Compression and Streaming of Polygon Meshes

by
Martin Isenburg

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2005

Approved by:

Jack Snoeyink, Advisor

Craig Gotsman, Reader

Peter Lindstrom, Reader

Dinesh Manocha, Committee Member

Ming Lin, Committee Member

ABSTRACT
MARTIN ISENBURG: Compression and Streaming of Polygon Meshes
(Under the direction of Jack Snoeyink)

Polygon meshes provide a simple way to represent three-dimensional surfaces and are the de-facto standard for interactive visualization of geometric models. Storing large polygon meshes in standard indexed formats results in files of substantial size. Such formats allow listing vertices and polygons in any order so that not only the mesh is stored but also the particular ordering of its elements. Mesh compression rearranges vertices and polygons into an order that allows more compact coding of the incidence between vertices and predictive compression of their positions. Previous schemes were designed for triangle meshes and polygonal faces were triangulated prior to compression. I show that polygon models can be encoded more compactly by avoiding the initial triangulation step. I describe two compression schemes that achieve better compression by encoding meshes directly in their *polygonal* representation. I demonstrate that the same holds true for volume meshes by extending one scheme to hexahedral meshes.

Nowadays scientists create polygonal meshes of incredible size. Ironically, compression schemes are not capable—at least not on common desktop PCs—to deal with giga-byte size meshes that need compression the most. I describe how to compress such meshes on a standard PC using an out-of-core approach. The compressed mesh allows streaming decompression with minimal memory requirements while providing seamless connectivity along the advancing decompression boundaries. I show that this type of mesh access allows the design of IO-efficient out-of-core mesh simplification algorithms.

In contrast, the mesh access provided by today's indexed formats complicates subsequent processing because of their IO-inefficiency in de-referencing (in resolving all polygon to vertex references). These mesh formats were designed years ago and do not take into account that a mesh may not fit into main memory. When operating on large data sets that mostly reside on disk, the data access must be consistent with its layout. I extract the essence of our compressed format to design a general *streaming format* that provides concurrent access to coherently ordered elements while documenting their coherence. This eliminates the problem of IO-inefficient de-referencing. Furthermore, it allows to re-design mesh processing tasks to work as *streaming*, possibly *pipelined*, modules on large meshes, such as on-the-fly compression of simplified mesh output.

ACKNOWLEDGMENTS

Five years have passed since I arrived in Chapel Hill and each one of them has been great. While I am super happy that my dissertation is now complete, I am also a little bit sad that my time here is coming to an end. Whenever asked why I had not yet defended my thesis, my supervisor would answer "Martin is just having too much fun." But he never mentioned that in fact he was to blame for that. After all, he was the one who made sure that those years were so much fun. So let me set the record straight.

As a visiting student at UBC in 1996 I enrolled in the "Computational Geometry" class that was taught by then associate professor Jack Snoeyink. He excited all of us about the material and the final course project with Marie-Claude was my first "real research experience" with polygon meshes. Shortly afterward, Jack took me, academically speaking, under his wing. First, he helped me to get into the Masters program at UBC, then became my thesis advisor, and eventually brought me along to UNC, where he would ultimately guide me to complete my doctorate.

Jack gave me the best guidance a student could hope for. He always had an open door when I needed advise but also let me do my thing when I was on a roll; yet he was immediately available when I ran into an unforeseen obstacle, wanted to share an exciting result, or simply needed reassurance that I was on the right track. He always encouraged me to pursue all opportunities, be it working for a summer in industry, spending time in other research labs, or attending academic meetings. He also knew to motivate me with the occasional conference trip to more "exotic" places.

Jack always kept a good balance between hard work and good fun. There was time for a game of foosball, an ultimate match, or an evening in the brew pub. His support extended well beyond academic affairs. He helped me move from Vancouver to Chapel Hill, sent someone to the airport to pick me up, registered and drove me to the "unavoidable" GRE test in Greensboro, and much more. He also sprang into action to save our SIGGRAPH 2000 presentation virtually in the last minute with a high-speed cab ride back the hotel. Yes, I was having fun. Thank you for all this, Jack!

There are two other people who directly contributed to the work presented here. Stefan Gumhold single-handedly implemented the Out-of-Core Mesh described in Chapter 7 and Peter Lindstrom wrote most of the streaming simplification algorithms described in Chapter 8 and the out-of-core mesh re-ordering tools from Chapter 9.

I also want to thank Craig Gotsman who invited me to work for half a year with Stefan Gumhold at the Technion in Haifa in 2000. This cooperation was quite synergetic and resulted in a fun paper. I enjoyed many espressos that "Chaim" prepared for me in his office with his high-tech coffee maker. I also thank Pierre Alliez and Olivier Deviller who welcomed me to their group at INRIA Sophia-Antipolis and helped me in a thousand ways to get settled there. They are ultimately to blame that I now speak some basic French. I also thank Peter Lindstrom for being so excited about streaming and for inviting me to California for six month of enjoyable joint-work. Peter also did an excellent job in reviewing this manuscript and finding many little mistakes.

Big thanks go to Anveesh, Miguel, and Adrian who lived for six month with my mattress in their hallway, to Michael North who stood with my name on a sign at RDU airport when I first arrived, to Zachi Karni who was a good friend during my time in Israel, to Ming for challenging me repeatedly with pointed questions during my defense, to Dinesh for not being too angry about me reading the "Daily Tar Heel" during his lectures, to Janet Jones for loaning me her antique furniture and for infinite patience despite my chronic slack with filing paperwork, to my co-workers at EAI, Intel, and Nvidia for making my work experiences in the "real-world" so enjoyable, to all the baristas around town that kept my research going with numerous free coffee refills, to my fellow students with whom I spent fun times in the department and in local coffee shops, bars, and parties around Chapel Hill and Carrboro, to Scott who has been a true friend and a great roommate in 20B Davie Circle for fun roadtrips and those "burned" week-old Driade pastries that kept me going under SIGGRAPH submission stress, to my childhood friends Marcus and Oliver for supplying me with hometown gossip, to all my fellow hikers on backpacking and hot spring trips in California and Oregon, to Henna who ironed the tie that made me look so distinguished during my defense talk, and to the five highly-educated scientists that it took to get it properly tied.

And finally, big thanks go to my mom, my dad, and my sister for their everlasting love and support from far away. Sometimes weeks went by without a note from me during the crazy rush to meet some conference deadline. But they would always be there for me to relieve some of that stress, to share joyous moments of accomplishment, and to come for the occasional visit no matter where I happened to be.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

```
#  triceratops.obj
#
#  2832 vertices
#  2834 polygons
#
v  3.661  0.002  -0.738
v  3.719  0.347  -0.833
v  3.977  0.311  -0.725
v  4.077  0.139  -0.654
    ⋮      ⋮      ⋮      ⋮
f  2806  2810  2815  2821
f  2797  2801  2811  2805
f  2789  2793  2802  2796
f  2783  2794  2788
    ⋮      ⋮      ⋮      ⋮
```

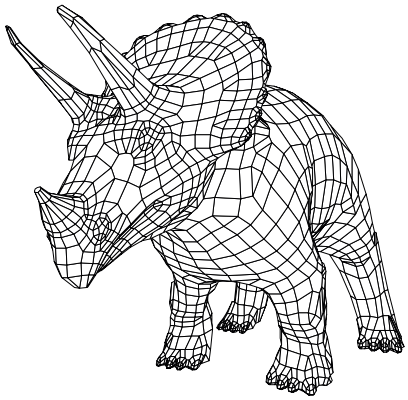


Figure 1.1: Indexed mesh formats are commonly used for storing and distributing polygon meshes. Shown is an excerpt of the OBJ file for the "triceratops" model.

Irregular polygon meshes are used to represent surfaces in many applications such as geographic information systems, virtual reality, computer-aided design, finite element analysis, and computer games. They provide a simple mechanism for describing three-dimensional objects and are easily derived from other surface representations. The large number of polygons that is required to accurately represent a smooth surface is a major drawback as bandwidth to the rendering pipeline is *the* limiting factor of most graphics applications. Nevertheless, polygon meshes remain the de-facto standard for interactive visualization of geometric models. One one hand, this has put manufacturers of graphic acceleration boards in fierce competition on the number of polygons per second that their hardware can render. On the other hand, it has motivated researchers to find suitably compact representations for polygon meshes.

The standard representation of a polygon mesh uses an array of floats to specify the vertex positions and an array of integers containing indices into the vertex array to specify the polygons. This is illustrated in Figure 1.1 at the example of the popular "triceratops" model. The array of positions is called the *geometry* and the array of

indexed polygons is called the *connectivity* of the mesh. Optional *properties* (e.g. surface normals, texture coordinates, ... that further refine the visual appearance of a model) and how they are attached to the mesh are specified in a similar manner.

A number of indexed mesh formats have emerged over the past decades, such as VRML, OBJ, PLY, or X3D just to name a few. They are simple to create and parse and map almost directly to the graphics APIs commonly used for rendering. These formats are basically identical and each graphics researcher has spent some quality time converting models from one format to another. Unfortunately indexed formats are not the most concise description for polygonal meshes. Large and detailed models can result in files of substantial size that are slow to transmit and expensive to store.

The bloat of an indexed mesh representation becomes more pronounced as the mesh becomes bigger because the storage costs for the indices increases super-linearly with the number of vertices. While the per-vertex costs for specifying additional vertex positions is constant, the per-vertex cost for referencing them increases logarithmically. The most concise indexed format stores $3b$ bits per vertex for the geometry, where b is a constant that stands for the number bits used to represent each coordinate, and $k \log_2(v)$ bits per vertex for the connectivity, where v is the number of vertices in the mesh and k is the average number of times each vertex is referenced, which is about 6 for triangle meshes and about 4 for polygon meshes such as the "triceratops". Hence, for large meshes the connectivity quickly dominates the overall storing costs.

The particular order in which polygons and vertices appear in their array makes no difference to the geometric shape that the polygon mesh describes. An indexed format imposes no constraints on the order in which the polygons are listed. Neither does it matter which of a polygon's vertices is listed first, as long as they are listed in a consistent, typically counterclockwise order around that polygon. Furthermore, the vertices of a polygon may be located anywhere in the vertex array. Subsequent polygons could potentially reference vertices at opposite ends of the array, whereas the first and the last polygon could reference the same vertex. That this happens in practice is apparent in Figure 1.1 where the first polygons of the "triceratops" reference some of the last vertices.

Indexed formats are so expensive *because* of this ability to arrange polygons and vertices pretty much at random. These formats accommodate an incredible number of different descriptions for one and the same polygon model. For a mesh with p polygons and v vertices there are $p!$ possible ways in which the polygons can be arranged and $v!$ possible ways to permute the vertex array. Each arrangement of the polygons

can be combined with any permutation of the vertices which already gives us $p! \cdot v!$ different descriptions for a particular polygon model. And that is before considering the flexibility that the d indices of each individual polygon can be listed in d different rotations. The large number of different descriptions that each single mesh has directly leads to the bloat of an indexed format, because it not only specifies the mesh, but also one of these many descriptions.

1.1 Compression

To reduce transmission times either between several networked computers or between the main memory and the graphics card, a number of *mesh compression* schemes have been proposed that reduce the amount of data needed to describe a particular polygonal model. The more compact this description, the smaller the delay when transmitting it across a network from one computer to another or when sending it across an internal bus to the graphics adapter. Mesh compression is a relatively young research area that was started by the pioneering work of (Deering, 1995).

Compression schemes completely ignore the original orderings of polygons and vertices. Using deterministic ordering rules they reduce the exponential number of different descriptions for a single mesh that an indexed format can accommodate to a linear number. By restricting the mesh elements to appear in a "somewhat" canonical order they can replace global indices with a small set of symbols that stores the local incidence relation between the mesh elements. Polygons are encoded in the order in which they are encountered during the encoding of the *connectivity graph* of the mesh and vertices are stored in the order in which they are first referenced by the re-ordered polygons. This effectively correlates both the ordering of the polygons among each other as well as the ordering of the vertices in respect to that of the polygons. Coding the connectivity graph of a mesh can usually be done with a constant number of bits per vertex (bpv) with bit-rates between 0.5 to 4.0 bpv being typical. In comparison, an indexed format uses between $4 \log_2(v)$ and $6 \log_2(v)$ bpv for a mesh with v vertices.

Previously the focus has been on compressing fully triangulated data sets (Taubin and Rossignac, 1998; Touma and Gotsman, 1998; Gumhold and Strasser, 1998; Li and Kuo, 1998; Rossignac, 1999; Bajaj et al., 1999), a natural candidate for the lowest common denominator. However, many polygonal meshes contain only a small percentage of triangles, like the "triceratops" model shown in Figure 1.1, for example. The "Premier Collection" from Viewpoint Datalabs—a well-known source of high quality 3D models—consists mostly of meshes with relatively low triangular face counts. Likewise,

few triangles are found in the output formats of many geometric modeling packages. The dominating element of these meshes is the quadrangle or quadrilateral, but pentagons, hexagons and higher degree faces are also common.

To encode polygonal meshes, previously reported mesh compression schemes triangulate all non-triangular faces *prior* to compression. Since there are many possible ways to triangulate a polygon this can lead to bloat in the representation. First, this approach does not encode the original polygonal connectivity but one of the many possible triangulations that was chosen to represent it. And second, in order to be able to recover the original polygons, additional information needs to be stored that marks all the edges that were added during the triangulation step for later removal. For maximal compression it is beneficial to keep a mesh in its native polygonal representation and delay the conversion to triangles until this becomes necessary.

(King et al., 1999) were first to report a compression scheme that the can represent the connectivity information of quadrangular meshes with fewer bits than that of their triangulated counterparts. Furthermore, knowledge about polygonal faces, which tend to be fairly planar and convex, can also be useful for more accurate geometry prediction. While a non-triangular face is usually not perfectly planar, major creases are improbable to occur across it—otherwise it would likely have been triangulated when the model was designed. The geometry predictor introduced by (Touma and Gotsman, 1998), for example, is based on the assumption that four neighboring vertices form a parallelogram. While the four vertices of two adjacent triangles can violate this assumption quite drastically, the four vertices of a convex quadrangle can not.

The first part of my thesis is

> *Polygon models can be compressed more compactly by avoiding the initial triangulation step and operating directly on polygonal connectivity.*

In support of my thesis I have designed and implemented novel mesh compression schemes, as well as improved and extended existing schemes. I present the main results of this work in Chapter3 3 through 6. I show that both the connectivity and the geometry of polygon meshes can be compressed more efficiently by operating directly on the polygonal representation of the mesh. I also describe techniques for efficient compression of mesh properties, which have been somewhat neglected in previous work on compression, but which are important for fast delivery of the property-rich 3D content of Web applications. Finally, I demonstrate how to generalize the most successful techniques for compressing polygonal surface meshes to hexahedral volume meshes.

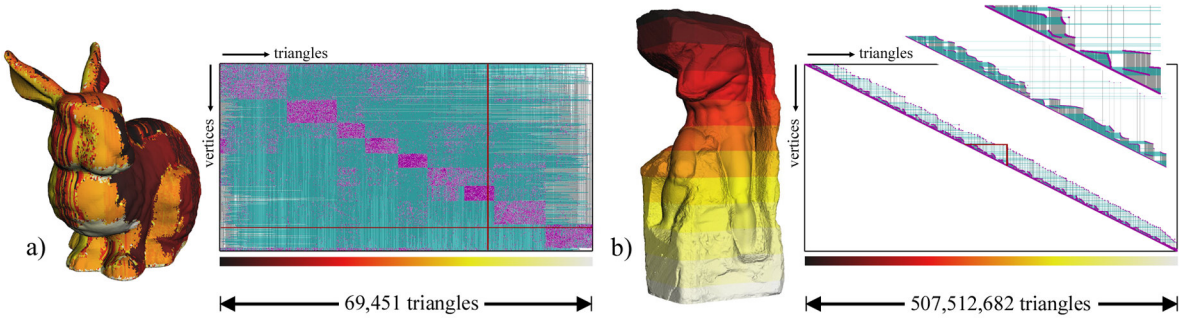


Figure 1.2: Illustration of incoherence in the element ordering of meshes stored in indexed formats: (a) The "Stanford bunny" and (b) the 10,000 times more complex "Atlas". Renderings color-code triangles based on their position in the array. Layout diagrams connect triangles that share the same vertex with horizontal line segments (green) and vertices referenced by the same triangle with vertical line segments (gray).

1.2 Streaming

Modern scanning technology has enabled scientists to create polygonal meshes of incredible size. Recent examples include statues scanned for historical reconstruction, isosurfaces displayed to understand scientific simulations, and terrain measured to predict flood impact. The large "Atlas" statue shown in Figure 1.2 from Stanford's Digital Michelangelo Project (Levoy et al., 2000), for example, has over 250 million vertices and more than 500 million triangles. Ironically, current mesh compression schemes are not capable—at least not on common desktop PCs—of dealing with meshes of the gigabyte size that would benefit from compression the most. Currently, mesh compression algorithms can be used only when connectivity and geometry of the mesh are small enough to reside in main memory.

Storing polygon models that contain millions of vertices in a standard indexed format not only has the disadvantage of requiring large amounts of disk space. The more serious problem is that this representation does not scale well with increasing model size. Today's mesh formats were designed years ago, when meshes were orders of magnitude smaller. They implicitly assume that it is possible to completely load the mesh into the main memory. The most fundamental operation for any indexed mesh format to support is *dereferencing* of the input mesh (i.e. resolving all triangle to vertex references). To efficiently do this for large data sets that mostly reside on the hard disk, the mesh format must take into account the memory hierarchy of a computer. The order in which the data is accessed must be consistent with its layout on disk in order to avoid frequent reloading of mesh data from slow external memory.

The flexibility of being able to randomly order vertices and triangles in their respective array was convenient for working with smaller models such as the 70 thousand triangle "Stanford bunny". This polygon model has helped popularize the PLY format and abuses the flexibility in laying out the mesh elements like no other data set. Its *mesh layout* is highly incoherent, which we demonstrate visually with the help of two illustrations in Figure 1.2. Storing the 500 million triangle "Atlas" statue in the same format means that a six gigabyte array of triangles references into a three gigabyte array of vertex positions. The visualizations reveal that some vertices are referenced over spans of up to 550,000 triangles—equaling 700 MB of the triangle array. Because common desktop PCs can not operate on nine gigabyte of indexed mesh data, the "Atlas" statue is provided in twelve pieces in Stanford's Webarchive.

One could argue that simply memory mapping the indexed mesh and having the virtual memory functionality of the operating system swap in the relevant sections of the vertex and triangle arrays would be sufficient. Such an approach is feasible if the layout of the mesh is sufficiently coherent. Obviously, current mesh formats do not *prevent* us from storing meshes in a coherent fashion. But they also do not reward us for doing so. The problem is not only the potential lack of coherence but also its unforeseeability. To operate robustly on large indexed meshes, an algorithm must either be prepared to handle the worst possible input or make assumptions about their coherence that are bound to fail on some inputs.

The second part of my thesis is

> *Ordering the elements of polygonal meshes in an interleaved and coherent manner while also documenting their coherence in the file format enables the design of IO-efficient processing modules that operate on the data in a streaming, possibly pipelined, fashion.*

In support of my thesis I have designed and implemented out-of-core techniques that can convert large indexed meshes into a streaming representation schemes using only limited memory resources. I have adapted several mesh processing tasks to consume, operate on, and produce meshes in a streaming manner and have shown that this improves run times, memory efficiency, and sometimes even processing quality (for mesh simplification). The results of this work are presented in Chapters 7 through 9.

I describe how to compress gigantic meshes in one piece on a standard PC using an out-of-core approach. The resulting compressed format allows streaming decompression with a small memory foot-print at speeds that are CPU- and not IO-limited. The

particular type of streaming access to a mesh that is provided by our decompressor makes simple operations on large meshes surprisingly easy. This promises potential for using this type of out-of-core mesh access to also perform other, more complex mesh processing tasks in a more efficient manner. I demonstrate that this is indeed the case by adapting two simplification algorithms to take advantage of the IO-efficient out-of-core mesh access that our compressed format provides.

The ease with which meshes in our compressed format can be processed suggests that a streaming representation is better suited for storing large models than current indexed mesh formats. I extract the essence of what makes our compressed format useful to design a new, more general *streaming mesh* format. While conceptually simple, this format eliminates once and for all the problem of de-referencing indexed meshes. Furthermore it allows to re-design certain mesh processing algorithms to work as *streaming*, possibly *pipelined*, modules on large meshes. I describe measures for different stream qualities and how they impact processing. I present several re-ordering algorithms and evaluate the stream quality of the meshes they produce. I also describe a scheme that can compress streaming meshes on-the-fly in their particular stream order.

The availability of a streaming format enables us to consider a new breed of algorithms that incorporate a streaming paradigm from ground up into their design, where all processing happens in a small in-core buffer into which original vertices and triangles stream from disk and out of which already processed elements stream back to disk. Such algorithms have to adapt their computations to respect this new type of mesh access. Doing so promises significant improvements in mesh processing speed and scalability to meshes of arbitrarily large size. Future work will address which part of the mesh processing pipeline can operate in a streaming fashion and which extensions or specializations are needed to make it useful for other tasks.

A streaming mesh format is a better *input format* than a standard indexed format for all large mesh processing tasks. But that does certainly not mean that all mesh processing tasks can be implemented in a streaming manner. Stream-based processing is mainly useful for time-consuming, off-line algorithms that operate on the entire data set, such as simplification, remeshing, smoothing, or compression. Streaming is not applicable for applications that support real-time interaction with large models or require selective access to parts of the model. Examples for this include out-of-core mesh editing, view-dependent level-of-detail rendering, interactive exploration of complex structures or large terrains, visibility computations, and collision detection. Such applications perform *online* as supposed to *streaming* processing and therefore need a

mesh representation that supports online rather than streaming mesh access. But at preprocessing time it is still beneficial when the original data arrives in a streaming format. These applications need to build data structures that support online access. This typically involves partitioning the input mesh into a large number of smaller pieces of which only a small number is later kept in memory with the majority residing on disk. Building these initial on-disk representations can be done much more efficiently when the input mesh is in a streaming format.

1.3 Overview

My contributions are collected in the following chapters. Each chapter ends with a section that contains hindsights of what I would have done differently or what needs further investigation. Over the last four years most of these works were made public in form of conference and journal publications.

Chapter 2 gives a detailed overview of previous research in the area of mesh compression and the graph theoretical roots of connectivity coding. I show the intimate connections between the classic work on planar graph coding by Turan, recent schemes like Edgebreaker, and the optimal coding method of Poulalhon and Schaefer by fitting them into a common framework. This allows me to uncover simple improvements and surprising similarities that were previously undetected. Furthermore, I form parallel classifications of the main schemes into face-based, edge-based, and vertex-based and into one-pass and multi-pass coders. The latter divides the existing body of compression schemes into those that can be used out-of-core and those that can not.

Chapter 3 describes our edge-based 'Face Fixer' scheme, which is joint work with my supervisor Jack Snoeyink that was published at the SIGGRAPH conference (Isenburg and Snoeyink, 2000). It was the first scheme to encode mesh connectivity directly in its polygonal representation and to improve compression rates for storing arbitrary polygonal connectivity by avoiding the initial triangulation step. We also describe a variation of this edge-based encoding scheme that allows to compress the connectivity of triangle meshes together with information about pre-computed triangle strips. 'Triangle Strip Compression' was first published as a single-authored paper at the Graphics Interface conference (Isenburg, 2000) and appeared later as a longer journal version in Computer Graphics Forum (Isenburg, 2001).

Chapter 4 shows that the degree-based coder for triangle mesh connectivity by (Touma and Gotsman, 1998), which mainly stores a sequence vertex degrees, can also be generalized to directly code polygonal connectivity by storing a separate sequence of face degrees. This approach gives significantly better bit-rates than 'Face Fixer'. First, it can adapt to regularity in either degree sequence and second, it can exploit the duality between these two degree sequences for mutual predictive coding. The 'Degree Duality Coder' was published as a single-authored paper at the Graphics Interface conference (Isenburg, 2002). We also prove the necessity of split operations and disprove the suspected redundancy of split-offsets for degree-based coding.

Chapter 5 looks into various other aspects of mesh compression. I describe a simple trick for improving predictive compression of vertex positions of polygonal meshes, which is joint work with Pierre Alliez that was published at the Visualization conference (Isenburg and Alliez, 2002b). In this chapter I also explain a novel predictive technique for compressing per-corner mappings of mesh properties such as normals or texture coordinates, which grew out of my work on a CAD mesh compression engine at EAI Inc. and was published as a joint paper with Jack Snoeyink at the Pacific Graphics conference (Isenburg and Snoeyink, 2001a) and also appeared as a longer journal paper in Graphical Models (Isenburg and Snoeyink, 2002). Finally, I show how predictive compression of texture coordinates needs to be modified to avoid unreasonable predictions in the presence of mapping discontinuities, which is also joint work with my supervisor and was presented at Computer Graphics International (Isenburg and Snoeyink, 2003).

Chapter 6 takes compression into the next dimension using hexahedral volume meshes as the example. In particular, I show here that the concept of coding with degrees can be extended to volume meshes by using edge degrees instead of vertex degrees. This is joint work with Pierre Alliez and was done during my six month stay at INRIA Sophia-Antipolis in France. It was initially published at the Pacific Graphics conference (Isenburg and Alliez, 2002a) and appeared later in revised form as a journal publication in Graphical Models (Isenburg and Alliez, 2003).

Chapter 7 is the first of three chapters where I start looking at larger data sets of up to 500 million triangles. Such gigabyte sized data sets can no longer be processed with standard approaches to mesh compression, at least not on a standard PC. I describe how to compress gigantic polygon meshes using a dedicated external memory structure

and a compression scheme that queries that structure as coherently and as infrequently as possible. This out-of-core compression scheme is joint work with Stefan Gumhold that was published at the SIGGRAPH conference (Isenburg and Gumhold, 2003). The results of this work gave me the initial insights on the benefits of a streaming mesh representation, which is manifested in the following two chapters.

Chapter 8 shows that the order in which the decompressor decodes vertices and triangles from the compressed representation described in the previous chapter is useful for efficient out-of-core mesh processing. While at any time only a small portion of the mesh needs to be kept in main memory, seamless connectivity can be maintained between the active mesh elements of the traversal. For algorithms that can adapt their computations to a fixed element ordering such *processing sequences* provide highly IO-efficient out-of-core access to large meshes. The two abstractions that are naturally supported by this representation are *boundary-based* and *buffer-based* processing. We illustrate both abstractions by adapting two simplification methods to perform their computation in processing sequence order. Both algorithms benefit in terms of improved quality, more efficient execution, and smaller memory footprints. This is joint work with Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink that was published at the Visualization conference (Isenburg et al., 2003).

Chapter 9 extracts the essence of what made our compressed format so useful to design a new, more general *streaming mesh* format. When we originally designed our compressed format we were aiming for maximal compression and efficient decompression, but without considering the potential needs of streaming processing. Here I describe desirable qualities for a stream ordering of mesh elements and present metrics and diagrams that characterize them. This shows that the traversal heuristics of our compressor in fact systematically creates poor orderings that are not suited for all kinds of streaming processing. Furthermore, I present different methods for converting meshes from a traditional to a streaming format and a novel technique for streaming on-the-fly compression. This work was done at Lawrence Livermore National Laboratory together with Peter Lindstrom and was the basis for a fully funded joint NSF proposal with UC Berkeley where I will further work on this topic.

Chapter 10 summarizes my contributions, discusses the limitations of streaming processing, and describes what should definitely be pursued in future work and what may be worthwile to investigate.

Chapter 2

Mesh Compression

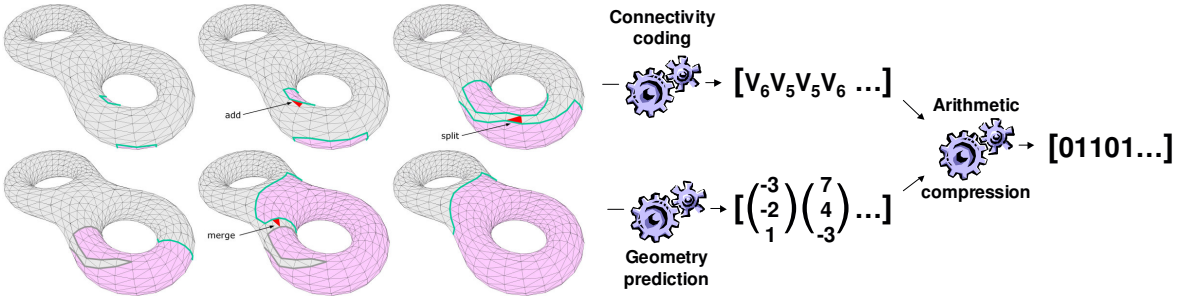


Figure 2.1: Workflow of a typical mesh compressor: Connectivity is coded as sequence of symbols and geometry is coded as a sequence of corrective vectors. Both sequences are compressed into a compact bit-stream using, for example, an arithmetic coder.

A polygon mesh is the most widely used primitive for representing three-dimensional geometric models. Such polygon meshes consists of mesh *geometry* and mesh *connectivity*, the first describing the positions in 3D space and the latter describing how to connect these positions together to form polygons that describe a surface. Typically there are also mesh *properties* such as texture coordinates, vertex normals, or material attributes that describe the visual appearance of the mesh at rendering time.

The standard representation of a polygon mesh uses an array of floats to specify the positions and an array of integers containing indices into the position array to specify the polygons. A similar scheme is used to specify the various properties and how they are attached to the mesh. For large and detailed models this representation results in files of substantial size, which makes their storage expensive and their transmission slow. The need for more compact representations has motivated researchers to develop efficient mesh compression techniques. Their research has aimed for three different objectives: efficient rendering, progressive transmission, and maximum compression.

Efficient rendering: Encodings for efficient rendering try to reduce the amount of data that needs to be sent to the graphics card. When each triangle of the mesh is

rendered individually the card must process every mesh vertex an average of six times. Processing a vertex involves passing its data from the main memory to and through the graphics pipeline. Besides the vertex coordinates this typically also includes normal, color, or texture information. The most common technique to reduce the number of times this data needs to be transmitted is to send long runs of adjacent triangles. Such *triangle strips* (Woo et al., 1996) are widely supported by today's graphics hardware and can reduce the number of vertex repetitions by a factor of three.

In his pioneering work, (Deering, 1995) introduces techniques that further reduce the amount of data that needs to be sent to the graphics card. These techniques not only eliminate a much larger number of vertex repetitions but also reduce the amount of data needed for representing each single vertex. His *generalized triangle mesh* format is designed for a highly specialized graphics adapter that can cache sixteen previously processed vertices and includes explicit information for managing this cache. In addition, his mesh format specifies quantizations and encodings for coordinates, normals, and colors that allow decompression to be implemented directly in hardware.

Progressive transmission: Encodings for progressive transmission use incremental refinements of mesh connectivity and geometry so that partial data already represents the entire mesh at a lower resolution. The 'Progressive Mesh' scheme by (Hoppe, 1996) encodes a mesh by collapsing edges one by one. Decoding starts with a small base mesh and expands the collapsed edges in reverse order. While this progressive scheme was not designed for compression and used a large number of bits per vertex, more recent schemes (Taubin et al., 1998a; Pajarola and Rossignac, 2000; Cohen-Or et al., 1999; Alliez and Desbrun, 2001a) group the refinement operations into large batches and achieve bit-rates that come close to those of non-progressive encodings. Even though more bits are used for the connectivity information, the progressive nature of the decoding allows more accurate geometry and property prediction.

For the special case of terrains models based on Delaunay triangulations, (Snoeyink and van Kreveld, 1997) use ideas from the point location scheme of (Kirkpatrick, 1983) to encode all topology information in a permutation of the vertices, from which the mesh is progressively reconstructed. The work of (Denny and Sohler, 1997) extends this scheme to arbitrary planar triangulations. Although the cost of storing the topology is zero, the unstructured order in which the vertices are received and the absence of adjacency information during their decompression prohibits predictive geometry encoding. This makes these schemes overall more expensive. Moreover, it is not clear whether it is possible to extend this idea to general surface meshes.

Maximum compression: Encodings for maximum compression try to squeeze a given mesh into as few bits as possible for faster network transmission and more compact storage. Most efforts have focused on connectivity compression (Taubin and Rossignac, 1998; Touma and Gotsman, 1998; Li and Kuo, 1998; Gumhold and Strasser, 1998; Rossignac, 1999; Bajaj et al., 1999; Kronrod and Gotsman, 2000). There are two reasons for this: First, this is where the largest gains are possible, and second, the connectivity coder is the core component of a compression engine and usually drives the compression of geometry (Deering, 1995; Taubin and Rossignac, 1998; Touma and Gotsman, 1998), of properties (Taubin et al., 1998b; Bajaj et al., 1999), and of how properties are attached to the mesh (Taubin et al., 1998b). The connectivity is usually compressed through a compact and often interwoven representation of two dual spanning trees: one tree spans the vertices, and its dual spans the polygons. The pair of spanning trees is obtained by traversing the elements of the mesh with some deterministic strategy. The geometry and the property data are typically quantized before they are compressed with a prediction based on local neighborhood information.

Finally, I should mention *shape compression* techniques that try to achieve these three objectives not for a given mesh, but rather for the shape that it represents. Here the input mesh is considered to be merely one particular instance of a geometric shape, which may be changed as long as the shape is still represented faithfully. Such methods *re-mesh* the input to construct a highly regular mesh that can be more compactly stored and transmitted (Khodakovsky et al., 2000; Szymczak et al., 2002; Khodakovsky and Guskov, 2004) or even more efficiently rendered (Gu et al., 2002).

The remainder of this chapter (and also the two following chapters) focuses on maximum compression of mesh connectivity. Compression of mesh geometry, of mesh properties, and of how properties attach to the mesh is covered in Chapter 5. After reviewing some definitions in Section 2.1 and arithmetic coding in Section 2.2, I survey the state of the art in non-progressive connectivity coding. In Section 2.3 I describe the overall approach to connectivity coding. In Section 2.4 I go back to its graph theoretical roots and give a visual framework that I use to illustrate the intimate connections between the classic work on planar graph coding by (Turan, 1984) and more recent schemes. In Section 2.5 I describe the main compression schemes for triangular connectivity and organize them into two parallel categories. In Section 2.6 I explain what the scheme by (Poulalhon and Schaeffer, 2003) does differently to achieve optimality in coding. In Section 2.7 I discuss how these schemes extend to polygonal connectivity with my contributions being detailed further in Chapters 3 and 4.

2.1 Preliminaries

A *triangle mesh* or a *polygon mesh* is a collection of triangular or polygonal *faces* that intersect only along shared edges and vertices. Around each face we find a cycle of vertices and edges that we consider to have a counterclockwise orientation when seen from the outside. Each appearance of a vertex in this cycle is called a *corner*. Each appearance of an edge in this cycle is called an *half-edge*. It is oriented in the direction of the cycle and therefore has an *origin* and a *target* vertex. An edge is *manifold* if it is shared by two faces of opposite orientation or used only by one face, in which case it is also a *border edge*. An edge shared by more than two faces is *non-manifold* and an edge shared by two faces of identical orientation is *not-oriented*. A vertex is *manifold* if all of its incident edges are manifold and connected across faces into a single *ring*. A polygonal mesh is manifold if all of its edges and vertices are manifold.

Topologically, a manifold mesh is a graph embedded in a 2-manifold surface in which each point has a neighborhood that is homeomorphic to a disk or a half-disk. Points with half-disk neighborhoods are on the boundary. A mesh has genus g if one can remove up to g closed loops without disconnecting the underlying surface; such a surface is topologically equivalent to a sphere with g handles. A mesh is *simple* if it has no handles and no border edges. Euler's relation says that a graph embedded on a sphere having f faces, e edges, and v vertices satisfies $f - e + v = 2$. When all faces have at least three sides, we know that $f \leq 2v - 4$ and $e \leq 3v - 6$, with equality if an only if all faces are triangles. For a mesh with g handles (genus g) the relation becomes $f - e + v = 2 - 2g$ and the bounds on faces and edges increase correspondingly.

2.2 Arithmetic Coding

A mesh compressor typically encodes the connectivity of the mesh as a sequence of symbols and the geometry of the mesh as a sequence of small corrective vectors. Rather than uniformly mapping symbols and vectors to bit codes, one applies some sort of entropy coding so that they can be stored using the least number of bits that information theory allows. The entropy for a sequence of n symbols of t different types is

$$-n \sum_{i=1}^{t} (p_i \log_2(p_i)), \tag{2.1}$$

where p_i denotes the probability for a symbol of type i to occur. This simply corresponds to the number of times this symbol occurs divided by the total number of symbols n.

An arithmetic coder can compress a symbol sequence in an information theoretically optimal way. Given a sufficiently long input, the compression rate of arithmetic coding converges to the entropy of the input. Often the exact symbol probabilities are not known in advance, in which case an an adaptive version of arithmetic coding is used that learns the probabilities during compression. It starts with uniform probabilities for all symbols that are updated each time after a symbol was compressed.

Correlation among subsequent symbols can be exploited using a memory-sensitive coding scheme that approximates the order-k entropy of the sequence. An order-k arithmetic coder uses a different probability table for each combination of the preceding k symbols. For a sequence of n symbols of t different types the order-k entropy is

$$-n \sum_{i=1}^{t} \sum_{\alpha} (p_{i|\alpha} \log_2(p_{i|\alpha})), \tag{2.2}$$

where α denotes a particular combination of k symbols and $p_{i|\alpha}$ the probability for a symbol of type i to occur after the symbol combination α.

Often the likelihood of a particular symbol to occur is not only correlated to the preceding symbols but also to some state information that is available to both the encoder and the decoder. This can be exploited using a more general context-based arithmetic coder that switches probability tables based on a context that is provided together with each symbol. The context-based entropy for a sequence of n symbols of t different types where each symbol is accompanied by one of c contexts is

$$-n \sum_{i=1}^{t} \sum_{j=1}^{c} (p_{i|j} \log_2(p_{i|j})), \tag{2.3}$$

where $p_{i|j}$ is the probability for a symbol of type i to occur with a context of type j.

2.3 Connectivity Compression

The standard approach that represents the connectivity of a mesh with a list of vertex indices requires at least $kn \log_2 n$ bits, where n is the total number of vertices and k is the average number of times each vertex is indexed. For a simple triangular mesh, Euler's relation tells us that k will be about 6. For typical non-triangular meshes that contain a mix of polygons dominated by quadrangles, k tends to be about 4. One disadvantage of this representation is that it does not directly store face adjacency, which must be recovered by sorting around vertices. But the main problem is that the

space requirements increase super-linearly with the number of vertices, since $\log_2 n$ bits are needed to index a vertex in an array of n vertices.

Efficiently encoding mesh connectivity has been subject of intense research and many techniques have been proposed. Initially most of these schemes were designed for fully triangulated meshes (Taubin and Rossignac, 1998; Touma and Gotsman, 1998; Gumhold and Strasser, 1998; Rossignac, 1999), but more recent approaches (Isenburg and Snoeyink, 2000; Kronrod and Gotsman, 2000; Khodakovsky et al., 2002; Isenburg, 2002) handle arbitrary polygonal input. These schemes do not attempt to code the vertex indices directly—instead they only code the *connectivity graph* of the mesh.

For a manifold polygon mesh, every face (i.e. every edge loop) in the connectivity graph corresponds either to a polygon or a hole. For representing mesh connectivity, it is sufficient to specify (a) the connectivity graph of the mesh and (b) which of its faces are polygons/holes. The mapping from graph vertices to the 3D positions that are associated with each mesh vertex can be established using an order derived from the graph connectivity. Hence, mesh connectivity is compressed by coding the connectivity graph (plus some additional information to distinguish polygons from holes) *and* by changing the order in which the vertex positions are stored. They are arranged in the order in which their corresponding vertex is encountered during some deterministic traversal of the connectivity graph. Since encoding and decoding of the connectivity graph also requires a traversal, the positions are often reordered as dictated by this encoding/decoding process.

This reduces the number of bits needed for storing mesh connectivity to whatever is required to code the connectivity graph. This is good news: the connectivity graph of a polygon mesh with sphere topology is homeomorphic to a planar graph. It is well known that such graphs can be coded with a constant number of bits per vertex (Turan, 1984) and exact enumerations exist (Tutte, 1962; Tutte, 1963). If a polygon mesh has handles (i.e. has non-zero genus) its connectivity graph is not planar. Coding such a graph adds per handle a number of bits that is logarithmic in the size of the input mesh (Rossignac, 1999), but most meshes have only a small number of handles.

Encoding the connectivity of non-manifold polygon meshes requires additional attention. The bounds on the bit-rate for planar graphs no longer apply and we can not expect to do to this with similar efficiency. Coding non-manifold connectivity directly is tricky and there are no elegant solutions yet. Most schemes either require the input mesh to be manifold or use a preprocessing step that cuts non-manifold meshes into manifold pieces (Guéziec et al., 1998). Cutting a non-manifold mesh replicates

all vertices that sit along a cut. Since it is generally not acceptable to modify a mesh during compression, the coder also needs to describe how to stitch the mesh pieces back together. (Guéziec et al., 1999) report two approaches for doing this in an efficient manner. In Section 4.3 we describe a simpler scheme that is easily implemented but may not achieve the same compression. While it was originally only intended for smaller meshes with few non-manifold vertices we show in Section 7.4 that an improved version of this scheme is sufficient to deal with large and highly non-manifold models.

2.4 Coding Planar Graphs

The early enumeration results by (Tutte, 1962; Tutte, 1963) imply that an unlabeled planar graph can be represented with a constant number of 3.24 bits per vertex. Similarly, (Itai and Rodeh, 1982) prove that a triangular graph with v vertices may be represented by $4v$ bits. However, these existence proofs do not provide us with an effective procedure to construct such a representation. (Turan, 1984) is the first to report an efficient algorithm for encoding planar graphs using a constant number of bits.

A planar graph with v vertices, f faces, and e edges can be partitioned into two dual spanning trees. One tree spans the vertices and has $v - 1$ edges, while its dual spans the faces and has $f - 1$ edges. Summing these edge counts results in Euler's relation $e = (v - 1) + (f - 1)$ for planar graphs. Turan observed that the partition into dual spanning trees can be used to encode planar graphs and reported an encoding that uses 4 bits per edge (bpe) for general planar graphs. Applying his method to fully triangulated graphs results in an encoding that uses 12 bits per vertex (bpv). This bit-rate, which is often quoted in literature, is unnecessarily inflated. We can improve the Turan's bit-rate to 6 bpv simply by using of the fact that every face is a triangle.

The encoding method of Turan walks around a vertex spanning tree and records four different symbols as illustrated in Figure 2.2. Two symbols "+" and "–" describe walking down and up the edges of the vertex spanning tree. Two symbols "(" and ")" describe walking across edges that are not part of the vertex spanning tree for the first and for the second time. This information encodes both spanning trees and is sufficient to reconstruct the original graph. There are $v - 1$ symbols each of type "+" and "–", one pair for each edge of the vertex spanning tree. There are $e - v + 1$ symbols each of type "(" and ")", one pair for each edge not part of the vertex spanning tree. Coding each symbol with 2 bits leads to an encoding for planar graphs that uses $4v - 4 + 4e - 4v + 4 = 4e$ bits.

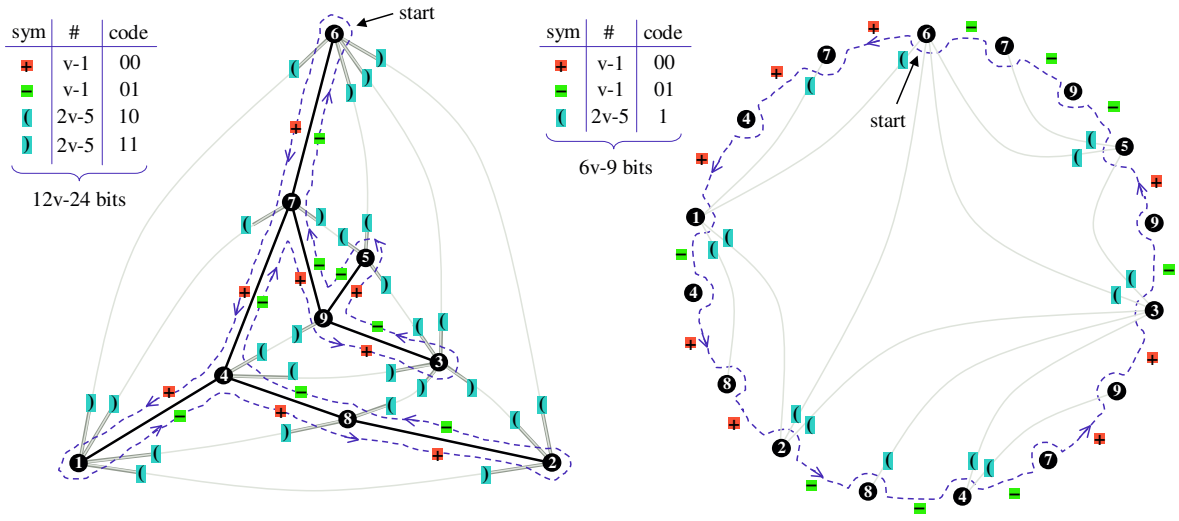


Figure 2.2: To code planar graphs Turan performs a walk around a vertex spanning tree during which he records four different symbols, "+", "–", "(", and ")". It's not hard to see that for fully triangulated graphs we may omit either all opening or all closing brackets. The respective other brackets can be derived from the fact that all faces are triangular. Turan's illustrates his method by "pulling open" the graph along the path a round the vertex spanning tree. We use this as a visual framework to illustrate how recent connectivity coders manage to encode the same information with fewer bits.

The straight-forward application of Turan's method to fully triangulated graphs where $e = 3v - 6$ results in an encoding that uses $12v - 24$ bits. However, when every face is a triangle, we only need to include either all "(" symbols or all ")" symbols in the code. The respective other can be omitted as it can be derived with simple book-keeping during decoding. This observation leads to a much tighter bound of $6v - 9$ bits by encoding the $v - 1$ occurrences of "+" and "–" with two bits and the $2v - 5$ occurrences of either "(" or ")" with one bit.

Given the choice about which of the two brackets to omit we suggest to keep the closing brackets because it leads to a more elegant implementation of the decoder. Using opening brackets requires the decoder to check whether a triangle is to be formed after each non-bracket symbol and after each formed triangle. Using closing brackets, we always form a triangle when the decoder reaches a bracket symbol by connecting back to the vertex that is reached when taking two backwards steps along the edges.

Improving on Turan's work, (Keeler and Westbrook, 1995) report a 3.58 bpe encoding for general planar graphs, which they specialize to a 4.6 bpv encoding if the graph is triangulated. Again, a small oversight on the authors' part results in the latter bit-rate being unnecessarily inflated. We can improve their bit-rate to 4 bpv simply by chang-

ing their mapping from symbols to bit codes. This oversight has also helped obscure the intimate similarities between Keeler and Westbrook's method and the Edgebreaker scheme by (Rossignac, 1999). For the case of encoding planar triangulations these schemes perform exactly the same traversal and distinguish exactly the same five cases. The only difference between them is how they map each case to a bit code.

The encoding method of Keeler and Westbrook and its equivalence to the Edgebreaker method are illustrated in Figure 2.3. Keeler and Westbrook construct a triangle spanning tree using a topological depth-first sort in the dual graph. It is the determinism with which this triangle spanning tree and the corresponding vertex spanning tree are created that allows them to improve on the bit-rates of Turan, which assumes no particular vertex spanning tree. Each dual edge that is not part of the triangle spanning tree crosses a primal edge that is part of the vertex spanning tree. Keeler and Westbrook's dual edges connect two nodes of the triangle spanning tree such that one of these nodes is an ancestor of the other. They declare the ancestor node to have a *missing child* where this dual edge connects (illustrated by a red square in the figure) and attach a leaf node to where this edge connects at the other end (illustrated by a green dot). The resulting tree has only five different types of non-leaf nodes and is encoded through a pre-order traversal that maps them to different bit-codes. The type of a non-leaf depends on its parenthood. A non-leaf node can either have:

1. two non-leaf children (**S**).
2. a non-leaf left child and a missing right child (**C**).
3. a non-leaf left child and a leaf right child (**L**).
4. a leaf left child and a non-leaf right child (**R**).
5. two leaf children (**E**).

Each of these node types corresponds to the indicated label of the Edgebreaker encoding. Keeler and Westbrook observe that half of all nodes will have leaves, meaning are of type L, R, and E. They devise a mapping from node types to bit codes that represents C as 00, S as 01 and either of L, R, or E with 1. To that encoding they append a bitstring that distinguishes between L, R, and E using $\log_2(3)$ bits each. The total results in the reported bitrate of 4.6 bpv. Would the authors have instead noticed that half of all nodes have a missing child, meaning are of type C, they could have just as easily proposed the more efficient 4 bpv encoding that was not discovered until Rossignac formulated a more elegant version of this algorithm, terming it Edgebreaker.

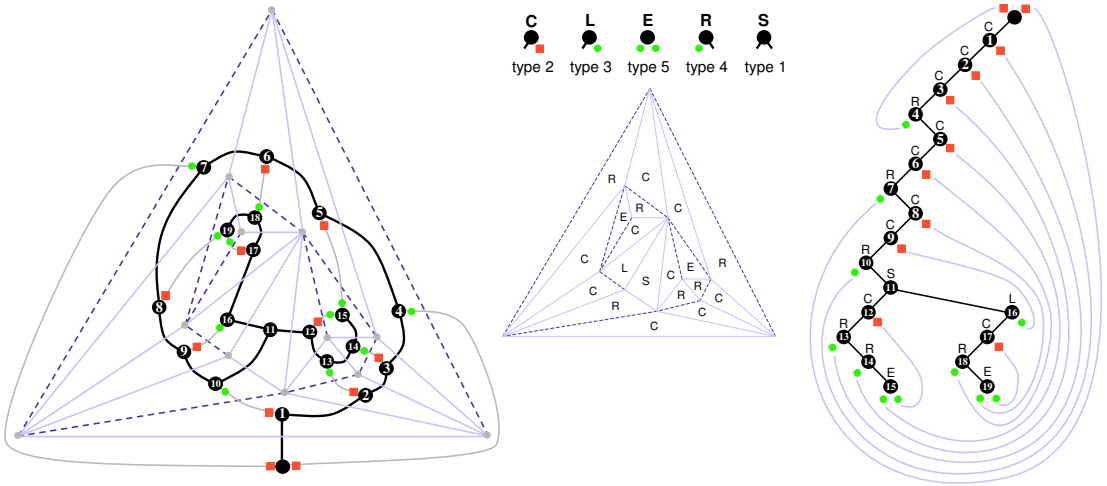


Figure 2.3: To code a triangulated planar graph Keeler and Westbrook construct a triangle spanning tree using a depth-first traversal and classify its nodes as types 1 to 5, which are identical to the five cases C, L, E, R, and S of the Edgebreaker scheme.

2.5 Coding Triangle Mesh Connectivity

These graph coding techniques were introduced to the graphics community for compressing triangle meshes by (Taubin and Rossignac, 1998). Like Turan they encode triangular connectivity using a pair of spanning trees, but unlike Turan they code the two trees separately. When using run-length coding, this results in bit-rates of around 4 bpv in practice but leads to no guaranteed bounds. (Rossignac, 1998) later pointed out that using a standard 2 bit per node encoding for each of the two trees also guarantees a 6 bpv bound. Most importantly, the work of (Taubin and Rossignac, 1998) showed how to integrate *topological surgery* into the encoding process for dealing with non-planar connectivity graphs. They do this by identifying pairs of triangles in the triangle spanning tree that are glued together to recreate a handle, which can always be done with $O(\log_2(v))$ bits per handle. Since most meshes have only a small number of handles, no efforts have been directed at establishing tighter bounds.

More recent encoding schemes (Touma and Gotsman, 1998; Gumhold and Strasser, 1998; Li and Kuo, 1998; Rossignac, 1999; Isenburg and Snoeyink, 2000; Alliez and Desbrun, 2001b) do not explicitly construct the two spanning trees. Instead they traverse the connectivity graph using a *region growing* approach during which they produce a symbol stream that implicitly encodes both trees in an interleaved fashion. The schemes iteratively encode faces or edges that are adjacent to the already *processed region* and produce a stream of symbols that describes the adjacency relation between

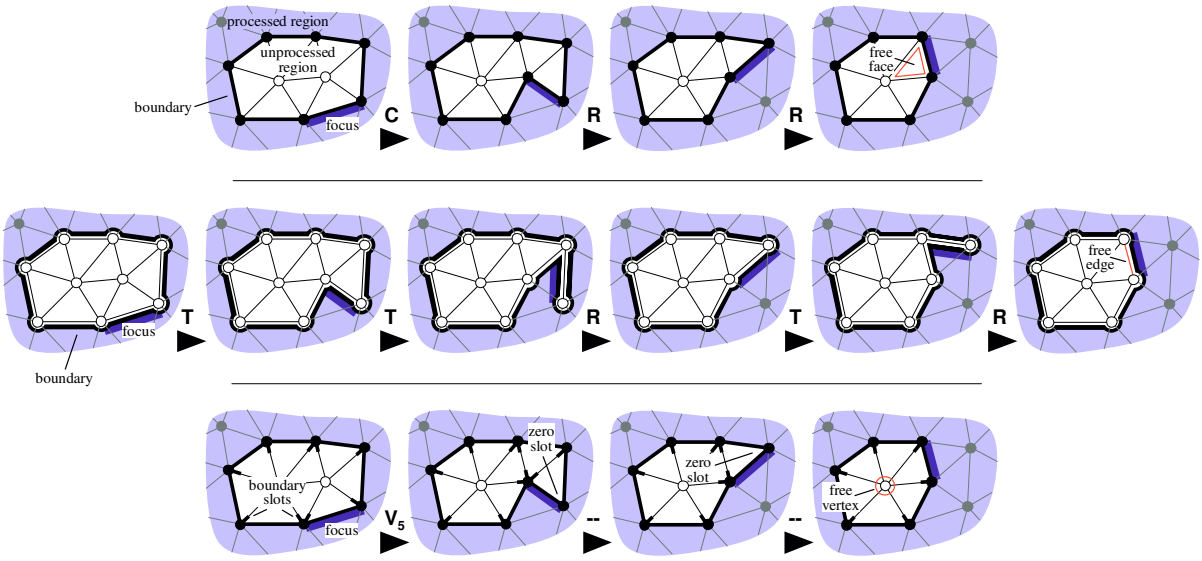


Figure 2.4: Three different approaches to connectivity coding by region growing: face-based (top), edge-based (middle), and vertex-based (bottom). Every iteration of the face-based coder processes the *free face* and describes its adjacency relation to the active boundary. Similarly every iteration of the edge-based coder processes the *free edge* also describing its adjacency relation. These descriptions specify how processing these elements changes the active boundary. The vertex-based coder only needs to describe how the boundary changes when it processes a face that has a *free vertex*.

the processed element and everything processed previously. For this they maintain one or more boundary loops that separate a single processed region of the mesh from the rest. The edges and vertices of these boundaries are called *boundary edges* and *boundary vertices*. Each boundary encloses an *unprocessed region* and in case the connectivity graph has handles, boundaries can also be nested, in which case an unprocessed region is enclosed by more than one boundary. Each boundary has a distinguished boundary element that is called the *focus* or the *gate*. The schemes work on the focus of the *active boundary*, while all other boundaries are kept in a stack. At each step they describe the adjacency between a currently processed element at the focus and the active boundary. For the case of non-zero genus meshes there will be one situation per handle in which this involves a boundary from the stack.

