM Press.

[Mey97] Scott Meyers. *Effective C++: 50 specific ways to improve your programs and designs. – 2. ed.* Addison-Wesley, Reading, MA, USA, 1997.

[NB94] X. Ni and M. S. Bloor. Performance Evaluation of Boundary Data Structures. *IEEE Computer Graphics and Applications*, 14(6):66–77, 1994.

[NDW97] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide — The Official Guide to Learning OpenGL, Version 1.1*. Addison-Wesley, Reading, MA, USA, 1997.

[PAC97] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Proceedings of SIGGRAPH 97*, pages 303–306. ACM SIGGRAPH, August 1997.

[Pas76] R. Pasco. *Source coding algorithms for fast data compression*. PhD thesis, Stanford University, Palo Alto, CA, 1976.

[PH97] Jovan Popović and Hugues Hoppe. Progressive simplicial complexes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 217–224. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[Pho75] Bui-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[PR00] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, 2000.

[PRS99] Renato B. Pajarola, Jarek Rossignac, and Andrzej Szymczak. Implant sprays: Compression of progressive tetrahedral mesh connectivity. In David Ebert, Markus Gross, and Bernd Hamann, editors, *Proceedings of the 1999 IEEE Conference on Visualization (VIS-99)*, pages 299–306, N.Y., October 25–29 1999. ACM Press.

[PS97] E. Puppo and R. Scopigno. Simplification, lod and multiresolution - principles and applications. In *Eurographics'97 Tutorial Notes*. Eurographics association, 1997.

[Ris76] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, May 1976.

[RL79] Jorma J. Rissanen and Glen G. Langdon, Jr. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, March 1979.

[Ros98] J. Rossignac. Edgebreaker: connectivity compression for triangle meshes. Technical Report GIT-GVU-98-35, Georgia Institute of Technology, October 1998.

[RR96] Remi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):C67–C76, C462, September 1996.

[Sei91] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1:51–64, 1991. Seidel's randomized algorithm runs in $O(n \log^* n)$ expected time and is simpler than the deterministic $O(n)$ algorithm due to B. Chazelle.

[SG98] Oliver G. Staadt and Markus H. Gross. Progressive tetrahedralizations. In *Proceedings IEEE Visualization '98*, pages 397–402. IEEE, 1998.

[SKS96] Andreas Schilling, Günter Knittel, and Wolfgang Straßer. Texram: A smart memory for texturing. *IEEE Computer Graphics & Applications*, 16(3):32–41, May 1996.

[SP91] N. S. Sapidis and R. Perucchio. Domain delaunay tetrahedrization of solid models. *Internat. J. Comput. Geom. Appl.*, 1(3):299–325, 1991.

[SR99] Andrzej Szymczak and Jarek Rossignac. Grow & fold: Compression of tetrahedral meshes. In Willem F. Bronsvoort and David C. Anderson, editors, *Proceedings of the Fifth Symposium on Solid Modeling and Applications (SSMA-99)*, pages 54–64, New York, June 9–11 1999. ACM Press.

[ST90] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics (San Diego Workshop on Volume Visualization)*, volume 24(5), pages 63–70, November 1990.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 65–70, July 1992.

[TG98] Costa Touma and Craig Gotsman. Triangle mesh compression. In Wayne Davis, Kellogg Booth, and Alain Fourier, editors, *Proceedings of the 24th Conference on Graphics Interface (GI-98)*, pages 26–34, San Francisco, June18–20 1998. Morgan Kaufmann Publishers.

[TGHL98] Gabriel Taubin, André Gueziec, William Horn, and Francis Lazarus. Progressive forest split compression. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 123–132. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[THJW98] Isaac J. Trotts, Bernd Hamann, Kenneth I. Joy, and David F. Wiley. Simplification of tetrahedral meshes. In *Proceedings of the 9th Annual IEEE Conference on Visualization (VIS-98)*, pages 287–296, New York, October 18–23 1998. ACM Press.

[TR96] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. Technical report, Yorktown Heights, NY 10598, January 1996. IBM Research Report RC 20340.

[TR98] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.

[Tur92] Greg Turk. Re-tiling polygonal surfaces. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 55–64, July 1992.

[Tut62] W. Tutte. A census of planar triangulations. *Canadian Journal of Mathemetics*, 14:21–38, 1962.

[Wei85] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, January 1985.

[Wil83] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17(3), pages 1–11, July 1983.

[Wil92] Peter L. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[XHM99] Xinyu Xiang, Martin Held, and Joseph S. B. Mitchell. Fast and effective stripification of polygonal surface models. In Stephen N. Spencer, editor, *Proceedings of the Conference on the 1999 Symposium on interactive 3D*

Graphics, pages 71–78 (Color Plate: 224), New York, April 26–28 1999. ACM Press.

[ZCK97] Yong Zhou, Baoquan Chen, and Arie Kaufman. Multiresolution tetrahedral framework for visualizing regular volume data. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization ́97*, pages 135–142. IEEE, November 1997.

Lebenslauf Stefan Gumhold

13.9.1971	geboren in Ruit auf den Fildern
9.1982-6.1991	Raichberg-Gymnasium Ebersbach/Fils
18.6.1991	Abitur
7.1991-10.1992	Zivildienst an der Medizinischen Klinik in Tübingen
10.1992-7.1995	Studium an der Eberhard-Karls-Universität Tübingen
20.9.1994	Vordiplom Physik
6.10.1994	Vordiplom Informatik
9.1995-6.1996	Auslandsstudium an der University of Massachusetts at Boston
26.12.1996	Master of Science (Department of Applied Physics)
10.1996-12.1997	Studium an der Eberhard-Karls-Universität Tübingen
1.1998-8.2000	Wissenschaftlicher Mitarbeiter im Sonderforschungsbereich 382 Teilprojekt D1 ”Objektorientierte Graphik” am Lehrstuhl für Graphisch Interaktive System des Wilhelm-Schickard-Instituts für Informatik an der Eberhard-Karls-Universität Tübingen
31.3.1998	Diplom in Informatik ausgestellt von der Fakultät für Informatik der Eberhard-Karls-Universität Tübingen
31.5.2000	Antrag auf Zulassung zum Promotionsverfahren beim Dekanat der Fakultät für Informatik der Eberhard-Karls-Universität Tübingen