One difference between these schemes is how the boundaries are defined and how processing of a face or edge updates them. Depending on the mesh element that the description of the boundary update is associated with, the schemes can be classified as *face-based*, *edge-based*, and *vertex-based*, which is illustrated in Figure 2.4. Another difference between these schemes is whether they use explicit "split offsets" or not.

Depending on this they can be classified into *one-pass* and *multi-pass* coders. We discuss these differences for the case of pure triangular connectivity. To further simplify this classification we assume a mesh of sphere topology without boundary, so that we can ignore how to deal with holes and handles.

Face-based schemes (Gumhold and Strasser, 1998; Rossignac, 1999) describe all boundary updates per face. The boundaries are loops of edges that separate the region of processed faces from the rest. Each iteration grows the processed region by the triangle adjacent to the focus of the active boundary. It is adjacent to the active boundary in one of five ways. Edgebreaker (Rossignac, 1999) encodes the boundary updates corresponding to these five configurations using the labels C, R, L, S, and E. The Cut-Border Machine (Gumhold and Strasser, 1998) associates an additional *split offset* with each label S that—from a coding point of view—is redundant if (and only if) the traversal processes the faces in a recursive, depth-first manner.

Edge-based schemes (Li and Kuo, 1998; Isenburg and Snoeyink, 2000) describe all boundary updates per edge. The boundaries are loops of half-edges that separate the region of processed edges from the rest. Each iteration processes the edge that is adjacent to the focus of the active boundary. This edge is either adjacent to an unprocessed triangle or to the active boundary in one of four different ways. Our Face Fixer scheme (Isenburg and Snoeyink, 2000), which is covered in great detail in Chapter 3, describes the boundary updates corresponding to these five configurations using the labels T, R, L, S, and E. The Dual Graph method (Li and Kuo, 1998) does the same but also associates a split offset with each label S that is again redundant. Note that the original Face Fixer algorithm described in Chapter 3 uses labels F_3, F_4, F_5, ... in place of label T because it was mainly designed to compress polygon meshes. But when all faces are triangles we can simply write T instead of F_3.

Vertex-based schemes (Touma and Gotsman, 1998; Alliez and Desbrun, 2001b), also called *degree-based* or *valence-based*, describe all boundary updates per vertex. The boundaries are loops of edges that separate the region of processed faces from the rest and that maintain *slot counts* reflecting the number of unprocessed edges incident at each boundary vertex. When the boundary has a *zero slot* there is an unprocessed face that shares two edges with the boundary. The boundary update for including this face does not need to be described. When the boundary has no zero slots there is a face at the focus that shares only one edge with the boundary and that has a *free vertex*. The boundary update for including this face needs to be described. Two scenarios are possible: either the free vertex has not been visited, in which case its

degree is recorded, or is has been visited, in which case an offset in slots along the active boundary is recorded and the active boundary is split. Unlike the split offsets in face-based and edge-based coding, these offsets are not redundant (see Section 4.7.2 for the proof). Since encoding the split operations and their associated offsets is expensive (Alliez and Desbrun, 2001b) first move the focus to the least likely place for a split to occur.

One-pass schemes (Gumhold and Strasser, 1998; Li and Kuo, 1998; Touma and Gotsman, 1998) have explicit offsets associated with the symbols that encode split operations. These offset values allow the decoder to perform an instant re-play of the split operation that happened during encoding. This makes it possible both to compress in a single pass over the mesh while immediately producing the final bit-stream, as well as to decompress in a single pass over this bit-stream while immediately producing vertices and triangles. This is crucial for out-of-core operation on large meshes and is discussed further in Chapter 7. The use of explicit split offsets also makes it possible to traverse the mesh triangles in a non-recursive, breadth-first manner. This is important to create coherent triangles orderings, as we show in Chapter 9.

For coding schemes that use explicit offsets it is difficult to establish tight worst-case bounds on the maximal code size. Assuming that the number of split operations of their vertex-based coder is negligible small (Alliez and Desbrun, 2001b) claim that coding with vertex degrees is optimal. They show the worst-case entropy of a vertex degree distribution satisfying Euler's relation asymptotically approaches the information theoretic minimum for coding triangulations due to (Tutte, 1962). However, (Gotsman, 2003) has since shown that Alliez and Desbrun's analysis includes degree distributions that do not correspond to actual triangulations. He incorporates additional constraints on the vertex degree distribution that lowers its worst-case entropy below Tutte's bound implying that the splits *must* contribute a small fraction to the encoding. Since there is no upper bound on the size of their contribution, Gotsman concludes that "the question of optimality of valence-based connectivity coding is still open."

Multi-pass schemes (Taubin and Rossignac, 1998; Rossignac, 1999; Isenburg and Snoeyink, 2000) do not store explicit offsets. Instead they recover these offsets that are implicit in their label sequences by decoding either in two passes (Taubin and Rossignac, 1998; Rossignac, 1999; Rossignac and Szymczak, 1999) or in reverse (Isenburg and Snoeyink, 2000; Isenburg and Snoeyink, 2001b). This requires the compressor to always complete one boundary part after a split operation before continuing on the other part. Given such a recursive traversal, the symbols S and E form nested pairs in

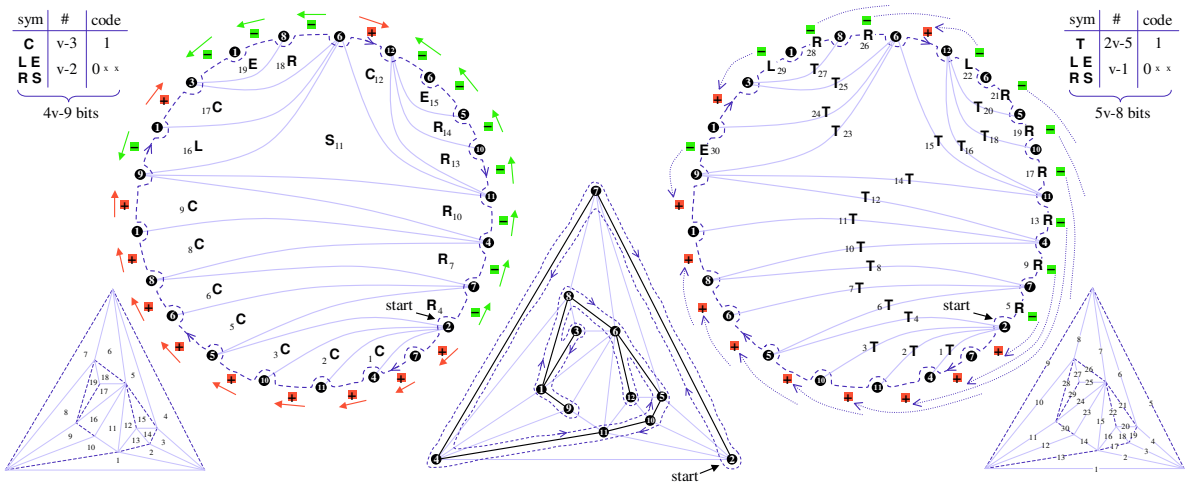


Figure 2.5: The Edgebreaker (left) and the Face Fixer (right) schemes traverse mesh triangles in the same depth-first order thereby constructing the same two spanning trees (middle). Because of the determinism in their construction, the labeling of these spanning trees can be compressed more efficiently than Turan's generic trees.

the label sequence that enclose subsequences of labels. For Edgebreaker and Face-Fixer, these label subsequences are self-contained encodings of the regions of the mesh enclosed by the boundary parts resulting from splits. Obviously this also determines the length of these boundary parts, which is exactly what is specified by the explicit split offsets. Having these offsets implicitly stored in the label sequence, however, prevents decoding from being an instant replay of the encoding process, which makes these coders unsuitable for out-of-core operation, as we point out in Chapter 7. The required recursiveness in traversal is also a guarantee for incoherence in the triangle ordering of the compressed mesh, as we illustrate in Chapter 9.

Because these schemes do not store offsets we can use a simple mapping from labels to bit codes that quickly establish interesting bounds on the coding costs. With the help of Figure 2.5 we give an intuitive explanation how recent offset-less schemes manage to improve on Turan's encoding method. Both Edgebreaker (Rossignac, 1999), the face-based scheme that does not use offsets, and Face Fixer (Isenburg and Snoeyink, 2000), the edge-based scheme that does not use offsets, can be looked upon as a labeling of the two spanning trees just like Turan's original method. These schemes manage to do so with fewer bits because they use spanning trees with certain properties (e.g. that are the result of recursively traversing the graph in a depth-first manner). The same can be said about the method by (Keeler and Westbrook, 1995) since the symbols it produces have exactly the same meaning as those produced by Edgebreaker.

Each Edgebreaker label C corresponds to a "+" and also marks an edge of the

triangle spanning tree. Labels R and L correspond to a "–" and also mark an edge of the triangle spanning tree. S marks two edges of the triangle spanning tree and E corresponds to two "–". Marking the edges of the triangle spanning tree may be thought of corresponding to either an opening or a closing bracket. There are two reason why the Edgebreaker labels allow a tighter bound on the code size than the Turan symbols: First, each of the five labels encodes a pair of Turan symbols, which—encoded independently—could form nine different combinations. Second, each C label pairs a "+" symbol with a bracket, which means that half of all labels are of type C. Note that the up and down labels "+" and "–" around the vertex spanning tree correspond exactly to the zip-directions used in the Wrap&Zip method (Rossignac and Szymczak, 1999). This method decodes mesh connectivity from the Edgebreaker labels by constructing the entire triangle spanning tree in a first pass and identifying edges whose zip-direction points to the same vertex in a second pass.

Each Face Fixer label T marks an edge of the triangle spanning tree, which may again be thought of representing either an opening or a closing bracket. Labels R, L, S, and E correspond to nested pairs of symbols "+" and "–". The reason that the Face Fixer labels allow a tighter bound on the code size than the Turan symbols is that pairs of "+" and "–" can be encoded with 3 bits whereas encoding them independently requires 2 bits per symbols, which makes 4 bits per pair.

Recently, more complex encodings of the Edgebreaker labels have been proposed (King and Rossignac, 1999; Szymczak et al., 2000; Gumhold, 2000; Gumhold, 2005). They establish tighter bounds that come closer to the information theoretical minimum without quite reaching it. Unlike Turan's method that allows the use of any vertex spanning tree and that does not only encode the connectivity, but also the particular spanning tree that was used, Edgebreaker and Face Fixer construct their spanning trees in a deterministic manner. This reduces the number of different descriptions that each connectivity has and therefore lowers the number of bits required for representing them. The next section describes a scheme that is even more deterministic in the choice of the vertex spanning tree that it uses for encoding a triangulation.

2.6 Optimal Coding of Planar Triangulations

The field of graph theory has recently seen renewed efforts towards optimal coding of planar graphs (He et al., 1999; Chiang et al., 2001). All these schemes make use of specially ordered spanning trees inspired by the work of (Schnyder, 1990). (Poulalhon and Schaeffer, 2003) finally show that a particular Schnyder decomposition of a

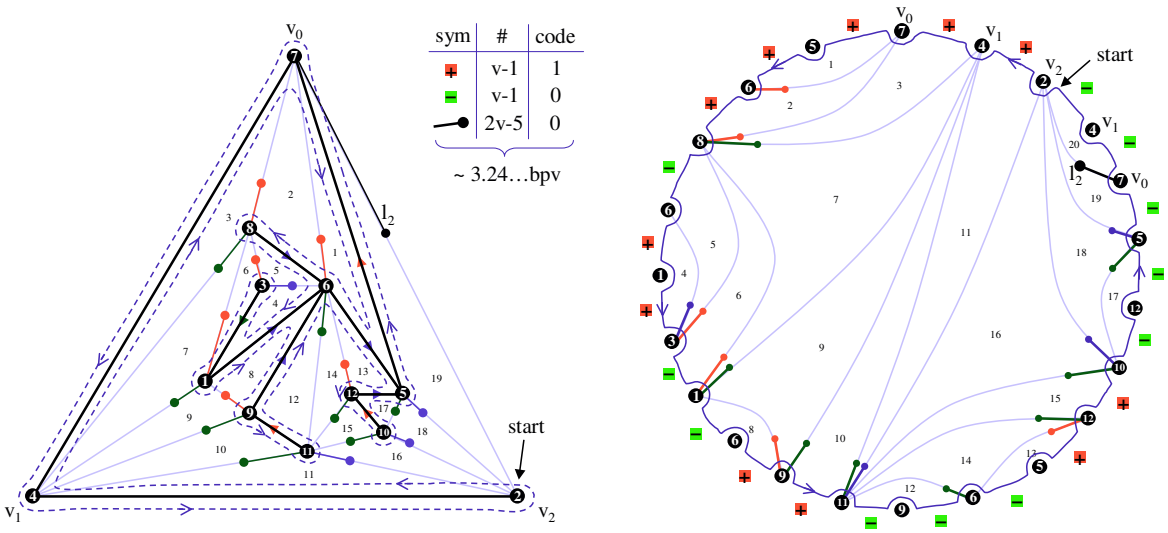


Figure 2.6: Poulalhon and Schaeffer construct a very particular vertex spanning tree. It has the property that a walk around it crosses at each node exactly two triangle spanning tree edges for the second time. The visualization on the right shows that this corresponds to a Turan-style labeling that uses closing brackets. The small numbers indicate the order in which the decoder reconstructs the triangles. Note that the original algorithm walks the opposite direction which corresponds to the use of opening brackets.

triangulation into three spanning trees can indeed be used for optimal coding.

Starting from a triangulation with an embedded *maximal realizer* (Schnyder, 1990) Poulalhon and Schaeffer construct a very particular vertex spanning tree that is shown in Figure 2.6. This vertex spanning tree has the property that a counterclockwise walk that starts at the root crosses at each node (but the first three) exactly two non-spanning tree edges for the second time. The original algorithm suggests walking in clockwise direction with the similar result that at each node (but the first three) exactly two non-spanning tree edges are crossed for the first time. An intuitive algorithm for constructing the maximal realizer of a triangulation is described in (Brehm, 2000).

The right illustration in Figure 2.6 shows that this corresponds to a Turan-style labeling that uses only closing brackets (or only opening brackets when using the walk direction employed in the original algorithm). The reason that Poulalhon and Schaeffer's encoding gives a tighter bound on the code size than the Turan symbols is that they manage to get away using just a single bit for each symbol "+", "–", ")". This is because their algorithm knows that in a spanning tree with two closing brackets per node, the first and the second occurrence of a zero bit at a node corresponds to a closing bracket, whereas the third zero bit must signal the "–". Therefore the one bit can be reserved to exclusively express the "+" symbols.

The resulting bit string contains $4n$ bits of which $3n$ bits are zeros while the remaining n bits are ones. Since the probability for one of the $3n$ zero bits to occur is 0.75 and the probability for one of the n one bits to occur is 0.25, the entropy of this bit string is $-3n \cdot \log_2(0.75) - n \cdot \log_2(0.25)$ or $\log_2(256n/27)$ or $3.24n$. This coincides with the optimal worst-case bounds for encoding a planar triangulation that can be derived from the enumeration work of (Tutte, 1962). Intuitively speaking, the reason why Poulalhon and Schaeffer are able to improve the worst-case bound for the code-size to the optimum of 3.24 bpv is that their choice in vertex spanning tree is even more special than that of Edgebreaker or Face Fixer.

2.7 Extensions to Polygonal Connectivity

Initially most connectivity compressionh schemes were only designed for purely triangular meshes. Several authors have reported simple extensions to their schemes in order to handle polygonal input. A naive approach arbitrarily triangulates the polygon mesh and then uses one bit per edge to distinguish the original edges from those added during the triangulation process. Marking every edge can be avoided by triangulating the polygons systematically. For the Topological Surgery method the extension to polygonal meshes as reported by (Taubin et al., 1998b) first cuts the mesh along a vertex spanning tree and then triangulates the tree of polygons that corresponds to the dual polygon spanning tree. Only the edges interior to the resulting triangle tree need to be marked. Obviously, all these edge-marking approaches will always require more bits for encoding the original polygonal connectivity than for encoding the triangular connectivity alone. However, (King et al., 1999) have shown that quadrangular meshes can be compressed more efficiently than their triangulated counterparts by not triangulating the mesh *prior* to compression.

The generalization of face-based coding to the polygonal case that was reported by (King et al., 1999) describes how to let the Edgebreaker scheme guide the triangulation process. When the encoding process encounters a polygonal face it systematically converts it into a triangle fan such that no longer all combinations of labels can occur. A quadrilateral face, for example, is systematically split into two triangles so that only 13 combinations are possible for assigning Edgebreaker labels to these triangles. They also show that the number of possible label combinations for systematically splitting a face of degree d equals the Fibonacci number $F(2d-1)$ and report a guaranteed 5 bits per vertex (bpv) encoding for simple polygon meshes without holes or handles. The

same extension was reported by (Kronrod and Gotsman, 2001) with a more polygonal mind-set. It directly enumerates the $F(2d-1)$ number of ways in which a polygon of degree d can interact with the active boundary. For purely quadrilateral meshes they report a mapping from enumerators to bit codes that guarantees a 3.5 bpv encoding. For meshes containing mainly quadrangles but also a few triangles they give another mapping that leads to rates of about four bits per polygon.

The generalization of our edge-based Face Fixer scheme to the polygonal case is straight-forward. After all, to directly code polygonal connectivity was our original motivation for developing this encoding scheme, which is described in detail in Chapter 3. Whenever the free edge is adjacent to an unprocessed face we also record the degree d of this face. This can simply be done by replacing the label T with a set of labels F_3, F_4, F_5, ... and including this face just as before into the boundary.

The generalization of the vertex-based coder of (Touma and Gotsman, 1998) to the polygonal case is described in Chapter 4. This can be done by *separately* recording a sequence of face degrees in addition to the sequence of vertex degrees and split symbols used by the original scheme. Encoding a triangle mesh will then correspond to the special case in which every face degree of this separately recorded face degree sequence is three. Since the entropy of such a sequence is zero and can essentially be encoded for free, our polygonal degree coder is a true generalization of the original scheme. A similar generalization of degree-based coding was reported independently by (Khodakovsky et al., 2002).

Recent work by (Fusy et al., 2005) extends the information theoretically optimal encoding method for planar triangulation of (Poulalhon and Schaeffer, 2003) to the polygonal case. We should point out, however, that from a practical point of view these optimal schemes are not so useful. Although they guarantee to encode *any* planar connectivity using no more bits than needed to distinguish between all connectivities, they also guarantee to *always* use this many bits. But the practical value of a mesh compression scheme is determined by its performance on typical mesh data. The mesh connectivities encountered in practice are of high regularity and represent a tiny subset of all connectivities. After all, these connectivities must allow a reasonable embedding in 3D since they are not random but belong to polygon meshes that represent piece-wise approximations of three-dimensional shapes. They can be encoded with bit-rates well below the theoretic worst-case bounds with coders that can adapt to this regularity.

Chapter 3

Edge-based Connectivity Coding

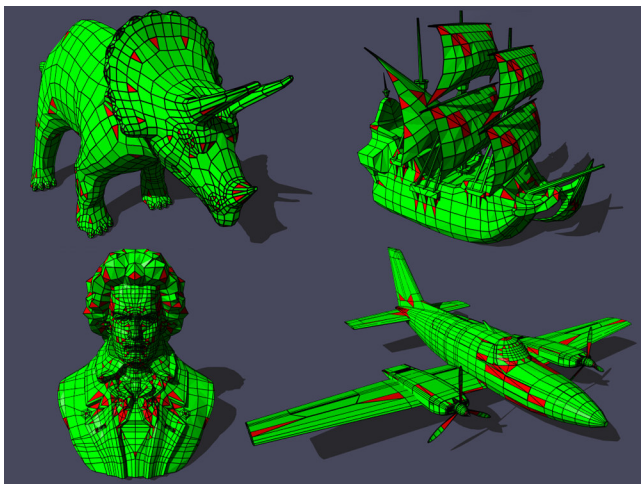


Figure 3.1: Many popular polygon meshes such as the triceratops, the galleon, or the cessna model contain a surprisingly small number of triangles, here shown in red.

Previous mesh compression techniques have focused on encoding fully triangulated data sets. Triangle meshes are naturally the lowest common denominator that any surface representation can be tessellated into. However, many models are represented by polygonal meshes that contain a surprisingly small percentage of triangles as, for example, the standard 'triceratops' and 'galleon' models shown in Figure 3.1, which are initially not triangulated. The dominating element of these models is the quadrangle or quadrilateral, but pentagons, hexagons and higher degree faces are also common.

Especially for storage purposes it is beneficial to keep a mesh in its native polygonal representation and delay the conversion to triangles until this becomes necessary. In this chapter we describe a simple scheme for encoding the connectivity of polygon meshes that we call 'Face Fixer'. In contrast to the face-based Edgebreaker scheme by (Rossignac, 1999), which assigns a label to each mesh face, the Face Fixer scheme assigns a label to each mesh edge and is therefore considered to be edge-based.

There is a small but crucial difference between our edge-based coding scheme and the face-based Edgebreaker scheme of (Rossignac, 1999). When Edgebreaker processes a face it immediately specifies the entire relationship between this face and the compression boundary. In the triangular case there are only five different configurations, which leads to the elegant CLERS labeling of the triangles. However, for a non-triangular face of degree d the number of ways the face can interact with the compression boundary quickly increases with the Fibonacci number $F(2d-1)$ (King et al., 1999; Kronrod and Gotsman, 2001). To accommodate large faces this requires either the use of a gigantic label set or the systematic break-down of larger faces into smaller ones.

Our method keeps the label set small by including faces into the active boundary without immediately specifying their entire adjacency relation with the compression boundary. This seems to make it a natural candidate for encoding polygonal connectivity where all we have to do differently is to also write down the degree of the polygon that is included. Later, in Section 3.5, we will see that having separate labels for including a face and for specifying how it is adjacent to the boundary also makes it possible to integrate triangle strip information into the encoding.

Face Fixer encodes polygonal connectivity as a sequence of labels F_n, R, L, S, E, H_n, and $\mathrm{M}_{i,k,l}$ whose length is equal to the number of edges in the mesh. This sequence represents an interwoven yet, compared to Edgebreaker, slightly de-coupled description of a polygon spanning tree and its complementary vertex spanning tree. For every face of n sides there is a label F_n and for every hole of size n there is a label H_n. Together they label the edges of the polygon spanning tree. For every handle there is a label $\mathrm{M}_{i,k,l}$ that has three integer values associated. These specify the two edges of the polygon spanning tree that need to be 'fixed' together to re-create the handle. The remaining labels R, L, S, and E label the edges of the vertex spanning tree and describe how to 'fix' faces and holes together. Finally, we use a standard adaptive, memory-sensitive arithmetic coder to compress this label sequence into a compact bit-stream.

3.1 Encoding and Decoding

Starting with a polygon mesh of v vertices, e edges, f faces, h holes, and g handles, the encoding process produces a sequence of e labels. This sequence contains f labels of type F_n, h labels of type H_n, g labels of type $\mathrm{M}_{i,k,l}$, and $v-2+g$ labels of type R, L, S, or E. The number of E labels equals the number of S labels minus the number of M labels plus one. The connectivity of the polygon mesh can be reconstructed with a single *reverse* traversal of the label sequence.

The algorithm maintains one or more loops of edges that separate a single processed region of the mesh from the rest. Each of these *boundary loops* has a distinguished *gate* edge. The focus of the algorithm is on the *active boundary*; all others are temporarily buffered in a stack. The initial active boundary, defined clockwise around an arbitrary mesh edge, has two *boundary edges*. The gate of the active boundary is the *active gate*.

In every step of the encoding process the active gate is labeled with either F_n, R, L, S, E, H_n, or $M_{i,k,l}$. Which label the active gate is given depends on its adjacency relation to the boundary. After recording the label, the boundary is updated and a new active gate is selected. Depending on the label, the boundary expands (F_n and H_n), shrinks (R and L), splits (S), ends (E), or merges ($M_{i,k,l}$). Table 3.1 summarizes the changes to the processed region and its boundaries for each operation. The encoding process terminates after exactly e iterations, where e is the number of mesh edges.

label	change to # of processed					# of boundary	
	faces	holes	vertices	edges	handles	loops	edges
F_n	$+1$	$\cdots$	$\cdots$	$+1$	$\cdots$	$\cdots$	$+(n-2)$
H_n	$\cdots$	$+1$	$\cdots$	$+1$	$\cdots$	$\cdots$	$+(n-2)$
R, L	$\cdots$	$\cdots$	$+1$	$+1$	$\cdots$	$\cdots$	-2
S	$\cdots$	$\cdots$	$\cdots$	$+1$	$\cdots$	$+1$	-2
E	$\cdots$	$\cdots$	$+2$	$+1$	$\cdots$	-1	-2
$M_{i,k,l}$	$\cdots$	$\cdots$	$\cdots$	$+1$	$+1$	-1	-2
init	$=0$	$=0$	$=0$	$=0$	$=0$	$=1$	$=2$
final	$=f$	$=h$	$=v$	$=e$	$=g$	$=0$	$=0$

Table 3.1: The changes in number of processed faces, holes, vertices, edges, handles, boundary components, and boundary edges that happen during encoding for each of the different label types. Initial and final counts are listed at the bottom.

In Figure 3.2 we illustrate for each label the situation in which it applies and the respective updates for gate and boundary. Both encoding and decoding are shown. The details for encoding are:

label F_n The active gate is not adjacent to any other boundary edge, but to an unprocessed face of degree n. The active boundary is extended around this face. The new active gate is the rightmost edge of the included face.

label R The active gate is adjacent to the next edge along the active boundary. The gate is 'fixed' together with this edge. The new active gate is the previous edge along the active boundary.

label L The active gate is adjacent to the previous edge along the active boundary. The

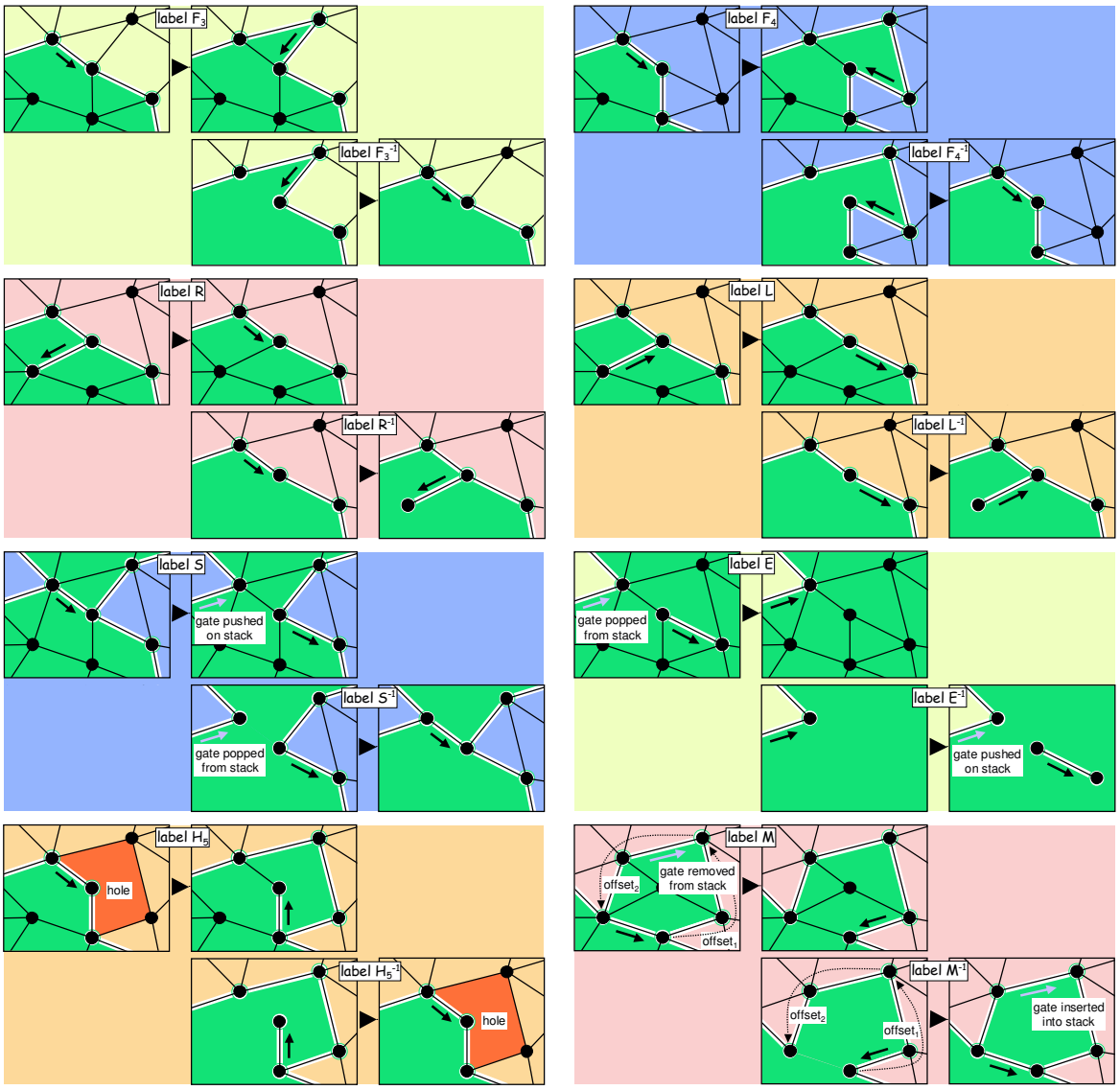


Figure 3.2: The labels of our edge-based coding scheme: when they apply and the corresponding updates to the compression boundary during encoding and decoding.

gate is 'fixed' together with this edge. The new active gate is the next edge along the active boundary.

label S The active gate is adjacent to an edge of the active boundary which is neither the next nor the previous. The gate is 'fixed' together with this edge, which splits the active boundary. The previous edge and the next edge along the active boundary become gates for the two resulting boundaries. The one that was the previous edge is pushed on the stack and encoding continues on the other.

label E The active gate is adjacent to an edge of the active boundary which is both the next

and the previous. Then the active boundary consists of only two edges which are 'fixed' together. The encoding process terminates if the boundary stack is empty. Otherwise it continues on the boundary popped from this stack.

label $\mathbf{H}_n$ The active gate is not adjacent to any other boundary edge, but to an unprocessed hole of size n. The active boundary is extended around this hole. The new active gate is the rightmost edge of the included hole.

label $\mathbf{M}_{i,l,k}$ The active gate is adjacent to a boundary edge which is not from the active boundary, but from a boundary in the stack. 'Fixing' the two edges together merges the two boundaries and the boundary is removed from the stack. Its former position i in the stack and two offset values l and k (see Figure 3.2) are stored together with the label. The new active gate is the previous edge along the boundary from the stack.

We use a simple half-edge structure (Guibas and Stolfi, 1985) during encoding and decoding to store the mesh connectivity and to maintain the boundaries. Besides pointers to the origin, to the next half-edge around the origin, and to the inverse half-edge, we have two pointers to reference a next and a previous boundary edge. This way we organize all edges of the same boundary into a cyclic doubly-linked list.

The decoding process reconstructs the connectivity of the polygon mesh with a single reverse traversal of the label sequence. As illustrated in Figure 3.2, each label has a unique inverse operation that does exactly the opposite of the gate and boundary updates that happened during encoding. The time complexity for decoding is linear in the number of mesh edges. An exception is the inverse operation for label $\mathrm{M}_{i,k,l}$ which requires the traversal of $k+l$ edges. However, labels of this type correspond to handles in the mesh, which are typically of rare occurrence.

3.2 Compression

The label sequence produced by the encoding process is subsequently mapped into a bit-stream. The frequencies with which the different labels occur are highly non-uniform, which invites some kind of entropy encoding. There is also a strong correlation among subsequent labels, which can be exploited using a memory-sensitive encoding scheme. Therefore we use a simple order-3 adaptive arithmetic coder (Witten et al., 1987) to compress the symbols in the order they are produced during encoding.

For an adaptive arithmetic encoder with three label memory the space requirement for the probability tables grows in $O(t^4)$ with the types of symbols t. Therefore we limit the number of labels in the input sequence to eight: F_3, F_4, F_5, F_c, R, L, S, and

E. This allows the implementation of the arithmetic order-3 entropy coder to be both space and time efficient. The probability tables need only 4 KB of memory and we can use fast bit operations to manage them. Labels F_n with $n > 5$ are expressed through the combination of a label F_5 and $n-5$ subsequent labels of type F_c. We observe that labels of type F_n are never followed by label L or label E. We exploit this to express the typically infrequent appearing labels H_n and $M_{i,k,l}$ using the combinations F_4L and F_4E. The integer values associated with these labels are stored using a standard technique for encoding variable sized integers with an arithmetic coder.

name	vertices	faces		edges		corners		bpv
triceratops	2832	2834	(5660)	5664	(8490)	11328	(16980)	**2.115**
galleon	2372	2384	(4698)	4733	(7047)	9466	(14094)	**2.595**
cessna	3745	3927	(7446)	7650	(11169)	15300	(22338)	**2.841**
beethoven	2655	2812	(5030)	5327	(7545)	10654	(15090)	**2.890**
sandal	2636	2953	(4952)	5429	(7428)	10858	(14856)	**2.602**
shark	2560	2562	(5116)	5120	(7674)	10240	(15348)	**1.670**
al	3618	4175	(7152)	7751	(10728)	15502	(21456)	**2.926**
cupie	2984	3032	(5944)	6004	(8916)	12008	(17832)	**2.307**
tommygun	4171	3980	(8210)	8085	(12315)	16170	(24630)	**2.611**
cow	2904	5804	(5804)	8706	(8706)	17412	(17412)	**2.213**
teapot	1189	1290	(2378)	2479	(3567)	4958	(7134)	**1.669**

Table 3.2: The number of vertices, faces, edges, and corners together with the achieved connectivity compression in bits per vertex (bpv) for the models shown in Table 4.1. In parentheses are the corresponding numbers for the triangulated version of the model.

The compression scheme described above produces compact encodings for the label sequence. In Table 3.2 we report connectivity compression results in bits per vertex (bpv) for a set of popular polygonal models. These meshes are pictured in Table 4.1 where their topology is characterized further and where their polygonal composition is described in detail. Our bit-rates are sometimes higher than those reported for other schemes. For example, the vertex-based coder of (Touma and Gotsman, 1998) reports a bit-rate of 2.1 bpv for the galleon model and a bit-rate of 2.4 bpv for the beethoven bust. However, this coder triangulates these polygonal models prior to compression and these reported bit-rates do not include the additional code that would be necessary to recover the original polygonal connectivity.

3.3 Quadrilateral Grids

Instead of fixing together faces the Face Fixer scheme can also fix together patches of faces. Then we have to describe in addition the interior of these patches. If a patch is

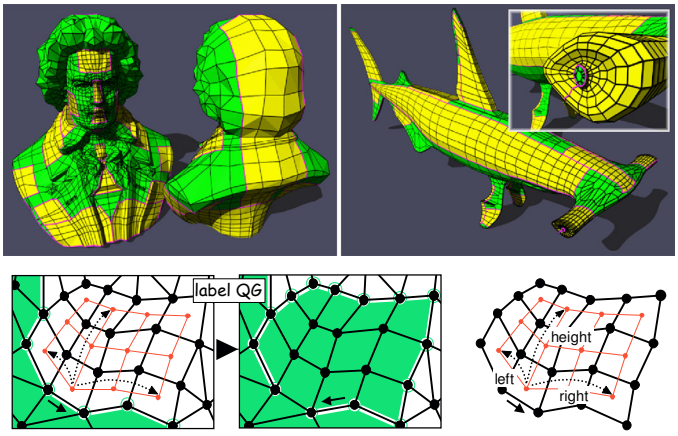


Figure 3.3: The beethoven bust and the shark model with quad grids marked in yellow. The label QG encodes a quad grid by specifying its left and right extend and its height.

a rectangular quadrilateral grid this can be done very efficiently through the number of rows and columns in this grid. The beethoven bust and the shark model shown in Figure 3.3 for example, contain large patches of quadrilateral grids. We introduce the label $\mathrm{QG}_{r,l,h}$ to include such *quad grids* into the active boundary. The associated integer values r, l, and h count the number of quadrangles that this grid extends to the right, to the left, and across as seen from the active gate.

Optimal selection of a set of non-overlapping quad grids on the model is not only NP-hard, we also lack a well-defined optimality criterion. Including quad grids into the active boundary breaks up the regularity of the label stream, which in turn hampers subsequent arithmetic coding. However, using a simple greedy method for finding a few large quadrilateral grids already leads to improved bit-rates: The connectivity of the teapot, for example, compresses down to 1.069 bpv using 10 quad grids, for the shark 1.374 bpv, for the galleon 2.190 bpv, and for the beethoven bust 2.591 bpv.

3.4 Coding Triangular and Quadrangular Meshes

A triangle/quadrangle mesh is a special kind of polygon mesh whose faces are all triangles/quadrangles. Since all of their faces have the same degree we use label T as a shorthand for label F_3 and label Q as a shorthand for label F_4. For simple connectivities without holes and handles a simple mapping from labels to bits gives us encodings with fixed bit-rates. A simple triangle mesh with v vertices has $3v-6$ edges and $2v-4$ triangles. Thus, $2v-4$ labels are of type T while the remaining $v-2$ labels are R, L, S, or E. A mapping that uses 1 bit for label T and 3 bits each for the other labels

guarantees a $5v - 10$ bit encoding. Similarly, a simple quadrangle mesh with v vertices has $2v - 4$ edges and $v - 2$ quadrangles. Here $v - 2$ labels are of type Q while the remaining $v - 2$ labels are R, L, S, or E. An encoding that uses 1 bit for label Q and 3 bits each for the other labels guarantees a $4v - 8$ bit encoding.

The correlation among subsequent labels is easily exploited with an adaptive order-k arithmetic coder that learns the probabilities with which each label type follows the preceding k labels. Compression results for various fully triangulated meshes that are achieved by applying an order-3 coder to the label sequence are listed in Table 3.3. Employing an arithmetic coder is computationally much more complex than a simple mapping from labels to bits and may not always be an option. However, the correlation among subsequent labels can also be exploited by making the bit mapping dependent on one or more preceding labels. For example, using 1 bit for label T and a varying assignment of 2, 3, 4 and 4 bits for labels R, L, S, and E guarantees a $6v - 12$ bit encoding, while being in practice closer to $4v$ bits. The small table on the right describes the bit assignment used for the results reported in Table 3.3.

after	TRLSE
T, R	1 2 4 3 4
L	1 4 2 4 3
S	1 4 3 4 2
E	1 2 4 4 3

The number of holes and handles of a mesh is generally small and so is the number of labels H_n and $M_{i,k,l}$. Since a label T can never be followed by a label L or E, we can encode labels H and M with the label combinations TL and TE without changing the above bit assignment. Their associated integer values are stored as before.

mesh characteristics					bit-rates (bpv)	
name	vertices	triangles	holes	handles	simple	aac-3
bishop	250	496	-	-	4.00	**1.86**
shape	2562	5120	-	-	3.99	**0.77**
triceratops	2832	5660	-	-	4.00	**2.52**
fandisk	6475	12946	-	-	4.00	**1.67**
eight	766	1536	-	2	4.09	**1.43**
femur	3897	7798	-	2	4.16	**3.05**
skull	10952	22104	-	51	4.22	**2.96**
bunny	34834	69451	5	-	4.00	**1.73**
phone	33204	66287	3	-	4.05	**2.70**

Table 3.3: The compressed connectivity in bits per vertex for various fully triangulated meshes with a simple bit assignment scheme and an order-3 adaptive arithmetic coder.

3.5 Coding Stripified Triangle Meshes

Using mesh compression allows faster distribution of 3D content in networked environments where transmission bandwidth is a limited resource. However, for distributed

interactive visualization not only the speed at which a triangle mesh can be received is important, but also the speed at which it can be displayed. At this stage the bottleneck becomes the rate at which the geometry data can be sent to the rendering engine. When each triangle of the mesh is rendered individually by sending its three vertices to the graphics hardware, then every mesh vertex is processed about six times, which involves passing its three coordinates and optional normal, color, and texture information from the memory to and through the graphics pipeline.

A common technique to reduce the number of times this data needs to be transmitted is to send long runs of adjacent triangles. Such triangle strips (Woo et al., 1996) are widely supported by today's graphics software and hardware. Here two vertices from a previous triangle are re-used for all but the first triangle of every strip. Depending on the quality of the triangle strips this can potentially reduce the number of vertex repetitions by a factor of three. Traditionally, an optimal stripification for rendering purposes covers the mesh with as few triangle strips using as few swap operations as possible. Computing such a set of triangle strips is NP-hard (Evans et al., 1996a) but various heuristics for generating good triangle strips have been proposed by (Evans et al., 1996b; Speckmann and Snoeyink, 1997; Xiang et al., 1999; Hoppe, 1999).

Given the difficulty of generating good triangle strips it would be desirable to do this just once and store the computed stripification together with the mesh. Especially for data sets with a larger distribution it would be worthwhile to provide a good precomputed stripification. However, currently available connectivity coding schemes do not support the encoding of stripified meshes. Obviously one can enhance any existing compression method by encoding the stripification separately and concatenating the results. But such a two-pass technique adds unnecessary overhead—it does not exploit the correlation between the connectivity and the stripification of a mesh. Next we describe a simple extension of our edge-based encoding scheme that can compress the connectivity and the stripification of a triangle mesh in an interwoven fashion while fully exploiting the correlation between the two.

3.5.1 Triangle Strips

Supported in software and hardware, triangle strips allow more efficient rendering of triangle meshes by reducing the data transfer rate between the main memory and the graphics engine. A triangle strip is a sequence of m vertices $(v_0, ..., v_{m-1})$ that represents the sets of triangles $\{(v_i, v_{i+1}, v_{i+2})\}$ for even i and $\{(v_{i+1}, v_i, v_{i+2})\}$ for odd i with $0 \leq i < m - 2$. The distinction between odd and even assures a consistent

orientation of all triangles. A strip is called *sequential* when it turns alternating to the right and to the left. A strip is is called *generalized* when it contains consecutive turns in the same direction. To make a strip turn twice into the same direction a degenerate zero-area triangle is inserted into the strip. The cost for such a *swap* operation is one vertex, which is cheaper than a *restart* operation that costs two vertices.

We say that an edge is a *strip-internal* edge if it is shared by two triangles that appear subsequently in a strip. The set of strip-internal edges marks either zero, one, or two edges of every triangle. A triangle without a strip-internal edge is a triangle strip by itself. A triangle with one strip-internal edge is either the start or the end of a strip. A triangle with two strip-internal edges is in the middle of a triangle strip.

3.5.2 Encoding Connectivity and Stripification

Representing mesh connectivity with indexed triangle strips rather than with individually indexed triangles reduces the amount of data by a factor between two and three. But the storage costs for indexed strips is still $2n \log n$ bits whereas typical connectivity coding schemes typically only need somewhere between $2n$ and $4n$ bits. However, current compression techniques have no means to preserve the information about how the triangles are arranged into strips. The straight-forward approach would be to append an encoding of the stripification to the encoding of the connectivity. This can be done with one bit per edge or $3n$ bits by marking all strip-internal edges. It is then still necessary to distinguish the start from the end of a generalized triangle strip, because one direction sometimes needs one fewer swap operation than the other. This can be determined in a single traversal of the triangle strip by counting the number of necessary swap operations.

However, encoding connectivity and stripification of a mesh separately fails to exploit the redundancy between the two: Every strip expresses the edge adjacency for each pair of subsequent triangles it contains. This local connectivity information also needs to be captured by the mesh compression scheme. The extension of our edge-based connectivity coding scheme for stripified triangle meshes specifies this information only once. Instead of traversing a triangle spanning tree using a deterministic strategy we let the underlying stripification be the guide. The adjacency information that is encoded while walking along a strip means progress for both the compression of connectivity and the compression of stripification. As before, the encoding process initially defines the active gate and the active boundary around some edge of the mesh. However, now this choice is not completely arbitrary. The edge must not be strip-internal.

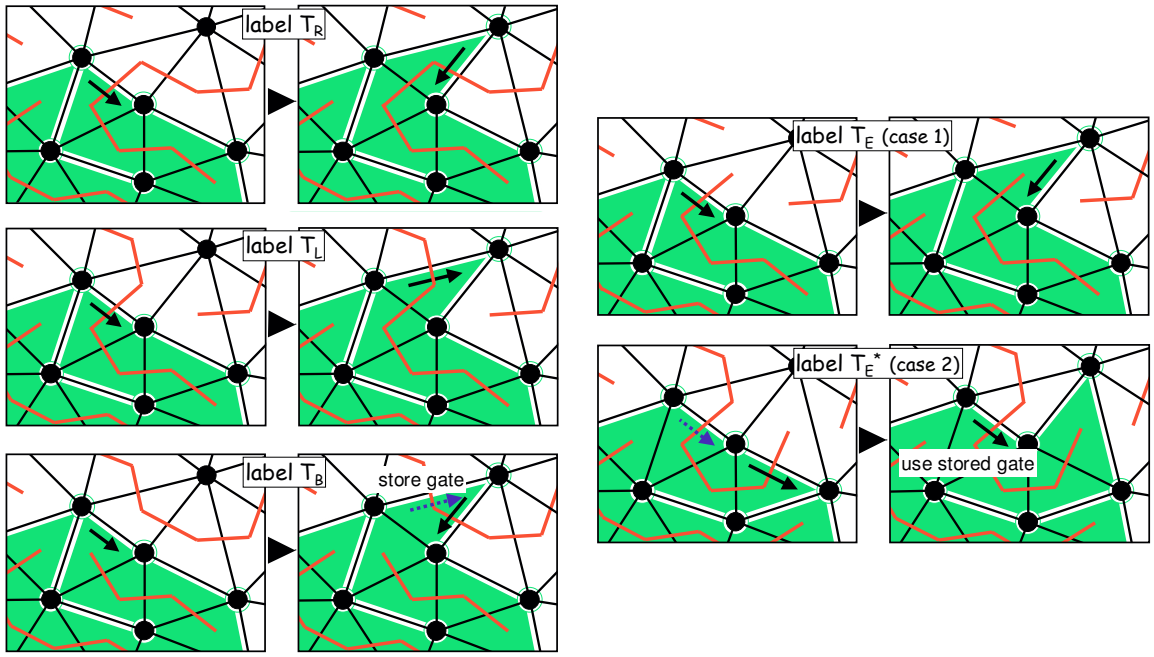


Figure 3.4: The labels T_R, T_L T_B, and T_E as used by the encoder. The black arrow denotes the active gate, the dashed arrow denotes a stored gate. Strips are marked red.

Again the active gate is labeled at each step of the encoding algorithm. Instead of label T we use the four labels T_R, T_L, T_B, and T_E. This subclassification captures the stripification of the mesh. The four labels direct the way the encoding process traverses the mesh triangles so that it follows the underlying strips. Once a triangle strip is entered, it is processed in its entirety using these labels. The total number of edges that receive labels T_R, T_L, T_B, or T_E is equal to the number of mesh triangles. The labels R, L, S, E, H, and M are used and assigned as before.

Each of the four new label updates the boundary just like label T (or rather label F_3) from Figure 3.2. The difference—illustrated in Figure 3.4—lies in the way the active gate is updated. They are as follows:

label T_R The triangle strip leaves the included triangle through the right edge. The new active gate is this right edge.

label T_L The triangle strip leaves the included triangle through the left edge. The new active gate is this left edge.

label T_B The triangle strip leaves the included triangle through the right and the left edge, which means we just entered this triangle strip somewhere in its middle. Both directions need to be considered. Therefore the left edge is stored and the right edge is the new active gate.

label T_E (case 1) The triangle strip leaves the included triangle neither through the right nor through the left edge *and* this is the last triangle of this strip. The new active gate is the right edge.

label T_E^* (case 2) The triangle strip leaves the included triangle neither through the right nor through the left edge, *but* this is not the last triangle of this strip. Then there was a preceding label T_B. The edge that was stored with label T_B is the new active gate.

strip characteristics				corners of		connectivity		stripification		interwoven	
name	strips	swaps	V/T	triangles	strips	simple	aac-3	simple	aac-3	simple	aac-3
bishop	1	72	1.15	1488	498	4.00	1.86	2.28	1.10	2.98	**1.78**
shape	2	220	1.04	15360	5124	3.99	0.77	2.08	0.45	3.09	**0.62**
triceratops	144	1424	1.30	16980	5948	4.00	2.52	2.64	2.05	4.12	**3.49**
fandisk	224	1630	1.16	38838	13394	4.00	1.67	2.28	1.35	3.61	**2.25**
eight	24	64	1.07	4608	1584	4.09	1.43	2.16	0.83	3.46	**1.78**
femur	237	1982	1.32	23394	8272	4.16	3.05	2.64	2.17	4.48	**4.02**
skull	600	6165	1.33	66312	23304	4.22	2.96	2.70	2.18	4.74	**4.18**
bunny	1229	10851	1.19	208353	71909	4.00	1.73	2.38	1.52	3.69	**2.40**
phone	1946	17519	1.32	198861	70179	4.05	2.70	2.65	2.13	4.42	**3.88**

Table 3.4: Triangle strip characteristics and compression results: # of strips, # of swaps, and vertex per triangle ratio (V/T) of the stripified mesh; the achieved compression rates in bits per vertex with a simple bit mapping scheme and an arithmetic coder for the connectivity alone, for the stripification alone, and for the connectivity interwoven with the stripification.

The decoder again processes the labels in reverse order by performing the inverse of each label operation. However, one initial traversal of the labels in forward order is necessary. For every label T_B we count the number of encountered T_R labels before the first occurrence of a label T_E. We add 2 to the count and associate this value w with the respective label T_E, marking it with a *. When during the decoding process a label T_E^* with associated value w is encountered, we walk from the active gate w edges along the active boundary. The edge we arrive at is the new active gate and we continue normally. This little variation becomes necessary to invert what happens during encoding: The first occurrence of a label T_E after a label T_B marks the completion of one end of a triangle strip. The active gate jumps to the edge that was stored with the preceding label T_B. The computed value expresses how many boundary edges were between the active gate and the stored edge at the time this jump occurred. The time complexity for decoding remains linear, since every triangle strip is traversed at most once. The example in Figure 3.6 and 3.7 leads step by step through the encoding and decoding process of a small mesh with two triangle strips.

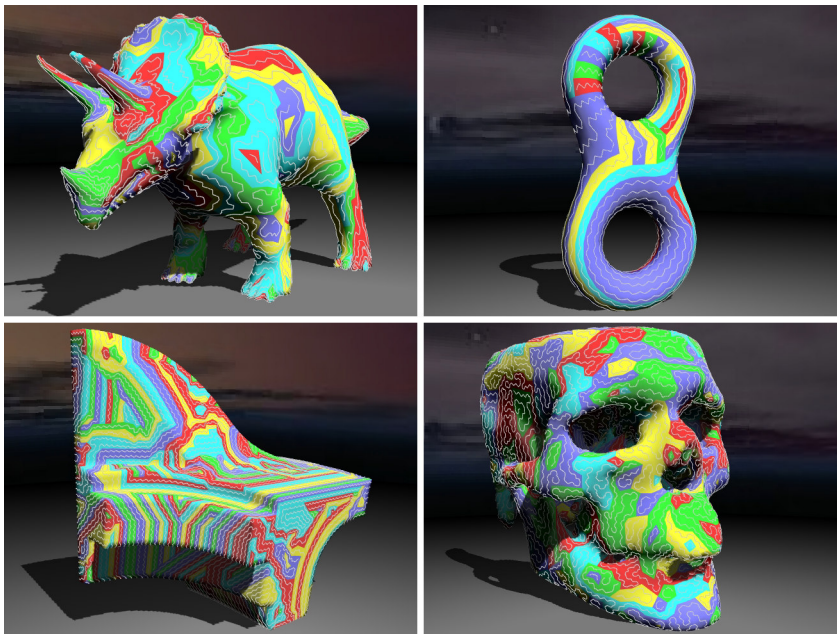


Figure 3.5: Some of the stripified meshes used in our experiments with triangle strips marked in white and rendered in different colors: triceratops, eight, fandisk, and skull.

While we have more label types to encode, the correlation among subsequent labels is stronger than before. We observe long runs of labels R and L, and long sequences of alternating labels T_R and T_L. The simple bit assignment scheme that is described in the small table below exploits these dependencies and achieves bit-rates between 3.0 and 5.0 bpv. This bit allocation scheme is geared towards long triangle strips with alternating left-right turns. The encodings are more compact for stripifications that have fewer strips and fewer swaps.

The resulting compression rates for connectivity and stripification increase by at most 0.6 bpv for the simple bit mapping and 1.3 bpv for the arithmetic coder compared to the rates for connectivity alone. For very regular meshes, like the bishop and the shape model, the compression rates even decrease. This is because the stripification software was able to decompose these meshes into long, almost sequential triangle strips. The label sequences encoding them are highly regular and have a lower entropy. The relation between the quality of triangle strips and the achieved compression becomes apparent in Table 3.4. It is noticeable that stripifications with a high vertex per triangle ratio do not compress as

after	T_R	T_L	T_B	T_E	R	L	S	E
T_R,T_E^*	2	1	–	2	–	–	–	–
T_L,T_B	1	2	–	2	–	–	–	–
T_E	4	5	3	6	2	7	1	7
R, E	7	6	5	7	1	4	3	2
L, S	6	7	5	7	4	1	3	2

well. Nevertheless, the compression rates we achieve for connectivity and stripification are significantly better than those of previously reported compression schemes for connectivity combined with an one bit per edge (3 bpv) encoding of the stripification.

3.5.3 Encoding the Stripification separately

While our scheme was designed to encode the connectivity together with the stripification of a mesh, it can also be used to compress the stripification separately. Given that the connectivity of the mesh is known, only the labels T_R, T_L, T_B, and T_E are needed to reconstruct the stripification. The decoder simply performs an exact replay of the *interwoven* encoding process. For this encoder and decoder also need to agree on the edge around which the initial active boundary is defined. Only when the active gate is adjacent to an unprocessed triangle, the next label is read and the mesh traversal directed accordingly. Otherwise the active gate is adjacent to a hole or a boundary edge and the applicable operation R, L, S, E, H, or M is carried out.

Compressing the subsequence of labels T_R, T_L, T_B, and T_E with the bit assignment scheme described in the table on the right always outperforms the one bit per edge (3 bpv) needed to mark all strip-internal edges (see Table 3.4). Especially for stripifications with few swaps and restarts this symbol sequence compresses aggressively with an order-3 arithmetic encoder.

after	T_R	T_L	T_B	T_E
T_R,T_E^*	2	1	–	2
T_L,T_B	1	2	–	2
T_E	2	2	2	2

3.6 Summary

We have presented an edge-based compression algorithm that encodes the connectivity of surface meshes directly in their polygonal representation. This has several benefits compared to methods that compress pre-triangulated polygon meshes: The original connectivity is preserved. Properties associated with faces and corners need not to be replicated. Subsequent stripification algorithms have more flexibility for generating better triangle strips. Predictive coding for geometry and property data can exploit additional convexity and planarity constraints. Furthermore, this method improves compression rates compared to approaches that triangulate meshes prior to compression and recover the polygons by marking edges. While the freedom to triangulate polygons on demand could lead to more compact encodings for the triangulated mesh, the number of bits required to mark the added edges could be as high as 3 bpv.

For triangular meshes, we have extended edge-based coding to include information about a pre-computed set of triangle strips into the compressed connectivity. This

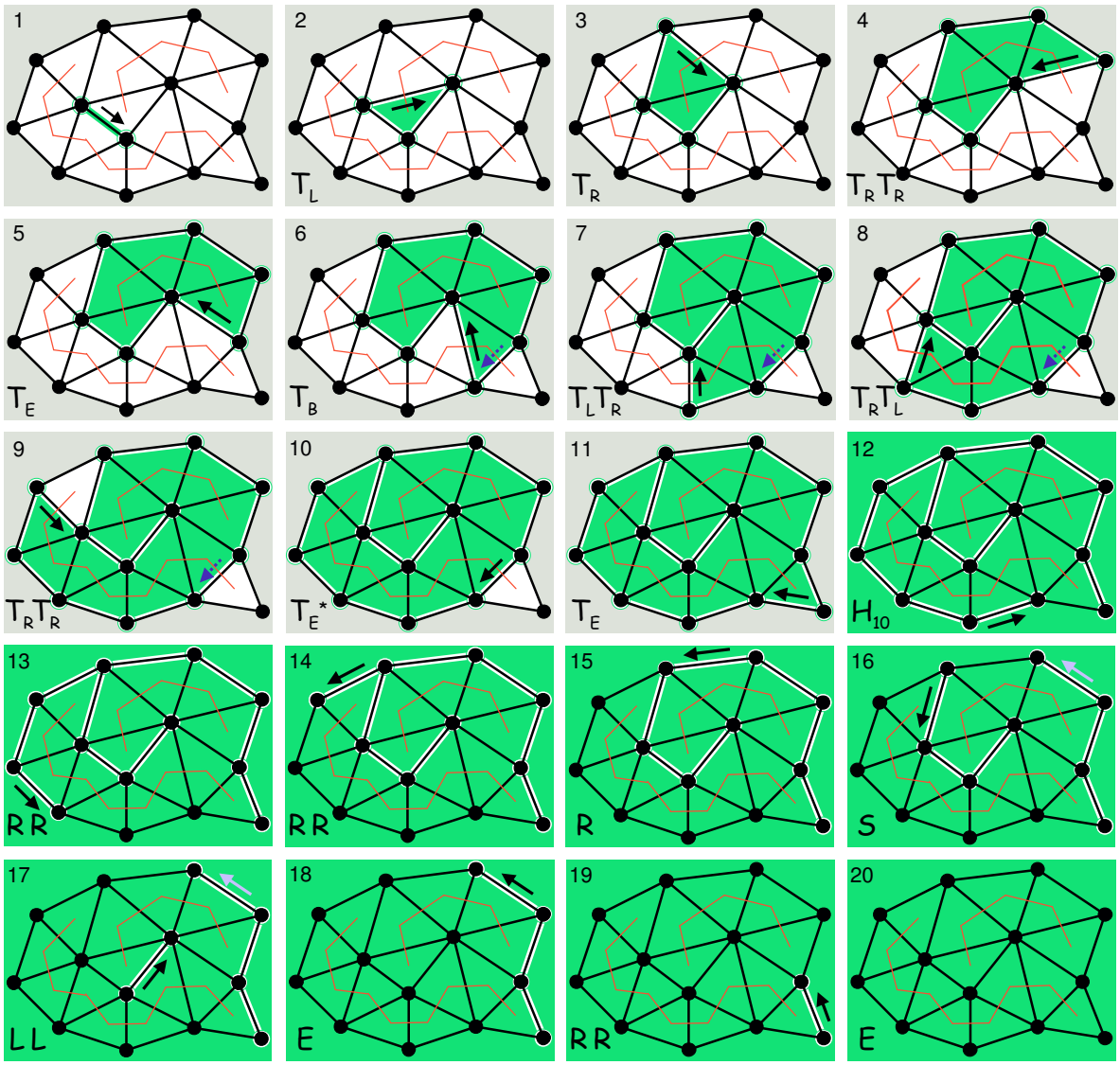


Figure 3.6: An example run of the encoding algorithm on a small mesh with two triangle strips. The interior of the active boundary is shaded dark, the active gate is denoted by a black arrow, a gate in the stack by a grey arrow, and a stored gate by a dashed arrow. The label(s) in the lower left corner of each frame express the performed update(s) since the previous frame. (1) Initial active boundary. (2-4) Boundary is expanded along the first triangle strip. (5) Reaching the last triangle of this strip. (6) Entering the second triangle strip in its middle. (7-9) Expanding this strip into one direction. (10) Finishing one side, the active gate jumps to expand other direction. (11) Finishing the other side. (12) Including a hole of ten edges. (13-15) Fixing the boundary with five R labels. (16) Splitting the boundary, one part is pushed on stack, continuing on other part. (17) Fixing the boundary with two L labels. (18) Ending this boundary, popping a boundary from stack. (19) Fixing the boundary with two R labels. (20) Ending this boundary, stack is empty, terminate.

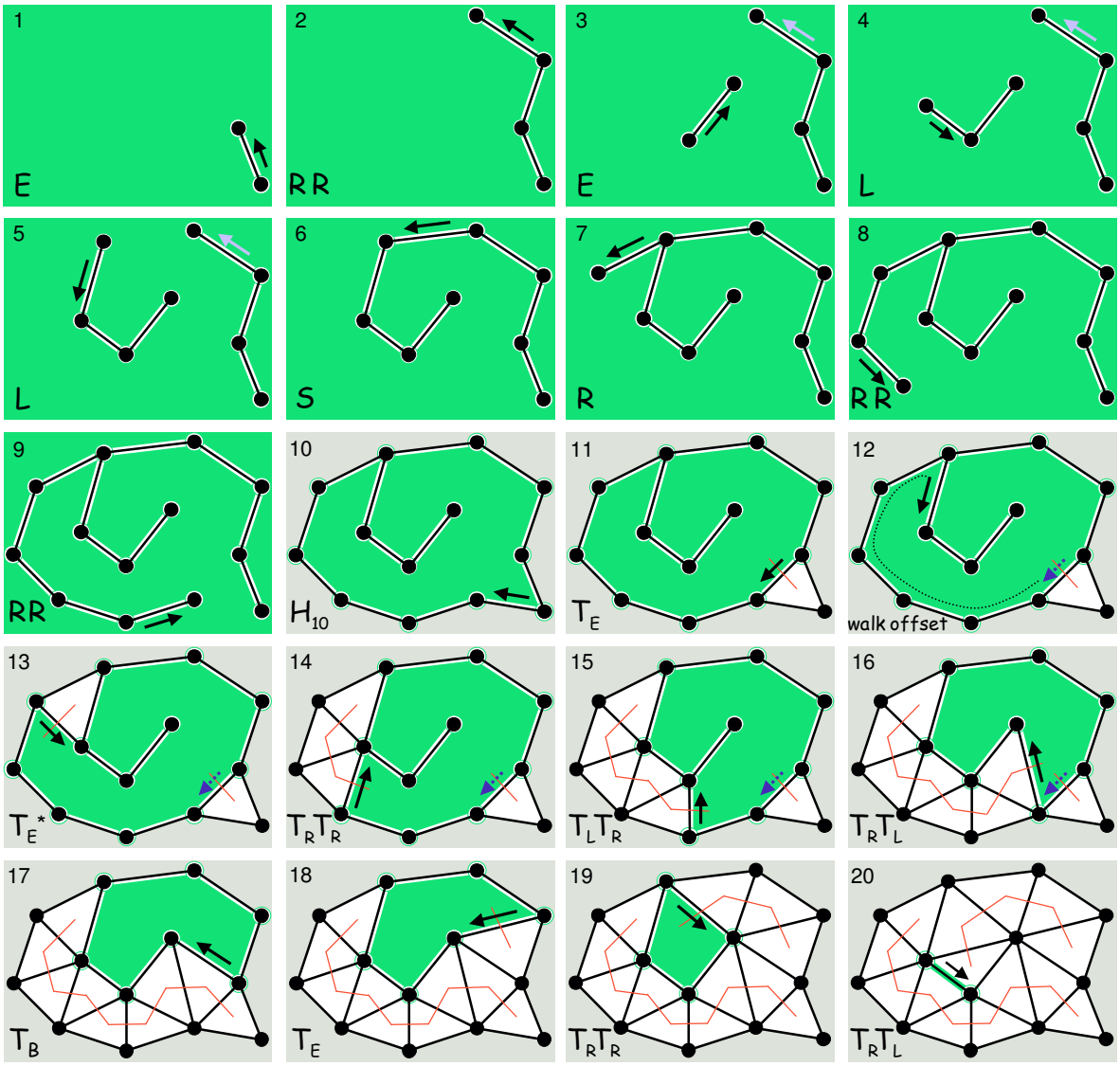


Figure 3.7: Reconstructing connectivity and stripification from the label sequence generated in Figure 3.6. The labels indicate the reversed operations since the previous frame. A forward pass counts the number of T_R labels between a T_B and the next T_E and stores this count plus 2 with the now marked T_E^* label. (1) Creating a boundary of length two reverses the last E operation. (2) Expanding this boundary reverses two R operations. (3) Pushing the boundary on the stack and creating a new boundary reverses another E operation. (4-5) Expanding the boundary. (6) Merging the boundaries that were split by the S label. (7-9) Further expansion of the boundary. (10) Recreating a hole of size ten. (11) Recreating a triangle that starts the first strip. (12) Walking the offset associated with the marked label. (13) Recreating the first triangle at the other end of the strip. (14-16) Recreating six more triangles of this strip. (17) Finishing the first strip, by gluing its two sides together. (18) Recreating a triangle that starts the next strip. (19-20) Recreating four more triangles of this strip, terminate.

approach can exploit the existing correlation between connectivity and stripification of a mesh. This is especially useful for compressing models that are widely disseminated. The computation of high quality stripifications is expensive and, in particular for triangle meshes with corner attributes, not trivial. Once a good set of triangle strips has been computed, our technique allows to store and distribute it together with the model at little additional storage or processing cost. The recovered triangle strip information can also be exploited for more efficient coding of the way per-corner properties are attached to the mesh as we see in Section 5.2.4. Furthermore, it can be used to guide predictive coding of vertex positions along the strips. Especially for CAD models that have sharp creases this gives good results. Since triangle strips typically do not cross creases in the model, predicting across discontinuities can easily be avoided.

3.7 Hindsights

Long after this work was completed, we realized that it is beneficial to compress the labels in the order they are consumed by the decoder instead of in the order they are produced by the encoder. On one hand this will eliminate the need to decompress and reverse the label sequence before beginning the actual decoding. On the other hand this will allow us to interleave the reconstruction of the mesh and the decompression of the label sequence so that the partially decoded mesh can be used to predict the next label, leading to better compression rates. It also allows improving compression of the integers associated with labels H_n and $M_{i,k,l}$ because their maximal possible value can then be derived from the state of the decoder. In (Isenburg and Snoeyink, 2005a) we give further evidence that reverse decompression, which was first suggested by (Szymczak, 2002) for the Spirale Reversi decoder of (Isenburg and Snoeyink, 2001b) to improve Edgebreaker compression, consistently gives the best compression rates.

With the advent of newer graphics accelerators that support both transparent caching of sixteen or more vertices in the pipeline and direct storage of vertex arrays on the card, locality in the access pattern of triangle strips has become a more important optimization criterion than maximal length and minimal number of swaps. Having a large number of smaller strips ordered in a breadth-first manner with maximal re-use of vertices between strips will usually lead to better rendering performance than a few long triangle strips that spiral around the mesh. Unfortunately, our encoding scheme neither preserves the ordering of the strips nor the information about the starting triangle of each individual strip. As presented our scheme is therefore not suited to

compress stripified meshes for which the global locality among the triangle strips needs to be preserved.

In retrospect, we generally advise against the use of recursive, depth-first compression schemes that need to maintain a stack of compression boundaries in order to encode the connectivity of polygon meshes such as the Face Fixer scheme presented here or the Edgebreaker scheme of (Rossignac, 1999). These schemes systematically create compressed meshes with incoherent element orderings, which especially as meshes become larger has practical disadvantages, as we learn in Chapter 9. In the next chapter we present a coding scheme for polygon mesh connectivity that not only has better compression rates, but also allows implementing coherent traversal strategies.

Chapter 4

Degree-based Connectivity Coding

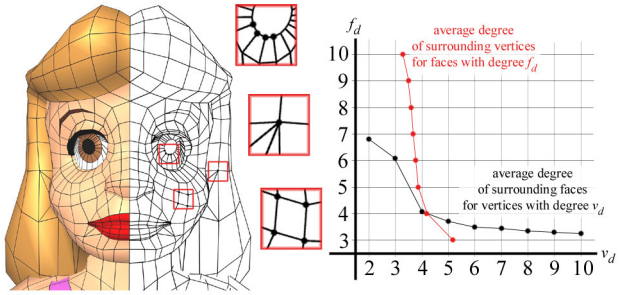


Figure 4.1: Low-degree vertices of a polygon mesh are usually surrounded by high-degree faces and vice-versa. The two plots show that this is generally the case. They report the average degree of faces (vertices) that surround vertices (faces) of degree d.

In this chapter I present a scheme for coding polygonal mesh connectivity that delivers the best connectivity compression rates reported to date. This coder is an extension of the vertex-based coder for triangle mesh connectivity by (Touma and Gotsman, 1998). Their scheme codes the connectivity of triangle meshes mainly as a sequence of vertex degrees. Our scheme codes the connectivity of polygon meshes by storing in addition a separate sequence of face degrees. Furthermore, we exploit the correlation between neighboring vertex and face degrees for mutual predictive compression of the two sequences. Because low-degree vertices are likely to be surrounded by high-degree faces and vice versa as illustrated in Figure 4.1, we predict vertex degrees based on neighboring face degrees and vice-versa.

While we use an adaptive traversal heuristic to improve compression by reducing the number of "split" operations, we also give a simple proof that such heuristics cannot guarantee to avoid "splits" altogether. Finally we put an end to the speculations whether the offset values that are associated with each "split" symbol are redundant or not. We show that split offsets are not redundant by giving example encodings that have two different decodings if the split offsets are not specified.

4.1 Coding with Vertex and Face Degrees

The vertex-based coder by (Touma and Gotsman, 1998) encodes the connectivity graph of a manifold triangle mesh as a sequence of vertex degrees. We describe how to extend their approach to encode the connectivity graph of a manifold *polygon* mesh using a sequence of vertex degrees *and* a separate sequence of face degrees. As for triangle meshes, an occasional split or merge symbol is needed in addition to the vertex degrees.

Encoding: Starting with a connectivity graph of v vertices and f faces, the encoder produces two symbol sequences: one is a sequence of face degrees F_d and the other is a sequence of vertex degrees V_d, split symbols S_j that have an associated offset j, and merge symbols $M_{i,k}$ that have both an associated index i and an associated offset k. If the encoding process performs s split operations and m merge operations then the first sequence contains $f - 1 - s + m$ face degrees and the second sequence contains v vertex degrees, s split symbols, and m merge symbols. The connectivity graph can be reconstructed by simultaneously working on both symbol sequences.

The coder maintains one or several loops of *boundary edges* that separate a single processed region from all unprocessed regions. Furthermore, it stores for every *boundary vertex* the number of *free degrees* or *slots*, which are unprocessed edges incident to the respective vertex. Each of these *boundaries* encloses an unprocessed region; its faces, vertices, and edges are called *unprocessed*. In the presence of handles one boundary can contain another, in which case they enclose the same unprocessed region. Each boundary has a distinguished boundary edge called the *focus*. The algorithm works on the focus of the *active boundary*, while the other boundaries are kept in a stack.

The initial active boundary is defined counterclockwise around an arbitrary edge and one of its two boundary edges is defined to be the focus. Each iteration of the algorithm processes the face adjacent to the focus of the active boundary. This involves recording its degree and processing its *free vertices* as illustrated by three example scenarios **A**, **B**, and **C** in Figure 4.2. Since including a face consumes two boundary slots, we sometimes need to widen the focus until there is a *start slot* and an *end slot* for the face. The number of *focus vertices* is called the *width* of the focus. In scenarios **A**, **B**, and **C** the focus has a width of 3, 2, and 4 respectively. The *free vertices* are those vertices of the processed face that are not part of the widened focus.

The free vertices are processed in clockwise order starting from the start slot. Three different cases can arise. In accordance with the original reference (Touma and Gotsman, 1998) we call them *add*, *split*, and *merge* (see Figure 4.2). By far the most frequent

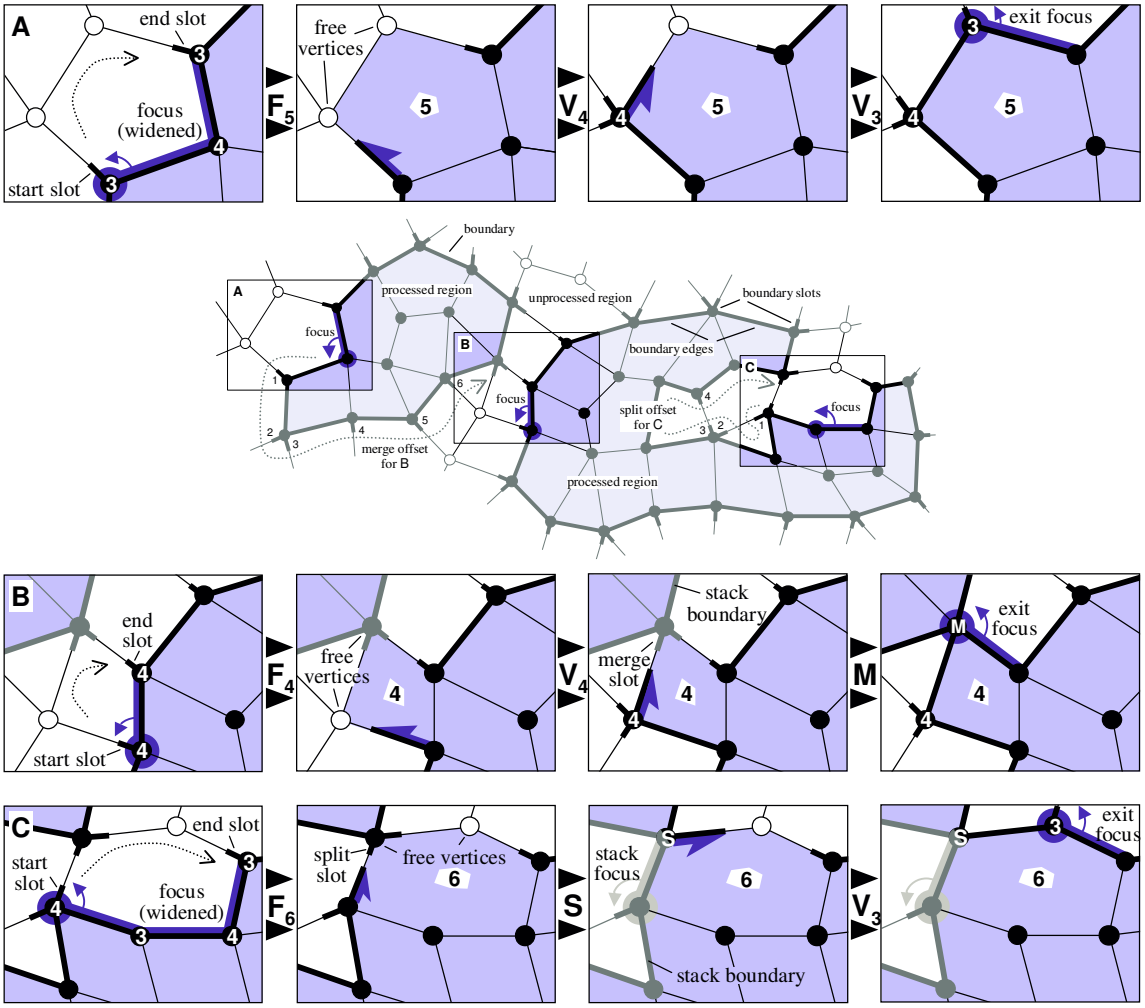


Figure 4.2: The three frame sequences **A**, **B**, and **C** illustrate different scenarios that can arise when processing a face. **A** is the most common one: The free vertices of the face have not been visited before, we *add* them to the boundary and record their degree. **B** only occurs for meshes with handles: A free vertex that has already been visited is on a boundary in the stack. The active boundary *merges* with this stack boundary. We record its stack index and the number of slots between the stack focus and the merge slot. **C** happens occasionally: A free vertex that has already been visited is on the active boundary. The active boundary *splits*. We record the number of slots between the new stack focus and the split slot.

case is *add*, which happens whenever the free vertex has not been previously visited. In this case we record the vertex degree d using the symbol V_d. When we encounter a free vertex that has already been visited we either have either a *split* or a *merge*. A merge can occur only for meshes with handles (i.e. with non-zero genus). In this case the free vertex is on a stack boundary, which causes the active boundary to *merge* with the respective stack boundary. We remove this boundary from the stack and record

the index i that the boundary had in the stack and the number of slots k between the focus of the stack boundary and the merge slot, denoted by symbol $M_{i,k}$. In the other case the free vertex is on the active boundary, which causes the active boundary to *split* into two. We push one part on the stack and record the number of slots j between the new stack focus and the split slot, denoted by symbol S_j.

After processing all free vertices, we exit the face and move to the next focus (see Section 4.4). This repeats until all faces have been processed. Notice that we do not need to record the degree of the very last face for each boundary. At this point a boundary has no slots left and wraps around this face. Therefore the number of recorded face degrees F_d equals at most the number of faces f minus one. Each split increases and each merge decreases the number of boundaries by one. Thus the exact number of face degrees recorded, given that we have s split and m merge operations, is $f - 1 - s + m$. This is discussed further in Section 4.5.

A hole in the mesh is processed like a large polygon. The encoder includes a face whose degree equals the size of the hole and processes the free vertices around the hole. Face degrees that correspond to holes in the mesh are marked so that the decoder does not mistake them for regular polygons.

Decoding: The decoder exactly replays what the encoder does by performing the boundary updates described by the two symbol sequences. A detailed example run that leads step by step through the decoding process is shown in Figure 4.3.

Complexity: We assume that the mesh genus is a small constant, so that there are only a constant number of merge operations. Each face is processed once. The cost of processing a face is proportional to its degree plus the cost for processing its free vertices. The sum of face degrees is linear in the number of vertices and each vertex is added once. This leaves us with the critical split operations that require walking the offset along the boundary. Since we know the length of the boundary we can always walk the shorter way. In the worst case the boundary consists of all v vertices and is recursively split into half, resulting in a time complexity of $O(v \log_2(v))$. However, typically there are few split operations that split the boundary in a unbalanced manner so that in practice we can expect the run-time to be linear in the number of vertices.

4.2 Compressing with Duality Prediction

The two symbol sequences are compressed into a bit-stream using a adaptive arithmetic coding with multiple contexts (Witten et al., 1987). Whenever a face is processed we

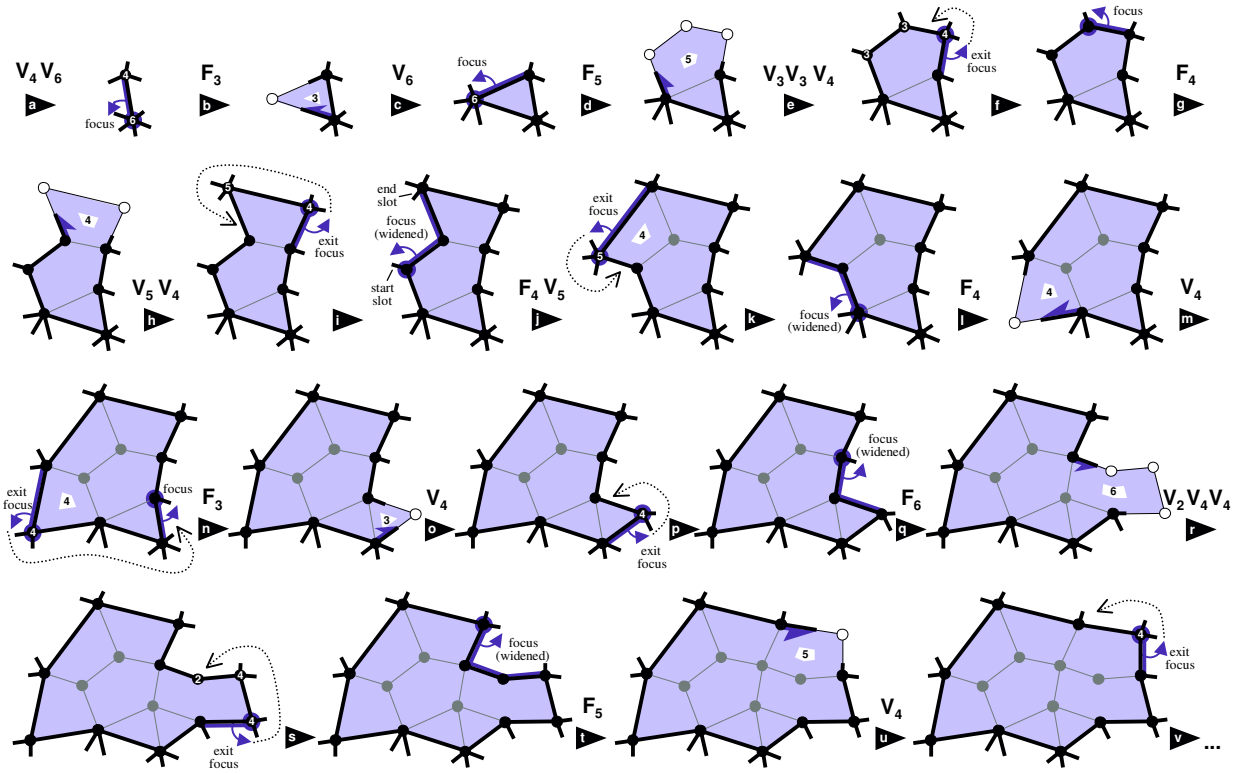


Figure 4.3: Decoding is an exact replay of encoding: **(a)** Create the initial boundary by uncompressing the first two vertex degrees. **(b)** Uncompress the first face degree. The average focus vertex degree of 5.0 determines which *face-degree context* (*fdc*) is used. **(c)** Uncompress the degree of the free vertex. The face degree of 3 determines which *vertex-degree context* (*vdc*) is used. The focus remains at the exit focus because there is no boundary vertex with 0 or 1 slots. **(d)** Uncompress the next face degree. The average focus vertex degree determining the *fdc* is 5.0. **(e)** Uncompress the degrees of the three free vertices. The face degree determining the *vdc* is 5. **(f)** The focus moves counterclockwise to the boundary vertex, which has just 1 slot. **(g)** Uncompress the next face degree. The average focus vertex degree 3.5 is the *fdc*. **(h)** Uncompress the degrees of the two free vertices. The face degree 4 is the *vdc*. **(i)** Move focus counterclockwise to the boundary vertex with 0 slots and widen it such that there is a start slot and an end slot for the next face to process. **(j)** Uncompress the next face degree ($fdc = 3.6$) and uncompress the degree of its free vertex ($vdc = 4$). **(k)** Move the focus counterclockwise to the vertex with 0 slots and widen it. **(l)** Uncompress the next face degree ($fdc = 4.6$). **(m)** Uncompress the degree of its free vertex ($vdc = 4$) and move focus. **(n)** Uncompress the next face degree ($fdc = 5.0$). **(o)** Uncompress the degree of its free vertex ($vdc = 3$). **(p)** Move and widen the focus. **(q)** Uncompress the next face degree ($fdc = 4.0$). **(r)** Uncompress the degree of its three free vertices ($vdc = 6$). **(s)** Move and widen the focus. **(t)** Uncompress the next face degree ($fdc = 3.5$). The focus has a width of 4, therefore the face degree is at least 4. Disable the entry of the chosen context that represents the *impossible* degree 3. **(u)** Uncompress the degree of its free vertex ($vdc = 5$). **(v)** And so on ...

need to specify if it represents a polygon or a hole in the mesh. Using the arithmetic coder we code this with a separate context of two symbols. Similarly, whenever a free vertex is processed we need to specify if an add, a split or a merge operation was used. We distinguish between the three possible operations using three different symbols that are also encoded with a separate arithmetic context.

What remains to be done is compressing the face degrees, the vertex degrees, and the offsets and indices associated with split and merge operations. The basic idea is to exploit the fact that high-degree faces tend to be surrounded by low-degree vertices, and vice versa, for predictive compression. For every vertex we know the degree of the face that introduces it. For every face we know the degrees of all vertices of the (widened) focus. We found that using four different prediction contexts each way is sufficient to capture the correlation in the duality of vertex and face degrees. Offsets and indices, on the other hand, are compressed with the minimal number of bits needed based on their known maximal range.

4.2.1 Compressing Face Degrees

When a face is processed the degrees of all vertices on the (widened) focus are known. The lower their average degree, the more likely this face has a high degree and vice-versa (see Figure 4.1). This can be exploited by using different contexts for entropy coding the face degrees, dependent on this vertex degree average. In practice the use of four such *face-degree contexts* seems to capture this correlation quite well. We have different contexts for an average vertex degree (a) below 3.3, (b) between 3.3 and 4.3, (c) between 4.3 and 4.9, and (d) above 4.9. These numbers were first chosen based on the plot in Figure 4.1 and then corrected slightly based on experimental results.

Each of the four face-degree contexts contains 4 entries: The first three entries represent face degrees 3, 4, and 5 and the last entry represents higher degree faces. These are subsequently compressed with a special *large-face-degree context*. This special context is also used for faces that correspond to holes in the mesh. All contexts are initialized with uniform probabilities that are adaptively updated. Four bits at the beginning of the code specify face degrees that do not occur in the mesh. Their representing entries are disabled in all contexts. For our set of example meshes, predictive coding of face degrees improves the bit-rates on average by 12.2 %.

There is another small improvement possible: The minimal degree of the face equals the width of the focus. If the focus is wider than 3 we can improve compression further by disabling those entries of the chosen arithmetic context that represents *impossible*

degrees. Although this improves the compression rates by only 1 or 2 percent, it was rather simple to integrate into the arithmetic coding process.

4.2.2 Compressing Vertex Degrees

When a free vertex is processed, the degree of the respective face is known. The lower its degree, the more likely this vertex has a high degree and vice-versa. Again we exploit this for better compression by using four different contexts. We switch the *vertex-degree context* depending on whether the face is a triangle, a quadrangle, a pentagon, or a higher degree face.

Each of the four vertex-degree contexts contains 9 entries: The first eight entries represent vertex degrees 2 to 9 and the last entry represents higher degree vertices. These are subsequently compressed with a special *large-vertex-degree context*. All contexts are initialized with uniform probabilities that are adaptively updated. Nine bits at the beginning of the code specify vertex degrees that do not occur in the mesh. Their representing entry is disabled in all contexts.

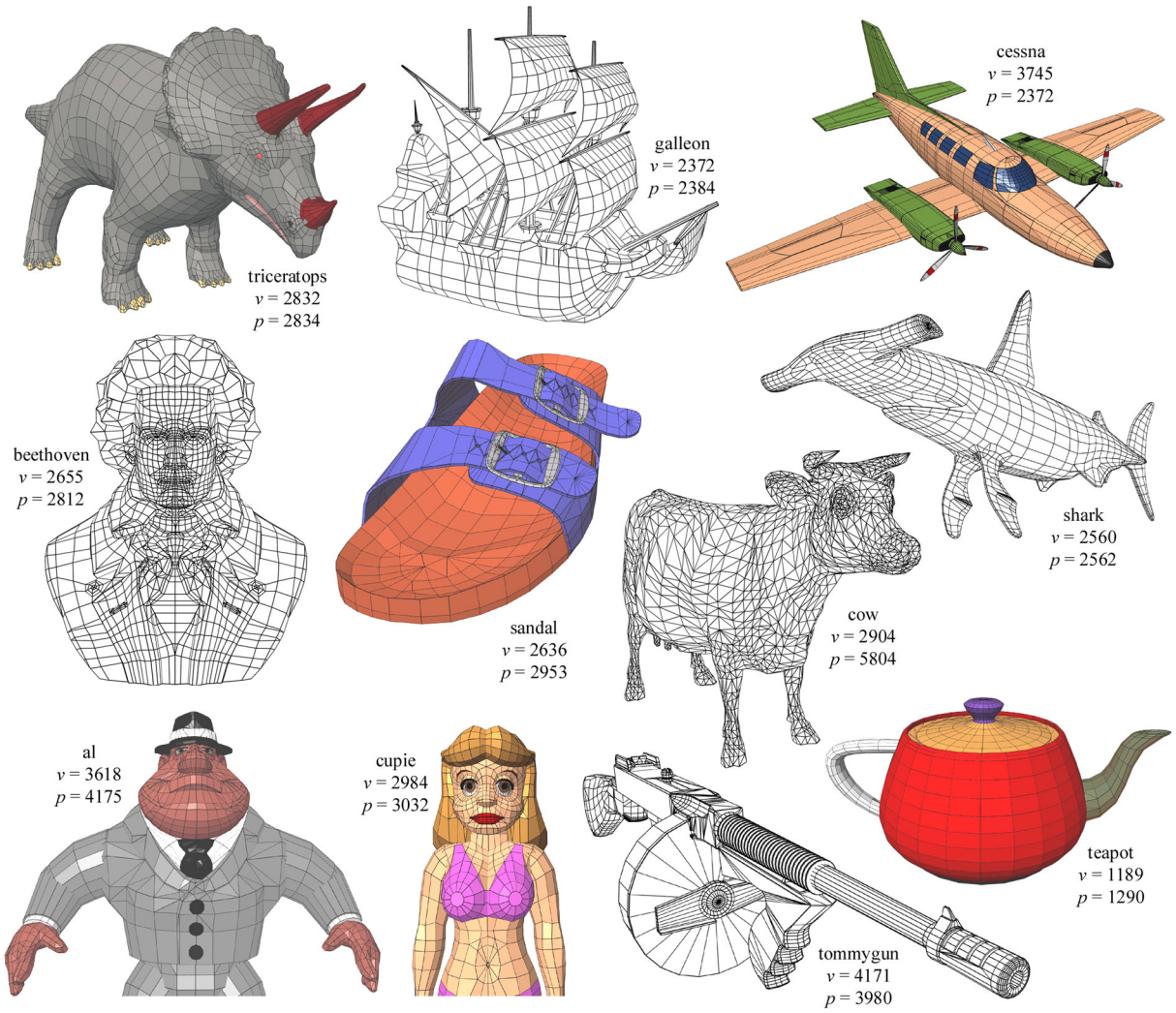
For our set of example meshes, predictive coding of vertex degrees improves the bit-rates on average by 6.4 %. Predictive coding of vertex degrees does not improve the compression rates as much as predictive coding of face degrees, because we use less information for each prediction. While each face degrees is predicted by an average of two or more vertex degrees, each vertex degree is only predicted by a single face degree.

4.2.3 Compressing Offsets and Indices

An integer number that is known to be between 0 and n can be encoded with exactly $\log_2(n+1)$ bits. We use this for compressing the offsets and indices associated with the split and the merge operation. Whenever a split offset j, a merge index i, or a merge offset k is encoded or decoded, the maximal possible value of this number is known. For the split offset j it equals the number of slots on the active boundary, for the merge index i it equals the size of the stack, and for the merge offset k it equals the number of slots on the indexed boundary in the stack.

4.3 Coding Non-Manifold Meshes

Compared to (Guéziec et al., 1999) our coder implements a much simpler stitching scheme to recover non-manifold connectivity that allows a robust, minimal-effort implementation at the expense of less efficiency. However, the number of non-manifold



mesh name	vertex degree distribution									face degree distribution					part, hole & handles		
	2	3	4	5	6	7	8	9	>9	3	4	5	6	>6			
triceratops	–	8	2816	8	–	–	–	–	–	346	2266	140	63	19	1	–	–
galleon	7	430	1595	270	66	4	1	–	–	336	1947	40	18	43	12	–	–
cessna	8	642	2470	384	178	41	18	1	3	900	2797	180	27	23	11	–	–
beethoven	21	279	1925	295	99	20	14	–	2	680	2078	44	4	6	8	10	–
sandal	–	280	1857	329	95	18	7	12	38	961	1985	7	–	–	9	14	12
shark	–	–	2560	–	–	–	–	–	–	188	2253	83	29	9	1	–	–
al	2	538	1999	720	268	69	15	1	6	1579	2505	44	11	36	21	–	–
cupie	16	272	2405	234	37	12	8	–	–	384	2506	114	10	18	6	–	–
tommygun	–	1557	2002	395	152	21	18	8	18	992	2785	84	21	98	39	–	6
cow	–	7	87	514	1796	364	98	23	15	5804	–	–	–	–	1	–	–
teapot	2	14	1022	125	18	5	1	–	2	215	1070	3	1	1	1	–	1

Table 4.1: The vertex count v and the polygon count p, the vertex and face degree distribution, and the number of holes and handles of the models used in our experiments.

vertices is typically small, which justifies the use of a simpler scheme. Whenever a free vertex is processed by an add operation we simply specify if this indeed is a new position or not using arithmetic coding. If it is a new position we increment the position counter. Otherwise it is an old position and its index needs to be compressed as well. We can do this with $\log_2(n)$ bits where n is the number of positions already encoded/decoded. A more efficient variation of this scheme that is geared towards large meshes with many non-manifold vertices is described in Section 7.4.

4.4 Reducing the Number of Splits

After processing a face, we could continue with the exit focus as the next focus. This is the strategy of the original coder for triangle meshes as proposed by (Touma and Gotsman, 1998). However, (Alliez and Desbrun, 2001b) propose a more sophisticated strategy for picking the next focus that can significantly reduce the number of splits. This is beneficial, because split operations are expensive to code: On one hand we need to specify where in the sequence of vertex degrees they occur and on the other hand we need to record their associated split offset. Since the decoding process has to follow this strategy, the quest for this better focus can only use information that is available to the decoder. Therefore (Alliez and Desbrun, 2001b) suggest moving the focus to the boundary vertex with the lowest number of slots. When there is more than one such vertex, they choose the least dense region by averaging over a wider and wider neighborhood. This strategy makes keeping track of the next candidate an expensive operation. Using a dedicated priority queue, for example, would require $O(\log(b))$ per boundary update, where b is the number of vertices on the active boundary.

Nevertheless, to reduce the number of splits is especially important in polygonal meshes, because here a split operation can pinch off parts of the boundary that do not enclose unprocessed vertices and that can be as small as a single unprocessed face. This does not happen in the pure triangular case where triangles that share two edges with the compression boundary (e.g. that "fill" a zero slot) are immediately included. We suggest a heuristic for picking the next focus that is similar to the one by (Alliez and Desbrun, 2001b) but does not affect the asymptotic complexity of the decoder.

The focus is moved to the boundary vertex with the smallest number of slots in counterclockwise direction as seen from the current focus. This current focus is usually the exit focus of the face processed last or the stack focus if a new boundary was just popped of the stack. However, we only move the focus if the smallest number of slots is 0 or 1, otherwise the focus remains where it is. Table 4.2 reports the success of

mesh	current		0 or 1 slot		coding	0 slot only		coding
name	# splits	bpv	# splits	bpv	gain	# splits	bpv	gain
triceratops	53	1.311	**25**	**1.189**	9.3 %	47	1.262	3.7 %
galleon	78	2.309	**18**	**2.093**	9.4 %	22	2.122	8.1 %
cessna	172	2.882	**28**	**2.543**	11.8 %	58	2.637	8.5 %
beethoven	99	2.431	**15**	**2.102**	13.5 %	27	2.155	11.4 %
sandal	85	2.295	**25**	**2.115**	7.8 %	33	2.173	5.3 %
shark	24	0.818	**13**	**0.756**	7.6 %	12	0.759	7.2 %
al	92	2.616	**14**	**2.429**	7.1 %	15	2.418	7.6 %
cupie	56	1.786	**15**	**1.640**	8.2 %	15	1.637	8.3 %
tommygun	131	2.449	**32**	**2.258**	7.8 %	37	2.251	8.1 %
cow	154	2.313	**13**	**1.781**	23.0 %	19	1.811	21.7 %
teapot	10	1.167	**3**	**1.127**	3.4 %	3	1.102	5.6 %
average					9.9 %			8.7 %

Table 4.2: The number of splits and the resulting bit-rate using the *current* focus compared to an adaptive strategy that moves the focus either to the next counterclockwise *0 or 1 slot* or to the next *0 slot only* and the coding improvement in percent.

this strategy in reducing the number of splits and the bit-rate. Starting a brute-force search along the boundary for the vertex with the smallest number of slots would mean a worst-case time complexity of $\mathrm{O}(n^2)$. Instead we keep track of the next 0 and the next 1 slot by organizing them into two cyclic linked lists. In each list we always point to the slot that is closest in counterclockwise direction and perform the necessary updates as the boundary changes. This data structure can be maintained without affecting the asymptotic complexity of the decoder.

4.5 Counts and Invariants

For a manifold mesh without boundary, the sum of all v vertex degrees and the sum of all f face degrees both equal twice the number of edges e. That means the two sums are equal.

$$\sum_{k=1}^{f} deg(\mathrm{f}_k) = \sum_{k=1}^{v} deg(\mathrm{v}_k) = 2e \tag{4.1}$$

If we know all vertex degrees and all face degrees but one we can compute it as the one completing the equality.

$$\mathrm{f}_f = \sum_{k=1}^{v} deg(\mathrm{v}_k) - \sum_{k=1}^{f-1} deg(\mathrm{f}_k) \tag{4.2}$$

mesh name	versus polygon ff	dd	coding gain	versus triangle tg	dd	coding gain
triceratops	2.115	**1.189**	43.8 %	2.167	**1.189**	45.1 %
galleon	2.595	**2.093**	19.3 %	2.088	**2.093**	-0.5 %
cessna	2.841	**2.543**	10.5 %	2.489	**2.543**	-2.2 %
beethoven	2.890	**2.102**	27.3 %	2.389	**2.102**	12.0 %
sandal	2.602	**2.115**	18.7 %	2.121	**2.115**	0.3 %
shark	1.670	**0.756**	54.7 %	1.513	**0.756**	50.0 %
al	2.926	**2.429**	17.0 %	2.105	**2.429**	-15.4 %
cupie	2.307	**1.640**	28.9 %	2.102	**1.640**	21.9 %
tommygun	2.611	**2.258**	13.5 %	2.066	**2.258**	-9.4 %
cow	2.213	**1.781**	19.5 %	1.879	**1.781**	5.1 %
teapot	1.669	**1.127**	32.5 %	1.063	**1.127**	-6.3 %
average			26.0 %			9.1 %

Table 4.3: The compression rates of the proposed Degree Duality coder (dd) in comparison to those of another polygon mesh coder, Face Fixer (ff), and those of a triangle mesh coder, the TG coder (tg), for the example models shown in Table 4.1. Reported are the achieved bit-rates in bits per vertex and the corresponding coding gain in percent. The TG coder compresses a triangulated version of these models and its bit-rates do not include the extra information necessary to reconstruct the polygons.

Furthermore we have the following invariants: The sum of degrees of all unprocessed faces minus the number of all boundary edges b equals twice the number of unprocessed edges u. And also the sum of degrees of all unprocessed vertices plus the number of all boundary slots s equals twice the number of unprocessed edges u.

$$\sum_{\mathrm{f}_k \subset \mathcal{B}} deg(\mathrm{f}_k) - b = \sum_{\mathrm{v}_k \subset \mathcal{B}} deg(\mathrm{v}_k) + s = 2u \tag{4.3}$$

where $\subset \mathcal{B}$ means unprocessed (or enclosed by some boundary). Furthermore, this invariant is true for the face and vertex degree count of every unprocessed region together with the edge and slot count of the respective boundaries that enclose it. In the moment a split occurs, one such equation $\mathcal{E}$ is split into two new equations $\mathcal{E}'$ and $\mathcal{E}''$ that are related with $s = s' + s''$, $b = b' + b''$, $u = u' + u''$, and the sums of unprocessed face degrees and vertex degrees are split correspondingly. The offset associated with the split operation specifies s'' and b''. Together with the two degree sequences they specify implicitly when each boundary ends. This explains why we can omit one face degree for every split operation—each split creates a new equation just like (4.2) that can be solved for a single face degree.

4.6 Results

On the set of example models shown in Table 4.1 the presented coder delivers compression rates that improve between 10 % to 55 % over those of the Face Fixer coder with an average improvement of 26 % as documented by the bit-rates reported in Table 4.3. Often the original polygonal connectivity is cheaper to encode than a triangulated version of it as evidenced by the bit-rate comparisons with the TG coder, in particular for connectivities that have strong regularity in both degree sequences, such as the "triceratops" or the "shark" model.

But degree coding does not always outperform other coders. We can construct pathological examples where other coders perform better. Using the cow model from Table 4.1 we generated a triangle mesh and a quadrangle mesh to demonstrate this. We generated the triangle mesh by placing a new vertex into every triangle of the original mesh and by connecting it to its three vertices. All new vertices have degree three, while the degree of every vertex of the original mesh doubles. This connectivity compresses to 0.988 bpv using my implementation of Edgebreaker (Rossignac, 1999) with adaptive order-3 arithmetic compression of the labels, whereas the Degree Duality coder needs 1.569 bpv. Similarly, we generated the quadrangle mesh by placing a new vertex into every original triangle and by connecting it to three new vertices that are placed on every original edge. All new vertices have either degree three or degree four, while the degree of the original vertices remains unchanged. This connectivity compresses to 1.376 bpv using Face Fixer (Isenburg and Snoeyink, 2000), whereas the Degree Duality coder needs 1.721 bpv. However, such pathological cases rarely occur in practice.

4.7 Splits and Split Offsets

There have been attempts to establish a guaranteed bound on the coding costs of a degree coder. However, the infrequently occurring "split" symbols make this a difficult task. For triangle meshes, the adaptive traversal heuristic of (Alliez and Desbrun, 2001b) significantly lowered the number of split operations and the remaining number of "splits" seemed negligible small. Therefore the authors restricted their worst case analysis to the vertex degrees. Surprisingly, the maximal entropy of a distribution of n vertex degrees whose sum fulfills Euler's relation for planar triangulations coincides with the information theoretic minimum of 3.24 bits per vertex that is due to Tutte's enumeration work (Tutte, 1962). Alliez and Desbrun's work has been extended by

(Khodakovsky et al., 2002) to show that the summed entropies of the face degree and the vertex degree sequences also converge to the corresponding bound for planar graphs (Tutte, 1963). They claimed that this would indicate that degree coding is in some sense optimal. However, all this is based on the assumption that the split operations can be neglected.

4.7.1 Splits can in general not be avoided

The obvious question is whether it is possible to avoid split operations altogether. If we can find some traversal heuristic that can guarantee that no split operations occur, then degree coding would indeed be optimal. However, we can easily prove that splits cannot be avoided with a strategy that only uses the already encoded/decoded part of the mesh. Given any such strategy we can always construct a mesh that is guaranteed to result in a split. This proof uses the fact that the connectivity of a non-zero genus mesh will have at least as many splits as the mesh has handles.

Imagine your favorite mesh of torus topology. The encoder eventually has to use the merge operation to code the handle. Every merge operation is preceded by a split operation. In the moment this split operation is performed, we pause the encoding process, perform an edge cut in the unprocessed region that opens the handle, insert two large polygons or holes into the cut, and continue the encoding process on the mesh (which now has sphere topology). The coder of course did not notice what happened, because the edge cut was performed in the region it has not yet seen. But now the coder has produced a split for a mesh of genus zero.

The occasional occurrence of split operations does not disprove the optimality claim of (Alliez and Desbrun, 2001b; Khodakovsky et al., 2002) since their number could be constant. However, (Gotsman, 2003) has shown that the entropy analysis for triangular connectivities of Alliez and Desbrun is slightly off, because it includes many vertex degree distributions that do not correspond to actual triangulations. He incorporates additional constraints on the distribution that lower the worst-case entropy of the vertex degree distribution below Tutte's bound. This means that there are fewer valid permutations of vertex degrees than triangulations and that additional information is necessary to distinguish between them. So the split information does contribute a small but necessary fraction to the encoding and is therefore not negligible. Obviously Gotsman's findings also prove that split operations can in general not be avoided.

4.7.2 Split offsets are in general not redundant

Since splits are a necessary part of the encoding, calculations for a guaranteed bound on the coding costs of a degree coder must take them into account. This is difficult, not only because their number is unpredictable but also because of their associated offset values. There has been speculation that it might be possible to modify the TG coder to operate without explicitly storing the offset values that are associated with each "split" symbol. Such speculations are motivated by the fact that Edgebreaker (Rossignac, 1999) manages to avoid storing such offsets, whereas the otherwise almost identical Cut-border Machine (Gumhold and Strasser, 1998) explicitly includes them. Similarly, Face-Fixer (Isenburg and Snoeyink, 2000) avoids storing offsets, whereas the Dual-Graph Method (Li and Kuo, 1998) includes them.

In an information theoretic sense, the split offsets used by the Cut-border Machine and the Dual-Graph Method are redundant because they are implied in their symbol sequence. Edgebreaker and Face Fixer recreate these offsets by decoding either in two passes (Rossignac, 1999; Rossignac and Szymczak, 1999) or in reverse (Isenburg and Snoeyink, 2000; Isenburg and Snoeyink, 2001b). In both schemes, each "split" symbol S has a corresponding "end" symbol E that signals the completion of a compression boundary. For a traversal that always completes one boundary part before continuing on the "split off" parts, the symbols S and E form nested pairs, each of which encloses a subsequence of symbols. In Edgebreaker and Face-Fixer, such a symbol subsequence is a self-contained encoding of the portion of the mesh enclosed by that boundary part. Obviously, this subsequence also determines the length of that boundary part, which is exactly what is specified by the split offset.

It was less clear whether the split offsets used by the TG coder would also be redundant. For one thing, the symbol sequences produced by the TG coder do not contain explicit "end" symbols because the completion of a compression boundary is automatically detected. Therefore it is not easy to identify the subsequences of symbols that complete a particular boundary part. But simply adding "end" symbols does not make things much easier, because the split offsets of the TG coder cannot be derived from a subsequence of symbols alone.

Unlike the symbol subsequences of Edgebreaker and Face Fixer, the symbol subsequences of the TG coder are not self-contained encodings of some portion of a mesh. Decoding also depends on the state of the compression boundary at the beginning of the subsequence. The TG coder stores significantly more state information on the

compression boundary than Edgebreaker or Face Fixer. It maintains *slot counts* that specify how many unprocessed edges are still incident to each boundary vertex. The split offsets also specify how to split the slot counts, not just the boundary vertices. The value of each count and their order around the boundary depends on all preceding symbols. Therefore is it impossible to derive the split offsets with computations restricted to subsequences of symbols or by processing the symbols in reverse.

For simple triangulations, we have implemented a decoding scheme that "tries out" all possible split offsets in a brute-force manner, backtracking as soon as it notices that it cannot complete a triangulation. This is slow due to the exponentially increasing search space and therefore impractical as a decoding algorithm. But this approach does find the counter-examples shown in Figure 4.4 that establish the non-redundancy of the offsets—namely, vertex degree sequences with "split" and "end" symbols that lead to more than one valid decoding if different split offsets are used.

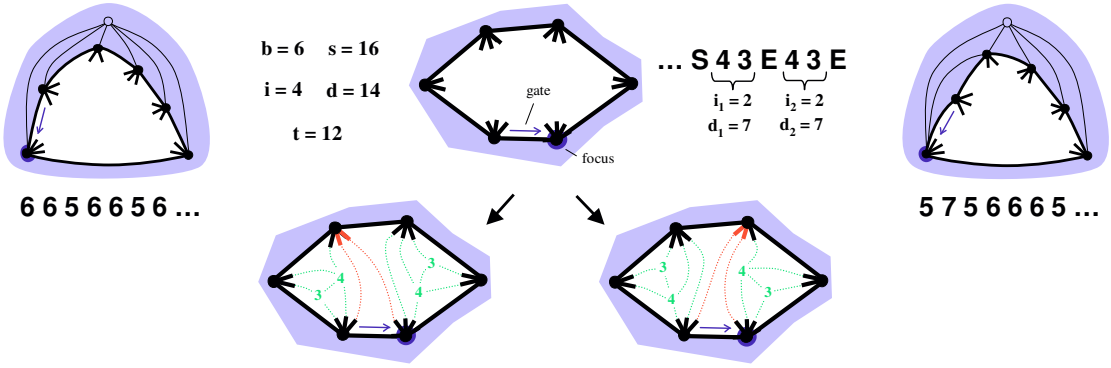
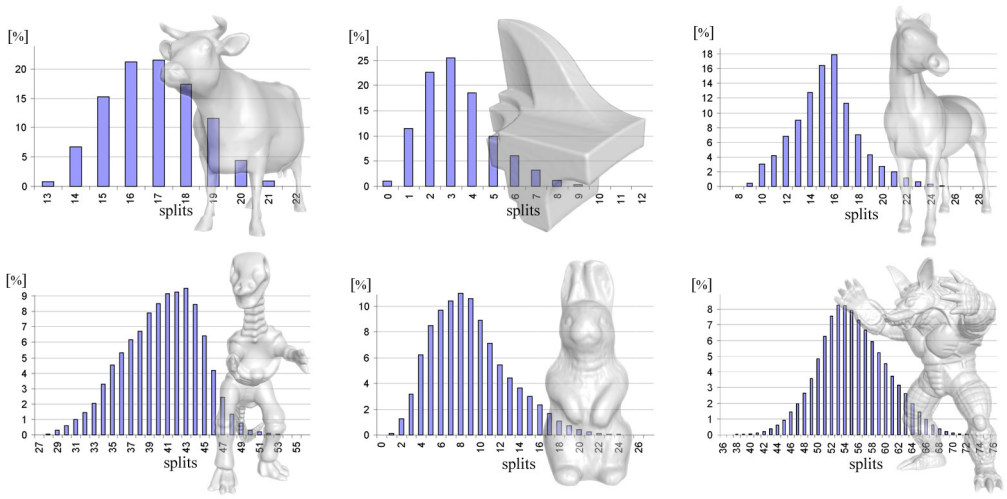


Figure 4.4: The smallest scenario where degree coding with "split" and "end" symbols but without offsets is not unique. This first occurs in triangulations with 11 vertices.

In practice, our implementation finds that only few sets of split offsets actually lead to valid decodings and that many offset-less encodings correspond to only one unique triangulation. For example, of the 290, 898 possible encodings of the well-known "horse" model (see Table 4.4), 290, 889 are unique when both "split" and "end" symbols are stored, and the remaining 9 have only two valid decodings each. For those we could replace the entire set of split offsets (on average 15 in case of the "horse" model) with a single bit that specifies which one of the two decodings it is. Furthermore there are many small triangle meshes—among them the popular "cow" and "fandisk" models—for which split offsets are redundant when both "split" and "end" symbols are stored.

Unfortunately we do not have an efficient algorithm for finding all possible valid decodings. We currently achieve this by exhaustive search through all potential splits



meshes			splits			non-unique
name	vertices	encodings	min	max	avg	encodings
cow	2,904	17,412	13	22	16.8	0
fandisk	6,475	38,838	0	12	3.3	0
horse	48,485	290,898	7	29	15.4	9
dinosaur	56,194	337,152	27	56	40.4	10
rabbit	67,039	402,222	0	27	9.0	56
armadillo	172,974	1,037,832	36	76	55.2	146

Table 4.4: The table lists for each mesh the number of vertices and the number of different encodings. The illustrations show which percentage of encodings has what number of splits. The minimum, the maximum, and the average number of splits are given in the table. Most importantly, we report the number of encodings that are not unique because they have valid decodings for two different sets of split offsets

and the complexity of this search increases quickly. Although we can significantly prune the search tree and even solve even relatively large triangulations in reasonable time, the high search costs mean that offset-less degree encodings are mainly of theoretical interest and do not lead to new practical compression algorithms. Further details of this work will be published in a forth-coming paper (Isenburg and Snoeyink, 2005b).

We must mention that there are incentives to include explicit split offsets in the encoding. Split offsets allow decoding in a single forward pass over the symbol sequence, which makes it possible to decompress in a streaming fashion (see Chapter 7). They give the freedom to choose a mesh traversal that is not recursive. The Cut-Border Machine and the TG coder can easily be modified to operate in a breadth-first manner, which leads to more coherent mesh layouts (see Chapter 9). Finally, explicit split offsets can result in better overall compression rates because they allow incorporating heuristics for predictive compression of the vertex degrees (Isenburg and Snoeyink, 2005a).

4.8 Summary

The main contribution of this chapter is the extension of degree coding to polygonal connectivity using a sequence of vertex degrees and a sequence of face degrees that are compressed separately. Also of importance is the observation that the correlation in the duality of the degrees can be used for mutual predictive compression. The coder that we have described here delivers the best connectivity compression rates reported for polygon mesh connectivity so far. We should mention that a similar coder was developed independently and during the same time period by a group of researchers at CalTech and USC (Khodakovsky et al., 2002). Furthermore, we proved that there is no traversal heuristic that can guarantee to avoid split operations and we also disproved the long suspected redundancy of split-offsets.

4.9 Hindsights

The use of an adaptive strategy that "jumps" around on the compression boundary in the attempt to reduce the number of splits typically lowers the compression rates, but it also tends to create incoherent triangle orderings. Sending the triangles in this order to the graphics hardware will have a negative impact on the success of the transparent vertex caching schemes employed on modern cards. Depending on the distance of these "jumps" this may also lead to incoherent memory accesses to the vertex array. We advice against moving the focus to the next 1 slot or the use of the strategy by (Alliez and Desbrun, 2001b). Instead we suggest a strategy that only moves the focus to a 0 slot and in case there are multiple such 0 slots to choose the "oldest" 0 slot that was least recently created. Pathological cases excluded, this will only require a tiny "hop" along the boundary, while already leading to good improvements in compression as documented in Table 4.2. For reasons of coherence we strongly recommend a breadth-first approach for advancing the focus in the absence of 0 slots rather than the depth-first approach described here. The problems with incoherent mesh layouts are discussed further in Chapter 9.

In retrospect we also advocate a different strategy for dealing with holes in the mesh. Our original approach treats a hole just like large polygon and simply marks it as a hole. However, that means that all the vertices around the hole need to be processed. In Section 5.1 we use the order in which the connectivity coder processes the vertices to compress their vertices with predictive coding. Especially for meshes

with large holes this results in poor vertex predictions around the hole. Therefore we suggest marking boundary edges that are adjacent to a hole and stopping the growth of the boundary there, as in (Gumhold and Strasser, 1998).

Chapter 5

Coding Geometry and Properties

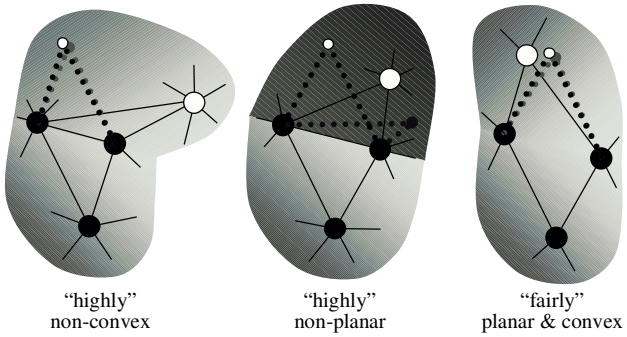


Figure 5.1: Two adjacent triangles are more likely to be in a non-convex or non-planar configuration than a polygonal face, which tends to be "fairly" planar and convex.

In the previous chapters we have looked at different ways to encode the connectivity of polygon meshes. In this chapter we look into the compression of vertex coordinates and the compression of mesh properties. For compressing vertex coordinates we describe a simple improvement to the prediction scheme by (Touma and Gotsman, 1998) for meshes that are not fully triangulated. We let the polygon information dictate where to apply the parallelogram rule that they use to predict vertex positions. Since polygons tend to be fairly planar and fairly convex, it is beneficial to make predictions "within" a single polygon rather than "across" two polygons. This, for example, avoids poor predictions due to a crease angle between polygons.

For compressing properties, such as shading normals, colors, and texture coordinates that can be associated with the vertices, faces or corners of the mesh, there really are two distinct kinds of information to compress. One describes how the properties are attached to the mesh—the property mapping, for which we introduce a novel predictive compression scheme. The other specifies each individual property—the property values, for which we report a compression technique that takes mapping discontinuities into account at the example of texture coordinates.

5.1 Compressing Vertex Positions

Recently we have seen a number of innovative approaches for compressing mesh geometry. There are spectral methods (Karni and Gotsman, 2000) that perform a global frequency decomposition of the surface, there are space-dividing methods (Devillers and Gandoin, 2002) that compress vertex positions without the aid of the connectivity, there are remeshing methods (Khodakovsky et al., 2000; Szymczak et al., 2002) that compress a regularly re-sampled version instead of the original geometry, there are angle-based methods (Lee et al., 2002) that represent geometry through the dihedral and internal angles of the mesh triangles, there are model-space methods (Lee and Ko, 2000) that compress prediction errors in a local coordinate frame using vector quantization, there are feature-based methods (Shikhare et al., 2001) that find and instantiate repeated geometric features in a model, and there are high-pass methods (Sorkine et al., 2003) that quantize coordinates after a basis transformation with the Laplacian matrix. We do not attempt to improve on these—rather complex—schemes. Instead we generalize the simple and popular triangle mesh geometry predictor by (Touma and Gotsman, 1998) to achieve better compression performance on polygonal meshes.

Predictive geometry compression schemes work as follows: First the floating-point positions are uniformly quantized using a user-defined precision of for example 8, 10, 12, or 16 bits per coordinate. This introduces a quantization error as some of the floating-point precision is lost. Then a prediction rule is applied that uses previously processed positions to compute an estimation for the next position. Only a corrective vector is stored that describes the difference between the predicted and the actual position. The values of these corrective vectors tend to spread around zero. This reduces the variation and thereby the entropy of the sequence of numbers, which means they can be efficiently compressed with, for example, an arithmetic coder.

The first prediction method for geometry compression was suggested by (Deering, 1995). It simply predicts the next position as the last position. While this technique, which is also known as delta coding, makes systematic prediction errors, it can easily be implemented in hardware. A more sophisticated scheme is the spanning tree predictor by (Taubin and Rossignac, 1998). A weighted linear combination of two, three, or more parent vertices in a vertex spanning tree is used for prediction. The weights used in this computation can be optimized for a particular mesh but need then to be stored as well. While such an optimization can be expensive, it is only performed by the encoder and not by the decoder. However, by far the most popular scheme is the parallelogram pre-

dictor by Touma and Gotsman. A position is predicted to complete the parallelogram that is spanned by three previously processed vertices of a neighboring triangle. This predictor gives the best overall trade-off between computational efficiency and achieved compression and has remained the accepted benchmark that recent approaches compare themselves with. Better compression rates have been reported, but it is often questionable whether these gains are justified given the sometimes immense increase in algorithmic and asymptotic complexity of the coding schemes.

Good predictions are close to the actual position of a vertex. In the triangle mesh case the parallelogram rule gives good predictions if used across pairs of triangles that are fairly planar and convex. It gives bad predictions if used across triangles that are highly non-planar and/or non-convex (see Figure 5.1 on page 65). Two approaches were proposed to increase the number of good parallelogram predictions. Instead of processing the vertices in the order encountered by the connectivity coder, (Kronrod and Gotsman, 2002) first locate good triangle pairs for parallelogram prediction and then try to use a maximal number of them. For this, they construct a *prediction tree* that directs the traversal to good predictions. Since these directives have to be encoded too, they devise a scheme that traverses the prediction tree while simultaneously encoding the mesh connectivity. Especially on meshes with many sharp features, such as CAD models, they achieve significant improvements in geometry compression. However, this scheme is considerably more complex than the original method.

Instead of using a single parallelogram prediction, (Cohen-Or et al., 2002) propose to average over multiple predictions. They define the *prediction degree* of a vertex to be the number of triangles that can be used to predict its position with the parallelogram rule. For typical meshes the average prediction degree of a vertex is two. In order to have as many multi-way predictions as possible, their geometry coder traverses the mesh vertices using a simple heuristic that always tries to pick a vertex with a prediction degree of two or higher. This approach slightly improves the compression rates, but at the same time increases the complexity of the encoding and the decoding algorithms, since connectivity and geometry need to be processed in two separate passes. In contrast, our generalization of the TG coder to the polygonal case only requires an extra *if ... else ...* statement and a second set of arithmetic probability tables.

5.1.1 Predicting within Polygons

The first vertex position of each mesh component has no obvious predictor. We simply predict it as the center of the bounding box. There will be only one such *center*

prediction per mesh component. The second and the third vertex positions cannot yet be predicted with the parallelogram rule since at least three vertices are needed for this. We predict them as a previously decoded position to which they are connected by an edge. This is simple delta coding and makes a systematic prediction error, but there will be only two such *last* predictions per mesh component. All following vertex positions use the parallelogram predictor. We distinguish two cases: a *within* and an *across* prediction that are illustrated in Figure 5.2.

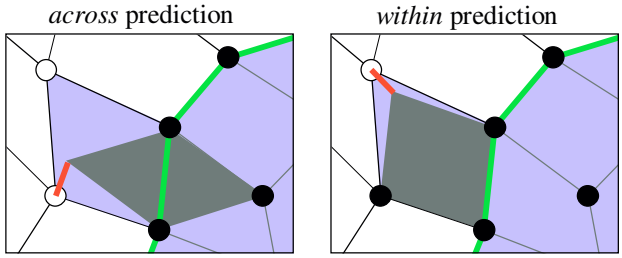


Figure 5.2: The parallelogram used for prediction is shown in dark grey and the corrective vectors are shown in red. Imagine the green shaded edges symbolize a sharp crease in the model: then *across* predictions will perform a lot worse than *within* predictions.

Polygonal faces tend to be fairly planar and convex. Although they might not be perfectly planar, major discontinuities are improbable to occur across them—otherwise they would likely have been triangulated when the model was designed. Furthermore, a quadrilateral, for example, is usually convex while two adjacent triangles easily form a non-convex shape. Therefore predicting *within* a polygon is preferred over predicting *across* polygons. At least three vertices of a (non-triangular) polygon must already be known before a *within* prediction is possible.

A simple greedy strategy for maximizing the number of *within* predictions grows the already processed mesh region by continuing (a) with a polygon that shares three or more vertices with the processed region or (b) with a polygon that creates (a) for the next iteration. Coincidentally, the traversal order of the connectivity coder that was described in Chapter 4 qualifies as such a strategy. In the attempt to improve connectivity compression rates it traverses the mesh with a heuristic that aims at reducing the number of *split* operations, which are expensive to encode. Whenever possible, this strategy continues with a polygon that completes a boundary vertex, which gives us (a). Otherwise, it continues with a polygon that brings a boundary vertex closer to completion, which gives us (b). The results in Table 5.1 illustrate the success of this strategy: on average 84 % of the vertices can be *within*-predicted.

mesh name	predicted total	within	across	last	center	% of within	bpv within	bpv other
triceratops	2832	2557	272	2	1	90	14.1	20.5
galleon	2372	2007	329	24	12	85	16.9	26.8
cessna	3745	3091	621	22	11	83	11.0	19.8
beethoven	2655	2305	326	16	8	87	21.0	24.2
sandal	2636	2084	525	18	9	79	14.1	22.8
shark	2560	2348	209	2	1	92	9.8	18.7
al	3618	2672	883	42	21	74	18.6	23.6
cupie	2984	2623	343	12	6	88	17.0	21.5
tommygun	4171	3376	678	78	39	81	10.9	19.5
cow	2904	0	2901	2	1	0	–	20.6
cow_poly	2904	2391	510	2	1	82	18.0	21.6
teapot	1189	1016	170	2	1	85	14.9	22.7
average						84	15.1	22.0

Table 5.1: This table reports how many vertices are predicted each way and the percentage of *within* prediction. The compression rates in bits per vertex (bpv) for *within*-predicted versus otherwise predicted vertices are given for a precision of 12 bits.

5.1.2 Compressing Corrective Vectors

The parallelogram prediction produces a sequence of correctors that has less variation than the sequence of positions. Specifically, the corrective vectors are expected to spread around the zero vector. The correctors produced by *within* predictions tend to be smaller than those produced by *across*, *last*, and *center* predictions. This implies that the entropy of the *within* correctors will be lower than that of the others. For entropy coding it is beneficial not to spoil the lower entropy of the *within* correctors with the higher entropy of the other correctors. Therefore we use two different arithmetic contexts depending on whether a corrector is the result of a *within* prediction or not. The results in Table 5.1 confirm the benefit of this approach: the correctors of *within* predictions compress on average 30 percent better than the others.

For quantization with k bits of precision we map the position coordinates to a number between 0 and $2^k - 1$. Instead of using corrective values ranging from $-2^k - 1$ to $+2^k - 1$ we use correctors that express the shortest distance between the predicted and the actual position modulo 2^k, which essentially folds the correctors into a range between $-2^{k-1} - 1$ and $+2^{k-1}$. These values are expected to be spread around zero without preference for either sign. Hence, the high-order bits of their absolute value are more likely to be zero than the low-order bits. The highest order bit, for example, will only be set if the prediction error is half the extend of the bounding box or more.

It benefits from almost all predictions. The lowest order bit, on the other hand, can be set whenever there is a prediction error.

For memory-efficient arithmetic compression we break the sequence of corrector bits into smaller sequences. This prevents the probability tables from becoming too large. For a precision of $k = 12$ bits, for example, we break the sequence of 11 value bits into sequences of 5, 3, 2, and 1 bits, plus one sign bit. In our experiments we initialized the arithmetic tables with uniform probabilities and used an adaptive coder that learned the actual distribution. An additional coding gain could potentially be achieved by initializing the table with the expected distribution.

mesh	connectivity			geometry (8 bits)			geometry (10 bits)			geometry (12 bits)		
name	TG	IA	gain	TG	IA	gain	TG	IA	gain	TG	IA	gain
triceratops	767	421	45	2990	2362	21	4936	3798	23	7095	5226	26
galleon	619	621	0	3666	2555	30	5436	3920	28	7396	5470	26
cessna	1165	1191	-2	3776	2269	40	6075	3824	37	8943	5856	35
beethoven	793	698	12	3589	3247	10	5607	5119	9	8072	7106	12
sandal	699	697	0	2996	2364	21	4648	3692	21	6709	5253	22
shark	484	242	50	2345	1515	35	3859	2346	39	5703	3384	41
al	952	1099	-1	4732	3957	16	7413	6369	14	10344	8997	13
cupie	784	612	22	3003	2505	17	5017	4361	13	7315	6535	11
tommygun	1077	1178	-9	4415	3076	30	7040	4653	34	10197	6517	36
cow	682	647	5	3096	3244	-5	5153	5316	-3	7397	7487	-1
cow_poly	667	554	17	3114	2673	14	5178	4628	11	7417	6776	9
teapot	158	168	-6	1392	981	30	2209	1650	25	3127	2387	24
average			10			24			24			23

Table 5.2: The table reports compression rates in bytes for connectivity and geometry for the Touma and Gotsman coder (TG) and our coder (IA). The coding gains of our coder over the TG coder are reported in percent. Geometry compression rates are given for three different precision levels of 8, 10, and 12 bits. For the cow, a purely triangular model, we get roughly the same bit-rates as the Touma and Gotsman coder. This validates that our improvement really comes from using the polygonal information.

5.1.3 Results

Aside from an additional switch statement and a second set of probability tables, our algorithm has the same simple implementation as the original TG coder. However, the results listed in Table 5.2 show that our generalization to polygon meshes gives an immediate improvement of more than 20 percent in the geometry compression rates. Note that for a purely triangular model (e.g. the cow model) we get roughly the same

bit-rates as the Touma and Gotsman coder. This validates that our improvement really comes from using the polygonal information.

Our approach improves on the TG coder similarly at 8, 10, and 12 bits of precision. This demonstrates that the coding gains are independent from the chosen level of quantization. However, the relative percentage of compression achieved by a geometry coder is strongly dependent on the number of precision bits. This is clearly demonstrated in Table 5.3, which reports relative geometry compression gains (i.e. the achieved reduction in size in proportion to the uncompressed size) at different quantization levels: with increasing precision the achieved compression gain decreases.

mesh	8 bit		10 bit		12 bit		14 bit		16 bit	
name	bpv	gain	bpv	gain	bpv	gain	bpv	gain	bpv	gain
triceratops	6.7	72	10.7	64	14.8	59	19.0	55	23.0	52
galleon	8.6	64	13.2	56	18.4	49	23.8	44	28.9	40
cessna	4.8	80	8.2	73	12.5	65	17.3	59	22.3	54
beethoven	9.8	59	15.4	49	21.4	41	27.4	35	33.4	30
sandal	7.2	70	11.2	63	15.9	56	21.4	49	26.9	44
shark	4.7	80	7.3	76	10.6	71	13.9	67	17.3	64
al	8.7	64	14.1	53	19.9	45	25.8	39	31.7	34
cupie	6.7	72	11.7	61	17.5	51	23.5	44	29.6	39
tommygun	5.9	75	8.9	70	12.5	65	16.7	60	20.7	57
cow	8.9	63	14.6	51	20.6	43	26.6	37	32.7	32
cow_poly	7.4	69	12.7	58	18.7	48	24.6	41	30.7	36
teapot	6.6	73	11.1	63	16.1	56	21.2	50	26.2	46
average	7.2	70	11.5	62	16.6	54	21.8	48	27.0	44

Table 5.3: This table reports geometry compression rates in bits per vertex (bpv) at different quantization levels and the corresponding gain compared to the uncompressed geometry. The latter is simply three times the number of precision bits per vertex.

This means that predictive compression does not scale linearly with different levels of precision. Such techniques mainly predict away the high-order bits. If more precision (= low bits) is added, the achievable compression gain decreases. In order to make a meaningful statement about the average compression rates of a geometry coder it is necessary to clarify at which quantization they were achieved. In Table 5.3 we report the performance of our geometry compression scheme at commonly used levels of precision.

5.1.4 Discussion

The geometry compression schemes that are used in industry-strength triangle mesh coders are those with simple and robust implementations. The most popular of those is Touma and Gotsman's linear parallelogram predictor. We have described a simple

technique that exploits information about polygonal faces for improving the predictive compression rates achieved with the parallelogram rule.

Can we further improve the polygonal geometry compression rates using only a simple linear predictor? Assume a parallelogram prediction is performed within a regular polygon (i.e. a planar and convex polygon with unit-edge lengths) of degree d. The prediction within regular quadrilaterals ($d = 4$) is perfect, but for higher-degree polygons ($d > 4$) the prediction error grows with the degree. Measurements on our test meshes show a similar behavior: predictions within quadrilaterals have the smallest average prediction error and the error becomes larger as the degree increases.

This observation suggests two ways of improvement: On one hand one could traverse the mesh such that a larger number of vertices are predicted within a quadrilateral or a low-degree polygon. On the other hand one could change the linear prediction rule depending on the degree of the polygon a vertex is predicted within. The parallelogram rule can be written as the linear combination $P = \alpha * A + \beta * B + \gamma * C$ where $\alpha = \gamma = 1$ and $\beta = -1$. It has the advantage that it can be implemented with pure integer arithmetic. If we allowed α, β, and γ to be floating point numbers, we could formulate a *pentagon rule* or a *hexagon rule*. Such rules would not be limited to base their predictions on only three vertices. The prediction of the last unknown vertex within a hexagon, for example, could use a linear combination of all five vertices.

The challenge is then to find *generic* coefficients that improve compression on all typical meshes. We tried to compute such coefficients for various polygonal degrees by minimizing the Euclidean error over all possible predictions in our set of test meshes. During compression we then switched the coefficients α, β, and γ based on the degree of the polygon we predicted within. This approach slightly improved the compression rates on all meshes; even on those that were not part of the set used to compute the coefficients. Also predictions across polygons can be improved this way by switching between different floating point coefficients based on the degrees of the two polygons involved. Initial experiments show that such degree-adapted prediction rules result in small but consistent improvements in compression. In particular for pure triangle meshes we found that our optimized coefficients always gave coding gains of a few percents because they would less often "over-shoot" in their predictions. However, for polygon meshes these gains are bound to be moderate because on average more than 70 percent of the vertices are predicted within a quadrilateral. Our experiments confirmed that the best linear predictor for these vertices is the standard parallelogram rule.

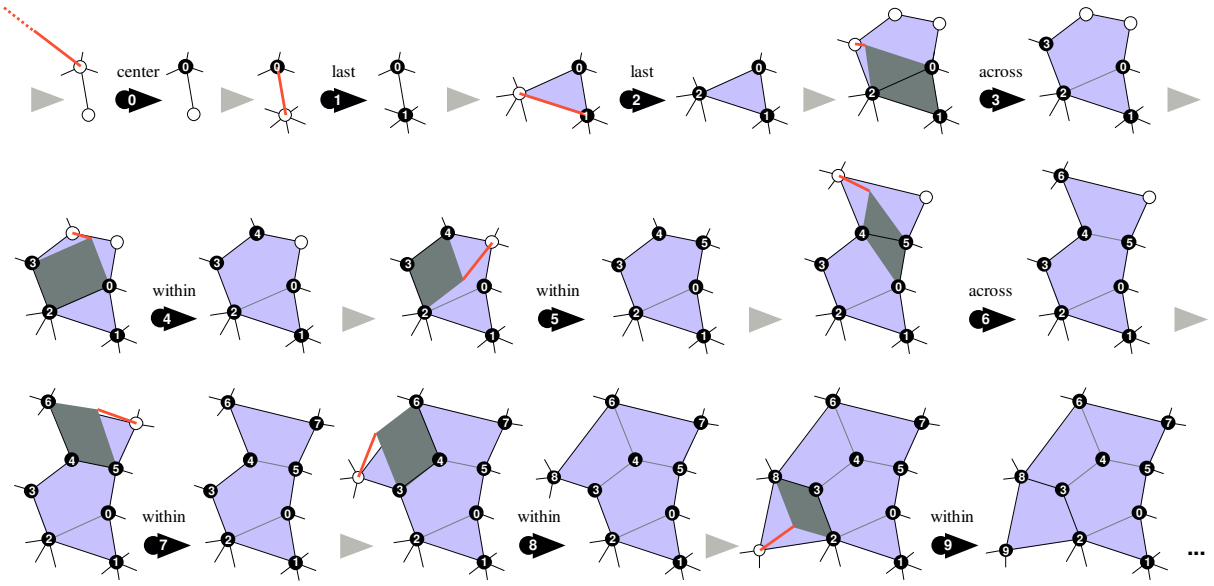


Figure 5.3: This example demonstrates the order in which the connectivity decoder traverses the vertices and which parallelogram predictions the geometry decoder uses to decode their position. The parallelogram used for prediction is shown in dark grey and the corrective vectors are shown in red. The light-grey arrows stand for one or more steps of the connectivity decoder, which are described in Figure 4.3. The black arrows with numbers denote a step of the geometry coder. They are described here: **(0)** The decoder predicts vertex 0 as the center of the bounding box. **(1)** The decoder predicts vertex 1 as vertex 0. **(2)** The decoder predicts vertex 2 as vertex 1. **(3)** The decoder *across*-predicts vertex 3 as completing the parallelogram spanned by vertices 0, 1, and 2. **(4)** The decoder *within*-predicts vertex 4 as completing the parallelogram spanned by vertices 0, 2, and 3. **(5)** The decoder *within*-predicts vertex 5 as completing the parallelogram spanned by vertices 2, 3, and 4. **(6)** The decoder *across*-predicts vertex 6 as completing the parallelogram spanned by vertices 5, 0, and 4. **(7)** The decoder *within*-predicts vertex 7 as completing the parallelogram spanned by vertices 5, 4, and 6. **(8)** The decoder *within*-predicts vertex 8 as completing the parallelogram spanned by vertices 6, 4, and 3. **(9)** The decoder *within*-predicts vertex 9 as completing the parallelogram spanned by vertices 8, 3, and 2. And so on ...

5.2 Compressing the Property Mapping

Mesh properties are used to refine the visual appearance of the mesh. Rather than directly specifying the color with which to display each polygon, they usually describe shading normals and material attributes that are attached to the polygon mesh, which are then used at run-time to compute colors in dependance on the view directions and the lights in the scene. In addition, a polygon mesh can have texture coordinates that further refine the appearance through references into an image or a light map.

The compression of mesh properties involves two kinds of information: the *property values* and the *property mapping*. The property values specify each individual property, such as the r, g, and b components for a color or the u and v components for a texture coordinate and we describe how to compress these in the next section. The property mapping specifies how these properties are attached to the mesh and we now describe how to encode this efficiently. (Deering, 1995) initiated research on compressing property values and was followed by others. These works, however, pay little attention to the problem of compressing the mapping. The few existing techniques (Taubin et al., 1998b; Gumhold and Strasser, 1998; Isenburg and Snoeyink, 2000) are fairly basic and use between 1.5 to 6 bits per vertex (bpv). This is surprisingly inefficient, especially given the abundance of works on mesh connectivity coding that scramble for improvements of compression rates that are already as low as 2 to 3 bpv.

We show how a set of simple predictions can be used to efficiently compress the property mapping of polygon meshes. Our predictive compression scheme results in bit-rates between 0.1 and 2 bpv, which improves by a factor of 2 to 10 over previous methods. After characterizing property mapping, we first discuss previously proposed methods for its compression, before we describe our predictive approach. Finally we consider the efficiency of this scheme for the special case of stripified triangle meshes.

5.2.1 Characterizing the Property Mapping

In the literature a property mapping is often classified as either *per-vertex*, *per-face*, or *per-corner* with the first two being special cases of the last. In the per-vertex case the properties are attached to the mesh vertices; a common property is shared by all corners around a vertex. In the per-face case the properties are attached to the mesh faces; a common property is shared by all corners around a face. In the per-corner case the properties are attached to the mesh corners; although each corner could have a different property, typically a common property is shared by a set of adjacent corners.

For shading normals, colors and texture coordinates there is usually a one-to-one mapping between the properties and the mesh elements. A per-vertex mapping has as many properties as vertices and each property is used by one vertex. Similarly a per-face mapping has as many properties as faces and each property is used by one face. For a per-corner mapping there are at least as many properties as vertices and at most as many properties as corners. Here each property is used by a set of adjacent corners that all sit around the same vertex. An exception is the mapping of material attributes, which are usually attached to the faces of the mesh. They are

mapped differently because each material attribute is used by many faces. All faces that represent the same real-world surface are given the same material attribute.

A per-vertex mapping for shading normals, colors, and texture coordinates results in a completely smooth shaded mesh when interpolated shading (e.g. Gouraud shading) is applied. It is smooth *along* each edge because the properties attached to the vertices at its ends are interpolated. It is smooth *across* each edge because the same properties are interpolated on both sides of the edge. Shading discontinuities may be introduced by cutting the mesh along a discontinuity and duplicating the affected vertices. These duplicates are given the same vertex location but different properties, which creates the shading discontinuity. A per-face mapping of shading normals or colors gives the mesh a faceted appearance unless it is highly tessellated. An example for a face-based property assignment is a pre-computed radiosity solution where each face is assigned the amount of light it emits or transmits. A per-corner mapping is an elegant way to specify shading discontinuities in otherwise smooth shaded meshes. Each vertex in a smooth region of the mesh has a single property that is shared by all its corners. Vertices along a discontinuity, however, have multiple properties each of which is shared by a set of adjacent corners. This avoids having to cut the mesh and deal with multiple copies of vertices. We make the following definitions to characterize the different configurations that can arise for a per-corner property mapping.

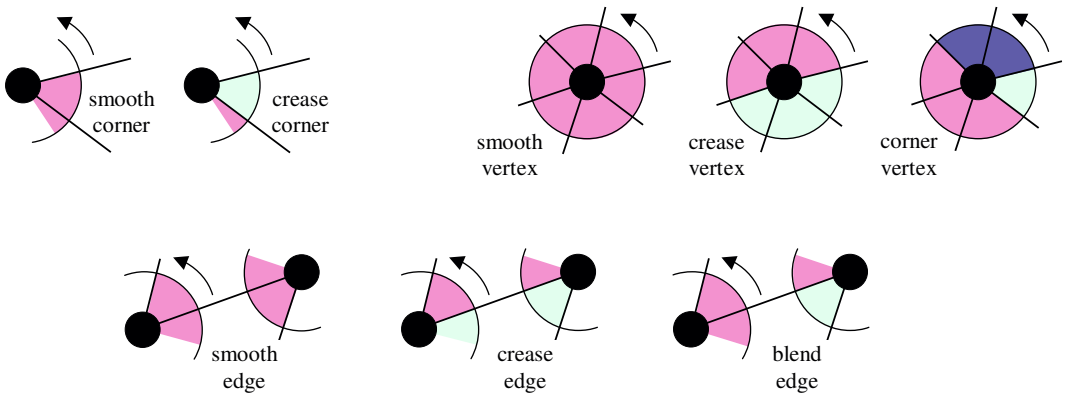


Figure 5.4: Different shaded corners have different properties associated. A smooth corner uses the same property as the previous corner, while a crease corner (a crease) uses a different one. Smooth vertices have no crease, crease vertices have two creases, and corner vertices have three or more creases. Smooth edges have no crease, crease edges have creases on both ends, while blend edges only have a crease at one end.

Around every vertex is a cycle of corners and edges. In this paper we use a counter-clockwise order to talk about a next/following and a previous/preceding edge or corner.

A vertex can be visited through any of its edges. For each edge there is a unique traversal of the corners surrounding the vertex. The traversal starts with the corner following the respective edge and ends with the corner preceding it. We say a corner is a *smooth corner* if it uses the same property as the previous corner, otherwise we have a *crease* and consequently call it a *crease corner* (see also the illustrations in Figure 5.4). A *smooth vertex* uses the same property for all adjacent corners—it has no crease. A *crease vertex* uses two different properties each associated with a set of adjacent corners—it has two creases. A *corner vertex* uses three or more properties each associated with a set of adjacent corners—it has more than two creases. A *smooth edge* has no property discontinuity on either end. The two corners that are next around the respective vertices are both smooth corners—this edge has no crease. A *crease edge* has a property discontinuity on each end. The two corners that are next around the respective vertices are both crease corners—this edge has two creases. And finally, a *blend edge* has a property discontinuity on only one end—it has one crease.

A mesh with a per-vertex mapping has no creases; all its corners are smooth corners, all its vertices are smooth vertices, and all its edges are smooth edges. Such meshes have a one-to-one mapping from vertices to properties. A mesh with a per-face mapping has only creases; all its corners are crease corners, all its vertices are crease or corner vertices, and all its edges are crease edges. Such meshes have a one-to-one mapping from faces to properties. A mesh with a per-corner mapping is somewhere between these two. Such meshes do *not* have a one-to-one mapping from corners to properties but instead a one-to-one mapping from smooth vertices *plus* crease corners to properties.

5.2.2 Encoding the Property Mapping

Neither a per-vertex nor a per-face property mapping need to be stored explicitly. The property values are simply stored in the order in which the vertices and faces that they are associated with are encountered during the traversal of the mesh. Every property is stored only once because of the one-to-one mapping between vertices/faces and properties. An exception is the mapping of material attributes. Since a material attribute is used by many faces it would be stored many times. In this case it is cheaper to store the property mapping from faces to materials explicitly but therefore each material attribute only once. This type of mapping can be efficiently encoded using *super faces* as suggested in (Isenburg and Snoeyink, 2000). A per-corner property mapping also needs to be stored explicitly. If every property is to be stored only once, we must specify which corners share a property. So far three different methods for

compressing per-corner mappings of properties have been proposed.

(Gumhold and Strasser, 1998) describe how to include this mapping information into the bit-stream of their one-pass coder. During the traversal of the mesh, all edges are classified as either smooth edges or as crease/blend edges using one bit. The latter are distinguished using two additional bits that specify, for each end of the edge, whether it is a crease or not. Encoding three possible configurations (i.e. the case that both ends have no crease cannot occur) with two bits is slightly wasteful and could be improved. For triangle meshes this approach requires at least 3 bpv (e.g. all edges are smooth) and at most 9 bpv (e.g. no edge is smooth). In praxis bit-rates range between 3 and 5.5 bpv. (Taubin et al., 1998b) store a *discontinuity bit* at the moment their mesh traversal reaches a corner for the first time. This bit is "0" for a smooth corner and a "1" for a crease corner. The property data that is associated with crease corners is then stored in the order in which the corresponding corners marked with "1" are encountered. This approach requires exactly as many bits as the mesh has corners, which implies a bit-rate of 6 bpv for triangular meshes.

Since meshes often have a significant fraction of smooth vertices we proposed in (Isenburg and Snoeyink, 2000) a simple improvement on Taubin et al.'s scheme that uses *vertex bits* and *corner bits*. One bit per vertex distinguishes smooth vertices ("1") from crease and corner vertices ("0"). The corners around a crease or corner vertex are marked as before, while the corners around a smooth vertex need no further treatment. The property data associated with smooth vertices and crease corners is again stored in the same order as the corresponding "1" bits appear in the bit sequence. For triangle meshes this approach requires at least 1 bpv (e.g. all vertices are smooth) and at most 7 bpv (e.g. no vertex is smooth). The performance gain/loss over (Taubin et al., 1998b) depends on the fraction of smooth vertices. For polygon meshes with an average vertex degree of d the break-even point is reached when this fraction is approximately $1/d$ with the gain increasing as the fraction gets larger. With an initial pass over the mesh we could always choose the better of the two methods. Otherwise, the bit-rate is at most 1 bpv above, but potentially 5 bpv below the bit-rates of (Taubin et al., 1998b).

Four simple rules, which are illustrated in Figure 5.5, can further reduce the number of vertex and corner bits needed:

rule $\mathbf{R}_1$ Vertices that have only one corner do not need a vertex bit. Such vertices are by definition smooth vertices.

rule $\mathbf{R}_2$ Crease vertices that have only two corners do not need corner bits. The vertex bit already determines whether it is a smooth vertex and both corners are smooth corners,

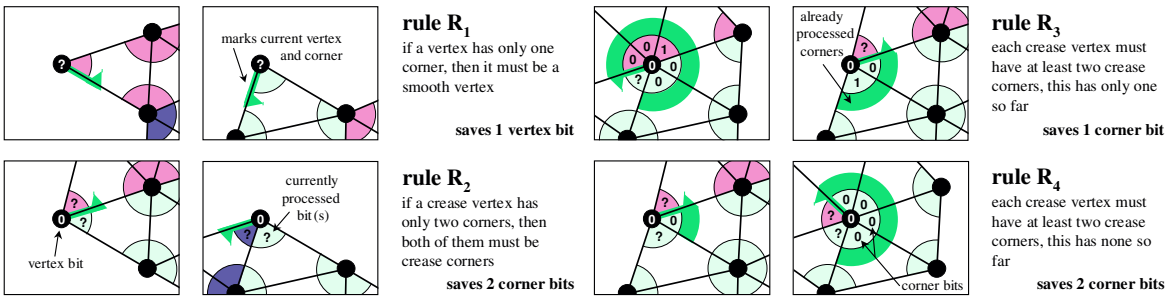


Figure 5.5: Simple rules to save vertex and corner bits. Using rule R_1 avoids unnecessary vertex bits, rules R_2, R_3, and R_4 avoid unnecessary corner bits.

or whether it is a crease vertex and both corners are crease corners.

rule R_3 If all but one corner of a vertex have been marked and there has been only one crease corner, then there is no need for the last corner bit. Because corner bits are only used for crease/corner vertices and such vertices have at least two crease corners, the last corner must be a crease corner.

rule R_4 Similarly, if all but two corners of a vertex have been marked and there has been no crease corner, then there is no need for the last two corner bits.

The rules R_1 and R_2 rarely apply for meshes without holes or boundary, since usually only vertices on the boundary have as few as one or two corners. However, these rules are very effective for compressing the property mapping of stripified triangle meshes.

5.2.3 Predicting the Property Mapping

Although heavily used for other aspects of mesh compression, predictive coding has previously not been used for compressing the property mapping. We predict since vertex bits and corner bits based on two simple observations that are also reflected in the statistics on the per-corner shading normal mapping given in Table 5.4. First, most edges are either smooth edges or crease edges, and second, most vertices are either smooth vertices or crease vertices. Using a set of eight simple predictions we exploit the correlation implied by these statistics. We have two predictions P_1 and P_2 for the vertex bits and six prediction P_3 to P_8 for the corner bits. Two example configurations for each of the two vertex bit predictions are illustrated in Figure 5.6. They are as follows:

prediction P_1 If the vertex connects to any previously processed vertex at a crease then we predict this vertex to be a crease or corner vertex because we assume that the

name	corners		edges			vertices		
button	66	34	66	34	—	—	100	—
dragknob	66	34	66	34	—	—	100	—
handle	60	40	60	40	—	—	70	30
handle1	66	34	66	34	—	—	100	—
handle2	98	2	98	2	—	92	6	—
part1	66	34	66	34	—	—	98	2
part4	83	17	83	17	—	50	50	—
part5	66	34	66	33	<1	1	94	3
spool	81	19	81	19	—	43	57	—
rotor	83	17	83	17	—	49	51	—
oilfilter	77	23	77	23	—	33	61	6
galleon	70	30	70	30	—	48	38	14
sandal	76	24	76	23	<1	56	33	11

Table 5.4: Statistics on the normal mapping for the meshes shown in Figure 5.8. Reported are the percentages of smooth and crease corners, and of smooth, crease, and blend edges, and also of smooth, crease, and corner vertices.

connecting edge is a crease edge and not a blend edge.

prediction $\mathbf{P}_2$ In all other cases we make no assumption.

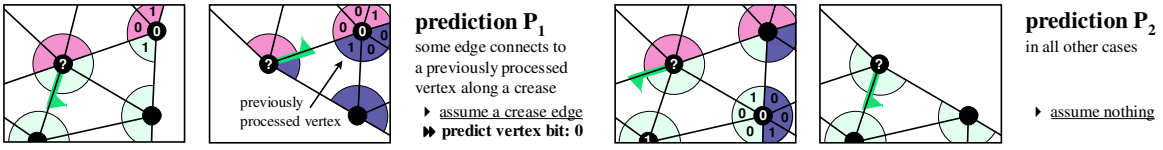


Figure 5.6: Two example scenarios for each of the two cases used to predict vertex bits.

Two example configurations for each of the six corner bit predictions are illustrated in Figure 5.7. They are as follows:

prediction $\mathbf{P}_3$ If the current edge connects to a processed vertex at a crease, we predict the next corner to be a crease corner, because we assume the edge is a crease edge.

prediction $\mathbf{P}_4$ If the current edge connects to a processed vertex not at a crease, we predict the next corner to be a smooth corner, because we assume the edge is a smooth edge.

prediction $\mathbf{P}_5$ If there have been already two (or more) crease corners, we predict the next corner to be a smooth corner because we assume this vertex is a crease vertex.

prediction $\mathbf{P}_6$ If there has been already one crease corner and if there have been less than $\lfloor (degree-1)/2 \rfloor$ smooth corners since then, we predict the next corner to be a smooth corner because we assume this vertex is a crease vertex.

prediction P_7 If there have been $\lfloor (degree-1)/2 \rfloor$ or more smooth corners, we predict the next corner to be a crease corner because we assume this vertex is a crease vertex.

prediction P_8 In all other cases we make no assumption.

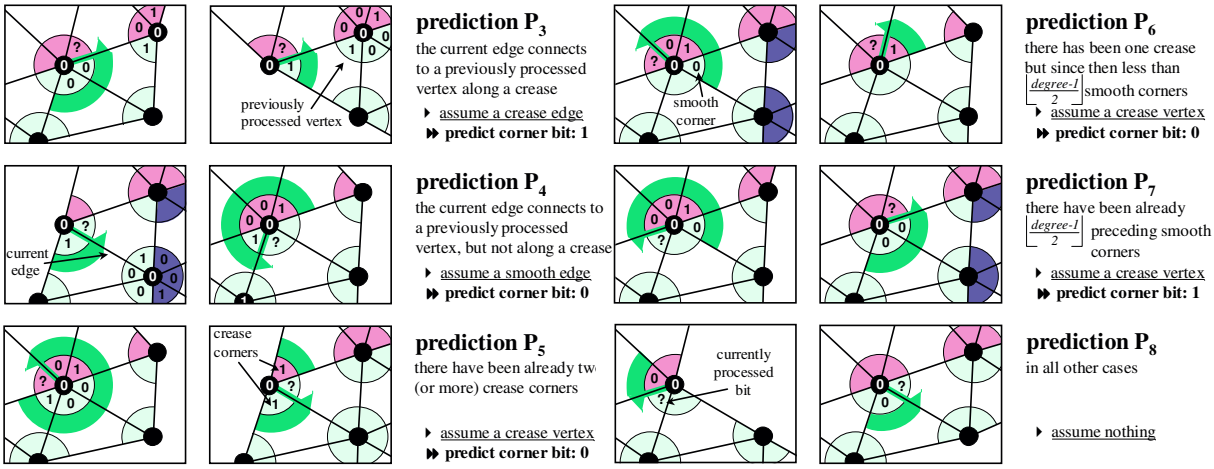


Figure 5.7: Two example scenarios for each of the six cases used to predict corner bits.

We use arithmetic coding with different probability tables for each of the eight predictions P_1 to P_8. Every probability table has two entries, one predicting the likelihood of a "0" bit and the other predicting the likelihood of a "1" bit. Initially we set these probabilities to roughly express our predictions. We use an adaptive version of an arithmetic coder that updates the respective probability table after every prediction.

Typical bit-rates for compressing the property mapping using this approach are reported in Table 5.5. For comparison we also give the bit-rates achieved using discontinuity bits as proposed by (Taubin et al., 1998b), using crease edge bits as proposed by (Gumhold and Strasser, 1998), and simply using vertex and corner as proposed earlier in (Isenburg and Snoeyink, 2000). The comparison between coders that simply store sequences of bits and a coder that applies context-based arithmetic coding to these bits is unfair. Obviously there is correlation among subsequent bits in the discontinuity bit sequence of (Taubin et al., 1998b) that could easily be exploited using a standard memory-sensitive entropy coding scheme such as an adaptive order-k arithmetic coder, which uses a different probability table for each combination of the preceding k bits.

In Table 5.6 we give compression rates that are the result of applying adaptive arithmetic coding of orders 0 to 5 to the sequence of discontinuity bits generated by the method of (Taubin et al., 1998b) and compare them with the compression rates achieved by our predictive coder. There are sudden improvements in the compression rates of

mesh characteristics			bits per vertex			
name	vertices	normals	T+	GS	IS	pred
button	99	198	6.0	4.9	6.6	1.2
dragknob	161	322	6.0	5.0	6.8	1.3
handle	100	236	6.0	5.3	6.3	2.1
handle1	128	256	6.0	5.0	6.6	1.5
handle2	1165	1235	6.0	3.1	1.3	0.1
part1	166	336	6.0	5.0	6.4	1.6
part4	330	495	6.0	4.0	3.8	0.9
part5	175	355	6.0	5.0	6.5	1.9
rotor	600	905	6.0	4.0	4.0	1.0
spool	649	1018	6.0	4.1	3.8	1.1
oilfilter	860	1484	6.0	4.4	4.7	1.5
galleon	2372	3974	4.0	3.2	2.8	1.0
sandal	2636	4096	4.1	3.0	2.7	0.9

Table 5.5: Compression results for the property mapping using discontinuity bits as proposed by Taubin et al. (*T+*), using crease edge bits as proposed by Gumhold and Strasser (*GS*), using vertex and corner bits as proposed by Isenburg and Snoeyink (*IS*) for mainly smooth mappings, and finally the predictive version of the latter (*pred*).

mesh	bits per vertex							
name	T+	aac0	aac1	aac2	aac3	aac4	aac5	pred
button	6.0	5.5	4.9	2.7	2.6	2.6	2.6	1.2
dragknob	6.0	5.5	4.6	2.5	2.5	2.5	2.5	1.3
handle	6.0	5.7	5.3	5.0	4.6	4.6	4.5	2.1
handle1	6.0	5.5	4.8	2.6	2.6	2.6	2.6	1.5
handle2	6.0	0.9	0.9	0.8	0.4	0.3	0.3	0.1
part1	6.0	5.5	5.0	3.5	3.3	3.3	3.3	1.6
part4	6.0	3.9	3.8	3.7	2.1	1.9	1.9	0.9
part5	6.0	5.5	4.7	4.1	4.1	4.1	4.1	1.9
rotor	6.0	4.2	4.0	3.6	1.3	1.1	1.1	1.0
spool	6.0	4.0	3.8	3.8	2.3	2.2	2.1	1.0
oilfilter	6.0	4.7	4.6	4.6	4.0	3.5	3.4	1.5
galleon	4.0	3.5	3.4	2.5	2.3	2.3	2.2	1.0
sandal	4.1	3.3	3.3	2.4	2.2	2.2	2.2	0.9

Table 5.6: The discontinuity bit sequence for coding the property mapping (*T+*) compressed with adaptive arithmetic coding of orders 0 to 5 (*aac0* to *aac1*) in comparison to our predictive coder (*pred*).

the discontinuity bit sequence. The biggest jumps occur when the coder increases its memory to either 2 or 3 bits. The order-2 coder has learned the likelihood of two smooth corners being followed by a crease corner around a crease vertex. The order-3 coder has learned the likelihood of three smooth corners being followed by another smooth

corner around a smooth vertex. However, our predictive scheme always outperforms any arithmetic order-k coding of the discontinuity bit sequence. First of all, an order-k coder has no means to learn predictions P_1 and P_2 from the discontinuity bit sequence, because they involve neighboring vertices. Moreover, this coder is constantly mislead. It processes a continuous sequence of discontinuity bits and does not know whether consecutive bits are from corners of the same or of different vertices. This causes it to predict and to learn from k-bit strings that contain corner bits from different vertices and therefore have little or no correlation.

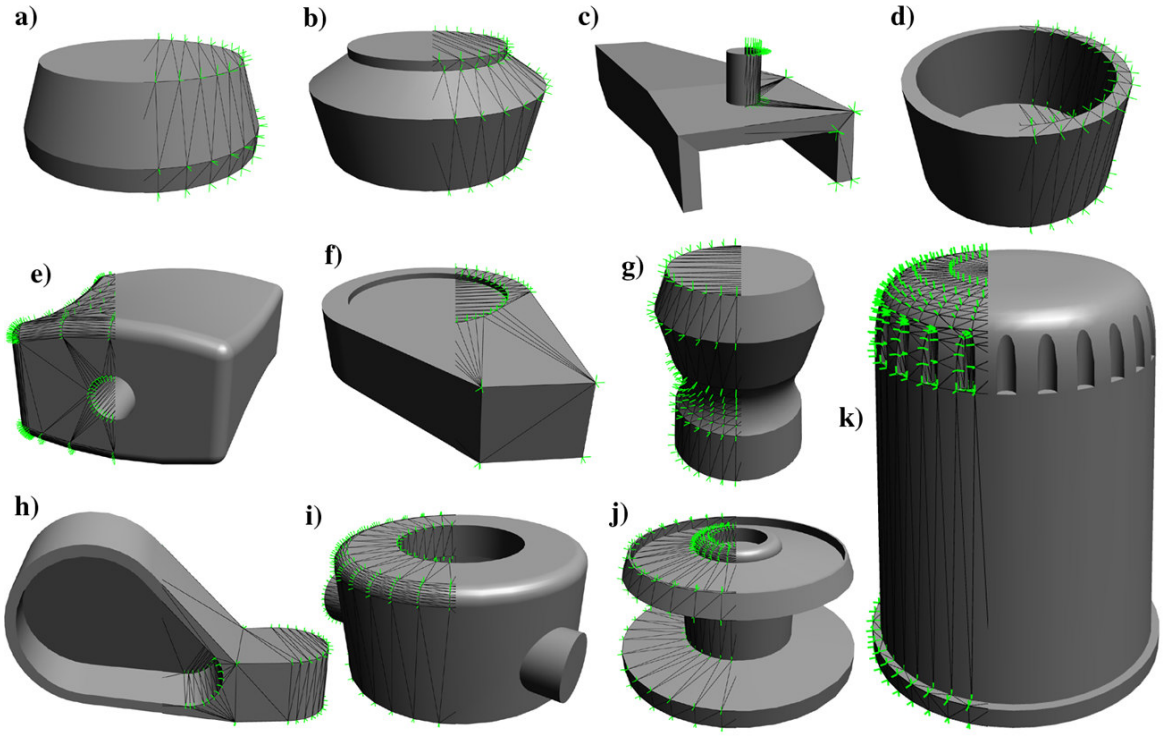


Figure 5.8: The example meshes above are courtesy of Engineering Animation Inc. All except the last are part of a fishing reel assembly: a) button, b) dragknob, c) handle, d) handle1, e) handle2, f) part1, g) part4, h) part5, i) rotor, j) spool, and k) oilfilter. Not shown here are the galleon and the sandal mesh courtesy of Viewpoint Datalabs.

5.2.4 Stripified Triangle Meshes

Generating good triangle strips is a difficult task. This was motivation in Section 3.5 to devise a compression technique that includes pre-computed triangle strip information into the encoding the mesh connectivity. Once such triangle strip information is available it can be used to further improve the coding of the property mapping.

Triangle strips arrange the mesh into long runs of adjacent triangles. When rendering a triangle strip, the vertices shared among subsequent triangles are stored on the graphics board. This way two vertices from a previous triangle are re-used for all but the first triangle of every strip. Re-using a vertex not only includes its coordinates, but also shading normals, colors, or texture coordinates. However, not every pair of triangles shares these properties across the common edge. When triangles are connected along a crease or blend edge they uses a different property at both or one of their common vertices. Therefore a triangle strip generator must ensure that strips do not run across such discontinuities. All corners of a vertex that are adjacent in a triangle strip must share all properties attached to them.

This complicates the process of creating triangle strips, but it reduces the number of bits needed to compress the property mapping. Since a property will always be shared by all corners of a vertex that are adjacent in a triangle strip, the property mapping can be thought of per *strip corner* rather than per corner. The number of different corners for a mesh with t triangles is $3t$. However, for a mesh decomposed into s strips we need to distinguish only $t + 2s$ strip corners for the mapping from properties to corners.

The bit-saving rules R_1 to R_4 are now applicable to strip corners instead of to corners. Since the number of strip corners per vertex is much lower than the number of corners, these rules apply much more often and save many more bits. We continue to predict vertex bits using P_1 and P_2 but predict strip corner bits instead of corner bits using P_3 to P_8. In Table 5.7 we list the compression rates achieved by simply storing the sequence of vertex and strip corner bits (i.e. without predictive compression) and the compression rates after applying the predictions. These results show that the availability of triangle strip information makes the compression of the property mapping a lot cheaper. However, this information is only available if it was encoded with the mesh in such a way that it can be decoded prior to decoding the property mapping. Compressing the mesh connectivity together with triangle strips makes its encoding more expensive. But the savings we get from the improved compression of the property mapping is often sufficient to offset this expense.

The predictions P_6 and P_7 involve comparisons that make use of the degree of the processed vertex. In experiments we replaced them with simpler predictions that are based only on absolute counts of preceding smooth and crease corners. The increase in bit-rates was less than 5 percent. This is not surprising, since most of our coder's efficiency results from predictions P_1 to P_5. It might be possible to improve compression further by directing the traversal to places where we expect the most correct predictions.

mesh characteristics				bits per vertex	
name	t	s	$(t+2s)/3t$	IS	pred
button	194	4	0.35	1.1	0.2
dragknob	318	6	0.35	1.1	0.2
handle	196	24	0.42	2.0	0.5
handle1	252	5	0.35	1.1	0.2
handle2	2326	125	0.37	1.0	0.1
part1	328	6	0.35	1.1	0.2
part4	656	34	0.37	1.1	0.3
part5	346	9	0.35	1.2	0.2
rotor	1200	41	0.36	1.1	0.3
spool	1294	24	0.35	1.1	0.2
oilfilter	1716	135	0.39	1.4	0.5

Table 5.7: Compressing the property mapping of stripified triangle meshes using a bit sequence of vertex and strip corner bits (IS) and the predictive version of this scheme (pred). Also reported are the number of triangles t and strips s for each mesh and the ratio $(t+2s)/3t$ between strips corners and corners.

A traversal order that follows the discontinuities on the mesh, for example, should make sure that most vertex bits are predicted correctly.

5.3 Compressing Texture Coordinates

The problem of compression of texture coordinates has received little attention. In this section we rigorously investigate texture coordinate compression in a quantitative, in-depth manner. Some papers on geometry compression (Taubin et al., 1998b; Bajaj et al., 1999) suggest that texture coordinates could be compressed with the same predictive scheme that is already used for vertex positions, but give no further details and report no experimental results. Although the predictors for vertex positions are in general suited for texture coordinates, the presence of discontinuities in the texture mapping can result in completely unreasonable predictions. The close-up views of the "lion" model shown in Figure 5.9 illustrate such discontinuities: Neighboring texture coordinates around the nose, the mouth, and the ear address distant locations in the texture image. Predictive schemes for compressing vertex positions assume that vertices that are *topologically close* are also *geometrically close*. This means, for example, that two vertices connected by an edge are assumed to have nearby positions in 3D space. The same assumption also holds for texture coordinates—unless there is a discontinuity. Instead of performing an unreasonable prediction near a discontinuity, our scheme switches to a less promising but at least reasonable predictor.

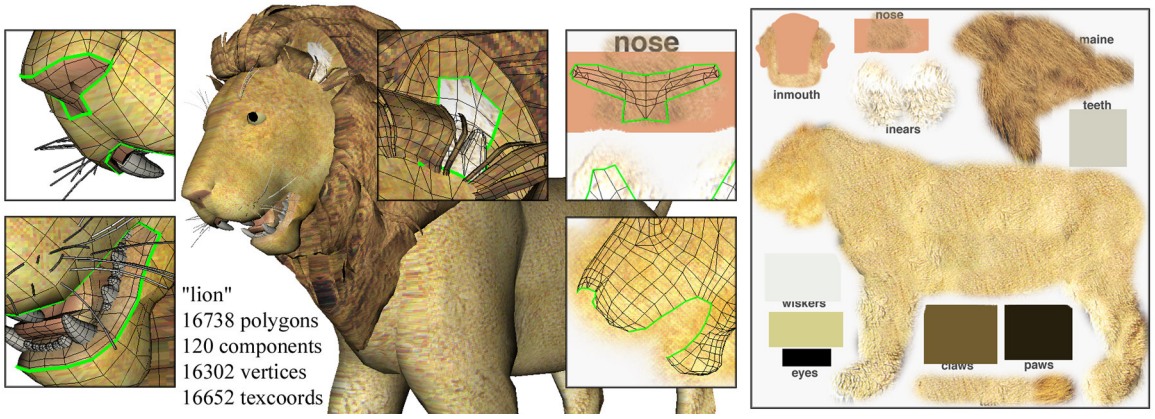


Figure 5.9: The lion model and its texture atlas. The close-ups show mapping discontinuities where neighboring texture coordinates address distant locations in the atlas.

The parallelogram rule predicts the position of a vertex to complete a parallelogram defined by the three vertices of a neighboring triangle. Intuitively it makes sense to apply this rule also to texture coordinates. Usually the texture coordinates of two adjacent triangles also form two adjacent triangles in texture space. But if the edge connecting the two triangles coincides with a discontinuity in the texture mapping, the parallelogram rule gives a completely "random" prediction. In this case it is better to fall back to a simpler but more meaningful prediction.

5.3.1 Discontinuities in the Texture Mapping

Texture images are a simple way to increase the realism of polygonal meshes. The process of applying a texture image to a mesh is called *texture mapping*. It consists of putting every polygon of the 3D mesh into correspondence with a polygon in the 2D texture image. Although each polygon could be mapped independently, it is usually beneficial to map neighboring polygons in the mesh into neighboring polygons in texture space. The problem of finding a suitable mapping or *parameterization* for texturing a polygonal surface is a much studied problem (Maillot et al., 1993; Sander et al., 2001; Desbrun et al., 2002; Levy et al., 2002; Gu et al., 2002; Sorkine et al., 2002).

Vertices whose surrounding polygons are mapped into neighboring polygons in texture space appear at a single location in the texture image. They have a single texture coordinate that is used by all their surrounding polygons. In order to flatten a mesh without boundary or of non-trivial topology it is often cut open. Such cuts introduce discontinuities or *seams* in the texture mapping. Vertices along these seams appear at several locations in the texture image. Therefore they have multiple texture coordinates

each of which is used by a subset of their surrounding polygons.

To minimize distortion in the texture-mapped image, the mapping between the polygons in 3D and the polygons in 2D is often sought to preserve angles and distances. Usually, it is impossible to flatten an entire mesh in one piece without creating overlapping or extremely distorted polygons. Distortion can be reduced by introducing additional cuts (Gu et al., 2002) or by cutting the mesh into several parts that are then parameterized separately (Maillot et al., 1993; Sander et al., 2001; Levy et al., 2002; Sorkine et al., 2002). The latter results in the parameterization being broken up into several *charts*, which are then assembled into a single texture image called an *atlas*. Both approaches create additional discontinuities in the texture mapping.

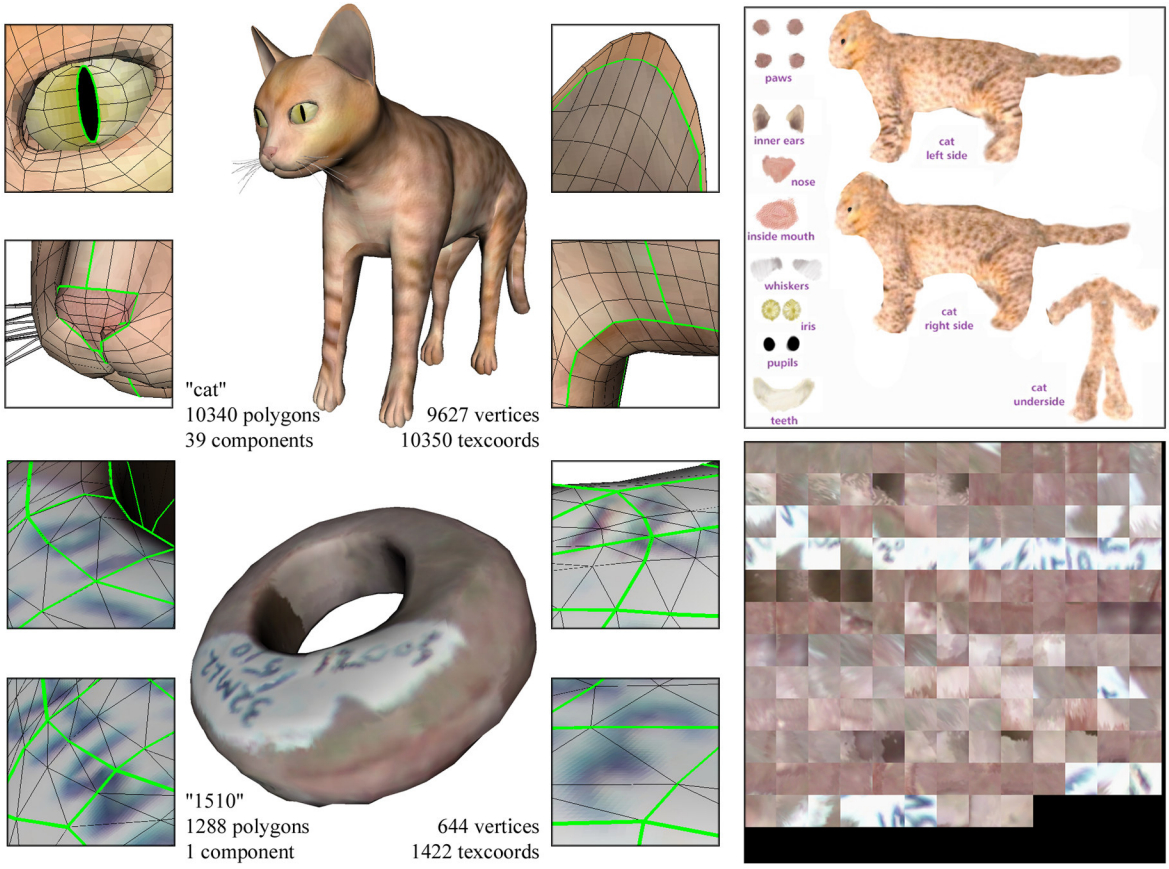


Figure 5.10: The closeup views show the texture mapping discontinuities of the "cat" model from the animal set and the "1510" model from the archaeological artifact set. These can also be inferred from their corresponding texture atlas.

There are also other reasons to perform a piece-wise texture mapping: From an artist's perspective it is often convenient to cut a mesh into several parts in order to paint each of them separately, as it was done for the "cat" model from Figure 5.10. Also,

commercial software packages for automated generation of texture maps sometimes create a lot of seams. The "1510" model from Figure 5.10 was generated from a scan of an archaeological artifact. Its parameterization is broken into a many small squares that are tightly packed into a space-efficient atlas. This creates an immense number of discontinuities in the texture mapping.

In our experiments we use two sets of meshes with differently generated texture mappings. One set consists of hand-crafted polygon models of animals that were carefully textured by a skilled artist. Their texture mappings are relatively smooth with only a small number of seams that either coincide with "natural" discontinuities or hide in the least conspicuous regions. The other set consists of automatically generated triangle models of scanned archaeological artifacts that were textured using an automated method. Their texture mappings are not smooth at all.

5.3.2 Previous Work

There are only a few papers that mention texture coordinate compression. Most authors employ their position predictor to compress mesh properties. (Deering, 1995) uses delta-coding for compressing quantized RBG colors. Similarly, (Taubin et al., 1998b) apply their spanning predictor to compress various quantized mesh properties. Although they capture discontinuities in the mapping with *discontinuity bits*, they do not use this information to prevent unreasonable predictions near discontinuities. (Bajaj et al., 1999) limit their linear predictor to meshes with perfectly smooth property mappings and perform all computations in spherical coordinates. For RGB colors quantized to 4, 6, and 8 bits per component they report surprisingly disappointing compression gains of merely 12, 11, and 10 percent.

A completely different approach for texture coordinate compression was proposed by (Sorkine et al., 2002). They completely re-texture a mesh by first computing a new piece-wise parameterization that is implicitly defined by the mesh and by then warping the texture image accordingly. This eliminates the need to explicitly store the texture coordinates as they can be computed from the mesh. However, the requirement to warp the texture image makes this method unsuitable as a general purpose compressor. Another approach that avoids explicit texture coordinates re-samples the mesh onto a regular grid in texture space (Gu et al., 2002). However, this method should be thought of as a different and more compact representation for geometric shapes rather than a mesh compression scheme.

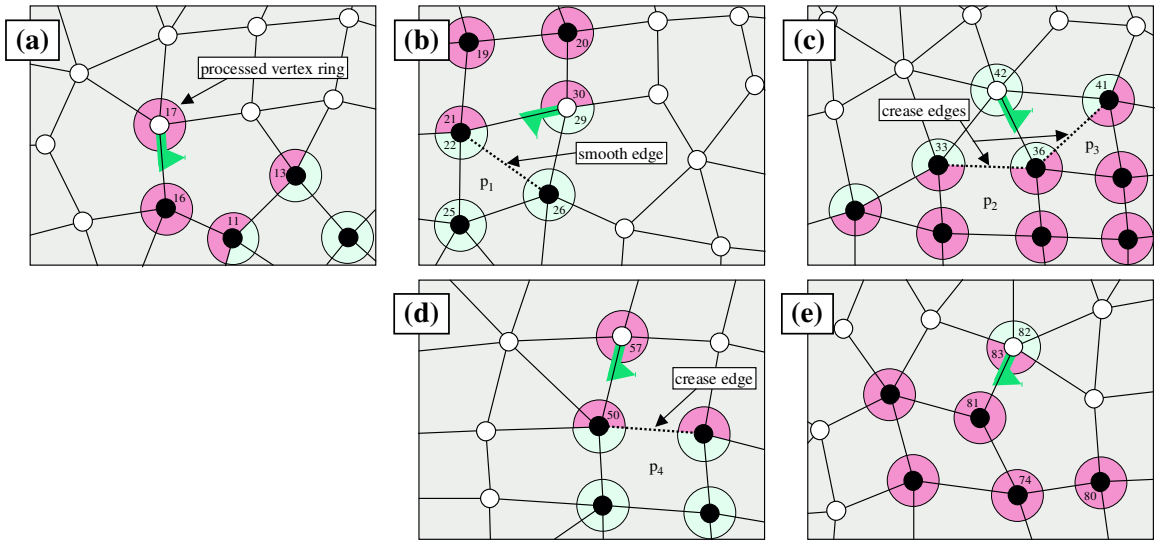


Figure 5.11: These five example scenarios illustrate the four different predictions that are used for compressing texture coordinates. The processed vertex ring and the edge through which it was encountered are marked with a green flag. Corners from the same vertex that are shaded with the same color use the same texture coordinate. Corners from different vertices that are shaded with the same color use a texture coordinate from the same chart. Corners that are shaded with different colors use texture coordinates from different charts. **(a)** a smooth vertex. Its texture coordinate T_{17} is used by four corners. It is *within*-predicted as $T_{16} - T_{11} + T_{13}$. **(b)** a crease vertex. Its texture coordinates T_{29} and T_{30} are each used by two corners. T_{29} is *across*-predicted as $T_{22} - T_{25} + T_{26}$ by involving polygon p_1, which connects to the vertex ring via a smooth edge. T_{30} is *within*-predicted as $T_{20} - T_{19} + T_{21}$. **(c)** another smooth vertex. Its texture coordinate T_{42} is used by six corners. It is *across*-predicted inside the vertex ring as $T_{33} - T_{36} + T_{41}$. The two *across*-predictions involving the polygons marked p_2 and p_3 are forbidden as they connect to the vertex ring via a crease edge. **(d)** again a smooth vertex. Its texture coordinate T_{57} is used by four corners. It is *nearby* predicted as T_{50}. An *across*-prediction involving the neighboring polygon marked p_4 is forbidden because of the crease edge. **(e)** a crease vertex. Its texture coordinates T_{82} and T_{83} are used by two and three corners respectively. T_{82} is *center*-predicted as the center of the texture image. There is no reasonable prediction for T_{82} because it could address any location in the image. T_{83} is *within*-predicted as $T_{81} - T_{74} + T_{80}$.

5.3.3 Predicting Texture Coordinates

We compress both the texture coordinate mapping and the texture coordinates by processing the vertex rings in the order they are encountered by the connectivity coder. The texture coordinate mapping is coded with vertex and corners bits, which are compressed with the predictive scheme described in the last section. Subsequently we predict the texture coordinate(s) associated with each vertex ring using one of four

prediction rules and compress the resulting corrective vectors with an arithmetic coder. Whenever possible, we predict a texture coordinates using the parallelogram rule. For polygonal meshes we again try to perform the parallelogram prediction *within* a polygon rather than *across* polygons. The intuition is that four texture coordinates from a single polygon are more likely to be in a parallelogram-shaped configuration in texture space than four texture coordinates from two adjacent polygons.

mesh	predicted [%]				bit-rate [bpt]			
name	within	across	nearby	center	within	across	nearby	center
lion	82	14	3	1	5.6	8.5	9.7	20.2
wolf	84	14	2	0.6	5.9	9.1	11.1	20.2
raptor	76	18	5	1	5.5	8.4	8.9	19.5
fish	90	10	0.3	0.1	6.6	10.6	14.6	20.1
snake	91	8	1	0.2	3.5	7.9	9.5	20.4
horse	86	11	3	0.5	4.3	7.4	10.1	20.4
cat	80	15	4	0.7	4.3	7.1	8.4	19.6
dog	59	37	4	0.7	6.2	7.2	10.0	20.6
average	**81**	**16**	**3**	**0.6**	**5.2**	**8.3**	**10.3**	**20.1**

mesh	predicted [%]				bit-rate [bpt]			
name	within	across	nearby	center	within	across	nearby	center
“1398”	–	55	35	10	–	7.5	9.8	18.9
“1412”	–	49	39	12	–	8.3	10.3	19.0
“1510”	–	53	36	11	–	7.3	10.7	18.3
“1568”	–	51	38	11	–	8.4	10.5	19.0
“17”	–	58	33	9	–	8.2	10.4	17.7
“1814”	–	52	38	10	–	7.5	10.7	15.9
“1823”	–	52	37	11	–	7.6	10.4	19.1
“2441”	–	49	38	13	–	8.3	10.2	19.2
average	**–**	**52**	**37**	**11**	**–**	**7.9**	**10.4**	**18.4**

Table 5.8: These tables report which percentage of texture coordinates is predicted *within* a polygon, *across* polygons, as a *nearby* texture coordinate, and as the *center* of the bounding box. The corresponding bit-rates at a precision of 10 bits confirm the different success of these predictions.

Parallelogram predictions across polygons are *only* used when the four texture coordinates belong to the same chart. This is the case when the edge connecting the two polygons is smooth. Applying the parallelogram rule to texture coordinates from different charts (e.g. across a crease edge) would result in a completely random prediction. Instead of performing an unreasonable prediction we fall back to a less successful but at least reasonable predictor. We simply use a *nearby* texture coordinate from the same chart that is connected by an edge as the prediction. If there is no such texture

coordinate, then no reasonable prediction is possible. In this case we predict it to lie in the *center* of the texture image. In Figure 5.11 the possible scenarios are illustrated.

mesh characteristics				8 bit		10 bit		12 bit	
name	c	v	t	bpt	gain	bpt	gain	bpt	gain
lion	120	16302	16652	3.8	77	6.3	69	9.7	60
wolf	35	7068	7234	3.8	76	6.6	67	10.3	57
raptor	79	7454	6984	3.7	77	6.3	68	10.0	58
fish	7	4685	4685	4.2	73	7.0	65	10.7	55
snake	6	11137	11610	2.3	85	3.9	80	6.5	73
horse	5	9199	9988	2.9	82	4.9	76	8.2	66
cat	39	9627	10350	3.0	81	5.0	75	8.2	66
dog	19	6650	6522	3.9	76	6.8	66	10.6	56
average				**3.5**	**78**	**5.9**	**71**	**9.3**	**61**

mesh characteristics				8 bit		10 bit		12 bit	
name	c	v	t	bpt	gain	bpt	gain	bpt	gain
“1398”	1	1487	3133	6.3	61	9.5	53	13.5	44
“1412”	1	1180	2712	7.0	56	10.4	48	14.4	40
“1510”	1	644	1422	6.7	58	9.9	51	13.8	43
“1568”	1	1394	2999	7.2	55	10.5	48	14.4	40
“17”	1	1178	2354	6.6	59	9.9	51	13.8	43
“1814”	1	1145	2475	6.4	60	9.6	52	13.4	44
“1823”	1	1202	2700	6.8	57	10.0	50	14.0	42
“2441”	1	1204	2727	7.0	56	10.5	48	14.5	40
average				**6.8**	**58**	**10.0**	**50**	**14.0**	**42**

Table 5.9: These tables list for each model the number of connected components c, of vertices v, and of texture coordinates t. The achieved compression rates in bits per texture coordinate (bpt) are given at the three common quantization levels of 8, 10, and 12 bits and the compression gain (%) in comparison to uncompressed texture coordinates is reported. The bit-rate for uncompressed texture coordinates is simply the number of quantization bits times two.

5.3.4 Results

It is crucial to the success of our method to compress the correctors with different arithmetic contexts depending on which prediction was performed. Using a single context for all correctors would spoil the potentially low entropy of correctors that are result of more promising *within* and *across* predictions with the anticipated poor outcome of the fall-back predictions. In Table 5.8 we list the percentages with which the different prediction rules were used. Note that *within*-predictions can occur only in polygonal meshes. Furthermore the tables report separate bit-rates for the different prediction

rules, which—as expected— confirm their varying success. In Table 5.9 we reports the performance of our compression scheme at different quantization levels. Since predictive compression mainly predict away the high-order bits, the relative compression gains decrease if more precision (= low bits) is added. The average compression gain achieved by our method is 71 % for our polygonal and 50 % for our triangular test set for texture coordinates quantized at 10 bits of precision.

A similar technique can also be used to compress RGB colors. However, on our test set of colored meshes the parallelogram predictor continuously "over-shot" the variation in color and was outperformed by the simpler nearby predictor. We achieved the best rates of 5.7 (11.4) bits per color at quantization levels of 4 (6) bit per color component by using an average of all possible nearby predictions—this equals a compression gain of 52 (37) % respectively.

It would be worthwhile to investigate whether previously decoded vertex positions can aid the prediction of texture coordinates. Many techniques for (semi-)automatic texture map generation take mesh geometry into account to compute texture coordinates that minimize some distortion metric. The correlation between vertex positions and texture coordinates is high when shape preserving metrics are used that minimize geometric stretch (Sander et al., 2001), angle distortion (Levy et al., 2002), or other intrinsic measures (Desbrun et al., 2002). (Sorkine et al., 2002) establish complete correlation between the two, because they define the texture coordinate mapping through the mesh geometry. Instead of using mesh geometry to define the texture coordinates, we can use it to predict them. This predictor works best if the shape of a polygon in 3D space is similar to its shape in texture space. Although initial results are promising we have to evaluate if the achievable gains are always worth the additional computational effort. For example *space-optimized* texture maps as proposed by (Balmelli et al., 2002) can have a fairly low correlation between texture coordinates and vertex positions.

5.4 Summary

We have presented a simple technique for exploiting polygonal information to improve predictive geometry compression with the parallelogram rule. This scheme is a natural generalization of the geometry coder by (Touma and Gotsman, 1998) to polygon meshes and gives compression improvements of up to 40 percent.

We have also introduced a predictive method for compressing the mapping from corners to properties. Our compression rates improve by a factor of 2 to 10 on previously

reported methods. We have described that information about triangle strips can be used to improve compression so significantly that it can potentially offset the expense of including this triangle strip information with the mesh.

Finally, we have introduced simple prediction rules for efficient compression of texture coordinates that take into account discontinuities in the texture mapping. The average compression gain achieved by our method is 71 % for our polygonal and 50 % for our triangular test set for texture coordinates quantized at 10 bits of precision. To our knowledge this is the first work that rigorously investigates texture coordinate compression in a quantitative, in-depth manner.

5.5 Hindsights

The predictive compressors described in the last three sections make heavy use of arithmetic coding in order to achieve the reported compression rates. In later chapters the model size will increase significantly and we noticed that the careless use of arithmetic coding quickly becomes the computational bottleneck of compression and decompression. There is a reasonable payoff in reducing the number of calls to the arithmetic coder by combining code symbols into larger correlation contexts and compressing them with a single call to the arithmetic coder. Instead of, for example, calling the arithmetic coder for every corner bit individually, it would be more efficient to compress corner bits in groups and combine their correlations into larger probability tables.

The worst investment in computing time is the expensive predictive compression of the low-order bits of the corrective vectors for positions and texture coordinates. While this technique gets nearly all its coding gains from removing the redundancy in correctly predicted high-order bits, it spends most of its time writing down the incompressible low-order bits. The computation times are amplified by our practice of breaking the sequence of corrector bits into smaller chunks to keep the probability tables small, which increases the number of calls to the arithmetic coder. In a recent paper on lossless floating point compression we describe a way of using fewer calls to the arithmetic coder for encoding the correctors (Isenburg et al., 2005a).

Chapter 6

Compression of Hexahedral Meshes

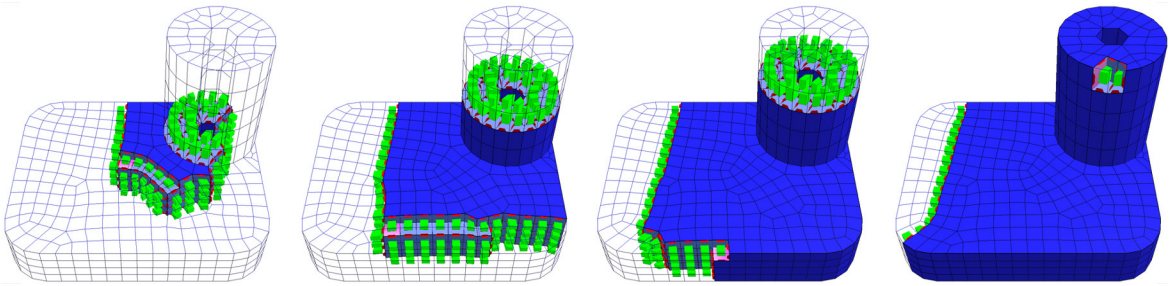


Figure 6.1: An unstructured hexahedral volume mesh during compression.

Unstructured hexahedral volume meshes are of particular interest for visualization and simulation applications. They allow regular tiling of the three-dimensional space and show good numerical behavior in finite element computations. Beside such appealing properties, volume meshes take up huge amounts of space when stored in a raw format. In this chapter we present a technique for encoding connectivity and geometry of unstructured hexahedral volume meshes.

For connectivity compression, we extend the idea of coding with degrees to volume meshes. Hexahedral connectivity is coded as a sequence of edge degrees. This naturally exploits the regularity of typical hexahedral meshes. We achieve compression rates of around 1.5 bits per hexahedron (bph) that go down to 0.18 bph for regular meshes. On our test meshes the average connectivity compression ratio is $1 : 163$. For geometry compression, we perform simple parallelogram prediction on uniformly quantized vertices within the side of a hexahedron. Tests show an average geometry compression ratio of $1 : 3.7$ at a quantization level of 16 bits. These results are significantly better than compression ratios achieved on typical tetrahedral meshes. This has to do with the "natural irregularity" of a tetrahedral mesh whose elements —in contrast to those of a hexahedral mesh—do not permit a regular tiling of the 3D domain.

6.1 Introduction

Unstructured volume meshes can be found in a broad spectrum of scientific and industrial applications including fluid mechanics, thermodynamics and structural mechanics, where such volumetric data is used for both computation and visualization. Traditionally unstructured volume meshes were composed of tetrahedral elements, but recently also other polyhedra have become popular. Especially hexahedral volume meshes are often used, because of their numerical advantages in finite element computations.

The generation of hexahedral meshes turned out to be much more complex than that of tetrahedral meshes, but the research efforts of the last years have produced several efficient techniques (Tautges and Mitchell, 1995; Schneider et al., 1996; Sheffer et al., 1998; Eppstein, 1999; Mueller-Hannemann, 2001). At the same time researchers have proposed strategies for efficient visualization of unstructured volume meshes using screen-based ray-casting (Garrity, 1990; Bunyk et al., 2000; Yang et al., 2000) or object-based sweeping (Wilhelms et al., 1996; Farias et al., 2000; Levy et al., 2001) (see (Farias and Silva, 2001) for a survey about rendering unstructured volume grids).

The basic ingredients of unstructured hexahedral volume meshes are are mesh *connectivity*, which is the incidence relation among the vertices, edges, faces, and hexahedra, mesh *geometry*, which is the 3D position associated with each vertex, and application-specific mesh *properties*, which are the density or pressure values that are typically attached to the vertices. The standard representation for hexahedral meshes uses three floating-point coordinates per vertex to store geometry and eight integer indices per hexahedron to store connectivity. For hexahedral meshes of v vertices and h hexahedra, this requires $96v$ bits for the geometry and $256h$ bits for the connectivity, if standard 4 byte data types are used. The mesh *c1* from our test set has $78,618$ vertices and $71,572$ hexahedra. The storage requirements for geometry and connectivity of this mesh can be estimated as 3.23 megabytes.

For archival and fast transmission of the data more compact representations are beneficial. In order to represent mesh geometry more compactly, each coordinate can be quantized with, for example, 16 bits. For data sets destined to be used in exact computations a loss in precision is obviously not acceptable. Care must be taken to use at least as many precision bits as present in the data without preserving precision that is not there. For volume mesh visualization, quantization is generally not a problem as long as visual artifacts are avoided. In order to represent mesh connectivity more compactly, each index can be specified with $\lceil \log_2 v \rceil$ bits by crossing the byte bound-

aries. For the mesh *c1* this more compact representation still requires 1.69 megabytes. Using the compression technique proposed here, this mesh can be represented at the same quality with less than 84 kilobytes—a compression by a factor of twenty.

Although we only focus on compression of connectivity and geometry, the same technique that is used to compress vertex positions can also be applied to efficiently compress properties. There have been several publications concerning the compression of tetrahedral volume meshes (Szymczak and Rossignac, 1999; Gumhold et al., 1999; Pajarola et al., 1999; Yang et al., 2000), but we are not aware of a compression scheme that can handle hexahedral volume meshes.

Degree-based connectivity coding has previously only been used for surface meshes. It was first proposed by (Touma and Gotsman, 1998) for purely triangular meshes and in Chapter 4 we have described its generalization to polygonal connectivity. In this chapter we show that degree coding can also be extended to volume mesh connectivity. We code the connectivity of a hexahedral mesh mainly as a sequence of edge degrees that is subsequently compressed with an arithmetic coder (Witten et al., 1987). We code the geometry of a hexahedral mesh as a sequence of corrective vectors that are also compressed with arithmetic coding. Whenever possible, we predict a vertex position "within" the side of a hexahedron using a single parallelogram prediction.

6.2 Related Work

Compared to the number of publications on compression of polygonal surface meshes (Deering, 1995; Taubin and Rossignac, 1998; Touma and Gotsman, 1998; Gumhold and Strasser, 1998; Mitra and Chiueh, 1998; Rossignac, 1999; Bajaj et al., 1999; Isenburg and Snoeyink, 2000; Karni and Gotsman, 2000; Alliez and Desbrun, 2001b; Isenburg, 2002; Khodakovsky et al., 2002; Kronrod and Gotsman, 2002; Lee et al., 2002) there are relatively few on compression of polyhedral volume meshes (Szymczak and Rossignac, 1999; Gumhold et al., 1999; Pajarola et al., 1999; Yang et al., 2000). Reason for this is probably that volume meshes are not as widely used as surface meshes. Volumetric data sets are mostly found in scientific and industrial applications.

The immense amount of data required to represent polyhedral volume meshes makes compression even more worthwhile than in the surface case. This is especially true for the connectivity: The standard indexed representation uses 6 indices per vertex for triangular surface meshes, but approximately 12 indices per vertex for tetrahedral volume meshes. And it uses 4 indices per vertex for quadrilateral surface meshes, but

approximately 8 indices per vertex for hexahedral volume meshes.

The challenge to compress the connectivity of tetrahedral volume meshes has first been approached by (Szymczak and Rossignac, 1999). Their "Grow&Fold" technique codes tetrahedral connectivity using only slightly more than 7 bits per tetrahedron for meshes with a manifold border surface. The encoding process builds a tetrahedral spanning tree that is rooted in an arbitrary border triangle. This tree is encoded with 3 bits per tetrahedron that indicate for all faces but the *entry face* whether the spanning tree will continue growing. The boundary of the tetrahedron spanning tree, a triangular surface mesh, has an associated a *folding string* that is represented with 4 bits per tetrahedron. This string describes how to "fold" and occasionally "glue" the boundary triangles of the spanning tree to reconstruct the original connectivity. The indices associated with the "glue" operations lift the bit-rate above 7 bits per tetrahedron, but their rare occurrence introduces only a small overhead.

(Gumhold et al., 1999) have extended their connectivity coder for triangular surface meshes (Gumhold and Strasser, 1998) to tetrahedral volume meshes. Their algorithm performs a space growing process that maintains a *cut-border*, a (possibly non-manifold) triangle surface mesh, that separates at any time the processed tetrahedra from the unprocessed ones. Each iteration of the algorithm processes a triangle on the cut-border either by declaring it a "border" face or by including its adjacent tetrahedron into the cut-border. The latter requires to specify the fourth vertex of the tetrahedron: Either it is a "new vertex" or it is already on the cut-border, in which case a "connect" operation is needed. This operation uses a local indexing scheme to specify the fourth vertex on the cut-border. Because of the order in which the cut-border triangles are processed, the fourth vertex is often very close to the processed triangle, which results in small local indices. The average bit-rate for connectivity is about 2 bits per tetrahedron, a result that has not been challenged since.

Besides coding the mesh connectivity, the authors also describe two approaches to compress mesh geometry. Vertex coordinates are compressed when a vertex is encountered for the first time (e.g. during the "new vertex" operation). The first approach uses pre-quantized vertices, predicts their position as the center of the currently processed cut-border triangle, and codes only a corrective vector. The second approach quantizes a vertex after expressing it in a local coordinate frame whose z-axis is the normal of the currently processed cut-border triangle. In both approaches the resulting 16-bits correction vectors are split into four packages of 4 bits, which are then entropy encoded with separate arithmetic contexts. The authors report that more sophisticated

prediction schemes failed, essentially because "tetrahedral meshes are too irregular to predict vertex coordinates much better than with the proximity information of the connectivity alone." For vertex coordinates that are uniformly quantized to 16 bits of precision, they report an average geometry compression ratio of 1 : 1.6.

(Yang et al., 2000) propose a compression technique for tetrahedral meshes that allows to streamline decoding and rendering of a volume mesh. Their technique can significantly reduce the memory requirements of a ray-casting-based volume mesh renderer. The contribution of tetrahedra to the intersected rays is incrementally composited as they are decompressed. As soon as a decoded tetrahedron is no longer needed it is discarded and its memory is de-allocated. This allows to render compressed tetrahedral meshes without ever having to store a completely uncompressed mesh.

First, they encode the surface formed by the border triangles using a triangle mesh compression scheme (Mitra and Chiueh, 1998). Then, they grow the border surface *inwards* by processing the adjacent tetrahedra using a breadth-first traversal. Similar to (Gumhold et al., 1999) a tetrahedron is encoded by specifying its fourth vertex. In case the fourth vertex was already visited they specify it using one of three different operations instead of the universal "connect" from (Gumhold et al., 1999). When the fourth vertex is connected across a "face" or an "edge", they use a local index into an enumeration of adjacent faces or adjacent edges. Otherwise they use a global "index" into the list of all already visited vertices. The resulting connectivity compression rates are slightly above those of (Gumhold et al., 1999).

Simplification techniques for tetrahedral meshes have been proposed independently by (Staadt and Gross, 1998) and (Trotts et al., 1998). An iterative process collapses edge after edge, thereby removing all tetrahedra incident to them. At each stage it picks the edge whose collapse results in the minimal error according to some cost function. This simplification technique can be used to create a single mesh of a certain resolution, but it also allows to construct a progressive multi-resolution representation from which meshes at various levels of resolution can be extracted on the fly. The latter requires to store a sequence of inverse edge collapse operations, often referred to as *vertex splits*.

A compact and progressive encoding of the sequence of vertex splits was proposed by (Pajarola et al., 1999). Instead of coding each vertex split individually, their *Implant Spray* technique codes entire batches of *independent* refinement operations at once. This reduces the average cost for identifying a split vertex from $O(\log_2 v)$ to $O(1)$. Additionally the *skirt* of each split vertex has to be encoded, which specifies the set of faces that are split. The bit-rates for this progressive representation of tetrahedral

mesh connectivity are reported to be less than 6 bits per tetrahedron. The authors note that the progressive nature of the connectivity encoding suggests that efficient geometry compression should be possible, but no experimental results are given.

6.3 Preliminaries

A *hexahedral mesh* or a *hexahedralization* is a collection of *hexahedra* that intersect only along shared faces, edges, or vertices. A hexahedron is a polyhedron that has six faces, eight vertices, and twelve edges, where each edge is adjacent to two faces, each vertex is adjacent to three faces and each face is a quadrilateral. A face is an *interior face* if it is shared by two hexahedra, otherwise it is a *border face*. Around each edge we find a cycle of faces and hexahedra. An edge is an *interior edge* if all its surrounding faces are interior faces, otherwise it is a *border edge*. A vertex is an *interior vertex* if all its incident edges are interior edges, otherwise it is a *border vertex*. In the following we denote the number of hexahedra with h, the number of faces with $f = f_i + f_b$, the number of edges with $e = e_i + e_b$, the number of vertices with $v = v_i + v_b$, where i stands for interior and b for border. A volume mesh has genus g if one can perform cuts through g closed border loops without disconnecting the underlying volume; such a volume is topologically equivalent to a sphere with g handles.

A volume mesh is *manifold* if each edge has a neighborhood that is homeomorphic to a cylinder or a half-cylinder and each vertex has a neighborhood that is homeomorphic to a sphere or a half-sphere. Edges with half-cylinder neighborhoods and vertices with half-sphere neighborhoods are on the border. The border of a manifold volume mesh is a manifold surface mesh.

The *degree* of an edge is the number of faces adjacent to the edge. For interior edges this corresponds exactly to the number of hexahedra that are adjacent to the edge. For border edges this corresponds to the number of hexahedra that are adjacent to the edge plus the number of border openings. In the manifold case a border edge has only one border opening. The degrees of interior edges tend to have a different distribution (e.g. tend to be higher) than the degrees of border edges.

Two hexahedra are *face-adjacent* if they share a face, *edge-adjacent* if they share an edge but no face, and *vertex-adjacent* if they share only a vertex. A hexahedral mesh may consist of one or more connected components. A component is *face-connected* if there is a path of face-adjacent hexahedra between any two hexahedra. A component is still *edge-connected* if there is at least a path of face- and edge-adjacent hexahedra

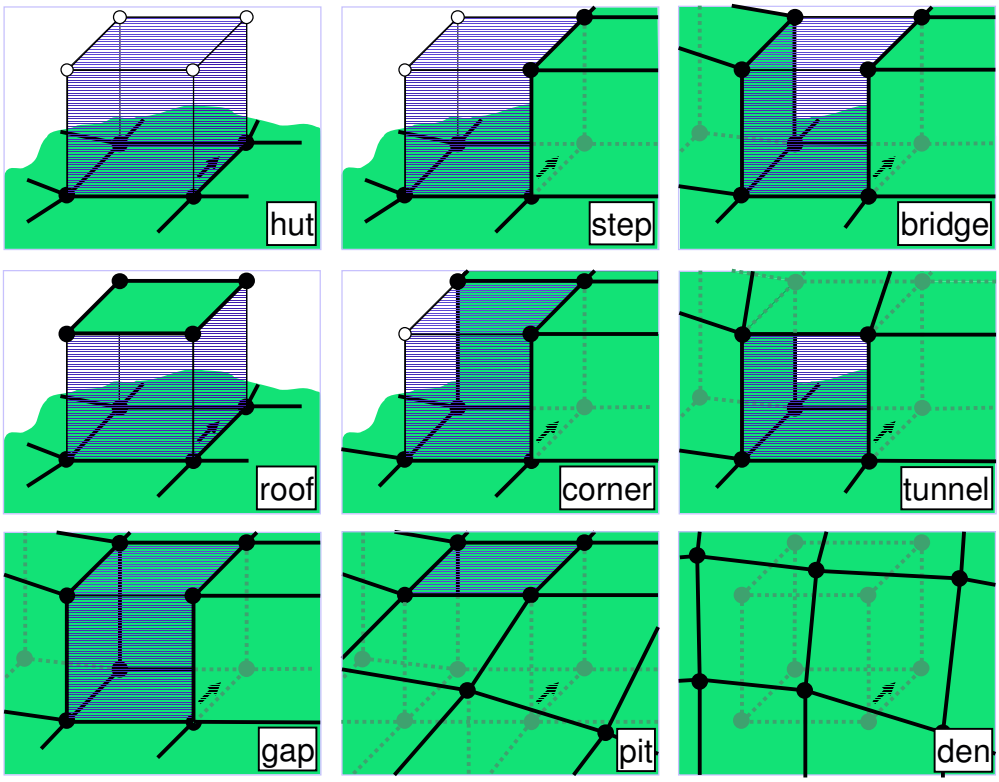


Figure 6.2: The nine different configurations in which a hexahedron (blue) can be face-adjacent to the hull (green). The characteristics of each configuration are summarized in Table 6.1. The faces of the hexahedron that are not adjacent to the hull are its *free faces*, the edges of the hexahedron that are adjacent to two free faces are its *free edges*, and the vertices of the hexahedron that are adjacent to three free faces are its *free vertices*. The focus face is the face on the hull that contains the arrow. It has no zero-slots for the configurations "hut" and "roof", one zero-slot for "step", two zero-slots for the "corner", "bridge", and "tunnel", three zero-slots for "gap", and four zero-slots for "pit" and "den".

between any two hexahedra. Otherwise the component is only *vertex-connected*.

6.4 Coding Connectivity with Degrees

The concept of coding connectivity with degrees was introduced by (Touma and Gotsman, 1998) for the case of triangular surface meshes, which can be coded through a sequence of vertex degrees and occasional "split" symbols. The achieved bit-rates are mainly dictated by the distribution of vertex degrees. This automatically adapts to *regularity* in the mesh, which we loosely define as how regular it tiles the domain it lives in. A surface mesh consisting of only equilateral triangles constitutes a perfectly

regular tiling of the 2D domain. Since the degree of all vertices of such a mesh is 6, the vertex degree distribution has an entropy of zero.

Degree coding was generalized to polygonal connectivity (Isenburg, 2002; Khodakovsky et al., 2002) by using both, a sequence of vertex degrees and a sequence of face degrees. The adaptivity of these coding schemes naturally extends to the other two regular tilings of the 2D domain: using squares, all face and all vertex degrees are 4; and using regular hexagons, all face degrees are 6 and all vertex degrees are 3.

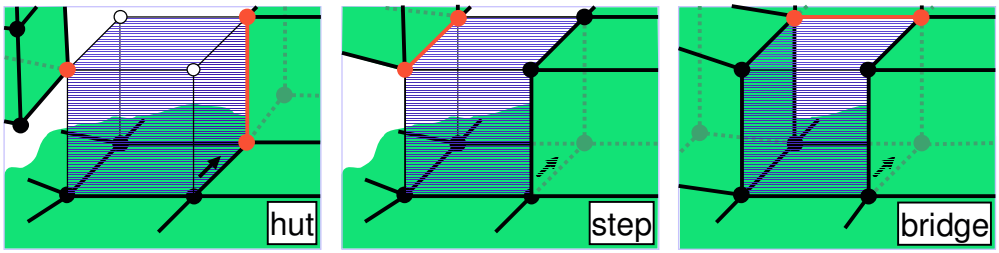


Figure 6.3: The shown "hut" configuration has a *local* edge-adjacency and also a vertex-adjacency with the hull (both marked in red), the "step" configuration has a *global* edge-adjacency, and the "bridge" configuration has a *known* edge-adjacency.

In the following we show that the concept of degree coding can be extended to compress the connectivity of hexahedral meshes using its *edge degrees*. Going from surface meshes to volume meshes we can think of polygons getting stretched into polyhedra, edges getting stretched into polygons, and vertices getting stretched into edges; what was a vertex degree in the surface mesh, becomes an edge degree in the volume mesh.

Hexahedral meshes allow a regular tiling of the 3D domain. A cube is a hexahedron whose six faces are square and meet each other at right angles. It is the only of the five platonic solids that regularly tiles the 3D domain. The interior edges of a perfectly regular hexahedral mesh all have degree 4. Fortunately, many hexahedral meshes found in practice are fairly regular and exhibit a low dispersion in edge degrees. The equilateral tetrahedron, on the other hand, does not permit a tiling of 3D space. In fact, tetrahedral meshes seem irregular by nature. Although degree coding can be adapted for tetrahedral connectivity, initial measurements on the edge degree distributions of various tetrahedral meshes suggests that the achievable compression rates will be lower than those of (Gumhold et al., 1999; Yang et al., 2000).

6.5 Compressing the Connectivity

The encoder and the decoder perform the same space growing process to compress and uncompress a connected component of a hexahedral mesh. Each iteration of the algorithm processes a hexahedron that is adjacent to one or more previously processed hexahedra. In face-connected components this hexahedron is always face-adjacent; in edge-connected or vertex-connected components this hexahedron is sometimes only edge-adjacent or vertex-adjacent. In order to simplify the description of our compression method we assume face-connected components. The two necessary extensions for dealing with components that are only edge-connected or vertex-connected are straightforward.

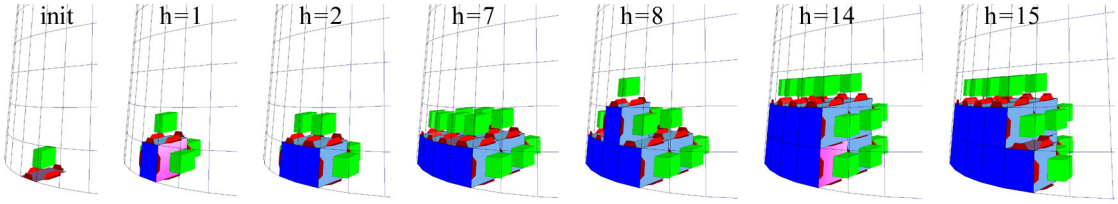


Figure 6.4: A close-up on the *fru* mesh at the beginning of the encoding process. Final faces are dark blue, incomplete faces are light blue, the focus face is pink, the slots are red, and all hexahedra face-adjacent to the hull are shown in green: The leftmost frame shows the initial hull. The next two frames show the hull after processing the first two tetrahedra. The rightmost frame shows the hull after processing 15 tetrahedra.

# of	hut	roof	step	corner	bridge	tunnel	gap	pit	den
adjacent faces	1	2	2	3	3	4	4	5	6
zero-slots	0	0	1	2	2	2	3	4	4
free faces	5	4	4	3	3	2	2	1	-
free vertices	4	-	2	1	-	-	-	-	-
free edges	8	4	5	3	2	-	1	-	-
(global)	4	-	1	-	-	-	-	-	-
(local)	4	-	4	3	-	-	-	-	-
(known)	-	4	-	-	2	-	1	-	-

Table 6.1: This table characterizes the nine configurations in which a hexahedron can be face-adjacent to the hull (see Figure 6.2). It lists the number of adjacent faces, the number of zero-slots of the focus face, and the number of free vertices, free faces, and free edges. The free edges are further classified into the number of potential candidates for global, local, or known edge-adjacency with the hull (see Figure 6.3).

Four arithmetic contexts (Witten et al., 1987) are used for compressing the symbols

that encode hexahedral connectivity. One for border edge degrees, one for interior edge degrees, and two binary contexts. One of the two binary contexts distinguishes border elements from interior elements, and the other marks the infrequent occurrences of "join" operations discussed below.

The algorithm maintains a *hull* that encloses at any time all processed hexahedra. This hull is a quadrilateral surface mesh, possibly non-manifold, whose edges and faces are called *hull edges* and *hull faces* respectively. The hull faces are classified as *final faces* and *incomplete faces*. A final face is a border face whose corresponding hexahedron has already been processed. An incomplete face is an interior face that has a processed hexahedron on one side and an unprocessed hexahedron on the other side. Each hull edge maintains a *slot-count* that specifies the remaining number of faces still to be added between its two adjacent hull faces. A hull edge is a *zero-slot* if its two hull faces are incomplete and its slot-count is zero. A hull edge is a *border-slot* if one of its hull faces is final and the other incomplete and its slot-count is one. The number of zero-slots and border-slots around an incomplete face is always between 0 and 4.

The initial hull is defined around a border face by recording the degrees of its four border edges. It has one final face, one incomplete face, and four hull edges. The slot-count of the hull edges is initialized to their degree minus one. In each iteration the algorithm selects an incomplete face as the *focus face* and processes the unprocessed hexahedron it is adjacent to (see Figure 6.4). Processing of a (face-adjacent) hexahedral mesh component is completed, when the hull consists only of final faces.

The currently processed hexahedron can be in one out of nine configurations face-adjacent to the hull; these are shown in Figure 6.2 and characterized in Table 6.1. Both, encoder and decoder, can determine the actual configuration based on the number of zero-slots in the vicinity of the focus face. Only when the focus face has no zero-slots, the ambiguity between the "hut" and the "roof" configuration needs to be coded explicitly. In case of the latter the encoder also needs to specify the incomplete face that the "roof" is formed with. Processing the hexahedron involves:

- recording if its free faces are border or interior faces;
- recording if its free edges are border or interior edges;
- recording the degrees of its free edges;
- predicting the positions of its free vertices;
- and updating the hull and the slot-counts appropriately.

The edge degree distribution of border edges is different from that of interior edges. While border edge degrees typically have a spread around 3, interior edge degrees average around 4 as documented in Table 6.2. It is therefore beneficial to compress them with different arithmetic contexts.

6.5.1 Propagating the Border Information

The proposed algorithm only needs to distinguish border faces from interior faces. Using this information all edges can eventually be classified as border or interior. However, in order to compress an edge degree with the appropriate arithmetic context, we need know this in the moment its degree is encoded. By using simple rules and by selecting a suitable focus face (see Subsection 6.5.3) we can propagate the information about the border. Most of the time encoder and decoder can deduce whether faces or edges are on the border without explicitly encoding this. The rules are:

rule $\mathbf{R}_1$ A free face is a border face if it connects to a border face across an edge that has a slot-count of zero.

rule $\mathbf{R}_2$ A free face is an interior face if any adjacent edge is known to be an interior edge.

rule $\mathbf{R}_3$ A free edge is a border edge if any adjacent face is known to be a border face.

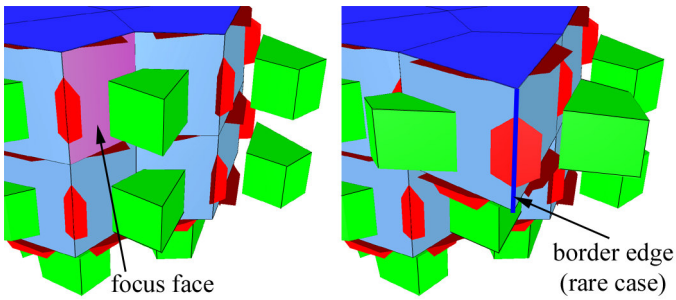


Figure 6.5: Propagating the border information in the "step" configuration: The free face at the top is a border face because of rule R_1. All other free faces are interior faces because of rule R_2. The two free edges at the top are border edges because of rule R_3. For the remaining three free edges this needs to be specified explicitly. Usually these would all be interior edges, but this example shows a rare scenario where one of them is a border edge, along which the included hexahedron "touches" the border.

The example in Figure 6.5 illustrates these rules. Whenever none of the rules applies a binary arithmetic context is used to encode explicitly if an edge or a face is on the border or not. Using the three rules on our test meshes classifies approximately 99

percent of all border elements correctly, so that the arithmetic coder is mainly used to repeatedly specify that an edge or a face is interior. This requires only few bits because the same symbol will be coded again and again.

6.5.2 Join Operations

For every "roof" configuration it is necessary to specify the incomplete face on the hull that forms the "roof". Furthermore, sometimes free edges are edge-adjacent or free vertices are vertex-adjacent to the hull as illustrated in Figure 6.3. Instead of recording the degree of such an edge or predicting the position of such a vertex, the encoder has to specify *how* they are adjacent to the hull such that the decoder can replay exactly the same updates. We use the following "join" operations for this:

Joining free vertices is done by identifying the respective vertex with an index between 0 and the current count of vertices v_{cc} minus one, which can be coded with $\log_2(v_{cc})$ bits. In theory we could improve compression slightly by excluding all interior vertices that have already left the hull from consideration. This would require to maintain all vertices that are eligible for a "join" in some kind of indexable data structure. But due to the regular nature of hexahedral meshes there are relatively few "join" operations, so that the improvement in compression would be small.

Joining free edges is done by identifying the respective hull edge, which has at least two slots, and by specifying how the "join" divides its slot-count.

We identify the respective hull edge in three different ways, depending on the type of edge-adjacency: *known*, *local*, or *global* (see Figure 6.3). For the *known* type we know the two vertices in whose linked lists the respective hull edge must appear. In most cases this will leave us with a unique candidate. For the *local* type we know only one vertex in whose linked list the respective hull edge must appear. Its position in this list is addressed with an index between 0 and the current number of edges $e_{s>=2}$ of this list that have a slot-count of 2 or higher minus one, which is coded with $\log_2(e_{s>=2})$ bits. For the *global* type we must furthermore explicitly address one of the vertices in whose linked list the respective hull edge appears using $\log_2(v_{cc})$ bits. Specifying how the "join" divides the s slots of the respective hull edge can be coded with $\log_2(s-2)$ bits, as 2 slots are consumed during the "join".

Joining the "roof" is done by identifying one of the hull edges of the respective incomplete face. We specify this edge, which has at least one slot, by addressing the

vertex in whose linked list it appears and its position in this list. Addressing the vertex is again coded with $\log_2(v_{cc})$ bits. The position of the respective hull edge in this list is addressed with an index between 0 and the current number of edges $e_{s>=1}$ of this list that have a slot-count s of 1 or higher minus one, which is coded with $\log_2(e_{s>=1})$ bits.

mesh	border edge degrees					interior edge degrees					
name	total	2	3	4	>4	total	2	3	4	5	>5
hanger	768	.17	.77	.06	–	149	–	.01	.98	.01	–
ra	792	.17	.79	.04	–	856	–	.03	.95	.02	–
bump2	1780	.08	.88	.03	.01	2708	–	.04	.94	.01	–
test	2928	.12	.87	.01	–	5774	–	–	1.0	–	–
mdg-1	3004	.06	.94	–	–	9676	–	.01	.98	.01	–
c2	3924	.07	.91	.02	–	10247	–	.02	.96	.02	–
fru	2872	.04	.97	–	–	11689	–	.03	.96	.02	–
shaft	8788	.08	.90	.02	.01	16392	.01	.03	.95	.02	.01
warped	4800	.05	.95	–	–	21660	–	–	1.0	–	–
hutch	2336	.03	.94	.02	–	23381	–	.01	.98	.01	–
c1	27428	.03	.97	.01	–	201190	–	.01	.98	.01	–
average		.08	.90	.02	.00		.00	.02	.97	.01	.00

Table 6.2: This table reports the degree distribution for border and interior edges in our data sets. Border edge degrees spread around 3; interior edge degrees spread around 4.

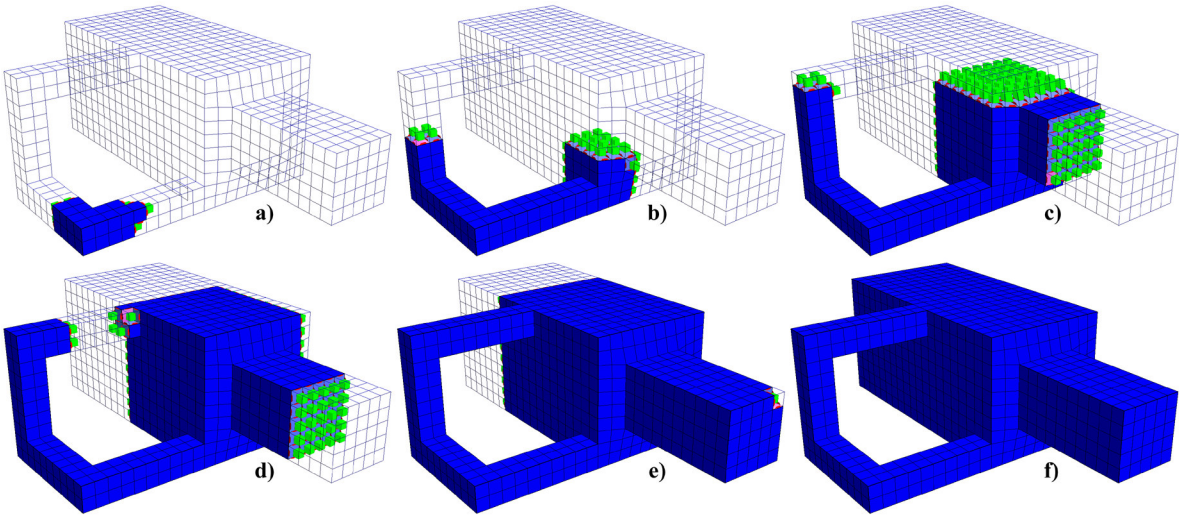


Figure 6.6: Six freeze-frames from the encoding process of the *test* mesh. Final faces are dark blue, incomplete faces are light blue, the focus is pink, and the slots are red. Furthermore all hexahedra face-adjacent to the hull are illustrated in green. Between frames **d)** and **e)** the handle of the mesh is processed with a "roof" configuration.

6.5.3 Reducing the Number of Join Operations

Coding "join" operations requires local or even global indexing. This is expensive and we would like to do this as seldom as possible. If the mesh has handles then there will always be at least one "roof" configuration, one global edge-adjacency, or one vertex-adjacency per handle (see Figure 6.6). Unfortunately these can also happen otherwise and the frequency of their occurrences is strongly dependent on the strategy used for selecting the next focus face. This problem is very similar to the occurrences of "split" operations in surface mesh connectivity coding (Touma and Gotsman, 1998; Gumhold and Strasser, 1998). Adaptive traversal strategies have been proposed that successfully reduce the number of these operations (Alliez and Desbrun, 2001b; Isenburg, 2002).

Such adaptive traversal strategies try to pick a focus that avoids the creation of "cavities" on the compression boundry during the region growing process. They use heuristics that move the focus to vertices on the boundary that are nearly completed (e.g. that have a low slot-count). We use a similar heuristic for avoiding the creation of "cavities" on the hull during our space growing process. Our heuristic moves the focus to the incomplete face with the highest number of zero-slots. This strategy is very successful on our set of hexahedral meshes. Only for one data set, the *hutch* mesh, we need a "join" operation that is not due to a handle. This happens because, as shown in Figure 6.9, during encoding the hull has temporarily the topology of a torus, while subsequent hexahedra completely fill and remove this handle.

In case there is no face with zero-slots, a face with border-slots is selected as the focus face. This increases the success rate of the border propagation described earlier. If there is also no face with border-slots, an arbitrary incomplete face is selected in some way that encoder and decoder agree upon.

6.6 Compressing the Geometry

We use the traversal order on the vertices induced by the connectivity coder to compress their associated positions with a predictive coding scheme. In order to use such a scheme the floating-point positions are first uniformly quantized using a user-defined precision of for example 10, 12, 14, 16, or even 18 bits per coordinate. This introduces a quantization error as some of the floating-point precision is lost. Then a prediction rule is applied that represents each quantized position as an offset vector that corrects the predicted position to the actual position. The values of these corrective vectors

tend to spread around zero, which means they can be efficiently compressed with, for example, an arithmetic coder (Witten et al., 1987). If for some reason it is absolutely not possible to uniformly quantize the floating-point coordinates, a lossless compression scheme can be used instead (Isenburg et al., 2005a).

For predicting the vertex positions of triangle meshes several different methods have been proposed. The simplest prediction method, which predicts the next position as the last position, was suggested by (Deering, 1995). This is also known as delta coding. A more sophisticated scheme is the *spanning tree predictor* by (Taubin and Rossignac, 1998) that uses a weighted linear combination of previously decoded vertices; the particular coefficients used can be optimized for each mesh. A similar, but much simpler scheme is the *parallelogram predictor* introduced by (Touma and Gotsman, 1998). This is the predictor we will use.

vertex	prediction rule
v_0	0
v_1	v_0
v_2	v_1
v_3	$v_0 - v_1 + v_2$
v_4	$2v_0 - v_8$ (*or* v_0)
v_5	$v_1 - v_0 + v_4$
v_6	$v_2 - v_1 + v_5$
v_7	$v_3 - v_2 + v_6$

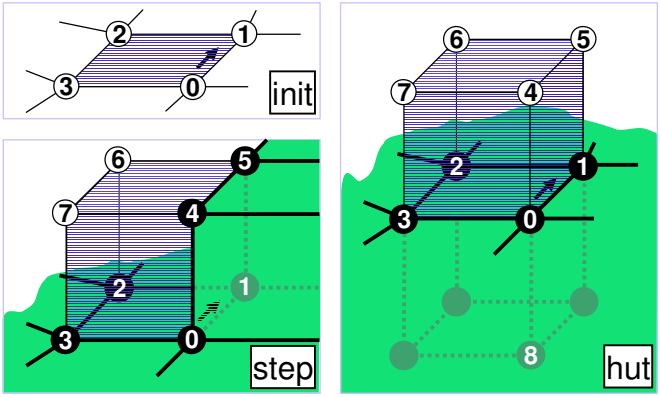


Figure 6.7: This figure illustrates how vertex positions are predicted: The rules for v_0 to v_3 are only used for the vertices of the initial hull. All other vertices are predicted during a "hut", a "step, or a "corner" configuration using the rules for v_4 to v_7. The first "hut" configuration uses a different prediction rule for v_4, since v_8 does not exist.

This scheme predicts vertex positions to complete a parallelogram spanned by the three previously processed vertices. Good predictions are those that predict a position close to its actual location. In the triangle mesh case the parallelogram predictor gives good predictions if used across two triangles that are in a fairly planar and convex position to each other. Consequently, the parallelogram predictor gives poor predictions if used across triangles that are in a highly non-planar and/or non-convex position.

When compressing polygonal meshes it is possible to improve the number of good predictions by letting the polygons dictate where to apply the parallelogram predictor as we saw in Section 5.1. Since polygons tend to be fairly planar and fairly convex, it is beneficial to make predictions *within* a polygon rather than *across* polygons. This,

for example, avoids poor predictions due to a crease angle between polygons.

In similar spirit we predict most vertex positions within the side of a hexahedron in the moment they are first encountered using the rules illustrated in Figure 6.7. Four vertices are encountered during initialization of the hull, all others are encountered as free vertices of a "hut", "step", or "corner" configuration. The first vertex v_0 has no obvious predictor and is predicted as 0. Also the next two vertices v_1 and v_2 cannot yet use parallelogram prediction and are predicted as a previously processed position. This makes a systematic prediction error, but there will be only two such predictions per mesh component. For most following vertex positions we use the parallelogram predictor. An exception is vertex v_4 of the "hut" configuration, which is predicted by extending the ray from v_8 to v_0 (if vertex v_8 exists).

Predictive geometry compression does not scale with increasing precision. The achievable compression ratio is strongly dependent on the number of precision bits. Since this technique mainly predict away the high-order bits, the compression ratios decrease if more precision (= low bits) is added. This is clearly demonstrated by the results in Table 6.3, which reports the performance of our geometry compression scheme at different levels of precision.

mesh	10 bits		12 bits		14 bits		16 bits		18 bits	
name	bpv	ratio	bpv	ratio	bpv	ratio	bpv	ratio	bpv	ratio
hanger	11.2	2.7	15.4	2.3	19.6	2.1	23.2	2.1	26.5	2.0
ra	14.5	2.1	19.9	1.8	25.2	1.7	30.8	1.6	36.2	1.5
bump2	9.5	3.1	14.2	2.5	19.1	2.2	24.4	2.0	29.8	1.8
test	1.8	17.0	3.3	11.0	4.3	9.8	5.9	8.2	6.5	8.3
mdg-1	5.3	5.6	7.7	4.7	10.1	4.2	12.3	3.9	14.4	3.8
c2	5.0	6.0	7.5	4.8	10.7	3.9	14.2	3.4	17.6	3.1
fru	7.1	4.2	12.0	3.0	17.1	2.5	23.1	2.1	29.1	1.9
shaft	6.8	4.4	10.6	3.4	15.2	2.8	19.9	2.4	24.8	2.2
warped	3.4	8.8	5.1	7.1	7.9	5.3	10.5	4.6	13.2	4.1
hutch	8.1	3.7	11.6	3.1	16.1	2.6	19.9	2.4	23.9	2.3
c1	1.5	19.7	2.7	13.3	4.1	10.2	5.9	8.1	8.0	6.8
average		7.0		5.2		4.3		3.7		3.4

Table 6.3: This table reports bit-rates for compressed geometry in bits per vertex (bpv) at different quantization levels and gives the corresponding compression ratio compared to uncompressed geometry. The bit-rate for uncompressed geometry is simply three times the number of precision bits.

```
class SpinEdge {
   Vertex* vertex;
   SpinEdge* next;
   SpinEdge* inv;
   SpinEdge* spin;
   SpinEdge* list;
   int on_border;
   int slots;
}
```

```
class Vertex {
   int index;
   SpinEdge* edge_list;
   float p[3];
}

SpinEdge* face_list[5];
Vertex* permutation[];
```

Figure 6.8: The data structures used for compression: The connectivity of the hexahedral mesh is captured by the `next`, `inv`, and `spin` pointers. The geometry is attached by the `vertex` pointer. The `list` pointer is used for all linked-lists: One list per vertex, starting at the `edge_list` pointers, links all incomplete edges incident to a vertex. Furthermore five lists, starting at the `face_list` pointers, link incomplete faces that have either border-slots, or one, two, three, or four zero-slots.

mesh	mesh characteristics					connectivity (bph)			geometry (bpv)		
name	g	h	v	e	f_b/h	raw	coded	ratio	raw	coded	ratio
hanger	2	171	382	917	2.25	72.0	5.30	13.6	48.0	23.19	2.1
ra	0	408	635	1648	0.97	80.0	2.89	27.7	48.0	30.83	1.6
bump2	1	1189	1665	4480	0.75	88.0	2.10	41.9	48.0	24.41	2.0
test	1	2386	3198	8702	0.61	96.0	0.87	110.3	48.0	5.85	8.2
mdg-1	0	3710	4510	12680	0.40	104.0	0.77	135.1	48.0	12.30	3.9
c2	0	4046	5099	14171	0.48	104.0	1.31	79.4	48.0	14.24	3.4
fru	0	4360	5124	14561	0.33	104.0	0.98	106.1	48.0	23.12	2.1
shaft	0	6883	9218	25180	0.64	112.0	1.70	65.9	48.0	19.93	2.4
warped	0	8000	9261	26460	0.30	112.0	0.18	622.2	48.0	10.45	4.6
hutch	0	8172	8790	25717	0.14	112.0	0.31	361.3	48.0	19.88	2.4
c1	0	71572	78618	228618	0.19	136.0	0.60	226.7	48.0	5.91	8.1
average					0.48	101.8	1.55	162.7	48.0	17.28	3.7

Table 6.4: The table lists the genus g and number of hexahedra h, vertices v, edges e for each of the models shown in Figure 6.9. Furthermore, the number of border faces per hexahedra f_b/h is given as an indicator of the mesh's compacity. The bit-rates for uncompressed and compressed connectivity are reported in bits per hexahedron (bph). The bit rates for uncompressed and compressed geometry at 16 bits of precision are reported in bits per vertex (bpv). The corresponding compression ratios are also listed.

6.7 Implementation and Results

The data structures used by encoder and decoder are shown in Figure 6.8. The spin-edges that store mesh connectivity are a straight-forward extension of standard twin-edges and are similar to those used in (Levy et al., 2001). Each hexahedron uses 24

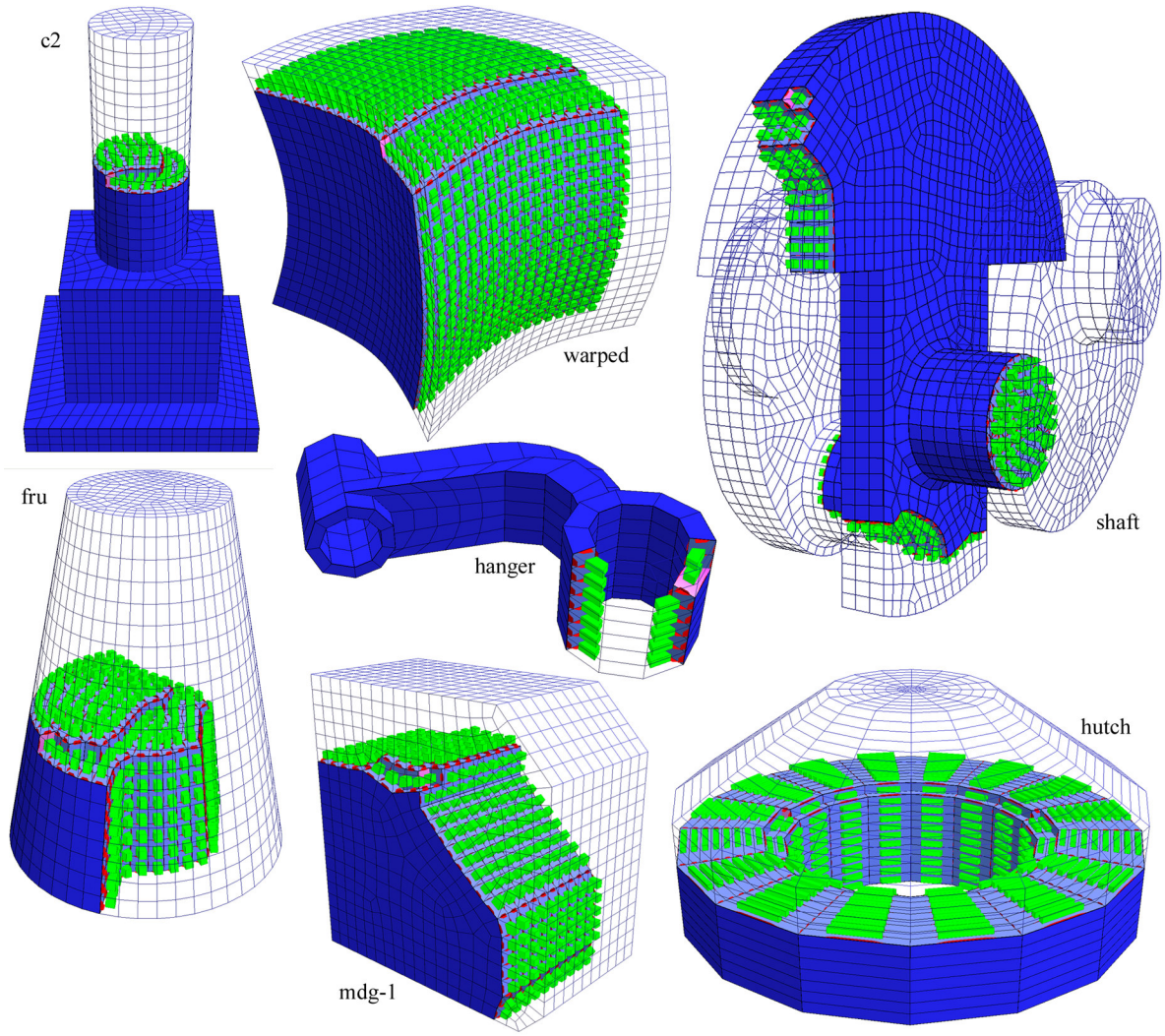


Figure 6.9: The set of example models used to evaluate our compression algorithm. The *bump2* model is used in the teaser on the first page and the *test* model is shown in Figure 6.6. The *c1* model is a much finer tessellated version of the *c2* model.

spin-edges, 4 per face, and also border faces are represented explicitly. This means that every face has two sets of 4 spin-edges, whose `list` pointers are used to maintain two kinds of single-linked lists during encoding and decoding. One set is used to link all spin-edges a vertex has on the hull to its `edge_list` pointer. These lists are used to address an edge during a "join" operation. The other set is used to link spin-edges on the hull that either have border-slots, or one, two, three, or four zero-slots into five priority lists. These five lists are used to select the next focus face. Spin-edges are inserted into and removed from a list at most once. After leaving the hull they are not

explicitly deleted, but marked invalid and removed the next time encountered. Hence, maintaining these lists has a linear time complexity.

This data structure has the advantage that it can represent general volume meshes containing arbitrary elements, such as prisms, pyramids, tetrahedra, or any other polyhedra. For volume meshes that are limited to one element type, like hexahedra in our case, the storage requirements for the data structure can be significantly reduced by storing the 24 spin-edges in of a hexahedron in a pre-defined order in the array of spin-edges. Then the `next` and `inv` pointers can be replaced by a fixed mapping within each block of 24 spin-edges. For the same reason the total number of `vertex` pointers can then be reduced to 8 per hexahedron.

Compression results for connectivity and geometry for a set of eleven test meshes are listed in Table 6.4. The bit-rates for connectivity are strongly dependent on the *compacity* of the mesh, which can be characterized by the ratio of border elements. The fraction of border vertices v_b/v and border edges e_b/e, for example, but also the number border faces per hexahedron f_b/h can be used as a measure of compactness. The less compact a mesh, the bigger the impact of the costs for encoding its border. The *hanger* mesh, for example, is closer to a surface mesh than to a volume mesh. Although its bit-rate of 5.30 bits per hexahedron seems high, expressed as 2.65 bits per vertex it is comparable to results in surface connectivity compression.

6.8 Summary

We have introduced the first scheme for compressing hexahedral volume meshes. The connectivity is coded using an edge-degree based approach that naturally adapts to the regularity typically found in hexahedral meshes. For regular meshes the bit-rates go down to 0.18 bits per hexahedron while averaging at around 1.5 on our test set of eleven meshes, which corresponds to a compression ratio of 1 : 163. The geometry is compressed by parallelogram prediction within a hexahedron, leading to a compression ratio of 1 : 3.7 at a quantization level of 16 bits. Furthermore, we describe a data structure well suited to efficiently implement the selection strategy for the focus face and maintain the hull during encoding and decoding.

We should point out that this compressor could be adapted to work *out-of-core* in a similar way as we will do it for surface meshes in the next chapter. In order to compress large volume meshes that are too large to fit in main memory an external memory data structure would be required that provides similar functionality as the out-of-core mesh

for surfaces that is described in Section 7.3. This would also require to separate the part of data structure from Figure 6.8 that is used to store hexahedral connectivity from the part that is used to maintain the hull. Such an approach would then allow to implement the decoder such that at any time during decompression only the hull is kept in memory. Decompressed hexahedra could, for example, be immediately rendered like it was proposed by (Yang et al., 2000) for tetrahedral meshes.

In the future we plan to generalize the degree-based approach to unstructured volume meshes containing arbitrary polyhedra. The final goal is an universal degree-based coder for irregular surface and volume meshes that obtains bit-rates competitive to those of a specialized coder. However, it seems that this will not be possible for tetrahedral meshes due to their notoriozus irregularity. The high entropy of their edge degree distribution suggests that the coder of (Gumhold et al., 1999) will always outperform a degree-based approach.

6.9 Hindsights

The described data structure for compression is unnecessarily bloated. A more efficient implementation—even for in-core operation—separates the data structure into a static part that stores hexahedral connectivity and a dynamic part that maintains the hull. In order to keep the dynamic part as small as possible, hull edges and hull vertices would then be de-allocated as soon possible. However, the current implementation does not allow us to safely deallocate vertices along the mesh border as they could potentially be referenced again later by hexahedra that are vertex-adjacent to the border in a non-manifold way. This could be fixed by explicitly marking all those border vertices where hexahedra meet in a vertex-adjacent, non-manifold manner.

In Chapter 9 we will see that the strategy with which the focus face is selected should take into account how it affects the global order of the hexahedral mesh elements. In particular, it should avoid giving vertices drastically varying "life times" on the hull. The current traversal strategy does not account for this and leads to coherent orderings only by chance. For the *shaft* model shown in Figure 6.9, for example, the space growing process will eventually stop growing the hull on the left and first complete the right half of the model, giving some hull vertices disproportionally long "life times".

Chapter 7

Out-of-Core Compression

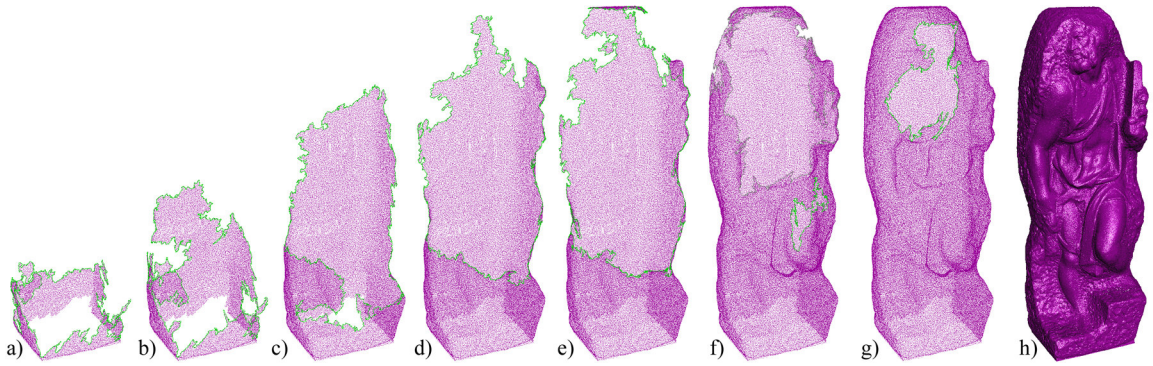


Figure 7.1: **(a) - (g)** Visualization of the decompression process for the St. Matthew statue. The in-core boundary is shown in green. **(h)** Example Out-of-Core Rendering.

Polygonal models acquired with emerging 3D scanning technology or from large scale CAD applications easily reach sizes of several gigabytes and do not fit in the address space of common 32-bit desktop PCs. In this chapter we describe an out-of-core mesh compression technique that converts such gigantic meshes into a streamable, highly compressed representation. During decompression only a small portion of the mesh (the green decompression boundaries shown in Figure 7.1) needs to be kept in memory at any time. As full connectivity information is available along these boundaries, this provides seamless mesh access for incremental in-core processing on gigantic meshes. Decompression speeds are CPU-limited and exceed one million vertices and two million triangles per second on a 1.8 GHz Athlon processor.

A novel external memory data structure provides our compression engine with transparent access to arbitrary large meshes. This out-of-core mesh was designed to accommodate the access pattern of our region-growing single-pass mesh compressor, which - in return - performs mesh queries as seldom and as localized as possible by remembering previous queries as long as needed and by adapting its traversal slightly. The achieved compression rates are state-of-the-art.

7.1 Introduction

Storing large and detailed models using standard indexed polygon mesh formats results in files of gigantic size that consume large amount of disk space. The St. Matthew model from Stanford's Digital Michelangelo Project (Levoy et al., 2000), for example, has over 186 million vertices resulting in more than six gigabytes of data. Transmission of such gigantic models over the Internet consumes hours and even loading them from the hard drive takes minutes. A number of efficient mesh compression schemes have been proposed, but ironically none of these schemes is capable—at least not on common desktop PCs—to deal with meshes of the gigabyte size that would benefit from compression the most. Current compression algorithms and some of the corresponding decompression algorithms can only be used when connectivity and geometry of the mesh are small enough to reside in main memory. Realizing this limitation, (Ho et al., 2001) propose to cut gigantic meshes into manageable pieces and encode each separately using existing techniques. However, partitioning the mesh introduces artificial discontinuities. The special treatment required to deal with these cuts not only lowers compression rates but also significantly reduces decompression speeds.

Up to a certain mesh size, the memory requirements of the compression process can be satisfied using a 64-bit super-computer with vast amounts of main memory. Research labs and industries that create gigabyte sized meshes often have access to such equipment. But to decompress on common desktop PCs, at least the memory footprint of the decompression process needs to be small. In particular, it must not have memory requirements in the size of the decompressed mesh. This eliminates a number of popular *multi-pass* schemes that either need to store the entire mesh for connectivity decompression (Taubin and Rossignac, 1998; Rossignac, 1999) or that decompress connectivity and geometry in separate passes (Isenburg and Snoeyink, 2000; Szymczak, 2002). This leaves us with all *one-pass* coders that can perform decompression in a single, memory-limited pass over the mesh. Such schemes (de-)compress connectivity and geometry information in an interwoven fashion. This allows *streaming* decompression that can start producing mesh triangles as soon as the first few bytes have been read. There are several schemes that could be implemented as one-pass coders (Touma and Gotsman, 1998; Gumhold and Strasser, 1998; Li and Kuo, 1998; Lee et al., 2002).

In this chapter we describe how to compress meshes of gigabyte size in one piece on a standard PC using an external memory data structure that provides transparent access to arbitrary large meshes. Our *out-of-core mesh* uses a caching strategy that accommodates the access pattern of the compression engine to reduce costly loads of

data from disk. Our compressor uses degree coding for the connectivity (see Chapter 4) and linear prediction coding for the geometry (see Chapter 5) to achieve state-of-the-art compression rates. The resulting compressed format allows streaming, small memory foot-print decompression at speeds of more than 2 million triangles a second.

The snap-shots in Figure 7.1 illustrate the decompression process on the St. Matthew statue. For steps (a) to (g) we displayed every 1000th decompressed vertex and the decompression boundaries, while (h) is an example out-of-core rendering. Using less than 10 MB of memory, this 386 million triangle model loads and decompresses from a 456 MB file off the hard-drive in only 174 seconds. At any time only the green decompression boundaries need to be kept in memory. Decompressed vertices and triangles can be processed immediately, for example, by sending them to the graphics hardware. The out-of-core rendering took 248 seconds to complete with most of the additional time being spent on computing triangle normals. These measurements were taken on a standard PC with a 1.8 Ghz AMD Athlon processor and an Nvidia Geforce 4200 card.

This compressed format has benefits beyond efficient storage and fast loading. It is a *better* representation of the raw data for performing certain out-of-core computations on large meshes. Indexed mesh formats are inefficient to work with and often need to be *de-referenced* in a costly pre-processing step. The resulting polygon soups are at least twice as big and, although they can be efficiently batch-processed, provide no connectivity information. Our compressed format streams gigantic meshes through limited memory and provides seamless mesh access along the decompression boundaries, thereby allowing incremental in-core processing on the entire mesh.

The next section summarizes relevant work on out-of-core processing, out-of-core data structures, and mesh compression. In Section 7.3 we introduce our out-of-core mesh and describe how to build it from an indexed mesh. Then, in Section 7.4, we describe the compression algorithm and report resulting compression rates and decompression speeds on the largest models that were available to us. The last section summarizes our contributions and evaluates their benefits for other algorithms that process gigantic polygon meshes.

7.2 Related Work

Out-of-core or external memory algorithms that allow to process vast amounts of data with limited main memory are an active research area in visualization and computer graphics. Recently proposed out-of-core methods include isosurface extraction, surface reconstruction, volume visualization, massive model rendering, and—most relevant to

our work—simplification of large meshes. Except for (Ho et al., 2001), out-of-core approaches to mesh compression have so far received little attention.

The main computation paradigms of external memory techniques are *batched* and *online* processing: For the first, the data is streamed in one or more passes though the main memory and computations are restricted to the data in memory. For the other, the data is processed through a series of (potentially random) queries. To avoid costly disk access with each query (e.g. thrashing) the data is usually re-organized to accommodate the anticipated access pattern. Online processing can be accelerated further by *caching* or *pre-fetching* of data that is likely to be queried (Silva et al., 2002).

Out-Of-Core Simplification methods typically make heavy use of batch-processing. (Lindstrom, 2000) first creates a vertex clustering in the resolution of the output mesh and stores one quadric error matrix per occupied grid cell in memory. Indexed input meshes are first dereferenced into a polygon-soup and then batch-processed one a triangle at a time by adding its quadric to all cells in which it has a vertex. Later, (Lindstrom and Silva, 2001) showed that the limitation of the output mesh having to fit in main memory can be overcome using a series of external sorts.

A different approach for simplifying huge meshes was suggested by (Hoppe, 1998) and (Bernardini et al., 2002): The input mesh is partitioned into pieces that are small enough to be processed in-core, which are then simplified individually. The partition boundaries are left untouched so that the simplified pieces can be stitched back together seamlessly. While the hierarchical approach of Hoppe automatically simplifies these boundaries at the next level, Bernardini et al. simply process the mesh more than once—each time using a different partitioning.

The methods discussed so far treat large meshes different from small meshes as they try to avoid performing costly online processing on the entire mesh. Therefore the output produced by an out-of-core algorithm is usually of lower quality than that of an in-core algorithm. Addressing this issue, (Cignoni et al., 2003) propose an octree-based external memory data structure that provides algorithms with transparent online access to huge meshes. This makes it possible to, for example, simplify the St. Matthew statue from 386 to 94 million triangles using iterative edge contraction.

Albeit substantial differences, our out-of-core mesh is motivated by the same idea: it provides the mesh compressor transparent access to the connectivity and geometry of gigantic meshes. Therefore our compressor will produce the same result, no matter if used with our out-of-core mesh or with the entire mesh stored in-core.

Out-Of-Core Data Structures for Meshes have also been investigated by (McMains et al., 2001). They reconstruct complete topology information (e.g. including non-manifoldness) from polygon soups by making efficient use of virtual memory. Their data structure provides much more functionality than our compressor needs and so its storage requirements are high. Also, using virtual memory as a caching strategy would restrict us to 4 GB of data on a PC and we will need more than 11 GB for the St. Matthew statue. The octree-based external memory mesh of (Cignoni et al., 2003) could be adapted to work with our mesh compressor. It has roughly the same build times and stores only slightly more data on disk. However, their octree nodes do not store explicit connectivity information, which has to be built on the fly. While this is acceptable for a small number of loads per node, the query order of our compressor might require to load some nodes more often—especially if we used their octree-based mesh: its clustering is created through regular space partitioning, which is insensitive to the underlying connectivity, while our clusters are more compact along the surface.

Mesh Compression techniques have always overlooked the memory requirements of the decompression process. So far meshes were moderately sized and memory usage is at most linear in mesh size. However, today's meshes most in need of compression are those above the 10 million vertex barrier. The memory limitation on common desktop PCs allows the decompression process only a single, memory-limited pass over such meshes. This eliminates all schemes that need to store the entire mesh for connectivity decompression (Taubin and Rossignac, 1998; Rossignac, 1999) or that decompress connectivity and geometry in separate passes (Isenburg and Snoeyink, 2000; Szymczak, 2002). Naturally, this constraint also prohibits the use of progressive approaches that require random mesh access for refinement operations during decompression (Taubin et al., 1998a; Cohen-Or et al., 1999; Pajarola and Rossignac, 2000; Alliez and Desbrun, 2001a). And finally, the sheer size of the data prohibits computation-heavy techniques such as traversal optimizations (Kronrod and Gotsman, 2002), vector quantization (Lee and Ko, 2000), or expensive per-vertex computations (Lee et al., 2002).

This leaves all those methods whose decompressor can be restricted to a single, memory-limited, computation-efficient pass over the mesh. This coincides with all those methods whose compressor can be implemented as a fast one-pass coder (Touma and Gotsman, 1998; Gumhold and Strasser, 1998; Li and Kuo, 1998). These compression algorithms require access to explicit connectivity information, which is usually constructed in a pre-processing step. However, if the mesh does not fit into main memory this is already not possible. Therefore, (Ho et al., 2001) suggest to cut large meshes

Figure 7.2: Less than 1 MB of memory is used by the out-of-core process that loads, decompresses, and renders this 82 million triangle "Double Eagle" model in 78 seconds from a 180 MB file. We do not store triangles in memory, but immediately render them with one call to glNormal3fv() and three calls to glVertex3iv(). Loading and decompression alone takes only 63 seconds. Most of the additional 15 seconds are spent on computing triangle normals. This frame was captured in anti-aliased 2048x768 dual-monitor mode on a 1.8 Ghz AMD Athlon processor with an Nvidia Geforce 4200.

into smaller pieces that can be dealt with in-core. They process each piece separately by first constructing explicit connectivity, which is then compressed with the two-pass coder of (Rossignac, 1999), before compressing the vertex positions with the parallelogram predictor of (Touma and Gotsman, 1998) in a third pass. They record additional information that specifies how to stitch the pieces back together after decoding.

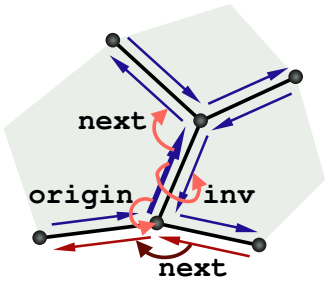
The compression scheme we describe in this chapter has several advantages over that of (Ho et al., 2001). As we do not break up the model, our compression rates are 20 to 30 percent better. As we can decode the entire model in a single pass, our decompression speeds are about 100 times faster. Finally, as our decompressor streams the entire mesh through main memory with a small memory foot-print, our compressed representation is useful beyond reduced file sizes and shortened download times. It supports efficient batch-processing for performing computation on large meshes while at the same time providing seamless access to mesh connectivity.

We should also mention *shape compression* methods (Khodakovsky et al., 2000; Gu et al., 2002; Szymczak et al., 2002; Khodakovsky and Guskov, 2004) as they are especially well suited for converting detailed scanned datasets into highly compressed representations. These approaches remesh prior to compression under the assumption that not a particular mesh but rather the geometric shape that it represents is of interest. Currently such scheme can only operate on models small enough to fit in memory, although the approaches to out-of-core processing presented here and in the following two chapters may change this in the future. However, remeshing methods are not applicable to CAD data such as the "Double Eagle" model (shown in Figure 7.2).

7.3 Out-of-Core Mesh

We use a half-edge data structure (Mantyla, 1988) as foundation for our out-of-core data structure, because it gives us the functionality needed by the compression algorithm at minimal storage space consumption. A static data structure with an array V of vertices and an array of half-edges H is basically sufficient. We provide the compression algorithm with the following (see also Figure 7.3):

1. enumeration of all half-edges and ability to mark them as visited
2. access to the next and the inverse half-edge, and to the origin vertex
3. access to the position of a vertex and whether it is non-manifold
4. knowledge of border edges




```
struct IndexPair {
      int ci : 15;
      int li : 17;
};
```

```
struct HalfEdge {
      Index origin;
      Index inv;
      Index next;
      Index myself;
};
```

Figure 7.3: At each edge (black lines) of the mesh two directed half-edges (blue arrows) are incident, one for each incident face (light grey background). From each half-edge the `next` and `inv`erse half-edges and the `origin` vertex are accessible. At the border additional border edges (red arrows) are created. Following their `next` pointers (dark red) cycles around the border loop. On the right is the syntax of an index-pair and of a half-edge. The `next` index-pair is used in explicit mode and for all border edges. The `myself` index-pair is only used for crossing half-edges.

7.3.1 Half-Edge Data-Structure

In order to efficiently support pure triangular meshes but also accommodate general polygonal meshes we have two modes for the out-of-core data structure: The *implicit mode* is designed for pure triangular meshes. Each internal half-edge consists of an index of its inverse half-edge and an index of its origin vertex. The three half-edges of a triangle are stored in successive order in the half-edge array H, such that the index of the next half-edge can be computed from the half-edge index `i` via $\texttt{next(i)} = 3 * (\texttt{i}/3) + (\texttt{i} + 1)\%3$. The *explicit mode* is used for polygonal meshes. A `next` index is explicitly stored with each half-edge, which means they can be arranged

in any order in H. The vertex array V contains the three coordinates x, y and z of each vertex in floating point or as a pre-quantized integer. In addition to V and H, a bit array for each vertex and each half-edge are necessary to maintain the status of manifoldness and visitation respectively. Border edges are also kept in a separate array. They always store an explicit index to the next border edge.

7.3.2 Clustering

The maximally used in-core storage space of the out-of-core mesh is limited to a user defined number of S_{incore} bytes. For efficient access to the mesh data during compression, a flexible caching strategy is necessary. For this we partition the mesh into a set of clusters. The total number of clusters c_{total} is

$$c_{\text{total}} = \frac{c_{\text{cache}}}{S_{\text{incore}}} \cdot S_{\text{vtx}} \cdot v, \tag{7.1}$$

where c_{cache} is the maximal number of simultaneously cached clusters, v is the number of mesh vertices, and S_{vtx} the per vertex size of our data structure. There are about six times as many half-edges as vertices, so S_{vtx} sums up to 60 bytes per vertex in implicit mode. For the St. Matthew model compressed with $S_{\text{incore}} = 384\text{MB}$ and $c_{\text{cache}} = 768$ this results in $c_{\text{total}} = 21381$ clusters.

Index-Pairs After clustering vertices and half-edges they are re-indexed into so-called *index-pairs* (c_i, l_i) consisting of a cluster index c_i and a local index l_i. If possible, the index-pair (c_i, l_i) is packed into one 32-bit index to reduce the required storage space for the half-edge data structure. The number of bits needed for the cluster index is simply $\lceil \log_2 c_{\text{total}} \rceil$. For the St. Matthew example this is 15 bits, which leaves 17 bits for the local indices. A perfectly balanced clustering needs about $6 \cdot v / c_{\text{total}}$ different local indices for the half-edges. For the St. Matthew model this would be $52,482$. As we would like to use no more than $2^{17} = 131,072$ local indices, a sophisticated clustering approach that achieves well-balanced cluster sizes is inevitable.

Caching Strategy For efficient access to the out-of-core mesh we cache the clusters with a simple LRU strategy. The vertex data, the half-edge data, and the binary flag data of a cluster are kept in separate files because they are accessed differently: The vertex data—once created—is only read. The half-edge data is both read and written when the out-of-core mesh is built, but only read when later queried by the compressor. The only data that needs to be read and written at compression time are the binary flags that maintain the visitation status of half-edges. We maintain separate caches

for this data, each having c_{cache} entries. For a given mesh traversal the quality of the caching strategy can be measured as the quotient $Q_{\text{read}}/Q_{\text{write}}$ of read/written clusters over the total number of clusters. For a full traversal the minimum quotient is one.

7.3.3 Building the Out-of-Core Mesh

Given the input mesh in an indexed format, we build the out-of-core mesh in six stages, all of them restricted to the memory limit S_{incore}:

1. vertex pass: determine the bounding box
2. vertex pass: determine a spatial clustering
3. vertex pass: quantize and sort the vertices into the clusters
4. face pass: create the half-edges and sort them into the clusters
5. matching of incident half-edges
6. linking and shortening of borders, search for non-manifold vertices

First Vertex Pass Each of the three vertex passes reads and processes the vertex array one time sequentially. In the first pass we only determine the number of vertices and the bounding box of the mesh. It can be skipped if this information is given. The required in-core storage for this pass is negligible.

Second Vertex Pass In this pass we compute a balanced, spatial clustering of the vertices into c_{total} clusters similar as (Ho et al., 2001). We subdivide the bounding box into a regular grid of cubical cells and count for each cell the number of vertices falling into it. Only for non-empty cells we allocate counters and keep them in a hash map. This ensures linear storage space consumption in the number of occupied cells. Then we partition the non-empty cells are into c_{total} compact clusters of balanced vertex counts using a graph partitioning package (MeTiS, v 40). As input we build a $k-$nearest neighbor graph on the centroids of occupied cells using an approximate nearest neighbor package (ANN, v 02) (with $k = 6$ and 1% precision) and weigh its vertices using the vertex counts of the associated cells. (Ho et al., 2001) derive the graph by connecting cells that are shared by a face. This could potentially give better cluster locality along the surface but would require an additional—expensive—face pass.

The second block in Table 7.1 shows results of cluster balancing. The time to build and cluster the graph is negligible. The standard deviation of the cluster sizes is fairly

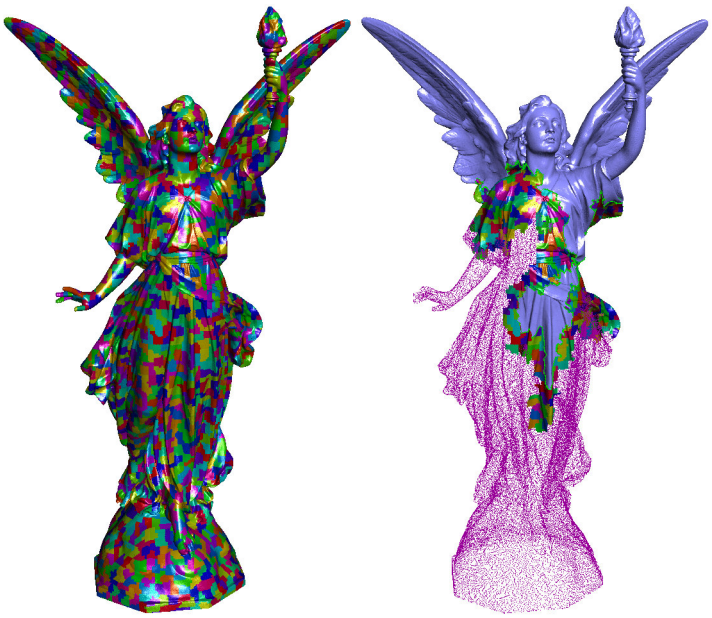


Figure 7.4: Visualization of the clustering and its usage during compression on the Lucy statue with $S_{\text{incore}} = 64MB$ and $c_{\text{cache}} = 128$. Already encoded regions are rendered with points while the rest is rendered with triangles. Cached clusters are colored.

small and the minimum and maximum are within 10% of the average, which is sufficient for our needs. Figure 7.4 illustrates an example clustering on the Lucy statue.

Third Vertex Pass In the final pass over the vertices we sort the vertices into clusters and determine their index-pairs (c_i, l_i) using the cell partitioning generated in the last pass. Since the vertices of each cluster are stored in separate files, we use a simple buffering technique to avoid opening too many files at the same time. If a vertex falls into cluster c_i, which already contains k vertices, we assign it index-pair (c_i, k), increment k, and store its position in the respective buffer. If a buffer is full, its contents are written to disk. The mapping from vertex indices to index-pairs is stored in a *map file* that simply contains an array of index-pairs. For the St. Matthew model the map file is 729 MB and cannot be stored in-core.

Face Pass There is only one pass over the faces. We read a face and map its vertex indices to vertex index-pairs according to the map file. Then we create one half-edge for each of its edges, determine a suitable cluster, store them in this cluster, and—if necessary—also store them in some other cluster.

For each cluster c_i we create two files of half-edges. The *primary half-edge file* stores the half-edges sorted into cluster c_i within which they are locally indexed with l_i in the same way as vertices. The *secondary half-edge file* is only temporary. It stores copies of

half-edges from other clusters that are needed later to match-up corresponding inverse half-edges. These so called *crossing half-edges* are incident to a half-edge of cluster c_i but reside in a different cluster. They are augmented by their own `myself` index-pair (see Figure 7.3) that is used later for matching the `inv` index-pairs.

As the map file is too large to be stored in-core, we split it into segments that are cached with a LRU strategy. For our test meshes the vertex indices of the faces were sufficiently localized, such that the read quotients Q_{read} of the map file cache was between 1 and 1.5. Cache thrashing will occur when the indices of the faces randomly address the vertex array. Then the mapping from indices to index-pairs needs to be established differently. One possibility is to perform several face passes, while each time storing a different chunk of the map file in memory and mapping only the stored indices. For the St. Matthew model three face passes would be sufficient when a chunk size of 256 MB is used. Another possibility would be to re-index the faces with three external sorts as proposed by (Lindstrom and Silva, 2001).

Before writing the half-edges to file, we store the index-pair of its origin vertex in the `origin` field and the index-pair of its target vertex in its `inv` field. The latter is only temporary and will be used during matching. Depending on the mesh mode we sort the half-edges differently into the primary and secondary half-edge files. In both modes, crossing half-edges receive their `myself` index-pair based on the cluster in which they are stored.

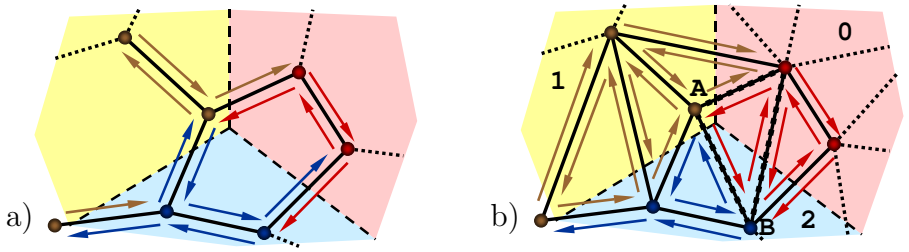


Figure 7.5: The sorting of the half-edges into the clusters. a) In explicit mode each half-edge is sorted into the cluster of its origin vertex. b) In implicit mode all half-edges of a triangle have to be in the same cluster, which is the cluster in which two or more vertices reside or any of three clusters otherwise.

In *explicit mode* the half-edges can be arranged arbitrarily within a cluster. We sort each half-edge into the cluster of its origin vertex. In this mode a half-edge is crossing when it has its target vertex in a different cluster. As they potentially have a matching inverse half-edge there, we insert them into the secondary file of that cluster. A small example is given in Figure 7.5a. The colors of the half-edges show in which of the three clusters they are sorted.

In *implicit mode* all three half-edges of a triangle must be in successive order in the same cluster. They are placed into the cluster in which the triangle has two or three of its vertices. In case all three vertices fall into different clusters we simply select one of them. Figure 7.5b shows an example, where the dashed triangle spans three clusters. The so called *external half-edges*, like the one from vertex A to vertex B, will require special attention later, because they are stored in a different cluster than either of their incident vertices. Again, a half-edge is crossing when its origin vertex and its target vertex are in different clusters. However, in implicit mode it is not obvious in which cluster its potential inverse match will be located. Therefore the secondary files are created in two stages. First we write crossing half-edges into a temporary file based on the smaller cluster index of their end vertices. Then we read these temporary files one by one and sort the contained crossing half-edges using their origin and target index-pairs ordered by increasing cluster index as key. Remember, the index-pair of the target vertex was stored in their `inv` field. Now all potential inverse matches among crossing half-edges are in successive order. Finally, all matching half-edges are entered into the secondary file of the cluster of their inverse, which can be determined from their `myself` index-pairs.

Matching of Inverse Half-Edges For each cluster we read the half-edges from the primary and the crossing half-edges from the secondary half-edge file. With the target vertex index-pairs in the `inv` fields, we again use the sorting strategy for matching inverse half-edges. We reduce the run time for sorting the half-edges with a single bucket-sort over all edges followed by a number of quick-sorts over the edges of each bucket. This results in a sort time of $O(n \log d_{\max})$, where n is the number of half-edges and $d_{\max}$ is the maximum vertex degree—usually a small constant. If origin and target vertex of an edge are both from the current cluster, the key used in the bucket-sort is the smaller of their local indices. Otherwise it is the local index from whichever vertex is in the current cluster. The key used in the quick-sorts is the index-pair of the vertex not used in the bucket-sort. External edges constitute a special case as they do not have any vertex in the current cluster necessary for the bucket-sort. These very rare external edges are gathered in a separate list and matched in the end using a single quick-sort.

All half-edges with the same vertex index-pairs have subsequent entries in the sorted array of half-edges. Looking at their number and orientation, we can distinguish four different types of edges:

1. border edge: an unmatched half-edge
2. manifold edge: two matched half-edges with opposite orientation
3. not-oriented edge: two matched half-edges with identical orientation
4. non-manifold edge: more than two matched half-edges

In case of a manifold edge we set the inverse index-pairs. In all other cases we pair the half-edges with newly created border edges, thereby guaranteeing manifold connectivity. This is similar to the cutting scheme proposed by (Guéziec et al., 1998).

Border Loops and Non-Manifold Vertices The final three steps in building the out-of-core mesh consists of linking and shortening border loops and of detecting non-manifold vertices. First we cycle for each border half-edge via `inv` and `next` around the origin vertex until we hit another border half-edge. Its `next` field is set to the half-edge we started from. This links all border loops.

The second step can shorten border loops that are the result of cutting non-manifold edges. We iterate again over all border half-edges, this time checking if a sequence of `next`, `next`, and `origin` addresses the same vertex as `origin`. In this case we can match the `inv` fields of their incident half-edges and discard the border half-edges. This can shorten or even close a border loop.

The third and last step detects and marks non-manifold vertices using two binary flags per vertex and one binary flag per half-edge. Each flag is administered in one LRU-cached file per cluster with a bit container holding as many bits as there are vertices/half-edges in the cluster. The first vertex flag specifies whether a vertex was visited before, the second whether a vertex is non-manifold, while the half-edge flag marks visited half-edges. For each non-manifold vertex we also maintain an occurrence counter. We iterate over all half-edges. If the current edge has not been visited before, we cycle via `inv` and `next` around its origin and mark all out-going edges as visited until we come back to the edge we started. Then we check if the visited flag of the origin vertex has already been set. If yes, we mark this vertex as non-manifold using the second flag and create or increase its occurrence counter. If no, we set its visited flag. This way we mark all types of non-manifold vertices including those, which cannot be found along non-manifold edges.

7.3.4 Results

Performance results of the out-of-core mesh are gathered in Table 7.1 for Lucy, David (1mm), and St. Matthew. The in-core memory was restricted to 96/192/384 MB and

mesh name	lucy	david (1mm)	st. matthew
vertices	14,027,872	28,184,526	186,836,665
in-core storage limit	96 MB	192 MB	384 MB
cached clusters	192	384	768
clusters	1,605	3,225	21,381
out-of-core size	871 MB	1.7 GB	11.2 GB
counter grid resolution	[283,163,485]	[340,196,856]	[409,1154,373]
ANN nearest neighbor	0:00:02	0:00:05	00:00:18
METIS graph partitioning	0:00:03	0:00:08	00:00:33
min vertices per cluster	8,569	8,550	8,360
max vertices per cluster	8,922	8,907	9,223
std over all clusters	0.00583	0.00529	0.01232
first vertex pass	0:00:14	0:00:34	0:03:24
second vertex pass	0:00:20	0:00:49	0:04:34
third vertex pass	0:00:51	0:02:04	0:53:56
face pass	0:05:22	0:11:01	2:09:20
matching	0:08:39	0:14:06	2:31:46
border link & shorten	0:00:01	0:00:39	0:09:06
non-manifold marking	0:03:26	0:06:36	1:02:06
total build time	0:18:57	0:35:52	6:54:17
compression time	0:48:46	0:13:42	3:36:24
Q_{read} half-edges	11.0	1.3	2.1
precision	20 bits	20 bits	20 bits
compressed size	47 MB	77 MB	456 MB

Table 7.1: Four blocks of measurements that characterize the out-of-core mesh: global parameters, performance of clustering stage, timings for different building steps, compression statistics. Times are in h:mm:ss taken on a Windows PC with 1 GB of memory and a 2.8 GHz Pentium IV processor. The system cache was software disabled.

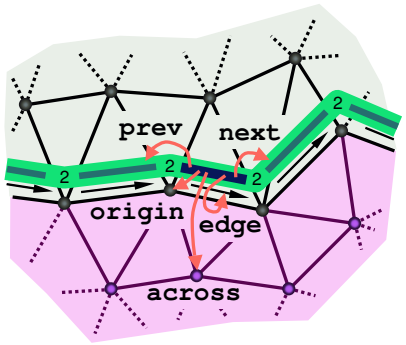
we allowed 192/384/768 clusters to be cached simultaneously. The resulting out-of-core meshes consumed 0.8/1.7/11.2 GB on disk with the build times being dominated by the face pass and the inverse half-edge matching.

The best compression times are achieved when enough clusters are cached to cover the entire compression boundary. But since its maximal length is not known in advance, this cannot be guaranteed. If too few clusters are cached, the compression process becomes heavily IO-limited. However, even then compression times are acceptable given the small in-core memory usage. Lucy, for example, has a poor cache quality factor Q_{read} of 11.0. Although Q_{read} for Lucy is much better with $c_{\text{cache}} = 384$, handling the larger number of files results in an overall longer running time. Twice the number of clusters as MB of in-core storage seemed a good trade-off between the two. When

increasing S_{incore} to 128 MB and caching 256 clusters, then Q_{read} is 2.1 and it takes only about 5 minutes to compress the Lucy model.

7.4 Compression

In order to enable fast out-of-core decompression with small memory foot-print, our mesh compressor performs a single pass over the mesh during which both connectivity and geometry are compressed in an interleaved fashion. It grows a region on the connectivity graph by including faces adjacent to its boundaries one by one. Whenever a previously unseen vertex is encountered, its position is compressed with a linear prediction. The decompressor can be implemented such that at any time it only needs to have access to the boundaries of this region. In order to decompress out-of-core these boundaries are equipped with some additional information. Maintaining such extra information also at compression time reduces the number of required queries to the out-of-core mesh.




```
struct Boundary {
   BoundaryEdge* gate;
   int length;
   int zero_slots;
   int one_slots;
};
```

```
struct BoundaryEdge {
   BoundaryEdge* prev;
   BoundaryEdge* next;
   Index edge;
   bool border;
   Index origin;
   Index across;
   int slots;
};
```

Figure 7.6: The minimal data structures required for out-of-core compression and decompression: The `prev` and `next` pointers organize the boundary edges into double-linked loops. The `edge` index refers to the mesh edge that a boundary edge coincides with. The `origin` index refers to the vertex at the origin of this edge. The `across` index is used for (de-)compressing vertex positions with the parallelogram rule. It refers to the third vertex of an adjacent and already (de-)compressed triangle. For an out-of-core version of the decompressor, the indices `origin` and `across` are replaced with vectors containing copies of the actual positions. The `slots` counter and the `border` flag are required for (de-)compressing the mesh connectivity with degree coding. The `Boundary` struct is used to store information about boundaries on the stack.

7.4.1 Connectivity Coding

Our connectivity coder is based on the degree coder by (Touma and Gotsman, 1998) that was extended to polygon meshes in Chapter 4. Starting from an arbitrary edge

it iteratively grows a region by including the face adjacent to the *gate* of the *active boundary*. This boundary is maintained as loop of *boundary edges* that are doubly-linked through a *previous* and a *next* pointer. Each boundary edge maintains a *slot* count that specifies the number of unprocessed edges incident to its *origin*. The boundary edges also store an index to their corresponding half-edge in the mesh, which is used for queries. If the compressor is used in conjunction with the out-of-core mesh we want to make as few queries as possible. Therefore each boundary edge keeps a copy of the index to the *origin* and the *across* vertex as illustrated in Figure 7.6.

The face to be included shares one, two, or three edges with the active boundary. If it shares three edges, the boundary ends and a new boundary is popped from the stack. If the stack is empty we iterate over the half-edges of the mesh to find any remaining components. If there are none, compression is completed. If the face shares two edges with the active boundary, no explicit encoding is needed. Otherwise it shares only one edge and has a *free vertex*, which can lead to three different cases: *add*, *split*, or *merge*.

In the most common case the free vertex is not on any boundary. Here we *add* the vertex to the boundary, record its degree, and update the slot counts. A more complex case arises if the free vertex is already on some boundary. If it is on the *active* boundary it *splits* this boundary into two loops that will be processed one after the other. A stack is used to temporarily buffer boundaries. We record the shorter direction and the distance in vertices along the boundary to reach the free vertex. If, however, the free vertex is on a boundary from the stack it *merges* two boundaries. This happens exactly once for every topological handle in the mesh. In addition to how the free vertex can be reached starting from that boundary's gate, we record the index of this boundary in the stack. The compressor does not query the out-of-core mesh to search for a free vertex along the boundaries. It uses the *origin* indices, which are stored (mainly for this purpose) with each boundary edge, to find this vertex.

The resulting code sequence contains the degree of every vertex plus information associated with the occurrence of split and merge operations. While it is not possible to avoid splits altogether, their number can be significantly reduced using an adaptive region growing strategy (Alliez and Desbrun, 2001b). Instead of continuing the mesh traversal at the current gate, one selects a gate along the boundary that is less likely to produce a split. We implemented the simpler heuristic proposed in Section 4.4, which picks a boundary edge with a slot count of 1 if one exists, or stays where it is otherwise. However, we restrict this adaptive conquest to +/- 10 edges along the boundary, as moving the gate to a more distant location could cause a cache-miss on the next query

to the out-of-core mesh. By keeping track on the number of these *one-slots* currently on the boundary we avoid searching for them unnecessarily.

Continuing compression on the smaller of the two boundary loops resulting from a split operation keeps the boundary stack shallow and the number of allocated boundary edges low. This helps further lowering the memory foot-print of the decoding process.

Holes in the mesh require special treatment. in Section 4.1 we suggested to include a hole into the active boundary in the same way it is done for faces. However, this requires immediate processing of all vertices around the hole. Since holes can be as large as, for example, the hole at the base the St. Matthew statue shown in Figure 7.1, this would result in an bad access pattern to the out-of-core mesh—potentially causing many cache-misses. Furthermore, it would lead to poor geometric predictions for all vertices around the hole since the parallelogram rule could not be applied.

Instead, similar to (Gumhold and Strasser, 1998), we record for every edge whether it is a border edge or not in the moment it joins the active boundary using a binary arithmetic context. If an edge has a slot count of zero on either end, we do not need to record this information explicitly. In this case the edge will be of the same type as the boundary edge it connects to via this *zero-slot*.

Non-manifold vertices are present when the neighborhood of a vertex is not homeomorphic to a disk or a half-disk. The out-of-core mesh provides our compressor with manifold connectivity and marks multiple occurrence of originally non-manifold vertices. We encode how to *stitch* these vertices back together using a slightly more involved variation of the simple approach that was outlined in Section 4.3. Whenever a vertex is processed with an *add* operation we record whether it is manifold or not with a binary arithmetic context. For non-manifold vertices we specify whether this is its first appearance using a second arithmetic context. Only the first time a non-manifold vertex is encountered its position is compressed. These first-timers are then inserted into an indexable data structure. Each subsequent time this vertex makes a non-manifold appearance it is addressed with $\log_2 k$ bits among the k entries of that data structure. Then a third binary context is used to specify whether this was its last appearance or not. If yes, it is deleted from this indexable data structure.

7.4.2 Geometry Coding

Quantization of the vertex positions into integer values is needed before they can be efficiently compressed with predictive coding. But especially for large datasets any

mesh		number of samples per millimeter				
name	extent [m]	16 bit	18 bit	20 bit	22 bit	24 bit
happy buddha	0.2	327	1311	5243	20972	83886
lucy	1.6	41	164	655	2621	10486
david	5.2	13	50	202	807	3226
double eagle	195	0.3	1.4	5.4	22	86
st. matthew	2.7	24	97	388	1553	6214

Table 7.2: This table reports the length of the longest bounding box size of a model in meters and the resulting number of samples per millimeter for different quantizations.

loss in precision is likely to be considered unacceptable. Vertex positions are usually stored as 32-bit IEEE floating point numbers. In this format the least precise (e.g. the widest spaced) samples are those with the highest exponent. Within the range of an exponent all points have at most 24 bit of precision: 23 bit in the mantissa and 1 bit in the sign. Once the bounding box (e.g. the highest exponent) is fixed we can capture the uniform precision of the floating point samples by uniform quantization with 25 bits. The extra bit of comes from all numbers with smaller exponent whose combined range equals exactly that of the highest exponent. Of course, quantization is not an option for data with non-uniform precision that was specifically aligned with the origin to provide higher precision in some areas. In this case we would have to compress the floating-point numbers in a lossless manner as proposed in (Isenburg et al., 2005a). But in general we can assume that the sampling within the bounding box is uniform.

For scanned datasets it is often not necessary to preserve the full floating point precision. The samples acquired by the scanner are typically not that precise, in which case the lowest-order bits contain only noise and not actually measured data. A reasonable quantization level keeps the quantization error just below the scanning error. In Table 7.2 we lists the resulting sample spacings per millimeter for different levels of quantization. For the 20 cm tall "Buddha" statue even 16 precision bits seem an overkill, whereas the 195 meter long "Double Eagle" may make use of all 24 bits.

The quantized vertices are compressed with the parallelogram rule (Touma and Gotsman, 1998) in the moment they are first encountered. A vertex position is predicted to complete the parallelogram formed by the vertices of a neighboring triangle. The resulting corrective vectors are subsequently compressed with arithmetic coding. Vertices are always encountered during an add operation. The first three vertices of each mesh component cannot be compressed with the parallelogram rule. While the first vertex has no obvious predictor, the second and third can be predicted with delta-coding (Deering, 1995). To maximize compression it is beneficial to encode correctors

of less promising predictions with different arithmetic contexts (Isenburg and Alliez, 2002b). For meshes with few components this hardly makes a difference, but the "Power Plant" and the "Double Eagle" model each consist of millions of components.

Other properties such as colors or confidence values can be treated similarly to vertex positions. However, for models of this size one might consider to store properties in separate files. Not everybody is interested, for example, in the confidence values that are stored for every vertex with all of Stanford's scanned datasets. Despite being in separate files, the decoder can add them on-the-fly during decompression, if the appropriate file is provided.

mesh name	number of					
	vertices	triangles	components	holes	handles	non-manifold
happy buddha	543,652	1,087,716	1	0	104	0
david (2mm)	4,129,614	8,254,150	2	1	19	4
power plant	11,070,509	12,748,510	1,112,199	1,221,511	10	37,702
lucy	14,027,872	28,055,742	18	29	0	64
david (1mm)	28,184,526	56,230,343	2,322	4,181	137	1,098
double eagle	75,240,006	81,806,540	5,100,351	5,291,288	1,534	3,193,243
st. matthew	186,836,665	372,767,445	2,897	26,110	483	3,824

mesh name	size of raw and compressed files on disk [MB]						load time	foot-print
	ply	16 bit	18 bit	20 bit	22 bit	24 bit	[sec]	[MB]
happy buddha	20	1.6	1.9	2.2	2.5	3.0	0.68	0.7
david (2mm)	150	7	10	12	15	17	4.1	1.3
power plant	285	19	23	28	32	35	8.9	0.7
lucy	508	28	37	47	58	70	14.6	1.5
david (1mm)	1,020	44	61	77	93	108	27	2.8
double eagle	1,875	116	146	180	216	244	63	0.7
st. matthew	6,760	236	344	456	559	672	174	9.4

Table 7.3: This table lists vertex, triangle, component, hole, handle, and non-manifold vertex counts for all meshes. Furthermore, the size of a binary ply file containing three floats per vertex and three integers per triangle is compared to our compressed representation at 16, 18, 20, 22, and 24 bits of precision. Also the total time in seconds required for loading and decompressing the 20 bit version on a 1.8 GHz AMD Athlon processor is reported. Finally, we give the maximal memory foot-print of the decompression process in MB.

7.4.3 Results

The compression gains of our representation over standard binary PLY are listed in Table 7.3. Depending on the chosen level of precision the compression ratios range from $1:10$ to $1:20$. Comparing measurements on the same models to (Ho et al., 2001),

mesh name	compression rates [bpv]					
	conn	16 bits	18 bits	20 bits	22 bits	24 bits
happy buddha	2.43	21.79	26.44	32.15	36.92	43.95
david (2mm)	1.50	12.54	17.81	23.22	28.37	34.13
power plant	2.50	11.57	15.26	18.54	21.48	24.23
lucy	1.88	14.60	20.41	26.51	32.87	39.08
david (1mm)	1.79	11.32	16.50	21.20	25.99	30.43
double eagle	3.39	9.58	12.92	16.66	20.67	23.84
st. matthew	1.84	8.83	13.71	18.86	23.63	28.61

Table 7.4: This table details compression results on all our example models. The achieved bit-rates are reported in bits per vertex (bpv) separately for connectivity and for geometry that was quantized with 16, 18, 20, 22, and 24 bits of precision.

our bit-rates are about 25% better. Another advantage of our compressed format is the reduced time to load a mesh from disk. Decompression speeds are CPU-limited and exceed one million vertices and two million triangles per second on a 1.8 GHz Athlon processor. The compression rates for connectivity and for geometry at different precision levels are detailed separately in Table 7.5. One immediate observation is that an additional precision of 2 bits increases some geometry compression rates by about 6 bits per vertex (bpv) or more. While predictive coding is known not to scale well with increasing precision, here this is likely a sign for the precision of quantization being higher than that of the data samples. In this case the additional two bits only add incompressible noise to the data.

mesh name	**decompression time [sec]** (decompression + rendering time [sec])									
	16 bits		18 bits		20 bits		22 bits		24 bits	
happy buddha	**.75**	(1.15)	**.81**	(1.21)	**.97**	(1.37)	**1.13**	(1.53)	**1.38**	(1.79)
david (2mm)	**4.9**	(7.7)	**5.3**	(8.0)	**5.7**	(8.5)	**6.2**	(9.0)	**7.1**	(10.3)
power plant	**11.1**	(14.9)	**11.8**	(15.7)	**12.5**	(16.5)	**13.4**	(17.4)	**15.1**	(19.2)
lucy	**17.8**	(26.7)	**18.9**	(28.0)	**21.1**	(30.2)	**22.8**	(32.1)	**27.3**	(36.7)
david (1mm)	**33**	(51)	**35**	(53)	**38**	(56)	**41**	(60)	**45**	(64)
double eagle	**77**	(94)	**81**	(105)	**88**	(110)	**94**	(119)	**113**	(134)
st. matthew	**215**	(327)	**228**	(351)	**242**	(363)	**259**	(384)	**294**	(419)

Table 7.5: This table details decompression and rendering times on our example models. Timings are reported both for loading/decompression alone as well as for loading/decompression and out-of-core rendering. Timings are taken on a Dell Inspiron 8100 laptop with a 1.1 Ghz Mobile Pentium III processor and a Geforce2go card.

7.5 Summary

We presented a technique that is able to compress very large models such as the gigantic St. Matthew statue in one piece on standard desktop PCs. Our compression rates are about 25% lower and our decompression speeds about 100 times faster than the technique by (Ho et al., 2001) that processes such models by cutting them in pieces.

For this, we introduced an external memory data structure, the out-of-core mesh, that provides our compressor with transparent access to large meshes. We described how to efficiently build this data structure from an indexed mesh representation using only limited memory. Our out-of-core mesh may also be useful to other algorithm that process large meshes. To use it efficiently the order of mesh queries should be localized, but most traversal-based processing is readily accommodated. While our current implementation only allows index-pairs to use a combined maximum of 32 bits, this data structure can theoretically handle arbitrary large meshes. Storing more bits per index-pair, however, will increase in-core and on-disk storage and make its build-up/usage more IO-limited.

Our compressed format has benefits beyond efficient storage and fast loading. It provides *better* access to large meshes than indexed formats or polygon soup by allowing to stream gigantic meshes through limited memory while providing seamless connectivity information along the decompression boundary. This *streaming mesh* representation offers a new approach to out-of-core mesh processing that combines the efficiency of batch-processing with the advantages of explicit connectivity information as it is available in online-processing. This suggests the concept of *sequenced* or *streaming* processing where access to the mesh is restricted to a fixed traversal, while at the same time full connectivity for the active elements of this traversal is provided.

Traversing mesh triangles in a particular order is already used for fast rendering on modern graphics cards. The number of times a vertex needs to be fetched from the main memory is reduced by caching previously received vertices on the card. The triangles are sent to the card in a *rendering sequence* that tries to minimize cache misses (Hoppe, 1999). Misses cannot be avoided altogether due to the fixed size of the cache (Bar-Yehuda and Gotsman, 1996). In a similar spirit our compressed format provides a *processing sequence* for more efficient mesh processing—but at a much larger scale. With the main memory as the "cache" we usually have more than enough storage space for all active elements throughout the traversal of a mesh. Therefore the analogue of a "cache miss" does not exist. Any algorithm that requires a complete mesh traversal without being particular about its order can perform computations at decompression

speed—we can envision an entire breed of mesh processing algorithms adapted to this kind of mesh access. In the next chapter we show that mesh simplification algorithms can indeed be implemented to take advantage of this type of processing.

7.6 Hindsights

When we started this work our main goal was to gain the ability to compress gigantic meshes with an out-of-core method into a highly compressed format from which they could then be decompressed with a small memory foot-print. For this we designed a mesh traversal that would keep the number of accesses to the out-of-core mesh to a minimum while driving the achieved compression to a maximum and—just like all previous works on mesh compression—we paid no attention what that would do to the ordering of triangles. In Chapter 9 we will see that traversing the mesh triangles in a depth-first manner produces highly incoherent mesh layouts (see Figure 9.6 for an illustration). To prevent this we advocate in Chapter 9 that a mesh compressor should give all vertices a similarly long "life-time" on the compression boundary.

To achieve coherence in the layout of the compressed mesh requires three changes to the traversal order of our out-of-core compressor. First, the traversal needs to advance along all boundaries simultaneously instead of operating exclusively on one boundary. One possible way of achieving this is to continue on the least recently advanced boundary whenever completing a loop around a boundary. Second, the adaptive traversal that jumps around the boundary in an attempt to avoid split operation needs to be replaced with the coherence-preserving strategy suggested in Section 4.9. And third, once a non-manifold vertex has made its first appearance we can no longer just wait until the traversal runs into its other appearances. This can leave such vertices "hanging" for a long time, especially if they re-appear in separate mesh components that are not edge-connected. To reduce their "life-time" requires the start of additional compression boundaries, making this probably the most complex change.

Chapter 8

Processing Meshes in Stream Order

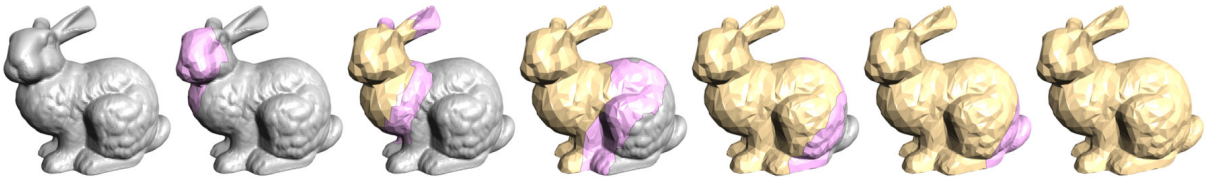


Figure 8.1: Illustration of mesh simplification using a fixed-size triangle buffer (pink) in main memory. Via processing sequences, original triangles (gray) stream into the buffer and simplified triangles (gold) stream out.

In this chapter I show that mesh processing algorithms can be adapted to perform their computations based on the processing sequence paradigm envisioned in the previous chapter, using mesh simplification as an example. We believe that this processing concept will also prove useful for other tasks, such as smoothing, parameterization, or remeshing, for which currently only in-core solutions exist.

A processing sequence represents a mesh as a particular interleaved ordering of indexed triangles and vertices. This representation allows streaming very large meshes through main memory while maintaining information about the visitation status of edges and vertices. At any time, only a small portion of the mesh is kept in-core, with the bulk of the mesh data residing on disk. Mesh access is restricted to a fixed traversal order, but full connectivity and geometry information is available for the active elements of the traversal. This provides seamless and highly efficient out-of-core access to very large meshes for algorithms that can adapt their computations to this fixed ordering.

The two abstractions that are supported by this representation are *boundary-based* and *buffer-based* processing. We illustrate both abstractions by adapting two different simplification methods to perform their computation using a prototype of our mesh processing sequence API. Both algorithms benefit from using processing sequences in terms of improved quality, more efficient execution, and smaller memory footprints.

8.1 Introduction

Polygonal models acquired with modern 3D scanning technology easily reach sizes of gigabytes—the most prominent examples are the detailed scans of Michelangelo's sculptures generated by teams at IBM (Bernardini et al., 2002) and Stanford (Levoy et al., 2000). Similarly large polygonal data sets result from extracting dense isosurfaces from volumetric data. A polygon mesh with hundreds of millions of vertices requires gigabytes of raw data, making subsequent processing difficult. The sheer amount of data not only exhausts the main memory resources of common desktop PCs, but also exceeds the 4 gigabyte address space of 32-bit machines.

A straightforward approach for processing meshes that are too large to fit in main memory is to cut them into pieces small enough to be processed in-core. However, mesh cutting tends to introduce processing artifacts along the cut boundaries. Another approach is to design computations to work in increments of single triangles. This allows efficient batch processing, as the CPU can be kept busy by loading and processing triangles as fast as possible. However, the absence of explicit mesh connectivity makes many mesh processing tasks either impossible or results in a lower quality outputs. Finally, there are approaches that use external memory data structures that provide transparent access for online processing of arbitrarily large meshes. However, building and using such complex data structures is typically inefficient.

In the previous chapter we envisioned a new processing paradigm for out-of-core computations on large meshes that combines the efficiency of batch processing with the advantage of explicit mesh connectivity that is available in online processing. The idea of *sequenced processing* was to restrict access to the mesh to a fixed traversal order, but to support access to full connectivity and geometry information for the active elements of this traversal. In this representation only a small fraction of the mesh is kept in main memory at any time with the bulk of the mesh data residing on disk. While the mesh streams through memory, we provide seamless mesh access for algorithms that can respect a fixed traversal order. This idea of *processing sequences* grew out of the particular mesh access provided by our compressed format. It supports two computational abstractions: *boundary-based* processing and *buffer-based* processing. In the previous chapter we have seen examples of simple operations on large meshes that are naturally supported by these abstractions, including loading, decompression, rendering, and connectivity reconstruction. In this chapter we show that they can be used for more complex tasks using out-of-core mesh simplification as an example.

The remainder of this chapter is organized as follows: The next section summarizes current approaches to out-of-core mesh processing. In Section 8.3 we describe how processing sequences provide access to large meshes. In Section 8.4 we detail current techniques for the simplification of large meshes. Then we adapt two different mesh simplification schemes to sequenced processing: In Section 8.5 we adapt the non-adaptive OoCS simplification algorithm of (Lindstrom, 2000) to boundary-based processing. Similarly, in Section 8.6, we map the adaptive stream simplification algorithm of (Wu and Kobbelt, 2003) to buffer-based processing. Both algorithms benefit from using processing sequences in terms of improved quality, more efficient execution, and smaller memory footprints. The last section concludes with a summary and an outlook on other types of mesh processing.

8.2 Out-of-Core Processing

There are three main approaches for processing meshes that are too large to fit in main memory (Silva et al., 2002): cutting the mesh into pieces, batch processing of polygon soups, and online processing using external memory data structures.

Mesh cutting is a straightforward approach for processing large meshes: cut the mesh into pieces small enough to fit in main memory and then process each piece separately while giving special treatment to the cut boundaries. This strategy has successfully been used to, for example, simplify (Hoppe, 1998; Prince, 2000; Bernardini et al., 2002) and compress (Ho et al., 2001) very large polygon models. Despite the apparent simplicity of this approach, the initial cutting step can be expensive when the input mesh is given in an indexed representation, as we will see later. Because mesh cutting typically lowers the quality of the output, many out-of-core algorithms try instead to process the data as a whole.

Batch processing aims to keep the memory footprint low and the processor busy by streaming the mesh data through main memory in one or more passes, and by restricting computations to the amount of data that is resident in memory at any time. This makes batch processing computationally very efficient.

Examples include a number of mesh simplification methods (Lindstrom, 2000; Lindstrom and Silva, 2001; Shaffer and Garland, 2001; Garland and Shaffer, 2002), which batch-process the input mesh as a sequence of individual triangles. If indexed meshes

are used that exhibit no locality in referencing the vertex array (e.g. where vertex indices of subsequent triangles address vertex array entries at random) an initial *dereferencing step* is required (Lindstrom and Silva, 2001). This can be computationally expensive and the resulting immediate mesh (i.e. *polygon soup*) requires at least twice the storage of an indexed mesh, and more if there are additional per-vertex properties such as texture coordinates or surface normals. The output of a batch simplification pass either is small enough to fit in memory, so that the remaining computation can be done in-core (Lindstrom, 2000; Shaffer and Garland, 2001; Garland and Shaffer, 2002), or is directly written to a file, which is then used as input for subsequent passes (Lindstrom and Silva, 2001).

Online processing accesses the data through a series of (potentially random) queries. In order to avoid costly disk seeks with each query (resulting in thrashing) the data is usually re-organized to accommodate an anticipated access pattern. Queries can be accelerated by *caching* or *pre-fetching* data that is likely to be accessed. Some schemes simply use the virtual memory functionality of the operating system and try to organize the data accesses such that the number of page faults is minimized (McMains et al., 2001; Choudhury and Watson, 2002). The performance of such schemes is operating system dependent and their input data is restricted to 4 gigabytes on a 32-bit machine. Going beyond that limit requires dedicated external memory data structures that explicitly manage a virtual address space for the data.

Such external memory data structures enable traditional in-core algorithms to be applied to large data sets. (Cignoni et al., 2003), for example, propose an octree-based external memory data structure that makes it possible to simplify a model of Michelangelo's St. Matthew statue (Levoy et al., 2000) from 386 to 94 million triangles using iterative edge contraction (Garland and Heckbert, 1997). Similarly, the out-of-core mesh that we have described in the last chapter allowed us to compress the St. Matthew statue from over 6.5 GB to 344 MB of data using a compressor based on region growing (Touma and Gotsman, 1998).

For comparison, out-of-core algorithms based on batch processing do their work on polygon soups without explicit connectivity information. Thus, they can perform their computations efficiently, but their output tends to be of lower quality than that of algorithms with access to explicit connectivity information. Out-of-core algorithms based on online processing, on the other hand, have explicit connectivity available. However, building these data structures is expensive in time and space, and using them

significantly slows down the computations. In the following we will show that we can combine the efficiency of batch processing with the advantages of explicit connectivity information available in online processing. Using a *processing sequence* we restrict the access to the mesh to a fixed traversal order, but support access to the full connectivity and geometry information for the active elements during this traversal.

Rearranging mesh triangles into a particular order is already used for improving rendering performance on modern graphics cards. The number of times a vertex needs to be fetched from main memory is reduced by caching previously received vertices on the card. The triangles are sent to the card in a *rendering sequence* in an attempt to minimize cache misses (Deering, 1995; Evans et al., 1996b; Hoppe, 1999; Bogomjakov and Gotsman, 2001). Due to the fixed size of a vertex cache, misses cannot be avoided completely (Bar-Yehuda and Gotsman, 1996). Our *processing sequences* exploit a similar strategy for more efficient mesh processing—but at a much larger scale. However, the main memory as a "cache" is much more flexible. The amount of storage necessary to maintain the active elements of a mesh traversal is usually small enough to fit in main memory. Therefore the analogue of a "cache miss" fortunately does not exist.

8.3 Processing Sequences

A *processing sequence* presents a mesh as a fixed interleaved sequence of indexed vertices and triangles that grow a region. The mesh edges that separate already processed triangles from unprocessed ones form the *processing boundary*. Mesh triangles *generated* by the processing sequence are either edge-adjacent to the processing boundary or start a new region. With each triangle, the processing sequence provides vertex information such as indices, coordinates, first and last time referenced, and non-manifoldness. Similarly, the topological type of edges and their relationship to the processing boundary are made available. Finally, a processing sequence supports storage and retrieval of user data on the evolving processing boundary.

Triangles can change the processing boundary in one of the five ways illustrated in Figure 8.2. A "start" triangle creates a new component of a processing boundary with three *new* vertices and edges. A new edge may be *entering* the processing boundary, to be paired with an incident triangle later in the sequence, or it may be part of the surface *border*, the topological boundary of the mesh. An "add" triangle completes a boundary edge and connects a new vertex with two new edges. The completed edge *leaves* the processing boundary. A "fill" completes two edges, replacing them and

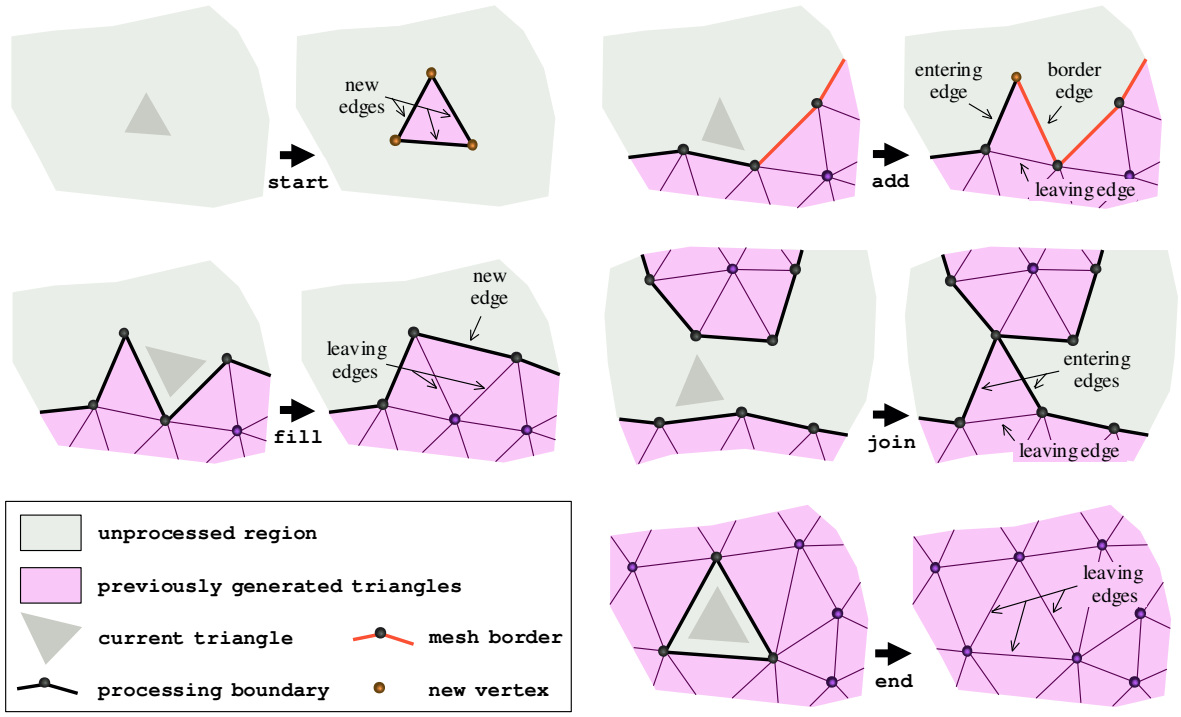


Figure 8.2: The five different ways generated triangles relate to the processing boundary.

the vertex reference between them with a new edge. A "join" completes one edge, adds two new edges, and either merges two components of the processing boundary into one (sometimes forming a handle), or splits one component into two. An "end" completes three edges. Given a "somewhat" compactly growing processing sequence, this representation allows streaming very large meshes through main memory. At any time only the processing boundary needs to be kept in-core. Yet, as explicit connectivity information can be maintained along the processing boundary, this provides seamless access to large meshes. We have only defined the allowable triangle sequences, which neither determines a particular triangle ordering, nor a particular file format. But whatever the underlying representation, a processing sequence reader, for example, will provide functionality similar to that of the API outlined in Figure 8.3.

Connectivity reconstruction is supported by letting users store their own data with the first appearance of any edge or vertex on the processing boundary. This data is made available when these mesh elements later reappear as part of another triangle, enabling full recovery of mesh connectivity in constant time per element. The pseudo code in Figure 8.3 illustrates how a typical application would reconstruct connectivity.

```
class PSreader {
    int     open(FILE* file);
    int     get_num_vertices();
    int     get_num_triangles();
    bool    has_triangles();
    int     generate_triangle();
    int     close();

    int     get_vertex_index(int i);
    float*  get_vertex_position(int i);
    vtype   get_vertex_type(int i);
    etype   get_edge_type(int i);

    void    set_vertex_data(int i, void* vdata);
    void*   get_vertex_data(int i);
    void    set_edge_data(int i, void* edata);
    void*   get_edge_data(int i);
}

            struct Vertex {
              int index;
              float pos[3];
            }

            struct HalfEdge {
              Vertex* origin;
              HalfEdge* next;
              HalfEdge* prev;
              HalfEdge* inv;
            }
```

```
PSreader ps = new PSreader();
ps.open(file);
while (ps.has_triangles()) {
  HalfEdge he[0] = new HalfEdge();
  HalfEdge he[1] = new HalfEdge();
  HalfEdge he[2] = new HalfEdge();
  he[0].next = he[1]; he[0].prev = he[2];
  he[1].next = he[2]; he[1].prev = he[0];
  he[2].next = he[0]; he[2].prev = he[1];
  ps.generate_triangle();
  for (int i = 0; i < 3; i++) {
    if (ps.get_edge_type(i) == ENTER) {
      ps.set_edge_data(i, (void*)he[i]);
    } else if (ps.get_edge_type(i) == LEAVE) {
      HalfEdge tmp = (HalfEdge*)ps.get_edge_data(i);
      tmp.inv = he[i];
      he[i].inv = tmp;
    } else {
      he[i].inv = 0;
    }
    if (ps.get_vertex_type(i) == NEW) {
      Vertex v = new Vertex();
      v.index = ps.get_vertex_index(i);
      v.pos = ps.get_vertex_position(i);
      ps.set_vertex_data(i, (void*)v);
      he[i].origin = v;
    } else {
      he[i].origin = (Vertex*)ps.get_vertex_data(i);
    }
  }
}
```

Figure 8.3: An outline of an API for a processing sequence reader and example code for reconstructing mesh connectivity using a simple half-edge structure. This is achieved by maintaining user data per-vertex and per-edge along the processing boundaries.

If processing sequences are read and written at the same time there are two processing boundaries: one is the *input* boundary, along which triangles are added, and one is the *output* boundary, where triangles are removed. The region between the two boundaries is the called the *triangle buffer*, which contains those triangles that are currently in memory. The triangle order of the input and the output sequence does not need to be identical. In particular, the two sequences can contain a completely different set of triangles and vertices, for example, if remeshing or simplification is performed on the triangle buffer. When the order in which an application outputs triangles and vertices does not immediately correspond to a processing sequence, we use a *processing sequence converter* that temporarily accumulates triangles and vertices in a small *waiting area* and reorders them appropriately, as illustrated in Figure 8.4. Processing sequences provide two useful computational abstractions: boundary-based and buffer-based processing.

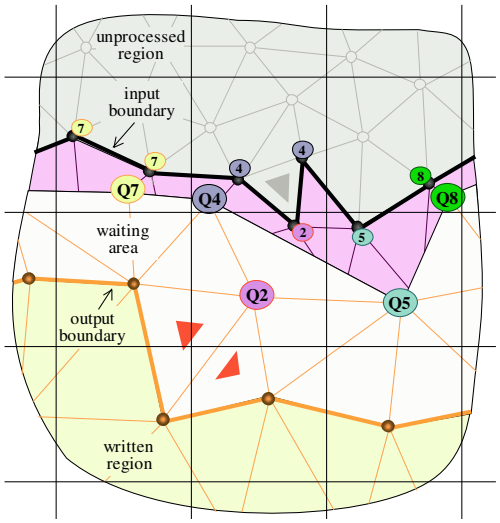


Figure 8.4: An illustration of how a *waiting area* is used to *on-the-fly* convert the triangle and vertex ordering produced by the simplification method described in Section 8.5 into a processing sequence. After processing the triangle marked in gray, the simplifier turns the quadric Q2 into a vertex and places it into the waiting area. In this moment the vertex becomes eligible for output, as do the two triangles in the waiting area marked in red that are connected to it. Furthermore, this also *finalizes* the vertex, in the sense that no triangles other than those already in the waiting area reference it.

Boundary-based processing performs its computations directly on the input boundary. It immediately processes the triangles generated at the input boundary and stores intermediate results only along these boundaries. Example applications are simplification methods using vertex clustering, non-iterative smoothing methods, gradient or surface normal computations, etc.

Buffer-based processing performs its computations on the triangle buffer between input and output boundary (see Figure 8.1 for an illustrating visualization). It generates triangles at the input boundary to fill the buffer and at the output boundary to empty the buffer. Example applications are simplification methods that use edge contraction, iterative smoothing methods, remeshing methods, etc. We can think of buffer-based processing as bridging the conceptual gap between boundary-based processing and in-core processing. Restricting the buffer size to a single triangle is equivalent to boundary-processing. A buffer size that is large enough to contain the entire mesh is equivalent to in-core processing. Any buffer size in between these extremes provides a compromise that "adapts" to the available resources.

Implementations of either abstraction can perform their computation in a single pass or in multiple passes over the data. For multiple passes, the output sequence of a previous pass becomes the input sequence of the next. Instead of sequentially performing multiple passes, a multi-stage approach streams the results of one pass directly to the next by making the output boundary of one the input boundary of the other. Immediate compression of the output of a simplification algorithm, for example, could be implemented using such a multi-stage approach.

Generating processing sequences can be done in a number of different ways, as the definition neither imposes a specific traversal order, nor a data format. The input sequences used for simplification experiments in this chapter were generated in a pre-processing step using the out-of-core compressor described in the last chapter. For now this compression scheme can be thought of as the means to obtain the triangle and vertex ordering that allows traversing the entire mesh with small memory footprint. Most *one-pass* compression schemes naturally generate orderings that conform to the definition of a mesh processing sequence. In fact, it was the memory-efficient decompression order of the decoder that originally inspired processing sequences. Although these particular processing sequences are compact and fast to load, their generation is not trivial and they are currently created offline. Furthermore their "stream quality" is far from optimal as we will see in the next chapter.

The processing sequence converter, mentioned earlier, provides an efficient mechanism for *on-the-fly* creation of processing sequences. It accepts indexed vertices and triangles ordered in some loosely localized form, temporarily accumulates them in a *waiting area*, where they are re-ordered into a proper processing sequence. A vertex from the waiting area becomes eligible for output when its first triangle is to be output. A triangle from the waiting area becomes eligible for output when all its vertices are already output *and* it conforms to one of the five configurations shown in Figure 8.2.

The sole requirement, besides some locality in the input, is that the converter is told when a vertex is *finalized*, i.e., used for the last time. This information is needed to correctly recover connectivity around vertices, as well as to safely deallocate the memory of mesh elements that are no longer used. For our output sequences, the simplification process tells the converter when a vertex is finalized. The converter automatically buffers as many triangles as needed to produce a valid processing sequence. Increasing the size of the waiting area beyond the minimum gives the converter freedom to choose among several potential output triangles. This allows, for example, sequences with fewer "start" or "join" configurations to be generated. Initially we stored output sequences in a verbose textual format, but in the next chapter we describe a scheme for *on-the-fly* compression of arbitrary output sequences.

This converter also provides an alternative to the out-of-core compressor for generating processing sequences "from scratch": First we create two spatially ordered sequences, one of vertices and one of triangles. Vertices are sorted together with their index i using one coordinate, for example x, as the sort key k. Triangles are sorted in indexed form using the minimal key k of their three vertices as the sort key. This

can be implemented using a few external sorts (Lindstrom and Silva, 2001). In a final pass over the two sorted sequences we load vertices and triangles into the waiting area. We read from the triangle sequence as long as the next triangle key is less than or equal to the next vertex key. Eventually the key of the next triangle is larger than that of the next vertex and we read from the vertex sequence. This vertex can now be finalized as all its triangles are already in the waiting area. The vertices and triangles leave the waiting area in processing sequence order, as described earlier. Other ways of generating such orderings are reported in the next chapter together with an in-depth investigation of the criteria that make orderings "good" for processing.

Non-manifold meshes are turned into manifold meshes simply by cutting along non-manifold vertices and edges. However, vertices and edges are not replicated, but re-appear multiple times as *new* mesh elements. This makes it possible to represent non-manifold meshes using only the five operations allowed for generating triangles. To accommodate processing tasks that require special treatment of non-manifold elements, the processing sequence API provides an additional flag per vertex and per edge. This flag informs whether an element is non-manifold and whether there are still future non-manifold occurrences of the element remaining.

8.4 Large Mesh Simplification

Early methods for simplifying large meshes were based on mesh cutting (Hoppe, 1998; Prince, 2000; Bernardini et al., 2002). In mesh cutting, the input mesh is partitioned into pieces small enough to be processed in-core, which are then simplified individually. The partition boundaries are left untouched such that the simplified pieces can be stitched back together seamlessly. The hierarchical approaches of (Hoppe, 1998) and (Prince, 2000) automatically simplify these boundaries at the next level, whereas (Bernardini et al., 2002) process the mesh more than once—each time using a different partitioning. Later, out-of-core simplification methods based on batch processing became popular. (Lindstrom, 2000) performs vertex clustering (Rossignac and Borrel, 1993) on a uniform grid and stores one quadric error matrix (Garland and Heckbert, 1997) per occupied grid cell in memory. Indexed input meshes are first dereferenced into polygon soups and then batch-processed one triangle at a time, adding each triangle's quadric matrix to the cells in which the triangle has a vertex. The output triangles are those that connect three different grid cells. Each cell is represented by a vertex

whose position minimizes the quadric error accumulated in the cell. In more recent work (Lindstrom and Silva, 2001) show that the limitation of the output mesh having to fit in main memory can be overcome using a series of external sorts.

Although the vertex clustering approach to simplification allows efficient out-of-core implementations, it delivers lower quality results than a typical in-core algorithm. Vertex clustering can not retain details smaller than a grid cell and lacks the adaptivity of an implementation based on, for example, iterative edge contraction. Addressing this issue, (Shaffer and Garland, 2001; Garland and Shaffer, 2002) suggest using batch processing to accumulate error quadrics with a vertex cluster resolution that is higher than that of the output mesh, but still fits in-core. From there a simplified mesh can be created in-core either top-down, using a variation of R-simp (Brodsky and Watson, 2000), or bottom-up, using QSlim (Garland and Heckbert, 1997). The accumulated quadrics pass information about the original surface to the in-core algorithm. This allows higher quality simplifications with an exact vertex budget, provided that the available memory is a constant factor larger than the output mesh.

As we will see in Section 8.5, processing sequences allow efficient implementations of simplification algorithms based on vertex clustering. As the processing boundary sweeps over the entire mesh, visiting every triangle exactly once, we can store, update, and propagate quadric error matrices along these boundaries only. This will significantly reduce the memory footprint, improve the quality of the simplified mesh, and enable pipelined processing by immediately feeding the output to another application.

The simplification methods discussed so far treat large meshes differently from small meshes as they try to avoid performing costly online processing on the entire mesh. Therefore the output produced by an out-of-core algorithm is usually of lower quality than that of an in-core algorithm. (Cignoni et al., 2003) propose an octree-based external memory data structure that provides algorithms with transparent online access to huge meshes. This makes it possible to, for example, simplify the St. Matthew statue from 386 to 94 million triangles using iterative edge contraction (Garland and Heckbert, 1997). However, the run times for both constructing an external memory data structure and using it during simplification are significantly longer than the run times of simplification methods based on batch processing.

(Wu and Kobbelt, 2003) propose an out-of-core simplification technique that is similar to the buffer-based abstraction of processing sequences. Starting with polygon soup as input, they keep a large in-core buffer of triangles on which they perform edge collapses. Since the input mesh is not indexed, connectivity between triangles must be

mesh name	number of						
	vertices	triangles	components	holes	handles	n.-m. v.	p. b. v.
buddha	544 K	1.1 M	1	0	104	0	2.3 K
blade	883 K	1.8 M	295	0	165	0	7.3 K
david (2mm)	4.1 M	8.3 M	2	1	19	4	21 K
lucy	14 M	28 M	18	29	0	64	23 K
david (1mm)	28 M	56 M	2.3 K	4.2 K	137	1.1 K	59 K
st. matthew	187 M	373 M	2.9 K	26 K	483	3.8 K	223 K
ppm isosurface	235 M	469 M	168 K	6.2 K	168 K	0	1.6 M

Table 8.1: Vertex, triangle, component, hole, handle, and non-manifold vertex counts, as well as maximum number of the vertices on the processing boundary in thousands (K) and millions (M) for all meshes used in our simplification experiments.

reconstructed by matching up the coordinates of their vertices. Their method assumes that the polygon soup is spatially ordered so that the triangles in the in-core buffer form connected regions. Thus, an input mesh may need to be pre-sorted using external sorting (Lindstrom and Silva, 2001).

One drawback of Wu and Kobbelt's method is that it can not distinguish actual mesh borders from the input boundary of the buffer. As borders cannot be recognized and simplified until the entire mesh has been read, they must keep all triangles along the mesh borders in the buffer. For a mesh with many small holes, which is common in large range scans, this can considerably inflate the memory requirements and may reduce the quality of the output. Processing sequences provide an ideal input to Wu and Kobbelt's stream-based method: The incoming triangles that populate the buffer are maximally connected. The mesh borders are known, which allows immediate simplification of holes. The connectivity reconstruction is either already provided by the API or can be done more efficiently as triangles are in an indexed format. Finally, their stream-based algorithm maps exactly to the abstraction of buffer-based processing, which is discussed further in Section 8.6.

8.5 Boundary-Based Processing

In this section we show how the out-of-core simplification method OoCS by (Lindstrom, 2000) can be adapted to mesh processing sequences using boundary-based processing. Capitalizing on the coherent geometric or topological ordering provided by processing sequences, as well as the connectivity information made available, we improve upon OoCS in a number of ways.

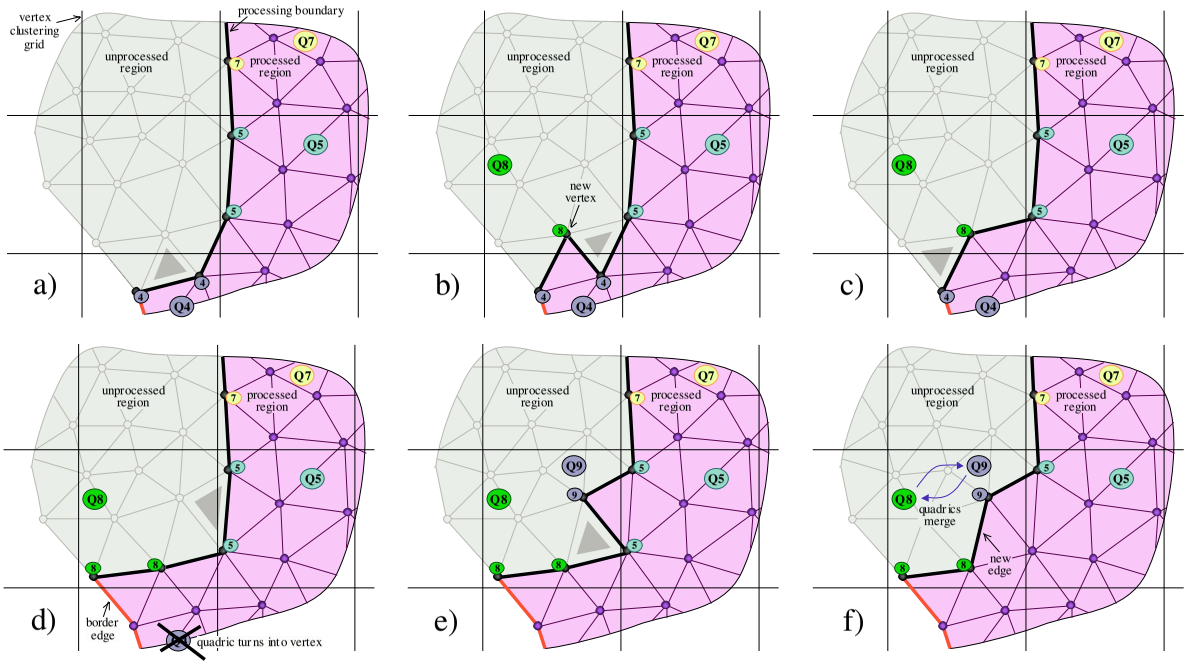


Figure 8.5: A 2D illustration of using boundary-based processing for vertex clustering based simplification: Quadrics, here depicted as colored ellipsoids, are allocated only for grid cells that contain processing boundary vertices. Each quadric maintains a counter for the number of vertices it is associated with. The counted vertices are marked with smaller colored ellipsoids. In every frame the triangle marked in gray is processed: **a→b)** A new quadric Q8 is allocated because the new vertex falls into a different grid cell. The quadric of the triangle is computed and added to Q4 and Q8. This triangle is not output as two of its vertices fall into the same grid cell. **b→c)** The triangle quadric is computed and added to Q4, Q5, and Q8. This triangle is output as all its vertices fall into different grid cells. **c→d)** The quadric of the triangle is computed and added to Q4 and Q8. Because of the border edge a border error quadric is also added to Q4 and Q8. No triangle is output. As the counter of Q4 drops to zero, we compute and output its representative vertex, and deallocate the quadric. **d→e)** A new quadric Q9 is allocated because the new vertex falls into a different grid cell from those it is connected to. The quadric of the triangle is added to Q5 and Q9. No triangle is output. **e→f)** The new edge connects two vertices of the same grid cell that have different quadrics. Therefore, quadrics Q8 and Q9 are merged. The triangle quadric is added to Q5 and Q8/Q9.

First, we make use of explicit mesh connectivity to detect and preserve surface boundaries. This is trivially accomplished using processing sequences, although it is an important improvement. Second, we avoid the common "pinching" problem that results when two or more (possibly unconnected) layers of the surface pass through the same grid cell and are pinched. This problem is particularly noticeable when simplifying "dense" meshes with many thin structures, such as CAD models and complex

isosurfaces (see, for example, Figure 8.7(a)). Finally, because of spatial coherence, we do not need to maintain the entire simplified mesh in memory, but output vertices and triangles whenever possible as the processing boundary advances through space. As a result, we require in-core storage only on the order of the length of the processing boundary. We will describe two extensions to the original clustering method—one simple and one somewhat more involved—and begin by explaining the general idea behind the two new techniques.

In both of our extensions, quadric error matrices are allocated, updated, and evaluated only along the processing boundaries, which sweep over the entire mesh, visiting every triangle exactly once. As in (Lindstrom, 2000), triangles add their quadric error to the respective matrices the moment they are processed. However, the life-time of each of these matrices is limited to the duration that a processing boundary pierces the grid cell associated with the matrix. More precisely, a grid cell is *active* whenever it contains vertices from the processing boundary and quadric matrices are stored only with currently active grid cells, thus obviating the need to explicitly store the entire sparse grid. Similar to the original method, but much more efficient since only the active subset of the cells intersected by the surface are stored, this sparse grid representation is implemented using a hash table.

With each active cell, we also store a counter that is incremented whenever a new vertex falls into this cell and decremented whenever a vertex from this cell is used for the last time (Figure 8.5). Thus, the active cells are those with non-zero vertex counters. When the value of the counter drops to zero, we compute the cell's representative vertex from the accumulated quadric matrix and place it on the output. The grid cell, including the counter and the quadric matrix, is then deallocated (i.e. removed from the hash table). Notice that the processing boundary may enter and leave any given cell several times when multiple layers of the mesh pass through the cell. Therefore we will often generate one representative vertex for each layer. This is in contrast to the original approach that represents all mesh layers passing through a grid cell with a single vertex. This difference becomes especially noticeable for aggressive simplification, as illustrated by the simplified blade model in Figure 8.7. The original approach collapses many layers into one vertex, which modifies the underlying topology and leads to poor positioning of the representative vertex.

The framework just described is the basis of our first extension to OoCS. As should be evident, it involves a minor change to the original algorithm—an additional per-cell counter and the ability to remove cells—yet it can have a dramatic impact on the

topological quality of the output mesh. For example, disconnected components are guaranteed not to be merged if the processing sequence traverses the mesh one component at a time. Nevertheless, it is still possible for pinching to occur, e.g. if the processing boundary wraps around and re-enters an already active cell, or if multiple boundaries simultaneously pass through a cell. Ideally, we would like to further partition each cluster of vertices within a cell into connected components, which would eliminate pinching altogether. This is accomplished in our second and more elaborate extension.

Conceptually, we construct connected components within a cell by initially assigning each new vertex introduced in the processing sequence to a unique cluster. Then, for each triangle processed, we collapse clusters that both share an edge of the triangle and are part of the same grid cell. As a result, vertices from the input mesh are merged only if they share an edge, which in effect renders our vertex clustering algorithm as an edge collapse method. That is, our method is functionally equivalent to collapsing all edges whose vertices are contained in the same grid cell. Indeed, for simplicity, our implementation explicitly makes use of edge collapse and a conventional mesh data structure for the partially simplified mesh near the processing boundary. Contrary to conventional edge collapse methods, however, we do not have access to the entire input mesh. In the context of processing sequences, this implies maintaining which cluster each of the vertices on the processing boundary belongs to, merging clusters (i.e. collapsing edges), and keeping track of when a single cluster (as opposed to all clusters) within a cell becomes inactive. We accomplish the latter by adding the vertex counters of two partial clusters when merging them.

Occasionally this approach can even generate more than one vertex per layer for a single cell. This happens each time that an edge with no endpoint in the cell divides the layer passing through the cell into two parts that both contain vertices in this cell (see Figure 8.6). Such additional vertices are generally beneficial since they serve to unfold what would otherwise become non-manifold mesh pieces. A single additional vertex can sometimes untangle multiple non-manifold vertices, as evidenced by Table 8.2.

The order in which triangles and vertices are finalized does not directly result in a proper output sequence. This is because output triangles are usually generated before their vertices are ready for output, i.e. before their clusters become inactive. Therefore, output triangles are first put into a waiting area, as illustrated earlier in Figure 8.4. Whenever a vertex is output, we check whether waiting triangles that reference the vertex are eligible for output, i.e. whether all three of their vertices have been output, and

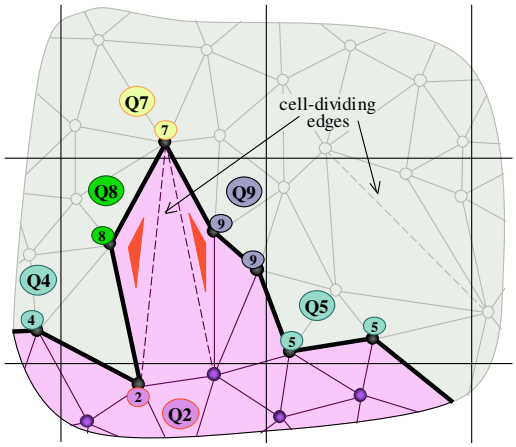


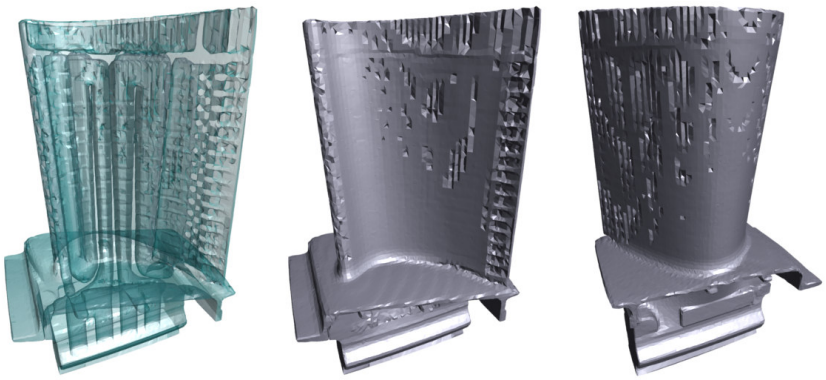
Figure 8.6: The presence of *cell-dividing* edges (shown stippled) results in more than one representative vertex per grid cell for a single mesh layer. Here they prevent Q8 and Q9 from merging. This is beneficial as it prevents the output triangles {Q7,Q8,Q2} and {Q2,Q9,Q7} (shown in red) from collapsing into a pair of oppositely oriented triangles with non-manifold edges. The grid cell on the right illustrates another example of a cell-dividing edge.

T_{out}	method	V_{out}	V_{nm}	$\frac{\Delta V_{\mathrm{nm}}}{\Delta V_{out}}$	RAM (MB)	time (s)	speed (T_{in}/s)
70,546	original	33,053	3,366	–	10.7	5.62	314 K
	active cells	34,682	1,665	104%	7.3	5.78	305 K
	connected	35,134	897	119%	3.4	6.18	285 K
122,470	original	59,675	3,103	–	11.0	6.97	253 K
	active cells	60,618	2,008	116%	8.0	7.07	250 K
	connected	61,109	1,172	135%	3.4	7.10	249 K
230,642	original	113,961	3,472	–	21.9	9.02	196 K
	active cells	114,695	2,436	141%	13.8	9.13	193 K
	connected	115,238	1,360	165%	3.5	9.15	193 K

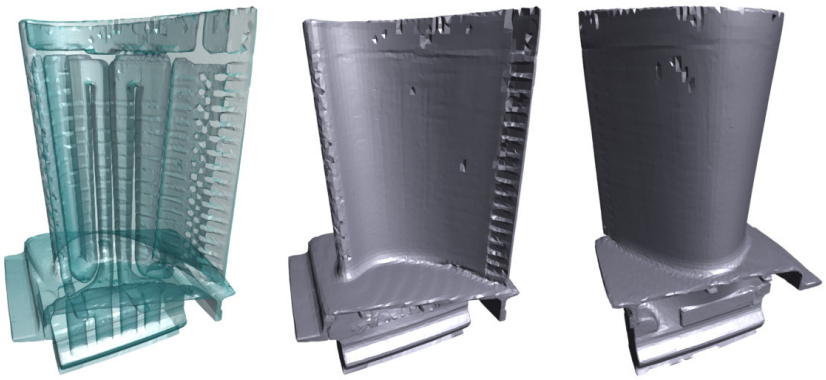
Table 8.2: Results of simplifying the blade model using the original OoCS algorithm and our "active cells" and "connected layers" extensions based on processing sequences. The fifth column lists the change in number of non-manifold vertices (ΔV_{nm}) over the change in total number of output vertices (ΔV_{out}) relative to the original method. Note that, on average, each added vertex generally makes more than one previously non-manifold vertex manifold. The last column reports the simplification speed as number of input triangles processed per second.

if so output and deallocate the triangle. Because the generated vertices and triangles can be written (almost) directly to disk, the memory requirements of this approach are independent of the size of the output mesh. Rather, the memory usage depends solely on the maximal length of the processing boundary.

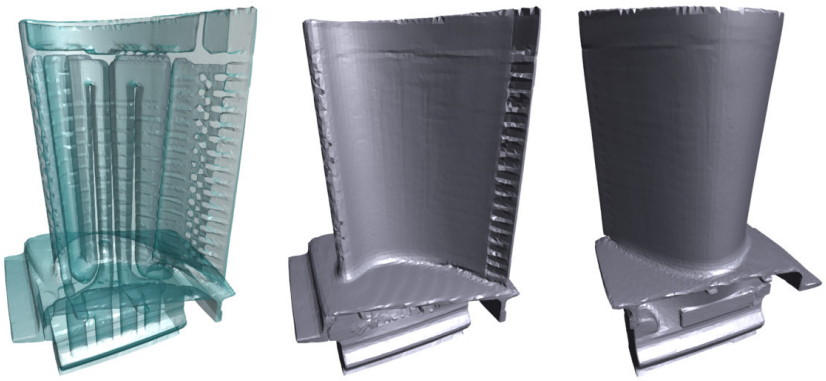
The information on border edges available during sequenced processing further improves the quality of the simplified mesh. Instead of adding tangential error terms for every edge that completely neutralize each other only across coplanar triangles, as suggested in (Lindstrom and Silva, 2001), we explicitly penalize deviation from surface borders using the specialized quadric error matrices of (Garland and Heckbert, 1997).



(a) original algorithm: 33,053 vertices, 3,366 non-manifold



(b) with active cells: 34,682 vertices, 1,665 non-manifold



(c) with connected layers: 35,134 vertices, 897 non-manifold

Figure 8.7: Semitransparent and opaque views of the turbine blade model, simplified using the original OoCS algorithm and our extensions to it. Notice the severe pinching in 8.7(a) as interior and exterior layers of the surface pass through single grid cells and are collapsed. The grid dimensions are $57 \times 96 \times 44$ in all three cases.

8.5.1 Results

Figure 8.7 and Table 8.2 highlight the results of using our boundary-based processing methods to simplify the turbine blade model. Notice the large reduction in non-manifold vertices relative to the small increase in total number of vertices (in all cases a higher than 100% efficiency). As can be seen in Figure 8.7, many of these non-manifold vertices are the result of pinching. These models were simplified on a 2 GHz Pentium 4 Windows 2000 PC with 1 GB of RAM.

In addition to higher quality meshes, our "connected layers" method is also more memory efficient than the original method, which requires storing the entire simplified mesh in-core. We simplified the St. Matthew model from 373 million triangles to 23 million using these two methods on a 250 MHz SGI Onyx2 with 40.5 GB of RAM. The original OoCS took 67 minutes and used 3,282 MB of RAM, while the boundary-based method took 83 minutes and used only 121 MB of RAM; a reduction in memory usage by a factor of 27.

8.6 Buffer-Based Processing

In this section we show how an adaptive simplification method based on iterative edge contraction (Garland and Heckbert, 1997) can use processing sequences. We modify the algorithm by (Wu and Kobbelt, 2003), which uses a buffering mechanism based on a geometric triangle ordering that directly maps to the buffer-based computation abstraction of processing sequences.

Their algorithm uses three operations, READ triangle, DECIMATE triangles, and WRITE triangle, to maintain an *active* portion of the mesh that is memory-resident and eligible for simplification. It stores a quadric error matrix with each active vertex.

READ inputs the next triangle in the triangle ordering, hooks it into the active mesh, and adds the quadric error of the triangle to the quadric matrices of its three vertices.

DECIMATE chooses an edge with minimal quadric error that is eligible for collapse, merges its two active vertices and their quadric matrices, and eliminates the triangles that share the edge. Constant-time complexity is achieved by choosing this edge from only a small, fixed-size set of random candidates.

WRITE chooses a triangle with maximal quadric error that has an edge on the output boundary and outputs it. Again, the search is restricted to a random set of potential output triangles for constant-time selection. When all triangles incident on a vertex have been written, the vertex is deleted together with its quadric.

Wu and Kobbelt READ triangles to keep an in-core buffer full, and interleave batches of WRITE and DECIMATE operations to maintain a simplified mesh whose resolution corresponds to a user-specified percentage reduction of the original mesh.

Figure 8.8 illustrates Wu and Kobbelt's algorithm adapted to the processing sequence paradigm. The unprocessed region is shown at the top. Shown in black is the processing boundary of the input sequence, where new triangles are read and where vertices accumulate the quadric error of incoming triangles in their quadric matrices. Furthermore, the input sequence provides information about connectivity and border edges to the in-core buffer (shown in the middle). Edge collapse operations are disallowed for edges that have vertices on the input or the output boundary. After decimation, the surviving triangles are output in the form of a second processing sequence. Again connectivity and border information is stored along the boundary of the output sequence, allowing for further processing such as on-the-fly compression.

In order to output a processing sequence, we slightly modify Wu and Kobbelt's method to select triangles to output. As in the original method, we try to minimize the number of "start" operations (compare with Figure 8.2) for output triangles in order to keep the output boundary as short and the triangle buffer as connected as possible. This is achieved by choosing an output triangle only from triangles incident to an edge of the output boundary, and allowing "start" operations only if no such triangle is available. Furthermore, we favor outputting triangles whose three vertices are on the output boundary, i.e. "end," "fill," and "join" operations (in that order), since its vertices can no longer be involved in an edge collapse. When no such triangle exists, we choose (using multiple choice selection) some triangle with one vertex between the input and output boundaries, i.e. we perform an "add" operation. To determine which such triangle to output from a set of multiple choice candidates, we choose the one with the largest quadric error at the non-boundary vertex rather than evaluating the quadric error for the entire triangle, as in Wu and Kobbelt's method. We decided on this approach since, in our method, vertices on the output boundary have no impact on the error involved in future potential edge collapses.

When a new vertex is encountered in the input, a corresponding vertex is allocated in the in-core mesh data structure. The processing sequence API optionally maintains

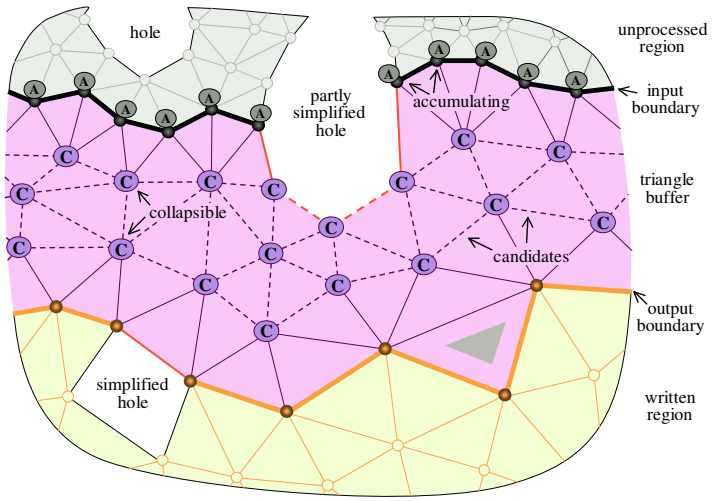


Figure 8.8: A 2D illustration of buffer-based computation using processing sequences. Such an algorithm, here the simplification algorithm of (Wu and Kobbelt, 2003), operates on a triangle buffer between an input and an output boundary. Triangles generated at the input boundary are read from disk. They are not immediately processed, but used to (re-)fill the buffer in which the actual processing takes place. Their quadric error is added to the accumulating error quadrics of vertices on the input boundary. Edge collapse operations are restricted to those edges (shown dashed) that are not incident to vertices on either boundary. They merge collapsible quadrics. Triangles adjacent to the output boundary empty the buffer and are written to disk. The next candidate for output is the triangle with all three vertices on the output boundary (shown in gray).

a mapping between the vertices it knows to be on the boundary and corresponding client-side vertices. This eliminates the need for the client to establish this mapping, e.g. via hashing on global vertex indices, for each previously visited vertex in the sequence, which gives us connectivity reconstruction essentially for free. Furthermore, using processing sequences, the mesh border edges are not (mis-)classified as input boundary edges, as in (Wu and Kobbelt, 2003). This allows border edges and nearby incident edges to be directly involved in decimation; we need not set aside precious space in the fixed-size mesh buffer to hold such edges until the entire input mesh has been read.

8.6.1 Results

Table 8.3 lists the results of running our adaptive simplification method on several meshes. The majority of these meshes were simplified on an 800 MHz Pentium 3 with 880 MB of RAM, running Red Hat Linux 7.1 (allowing a fair comparison with several

mesh name	T_{in}	T_{buf}	T_{out}	p (%)	RAM (MB)	time (h:m:s)	speed (T_{in}/s)
happy buddha	1,087,716	400 K	21,754	2	41	27	40,663
		400 K	217,544	20	41	26	42,434
blade	1,765,388	400 K	35,308	2	41	41	43,292
		400 K	353,078	20	42	45	39,396
david (2mm)	8,254,150	400 K	82,541	1	43	3:06	44,491
		400 K	825,415	10	44	3:50	35,915
lucy	28,055,742	400 K	280,557	1	43	10:05	46,408
		400 K	1,402,788	5	43	10:45	43,502
david (1mm)	56,230,343	400 K	562,303	1	48	14:40	63,898
		400 K	2,811,517	5	48	16:07	58,149
st. matthew	372,767,445	800 K	559,152	0.15	104	1:30:32	68,624
		800 K	1,863,837	0.5	105	1:33:00	66,804
ppm isosurface	467,614,855	4 M	2,346,907	0.5	776	2:25:11	53,883

Table 8.3: Results of buffer-based simplification. T_{buf} specifies the size (in number of triangles) of the in-core buffer, and p is the simplification ratio. For these results, we used 8 multiple choice candidates. The top four models were simplified on an 800 MHz Linux PC, while the bottom three were simplified on a 2 GHz Windows PC.

other methods, including (Wu and Kobbelt, 2003; Lindstrom and Silva, 2001; Cignoni et al., 2003)). The larger meshes were simplified on the same PC that was described in Section 8.5.1. Except for lower memory requirements and higher speed, these results generally agree with those published by Wu and Kobbelt. The performance differences may be attributed in part to our method not requiring hashing, but may also be the result of a more efficient implementation. Finally, Figure 8.9 shows a simplified mesh produced by our method.

8.7 Summary

In this chapter we have demonstrated that the mesh access provided by processing sequences allows highly efficient out-of-core computations on large meshes. We have illustrated this by adapting two simplification algorithms to access the mesh through a prototype of our processing sequence API: one using boundary-based, the other using buffer-based processing. In both cases using processing sequences was beneficial.

Boundary-based processing significantly reduces the memory-requirements of the vertex clustering based simplification method of (Lindstrom, 2000), enabling it to produce very large output meshes in a single pass. Furthermore, the quality of the simpli-

Figure 8.9: Adaptive simplification of David (2mm) to 1% of the input mesh with a stream-based simplifier using processing sequences and buffer-based processing.

fied mesh improves significantly—especially in the case of aggressive simplification—as multiple mesh layers that pierce one grid cell are no longer collapsed into a single vertex. Finally, information about border edges supports dedicated error quadrics that better preserve surface boundaries.

Buffer-based processing readily accommodates the stream-based simplification method of (Wu and Kobbelt, 2003), providing it with a triangle ordering that keeps the buffer maximally connected. Furthermore, the indexed nature of processing sequences removes the overhead associated with polygon soups. Additional speed-ups are gained through assistance in reconstructing connectivity. Finally, information about border edges solves the issue of uncollapsible triangles clogging the triangle buffer.

The maximal length of the processing boundary directly impacts the memory footprint of the simplification process. For the "ppm isosurface" data set of Table 8.1 this length is 1.6 million vertices, far above the theoretical worst-case bound of $O(\sqrt{n})$ that was established by (Bar-Yehuda and Gotsman, 1996). The processing sequences we have used in this chapter were generated by the compression scheme described in the previous chapter. This scheme traverses the mesh with a heuristic that primarily aims at lowering the bit rate and does not attempt to keep the maximal boundary length small. In the next chapter we investigate how to create processing sequences that have a smaller maximal memory footprint. Using different re-ordering strategies, we can easily reduce the maximal processing boundary length of the "ppm isosurface" to around one hundred thousand vertices (see Table 9.1).

In the future we would like to see these two computational abstractions applied to other types of mesh processing, in particular parameterization and remeshing algorithms. The needs of these processing tasks may result in improvements or changes to the definitions of processing sequences. Among other things, the next chapter addresses *on-the-fly* compression of processing sequences that are either output of an algorithm or created from scratch. When compressing processing sequences in a traversal order that is dictated by an application rather than deterministically chosen by the compressor we can expect lower compression rates. However, the benefits of such a scheme is that it will allow both the input and the output sequences to be in compressed form.

8.8 Hindsights

The idea of sequenced processing grew out of the streaming, small memory-footprint mesh access provided by our compressed format. However, it turns out that the depth-

first fashion in which our original scheme was decompressing triangles and vertices resulted in orderings that were far from optimal for some processing tasks. In our case such poorly ordered input sequences can have a negative impact on the quality of buffer-based simplification. When the processing boundary advances in a highly non-uniform manner, some mesh elements spend considerably more time in the triangle buffer than others. The randomized way in which edge-collapse operations are applied to this triangle buffer is likely to simplify those elements more heavily than those of areas where the processing boundary passes through more quickly.

But even if the input sequence is ordered more coherently, the buffer-based simplification method, as described in Section 8.6, tends to output highly incoherent sequences. The reason is that the WRITE operation, which decides in which order the mesh elements are output, takes only the maximal quadric error, but not the time spend in the triangle buffer into account. This results in newer mesh elements of slightly higher quadric error being repeatedly favored for output, which can leave behind many small islands of older elements that remain in the buffer for a long time. When we chose to follow a mainly error-driven output strategy as proposed by (Wu and Kobbelt, 2003) we did not yet have a clear understanding of the concept of *coherent streaming*. This is investigated in detail in the next chapter.

Finally, our definitions about which triangle sequences are allowable processing sequences were overly restrictive. Apart from the fact that growing the mesh in an edge-connected manner results in a smaller maximal footprint as it streams through memory, there is really no good reason to categorically disallow triangles that are only vertex-adjacent to the processing boundary. These restrictions grew out of the triangle orderings of our original compressed format that seemed ideal for processing, but that unnecessarily complicate creating and working with processing sequences. The main advantage of our compressed format was the coherent order of elements, the information on vertex finalization, the additional information about the topological type of vertices and edges, and the ability to store information along the evolving boundary. But we can provide the same functionality for a less restricted ordering of mesh elements as, for example, supported by the general streaming mesh format that we describe in the next chapter. In this sense, processing sequences really can be seen as an abstraction of the functionality that any streaming mesh format can provide.

Chapter 9

Streaming Meshes

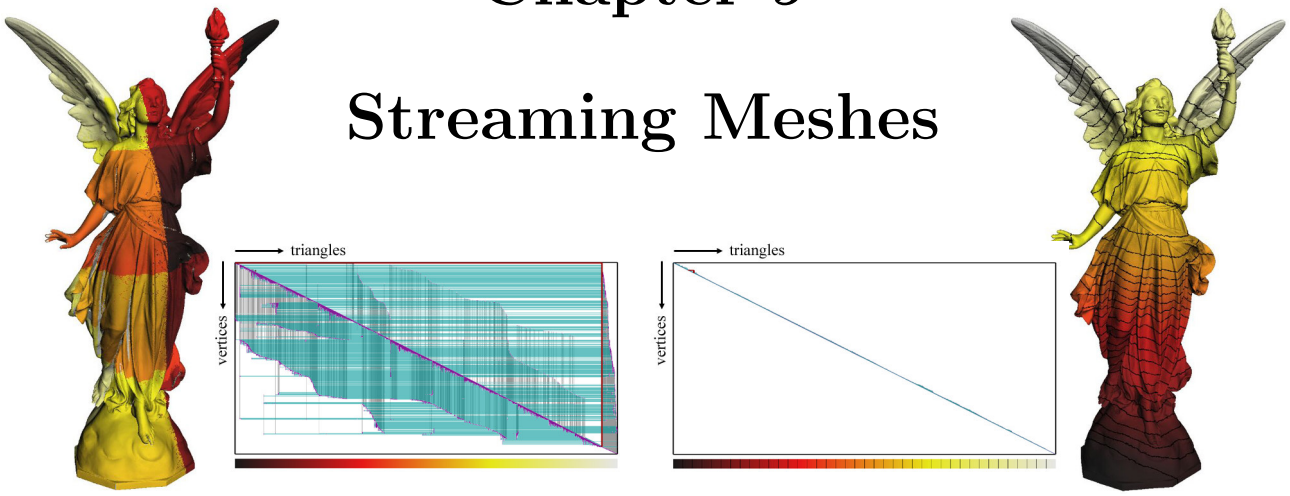


Figure 9.1: Illustrations of the coherence in the layout of the "Lucy" mesh: The original layout (left); the layout after reordering vertex and triangle arrays with spectral sequencing (right). Renderings color-code triangles based on their array position. Layout diagrams connect triangles sharing the same vertex with horizontal line segments (green) and vertices used by the same triangle with vertical line segments (gray).

Today's gigabyte-sized polygon models can no longer be completely loaded into the main memory of common desktop PCs. Unfortunately, current mesh formats were designed years ago and do not account for this. Using such formats to store large meshes is inefficient and unduly complicates all subsequent processing. In this chapter we describe a *streaming* format for polygon meshes that is simple enough to replace current mesh formats and more suitable for working with large data sets. Furthermore, it is an ideal input and output format for IO-efficient out-of-core algorithms that process meshes in the streaming, possibly pipelined, fashion discussed in the previous chapter.

The central theme in this chapter is the issue of *coherent* and *compatible* orderings of the mesh vertices and polygons. We present metrics and diagrams that characterize the coherence of a mesh layout and suggest appropriate strategies for improving its "streamability". To this end, we outline several out-of-core algorithms for reordering meshes with poor coherence, and present results for a menagerie of well known and generally incoherent surface meshes. We also describe novel technique that can compress streaming meshes on-the-fly and in their particular stream-order.

9.1 Introduction

The advances in computer speed and memory size are matched by the growth of data and model sizes. Modern scientific technologies enable the creation of digital 3D models of incredible detail and precision. Recent examples include statues scanned for historical reconstruction, isosurfaces visualized to understand the results of scientific simulation, and terrain measured to predict flood impact. These polygonal data sets easily reach sizes of several gigabytes. Such large amounts of data often exceed the main memory resources of the computing environment that is at a scientist's disposal, which makes the subsequent study of the produced data a challenging task.

In order to process geometric data sets that do not fit in main memory, one resorts to *out-of-core* algorithms. These arrange the mesh so that it does not need to be kept in memory in its entirety, and adapt their computations to operate mainly on the loaded parts. Such algorithms have been studied in several contexts, including visualization, simplification, and compression. A major problem for all these algorithms is to deal with the initial format of the input. Current mesh formats were designed in the early days of mesh processing when models like the Stanford bunny, with less than 100,000 faces, were considered complex. They use an array of floats to specify the vertex positions followed by an array of indices into the vertex array to specify the polygons. The order in which vertices and polygons are arranged in these arrays is left to the discretion of the person creating the mesh. This was convenient when meshes were relatively small. In the meantime, however, our data sets have grown in size by four orders of magnitude. Storing such large meshes in the same format means that a gigabyte-sized array of vertex data is indexed by a gigabyte-sized block of triangle data. This unduly complicates all subsequent processing.

Most processing tasks need to *dereference* the input mesh (e.g. resolve all triangle to vertex references). Memory mapping the vertex array and having the operating system swap in the relevant sections is only practical given a coherent *mesh layout*. The lack of coherence in the layout of the "Lucy" model is illustrated on the left in Figure 9.1. Loosely speaking, the farther the green and grey line segments are from the diagonal, the less coherent is the layout. In order to operate robustly on large indexed meshes an algorithm either needs to be prepared to handle the worst possible inputs or make assumptions that are bound to fail on some models.

In this chapter we present a *streaming* format for large polygon meshes that solves the problem of dereferencing. In addition, it enables the design of new IO-efficient algorithms for out-of-core stream-processing. The basic idea is to interleave indexed

vertices and triangles and to provide information when vertices are referenced for the last time. We call such a mesh representation a *streaming mesh*.

The terms "progressive" and "streaming" are often used synonymously in computer graphics. We point out that our streaming meshes are fundamentally different from the multi-resolution representations used for progressive geometry transmission. Progressive streaming adds more and more detail to a coarse approximation of a mesh stored in-core, possibly until exhausting the available memory (Hoppe, 1996). In our streaming model original triangles and vertices are added to, or removed from, a partial but seamless reconstruction of the mesh that is kept in a finite, fixed-size memory buffer (a "sliding window" over the full resolution mesh).

The advantage of a streaming representation for meshes was already identified in Chapter 7 where we proposed a compressed mesh format that allowed streaming decompression. During compression a set of boundaries was sweeping once over the entire mesh, which was accessed through a complex external memory data structure. The payoff for that initial work was that during decompression only those boundaries needed to be maintained in memory. In the previous chapter we confirmed that the streaming access provided by the decompressor are indeed useful for IO-efficient out-of-core processing of meshes. We showed that simplification algorithms can be adapted to operate in a streaming manner on a mesh. But in these two chapters we payed little attention to what makes good stream orders. In fact, the streaming meshes produced by our out-of-core compressor are not suitable for all types of stream-processing. Although they are reasonably low in width they have maximally poor span.

In this chapter we extract the essence of streaming to define a simple input *and* output format. We propose definitions and metrics that give us a language for talking about streaming meshes. We identify two fundamental stream characteristics, the *width* and the *span* of streaming meshes, and describe the practical impact of these metrics on stream processing. Some algorithms for stream-processing will require meshes with low span, while others will only be *width-limited*. We report the stream characteristic for a number of different mesh orderings and describe out-of-core techniques for creating such orders using limited memory. Furthermore, we describe a scheme for streaming compression. In contrast to previous schemes that dictate the order in which a mesh is compressed, we encode meshes in their stream order. While this does not achieve the same rate of compression, it allows immediate *on-the-fly* compression without cutting the mesh in smaller pieces, as suggested by (Ho et al., 2001) and without resorting to the complex external memory data structure we described in Chapter 7.

The remainder of this chapter is organized as follows: The next section summarizes related work in out-of-core algorithms and how they deal with incoherent mesh layouts. We also mention other work in mesh and graph re-ordering. In Section 9.3 we give useful measures for the coherence of a mesh layout that characterize how *streamable* a given mesh ordering is. These measures tell us how much work is needed to convert this mesh into a streaming format. In Section 9.4 we define streaming meshes, describe the tiny bit of extra information that needs to be included to turn a standard mesh format into a streaming format, and illustrate the big pay-off that this small change has on the workflow in large mesh processing. Although meshes are naturally generated in a streaming manner there are many incoherent data sets around. In Section 9.5 we look into approaches such as geometric sorting and spectral sequencing for creating streaming meshes from incoherent legacy data . We report their success and evaluate their complexity in an out-of-core setting. Finally, in Section 9.6 we describe how streaming meshes can be compressed *on-the-fly*. The last section summarizes our contributions and discusses potential future work such as spatial streaming, streaming in multiple resolutions, and extensions to regular and irregular volume meshes.

9.2 Related Work

While models from 3D scanning or iso-surface extraction have become too large to fit in the main memory of commodity PCs, storing the models on hard disk is always possible. Out-of-core algorithms are designed to efficiently operate on large data sets that mostly reside on disk. To avoid constant reloading of data from slow external memory, the order in which they access the mesh must be consistent with the arrangement of the mesh on disk. Currently the main approaches are: cutting the mesh into pieces, using external memory data structures, working on dereferenced triangle soup, and operating on a streaming representation. All these approaches have to go through great efforts to create their initial on-disk arrangement when the input mesh comes in a standard indexed format.

Mesh cutting methods partition large meshes into pieces that are small enough to fit into main memory and then process each piece separately. This strategy has been successful for distribution (Levoy et al., 2000), simplification (Hoppe, 1998; Bernardini et al., 2002), and compression (Ho et al., 2001). The initial cutting step requires dereferencing, which is expensive for standard indexed input.

In hindsight, once meshes became so large that it became impractical to store them in standard indexed formats, one should have designed a more scalable format instead

of cutting up the data. The practice of mesh cutting is also responsible for some of the poor mesh layouts that we have to deal with.

Approaches that use **external memory data structures** also partition the mesh, but into a much larger number of smaller pieces often called *clusters*. At run-time only a small number of clusters is kept in memory with the majority residing on disk from where they are paged in as needed. (Cignoni et al., 2003), for example, use such an external memory mesh to simplify large models with iterative edge contraction. In a similar manner we have used an out-of-core mesh in Chapter 7 to compress large models with region growing. Building these data structures from a standard indexed mesh involves additional dereferencing passes over the data.

One approach to overcome the problems associated with indexed data is not to use indices. Abandoning indexed meshes as input, such techniques work on **dereferenced triangle soup**, which streams from disk to memory in increments of single triangles with the processor operating at full capacity. (Lindstrom, 2000) showed how to implement vertex-clustering based simplification this way. Although his algorithm does not use indices, his input meshes usually come in an indexed format. Ironically, in this case an initial dereferencing step (Chiang and Silva, 1997) becomes necessary that does exactly what the algorithm itself later avoids: it resolves all triangle to vertex references. In order to take full advantage of this type of processing, the input must already be streamable.

While the entire mesh may not fit into main memory, one can easily store a working set of several million triangles. (Wu and Kobbelt, 2003) simplify large models by streaming coherent triangle-soup into a fixed-sized memory buffer, in which they perform randomized edge collapses. Connectivity between triangles is reconstructed through geometric hashing on vertex positions. Only vertices surrounded by a closed ring of triangles are deemed eligible for simplification. Mesh borders can not simplified until the entire mesh was read, which implies that border triangles and vertices remain in the buffer until the end. For meshes with borders their simplified output meshes are therefore guaranteed to have an incoherent layout. In the previous chapter we have shown that the processing sequence abstraction of our compressed format provides exactly the kind of information that Wu and Kobbelt's algorithm needs: finalization of vertices. Instead of the algorithm having to guess when a vertex is final, our compressed format informs when this is indeed the case. However, their simplified meshes will still be incoherent unless their output strategy is modified to also take coherence into account (see also Section 8.8).

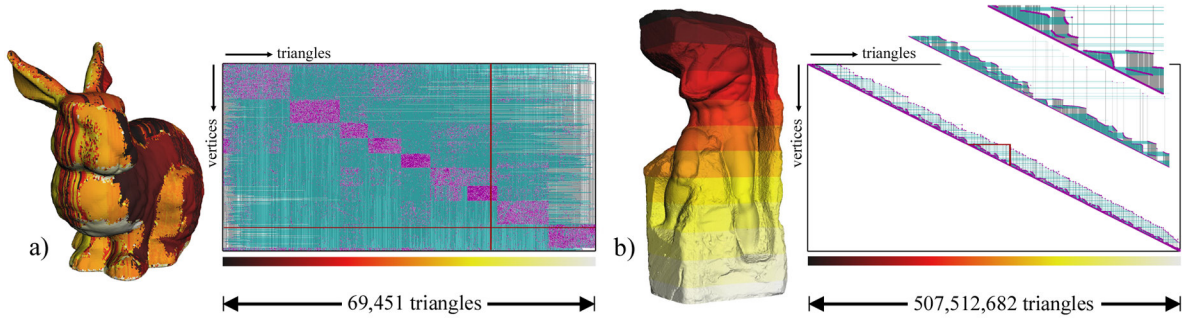


Figure 9.2: Visual illustrations of mesh layouts: (a) The "bunny" and (b) the 10,000 times more complex "Atlas" model. Successive triangles are rendered with smoothly changing colors. Layout diagrams intuitively illustrate incoherence in the meshes.

Coherence in reference has previously been investigated in the context of efficient rendering. Modern graphic cards use a vertex cache to buffer a small number of vertices. In order to make good use of the cache it is imperative for subsequent triangles to re-reference the same vertices. (Deering, 1995) stores triangles together with explicit instructions that tell the cache which vertices to replace. (Hoppe, 1999) produces coherent triangle orderings optimized for a particular cache size, while (Bogomjakov and Gotsman, 2001) create orderings that work well for all cache sizes.

An on-disk layout that is good for streaming seems similar to an in-memory layout that is good for rendering. But there are differences: For the graphics card cache it is expected that at least some vertices are loaded multiple times. In our case, each vertex is loaded only once as main memory can hold all required vertices for any reasonable traversal. Once a vertex is expelled from the cache of a graphics card, it makes no difference how long it takes until it is loaded again. In our case, the duration between first and last use of a vertex does matter. While local coherence is of crucial importance for a rendering sequence, it has little significance for streaming. What is of big practical difference here is whether the layout has global coherence or not. While the triangle orderings that (Bogomjakov and Gotsman, 2001) create with recursive graph partitioning are good for rendering, they have global incoherence and are therefore not good for all types of stream processing. Vice-versa, a geometric sort of the mesh triangles produces a good stream ordering but constitutes a rather poor rendering sequence.

9.3 Mesh Layouts

Indexed mesh formats impose no constraints on the order of either vertices or triangles. In particular, the three vertices of a triangle can be located *anywhere* in the vertex

array. They need not even be close to each other. And while subsequent triangles may reference vertices at opposite ends of the array, the first and the last triangle can use the same vertex. This flexibility was enjoyable as long as meshes were small or moderately sized. However, with the arrival of gigabyte-sized data sets this has become a major headache. Today's mesh formats have originated from a smorgasboard of legacy formats (e.g. PLY, OBJ, IV, OFF, VRML). They were designed in the early days of computer graphics when the polygon models were of the size of the Stanford bunny. This model, which has helped popularize the PLY format, abuses this flexibility like no other. Its layout is highly incoherent in every respect, which is illustrated in the form of a *layout diagram* in Figure 9.2.

A layout diagram intuitively visualizes the coherence in reference between vertices, which are indexed along the vertical axis, and triangles, which are indexed along the horizontal axis. Both are numbered in the order they appear in the file. We draw for each triangle a point (violet) for each of its three vertices and a vertical line segment (grey) that connects them. Similarly, we draw for each vertex a horizontal line segment (green) that connects the first and last triangle that reference it. Loosely speaking, the closer that points and lines group around the diagonal the more coherent is the layout.

Nowadays the PLY format is used to archive the scanned statues that were created by Stanford's Digital Michelangelo Project (Levoy et al., 2000). For the "Atlas" statue of 507 million triangles this means that a six gigabyte array of triangles references into a three gigabyte array of vertex positions. Its layout diagram, which is shown in Figure 9.2, reveals that vertices are used over spans of up to 550,000 triangles—equaling 700 MB of the triangle array. Since an indexed mesh of this size cannot be dealt with on commodity PCs, the statue is provided in twelve pieces.

9.3.1 Definitions

The layout of a mesh is defined by the ordering of its vertices *and* the ordering of its triangles. The following definitions are helpful to characterize the quality of a particular mesh layout (see Figure 9.3):

The *triangle span* of a vertex is the number of triangles between and including its first and last use. It corresponds to the green horizontal segments in a layout diagram. The triangle span of a layout is the maximal triangle span of any of its vertices. The *vertex span* of a triangle is the maximal index difference (plus one) amongst its vertices. It corresponds to the grey vertical segments in a layout diagram. The vertex span of a layout is the maximal vertex span of any of its triangles. The *vertex width* of a layout is

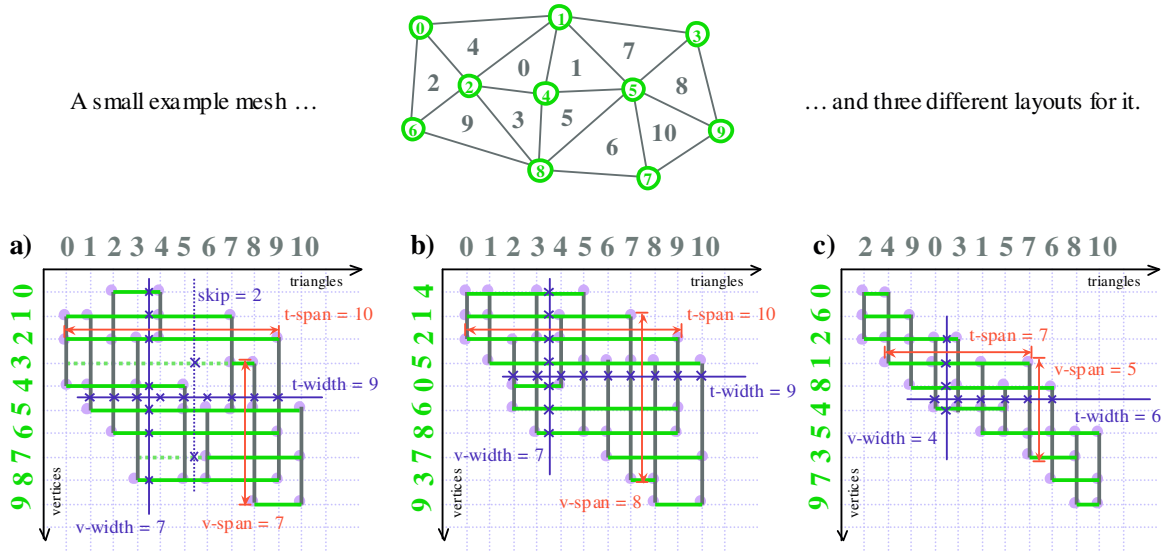


Figure 9.3: A small example mesh in three different layouts: (a) The vertex order is not compatible with the triangle order resulting in a skip. (b) Reordering the vertices can eliminate the skip but does not affect the triangle span or the vertex width. (c) Triangle span and vertex width can only be reduced by also changing the triangle ordering.

the maximal number of green segments that can be cut by a vertical line; the *triangle width* is the maximal number of gray segments cut by a horizontal line. The *skip* of a layout is the maximal number of "concurrently" skipped vertex indices. An index is skipped as long as its vertex is not referenced while a vertex with a higher index has already been referenced. When the skip of a layout is large we say that its vertex ordering is *incompatible* with its triangle ordering.

Three different layouts for a small example mesh are shown in Figure 9.3. The vertex ordering of the first layout is not compatible with its triangle ordering. The layout has a skip of 2 because vertex #8 is already used by triangle #3, while the vertices #3 and #7 are still unused. In the second layout we have re-ordered the vertices, which corresponds to vertically rearranging the green line segments. This neither changes the vertex width nor the triangle span of the layout, but affects the skip, the triangle width, and the vertex span. To reduce the vertex width and the triangle span we have to also reorder the triangles, which is illustrated by the third layout.

9.3.2 Incoherent Layouts

The incoherence in a mesh layout can often be explained by how the mesh was produced. The "horse", for example, is zipped together from multiple pieces that are result of

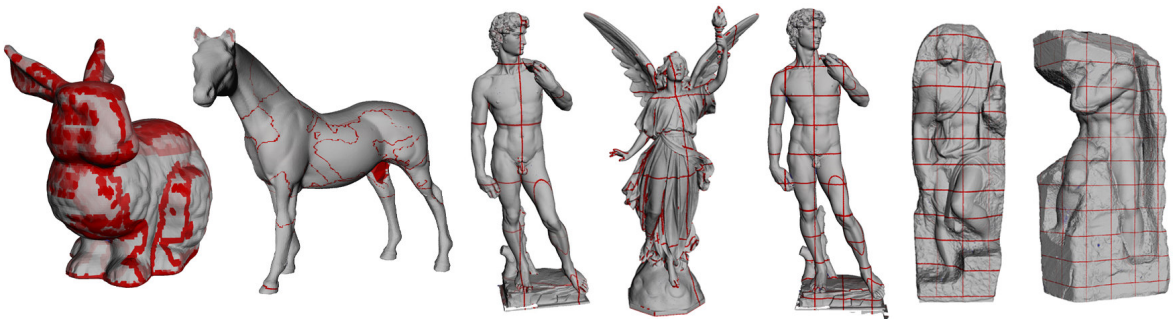


Figure 9.4: Highlighting triangles with high vertex span often reveals something about how the mesh was created or modified.

scanning from different viewpoints. While the zipping algorithm sorted the triangles spatially along one axis, it simply concatenated the vertex arrays—thereby creating triangles with high vertex spans along the zips. The "dinosaur" has its triangles ordered along one axis and its vertices along another axis. This projects the model along the third axis into vertex and triangle indices such that they capture a distorted 2D view of the shape. This layout is low in width and span, but has a high skip. For the most part, the "dragon" has its vertices and triangles loosely ordered along the z-axis. But there are a small number of vertices at the very end of the vertex array that are used all across the triangles array, leading to high vertex span. This is due to a post-processing operation for topological cleanup of holes in the mesh.

The large Stanford statues were extracted block by block from a large volumetric representation. The resulting surfaces were then stitched together on a supercomputer by concatenating triangle and vertex arrays and identifying vertices between neighboring blocks, which is evidenced by high vertex spans in Figure 9.4. For the two largest statues the vertex and triangle spans were somewhat reduced when their "blocky" layouts were multiplexed into several files by spatially cutting the statues into twelve horizontal slices.

9.4 Streaming Meshes

A streaming mesh format interleaves the vertices and the triangles that use them and provides explicit information about when vertices are no longer used. Such a format can be as simple as the ASCII examples in Figure 9.5. Despite its simplicity, a streaming mesh format has tremendous advantages over standard formats. Because the format tells us which vertices to keep in memory, the problem of repeated, possibly incoherent look-up of vertex data in a gigantic array does not exist. Furthermore, the fact

that a streaming mesh format contains information about when vertices are no longer used allows streaming, possibly pipelined, processing. Envision a scenario where one algorithm extracts an isosurface and pipes it as a streaming mesh to a simplification process, which in turn streams the simplified mesh to a compression engine that encodes it and immediately transmits the resulting bit-stream to a remote location where triangles are rendered as they decompress. In fact, we now have all components of this pipeline. The streaming format makes it possible to pipe them all together.

Simply put, a streaming mesh format makes operating on even the largest of all data sets a feasible tasks. For example, all images in this paper are interactively rendered on a laptop with 512 MB of memory by simplifying the full resolution input *on-the-fly* with vertex clustering: Read vertices are accumulated in a sparse uniform grid. The mapping from original vertex indices to grid cells is stored in a hash where it is looked up by incoming triangles. The fact that a hash entry can be removed as soon as its corresponding vertex is finalized keeps the overall memory requirements low.

Finally, a streaming format will make creators of large data sets aware of the mesh layouts they produce. It will encourage them to output large meshes into streaming rather than indexed formats and to take coherence in the output into consideration. Anyone who went through the pain of stitching together the "Atlas" statue from the twelve pieces that it is provided in, will appreciate this as an important contribution.

9.4.1 Definitions

A streaming mesh is a logically interleaved sequence of indexed vertices and triangles *plus* information about when vertices are *introduced* and when they are *finalized*. A vertex is either explicitly introduced when it appears in the stream or implicitly introduced if a triangle references the vertex before it appears. Vertices become *active* when they are introduced and cease to be active when they are finalized. We call the evolving set of active vertices the *front* F_i, which at time i partitions the mesh into finalized (i.e. processed) vertices and vertices not yet encountered in the stream. The *front width* (or simply the *width*) is the maximal size $\max_i\{|F_i|\}$ of the front, i.e. the maximal number of concurrently active vertices. The width gives a lower bound on the memory footprint as any stream process must maintain the front. The *front span* (or simply the *span*) is the maximal index difference $\max_i\{\max F_i - \min F_i + 1\}$ of vertices on the front, and intuitively measures the longest duration a vertex remains active.

We place no restriction on whether vertices precede triangles (as would normally be the case in a standard indexed mesh) or follow them. Streaming meshes are *pre-order*

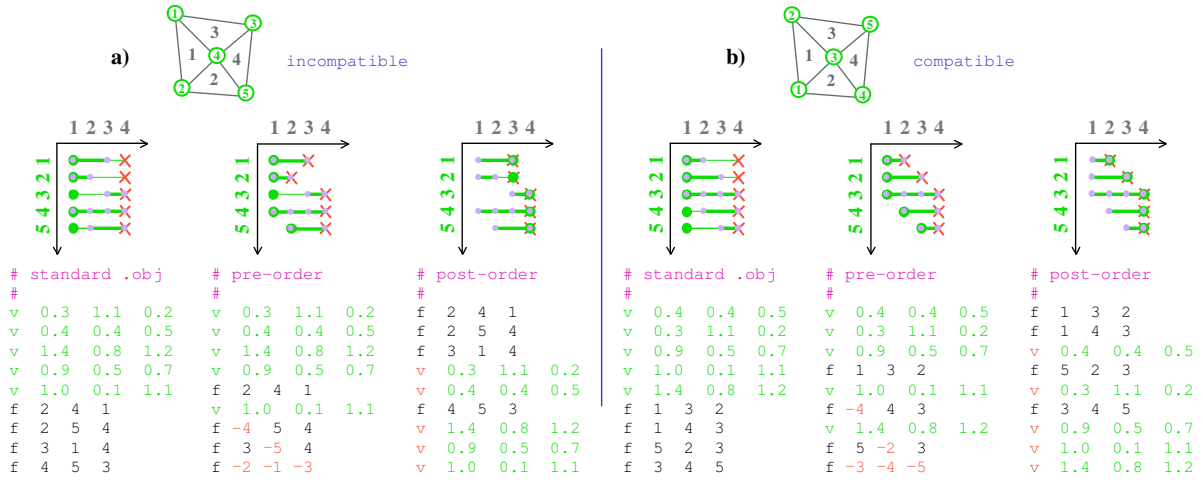


Figure 9.5: Simple ASCII examples illustrating two streaming mesh format in comparison to the standard OBJ format. Streaming pre-order format: finalization is explicitly coded through negative relative indices for the last vertex use, introduction implicitly corresponds to appearance. Streaming post-order format: all information is implicit, finalization given implicitly by appearance of a vertex, while introduction is detected as vertex first use. (a) The incompatible layout forces us to introduce vertex #3 earlier than necessary in pre-order and to finalize vertex #2 later than necessary in post-order. (b) By changing the order of the vertices the mesh can be streamed more efficiently.

if each vertex precedes all triangles that reference it, and are *post-order* if each vertex succeeds all triangles that reference it; otherwise they are *in-order*. The introduction of a vertex does not necessarily coincide with its appearance in the stream as triangles can reference and thus introduce vertices before they appear.

The latest that a vertex can be introduced is just before the first triangle that references it, and the earliest that a vertex can be finalized is just after the last triangle that references it. We can keep the front small in a pre-order mesh by delaying the appearance (introduction) of a vertex as much as possible, i.e. such that each vertex when introduced is referenced by the next triangle in the stream. Conversely, in a post-order mesh each finalized vertex would be referenced by the previous triangle. We say that a stream is *vertex-compact* if each vertex is referenced by the previous or the next triangle in the stream. Vertices can always be made compact with respect to the triangles by rearranging them, which eliminates the skip and causes front width to equal vertex width. Similarly, we say that a stream is *triangle-compact* if each triangle references the previous or the next vertex in the stream. For a pre-order mesh this means that each triangle appears immediately after its last vertex has appeared; for a post-order mesh each triangle appears just before its first vertex is finalized. Note that

vertex-compactness does not imply triangle-compactness, and vice versa. It is always possible to rearrange the triangles to make them compact with respect to a given vertex layout, which causes front span to equal vertex span (since the oldest active vertex could be finalized if it were not waiting on a neighbor). Finally, a streaming mesh is *compact* if it is both vertex- and triangle-compact (see Figure 9.5).

9.4.2 Working with Streaming Meshes

Streaming meshes are ideally suited for the type of processing that was introduced in the previous chapter. In this model, the mesh streams through an in-core *stream buffer*, which is large enough to hold all active mesh elements, i.e. its size is at least the front width. For straightforward tasks that simply need to dereference the vertices, such as rendering a flat shaded mesh, a minimal stream buffer is needed. For more elaborate processing tasks, a larger stream buffer is used that may hold as many additional mesh elements as there are memory resources available. We call the loops of edges that separate already read triangles from those not yet read an input *stream boundary*. For applications that write meshes, there is an equivalent output boundary.

Streaming meshes allow pipelined processing, where multiple tasks run concurrently on separate pieces of the mesh. One module's output boundary then serves as the downstream input boundary for another module. Because the mesh is only operated on in a single pass and because the streamed data is accessed sequentially, we can create fast out-of-core stream modules, for compression, simplification, or similar batch processing tasks, each with processing speeds on the order of 100,000 triangles per second.

The width of a streaming mesh is of particular interest as it is a lower bound for the minimal amount of memory required to seamlessly process a mesh, so in general we like the width to be as low as possible. But many processing tasks are inherently span-limited. Any process that requires maintaining the same order between input and output triangles while doing computation on them that involve neighboring elements must buffer a number of elements that is at least as large as the span. For example, conversion between pre-order and post-order streaming meshes is a span-limited operation if the triangle order is to be preserved. The same holds true for converting from non-compact to compact streaming meshes. These are common operations on streaming meshes as algorithms like to consume compact pre-order input but often produce non-compact post-order output. Although not inherently span-limited, randomized algorithms, such as the simplification algorithm described in Section 8.6 for example, can benefit from low span input meshes. When all processing boundaries advance at the

same speed, no complictated mechanism is needed to assure each mesh element roughly the same probability of being considered for processing.

Streaming meshes are a light-weight mesh representation that does not provide information such as manifoldness, valence, incidence relationships, and other useful topological attributes. But we can easily convert them to processing sequences, which provide this information and also let the user store data along the stream boundaries. We have an automated process for converting streaming meshes to processing sequences that was already mentioned in Section 8.5. It temporarily buffers vertices and triangles in a *waiting area* within which triangles await the finalization of their vertices. In practice, this buffer does not need to be much larger than the stream width. As a result we can read and write simple streaming meshes but retain the option to process them through the more powerful processing sequence API.

9.5 Generating Streaming Meshes

Many applications that generate large meshes can easily produce streaming meshes. They only need to interleave the output of vertices and triangles and provide information when vertices are no longer used. Even if this information is not exact, some conservative bounds often exist. For example, a marching cubes implementation for extracting iso-surfaces from a large regular volume grid could output all vertices of one volume layer, followed by a set of triangles, and then finalize the vertices before moving on to the next layer. This is the technique we used to produce the coherent "ppm" mesh from Table 9.1 that was extracted from a gigantic regular volume grid of size $2048 \times 2048 \times 1920$. Here, even *implicit* finalization in the form of a bound on the maximum number of vertices per layer would be sufficient to finalize vertices.

In this sense, streaming meshes are often the natural output of existing applications. Given limited memory resources, it is quite *difficult* to produce totally incoherent meshes as the mesh generating application can only hold and work on small pieces of the data at any time—unless, of course, this mesh generating application has plenty of main memory at its disposal. The reason that the large statues from Stanford's Digital Michelangelo Project (Levoy et al., 2000) are so incoherent is that they were generated on a supercomputer with gigabytes of main memory. But with such powerful equipment at hand it would have been simple enough to apply a post-processing step for bringing the mesh elements into a more coherent order before distributing these large data sets. Since there is a large body of incoherent meshes that are stored in various legacy formats, we now outline several out-of-core algorithms that are mainly based on external

sort and that can be used to convert these meshes from standard indexed formats to a streaming format. These algorithms may also be used to improve the layout of existing streaming meshes that either introduce/finalize vertices too early/late or that have an overly incoherent triangle ordering.

9.5.1 Interleaving

If the mesh layout is reasonably coherent, we can construct a streaming mesh from an indexed mesh by *interleaving* vertices and triangles to the greatest extend possible without reordering them. The interleaving step is straightforward given independent access to vertex and triangle arrays. The non-trivial step of the algorithm lies in computing when to finalize the vertices. We here sketch an out-of-core algorithm that outputs a pre-order streaming mesh.

In an initial pass over the input mesh we write vertices and triangles to separate temporary files. We also output all *corners* of the mesh to a temporary corner file, i.e. for each triangle $t = \{i, j, k\}$ we output $\langle i, t\rangle$, $\langle j, t\rangle$, $\langle k, t\rangle$. We then externally sort the corner file on the vertex field, which groups a vertex's incident corners together. Next, we scan the sorted corner file. We fetch all corners of a vertex, determine the triangle t_{max} with the last reference to this vertex, and store it to a temporary reference file. In a final pass, we scan the triangle file. For the current triangle we advance in parallel on the vertex and the reference files to the largest indexed vertex. For each read vertex v we output the coordinates to the streaming mesh and insert the record $\langle v, t_{max}\rangle$ into an in-core hash indexed by v. Then we look up each of the current triangle's vertices $\{i, j, k\}$ in the hash to see if their t_{max} equals the current triangle index. If so, we include finalization information when outputting this triangle to the streaming mesh. Before continuing with the next triangle, we remove all finalized vertices from the hash.

This algorithm uses in-core storage (for the hash table) proportional to the width of the created mesh. This dependence is not particular restrictive as there is no benefit in creating streaming meshes with high front width. It makes little sense to apply interleaving to meshes with high skip such as the "dinosaur" or the "Lucy" model. To stream these meshes we need to change either their vertex or triangle order or both.

9.5.2 Reordering

All of the streaming mesh reorderings tools that are described in the following rely on the on the same basic steps. To create a streaming mesh in pre- or post-order, we need: a vertex layout, a triangle layout, and finalization information.

Layout In an initial pass over the input mesh we write vertices and triangles to separate temporary files. We store with each vertex its original index so that after reordering it can be identified by its triangles. Usually we will specify only one layout explicitly and ensure the other layout is made compatible. Each explicit layout is specified as an array of unique sort keys, one for each input vertex or triangle, which we merge with the input elements into their temporary files and on which we perform an external sort (on increasing key value) to bring the elements into their desired order.

For a specified triangle layout, we assign (not necessarily unique) sort keys k to vertices v based on their new incident triangle indices t: for pre-order meshes we use $k_v = \min_{v \in t} t$; for post-order $k_v = \max_{v \in t} t$. Conversely, if a vertex layout is specified, we compute pre-order triangle keys $k_t = \max_{v \in t} v$ and post-order keys $k_t = \min_{v \in t} v$. These keys are based on the indices in the reordered layout. Thus, when an explicit vertex order is specified we must first dereference triangles and update their vertex indices. For a conventional indexed mesh, we accomplish this dereferencing step via external sorts on each vertex field (Chiang and Silva, 1997). If the input is already a streaming mesh, we can accomplish this step much faster by dereferencing (active) vertices, whose keys are maintained in-core, on-the-fly as the input is first processed.

Finalization For nonstreaming input we compute implicit finalization information by first writing all corners $\langle v, t \rangle$ to a temporary file and sorting it on the vertex field v. We then compute the degree $d = |\{t : v \in t\}|$ for each vertex, which will later be used as a reference count. For streaming input, degrees are computed on-the-fly.

Output We now have a file with vertex records $\langle k_v, v, d, x, y, z \rangle$ and a file with triangle records $\langle k_t, v_1, v_2, v_3 \rangle$ that we externally sort on the k fields. We then output a streaming mesh by scanning these files in parallel. Pre-order output is driven by triangles: for each triangle we read and output vertices one at a time until all three vertices of the triangle have been output (possibly also outputting skipped vertices not referenced by this triangle). Conversely, for a post-order mesh we drive the output by vertices: for each vertex we tap the triangle file until all incident triangles have been output. We maintain for each active vertex a degree counter that is decremented each time the vertex is referenced. Once the counter reaches zero the vertex is finalized and removed.

Compaction

We can always eliminate the skip in a mesh layout by reordering the vertices. To avoid skips, we must ensure that vertices appear in the order they are first referenced. We always can do this with pre-order *vertex compaction* by fixing the triangles and

reordering the vertices using the pre-order vertex sort keys defined above. Hence, during output each triangle's vertices have either already been output or appear next in the vertex file. As in the case of interleaving, this algorithm is width-limited. Note that the corresponding algorithm for post-order is span-limited (unless we allowed reordering of triangles) and either requires $O(span)$ memory or additional external sorts.

If the vertex layout is already coherent but the triangle layout is not, *triangle compaction* is worthwhile. For each vertex, in pre-order triangle compaction we immediately output all triangles that can be formed with this and previous vertices; in post-order compaction we output all triangles formed with this and later vertices. Because triangle compaction can shorten the spans of up to three vertices for each reordered triangle, it has the potential to be more effective at improving the streamability of a mesh than vertex compaction, for which moving a vertex only affects its own span.

Because of inter-dependencies creating streams that are fully compact is more challenging than ensuring vertex- or triangle-compactness alone. Given a triangle sequence we output a compact pre-order mesh as follows. For each remaining triangle t in the sequence, we output its vertices one at a time if they have not yet been output (thus ensuring vertex-compactness). For each newly output vertex v, we also output any triangle (other than t) incident on v whose other vertices have also been output (thus ensuring triangle-compactness). We then output t and continue with the next triangle. This (possibly rearranged) triangle order is therefore induced by the vertex-compact vertex order given by the original triangle order, i.e. $k_t = \max_{v \in t} \min_{v \in s} s$.

Spatial Sorting

If the vertex and triangle orders are both inherently incoherent, then compacting vertices or triangles is futile if we seek to reduce both width and span. Such meshes need to be reordered from scratch. Perhaps the simplest method for doing this is to linearly order its elements along a spatial direction such as its maximal x, y, or z extent. We first "rank" the vertices by sorting them in geometric order and use the rank both as the new (unique) index and sort key. Once vertices and triangles have been sorted, we drive the output by triangles (vertices) to produce a vertex-compact (triangle-compact) mesh. Because the sorted vertex order is close to but not guaranteed vertex-compact, we maintain a delay buffer for vertices that would be skipped if output immediately.

We also examine layouts based on space-filling curves. For simplicity, we chose the (Morton order) z-curve, for which we can compute sort keys by quantizing the vertex coordinates and interleaving their bits, e.g. as $x_n y_n z_n x_{n-1} y_{n-1} z_{n-1} \cdots x_1 y_1 z_1$.

Topological Sorting

An alternative to spatial sorting is topological traversal of the mesh. In Table 9.1 we report results for breadth-first vertex sorts and depth-first triangle sorts. Breadth-first traversals naturally lead to a low span as they always continue with the "oldest" element. One detail worth noting is that a breadth-first traversal must consider triangles that are only vertex-adjacent (e.g. triangles around non-manifold vertices) to avoid leaving such vertices "hanging," which would result in large triangle spans. A depth-first traversal by definition leaves mesh elements hanging and therefore guarantees layouts with high span as evidenced in Table 9.1. It also results in considerably higher width—especially for high genus meshes such as the "ppm" surface. For each topological handle in the mesh, the front elements of a traversal eventually split into two unconnected groups. A depth-first traversal leaves one group hanging on the stack until reaching it from the other side.

Our breadth-first sort has been designed to output fully compact meshes. The output is vertex-driven, thus ensuring triangle compactness. We use as primary sort key for each vertex the index of its least recently output neighboring vertex. To break ties among neighbors, we use as secondary key whether or not the vertex forms one or more triangles with already output vertices, which eliminates skips and guarantees vertex compactness. Finally, as tertiary key we use the most recently output neighbor, which results in coherent "walks" around each stream boundary.

We currently do not have out-of-core algorithms for these topological sorts. The results in Table 9.1 were obtained with an in-core algorithm on a supercomputer with large memory capacity. However, these topological traversals are readily accommodated by our out-of-core mesh, the external memory data structure that was described in Section 7.3. Alternatively, we can also use the clustering approach that we implemented for spectral sequencing (which is described next) to create these topological orderings.

Spectral Sequencing

For many meshes, geometric or topologic sorting will produce sufficiently coherent layouts. However, if the mesh is "curvy" (such as the "dragon" model) with changing principal direction or if the mesh is "spongy" (such as the "ppm" surface) with complex topology and space-filling geometry, these strategies produce orderings that are far from the best possible. The traversal shown in Figure 9.6, for example, follows the winding body of the "dragon" and achieves a much lower front width. Intuitively speaking, we want a rubber band to sweep over the connectivity graph so that it has the least

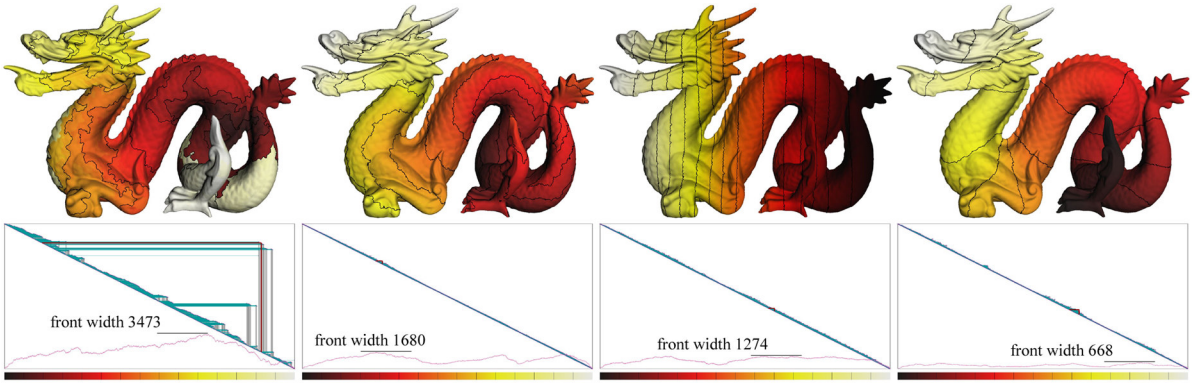


Figure 9.6: The "dragon" mesh reordered by (a) a depth-first sort compressor, (b) a breadth-first sort compressor, (c) spatial sort, and (d) spectral sequencing.

maximal expansion. We now describe a method particularly aimed at generating low-width orderings. Because stream boundaries have to turn corners to be as short as possible and do not advance uniformly, the span will suffer in favor of the width.

We first note that in a triangle-compact mesh the triangle order is "induced" by the vertex order, and we can therefore without loss of generality focus only on ordering vertices and treat this as a graph layout problem since triangle compaction can only further reduce the width and span. In a triangle-compact mesh front span equals vertex span, which in turn is equivalent to graph *bandwidth* (Díaz et al., 2002), while front width, also known in the finite element literature as *wavefront* (Scott, 1999), is equivalent to *vertex separation* (Díaz et al., 2002).

Both bandwidth and vertex separation are known to be NP-hard, and hence heuristics are needed. Intuitively, breadth-first sorting is a good heuristic for bandwidth, and is often used as an initial guess in iterative bandwidth minimization methods. One popular heuristic for vertex separation is *spectral sequencing*, which minimizes the sum of squared edge lengths in a linear graph arrangement. Spectral sequencing amounts to finding a particular eigenvector (the *Fiedler vector*) of the graph's Laplacian matrix. To solve this problem efficiently, we use the ACE multiscale method (Koren et al., 2002).

To make the problem more tractable for large meshes, we presimplify the input using a variation of the streaming edge collapse technique from (Isenburg et al., 2003), and contract vertices into clusters based purely on topological criteria aimed at creating uniform and well-shaped clusters. Clusters are maintained in a memory-mapped array as circular linked lists of vertices, using one index per vertex. We then apply ACE to order the clusters in-core, and finally order the mesh cluster by cluster, with

mesh description			original layout				spectral sequencing		geo. sort	topo. sort
name / genus / # comp. / # vertices / # triangles	skip / v-width / t-width / v-span / t-span	layout diagram	*inter-leaved* width span	*v-com-pacted* width span	*t-com-pacted* width span	snapshots	width span	snapshots	*linear* width span / *z-order* width span	*breadth* width span / *depth* width span
bunny	34,569		34,813	9,133	3,924		**228**		413	334
0	9,133		34,834	34,549	34,641		785		1,502	**354**
1	11,135								583	405
35,947	35,742								22,999	21,704
69,451	69,181									
horse	40,646		40,653	550	2,070		**303**		419	443
0	550		48,485	3,167	48,471		3,286		2,563	**466**
1	4,272								482	440
48,485	48,471								23,728	47,446
96,966	6,204									
dinosaur	55,196		55,331	496	1,028		**241**		568	357
0	496		55,680	1,083	4,353		1,382		1,825	**383**
1	2,048								991	409
56,194	4,353								42,083	51,315
112,384	2,017									
armadillo	171K		172K	51,951	17,873		**638**		1,042	1,115
0	51,951		172K	172K	172K		4,405		3,796	**1,199**
1	40,529								1,160	1,457
172,974	172K								124K	171K
345,944	345K									
dragon	434K		434K	4,586	3,918		**668**		1,274	1,680
46	4,586		434K	54,825	434K		11,617		9,243	**2,015**
151	7,147								1,713	8,583
437,645	434K								252K	435K
871,414	109K									
buddha	94,080		98,121	5,037	3,472		**883**		1,556	1,975
104	5,037		111K	102K	24,889		6,993		12,682	**2,335**
1	6,907								1,688	14,639
543,652	24,889								205K	543K
1,087,716	205K									
thai statue	0		53,416	53,416	29,337		**3,761**		6,003	7,051
3	53,416		4.70M	4.70M	4.70M		150K		43,970	**7,897**
1	54,832								4,989	35,461
4,999,996	4.70M								1.93M	4.99M
10,000,000	9.41M									
lucy	11.5M		11.6M	255K	113K		5,841		**4,985**	5,904
0	255K		13.5M	13.5M	13.5M		200K		20,362	**6,547**
18	231K								11,654	12,904
14,027,872	13.5M								5.63M	12.4M
28,055,742	26.8M									
david$_{1mm}$	1,568		26,405	26,383	26,375		**7,862**		8,919	8,282
137	26,383		15.8M	15.8M	15.8M		752K		36,421	**8,971**
2,322	52,515								10,705	35,770
28,184,526	15.8M								6.30M	28.1M
56,230,343	31.5M									
st. matthew	2,121		31,932	31,931	31,895		33,029		33,207	**23,602**
483	31,931		29.1M	29.1M	29.1M		3.85M		157K	**25,554**
2,897	58,916								23,858	110K
186,836,665	29.1M								32.8M	185M
372,767,445	58.3M									
ppm	306K		616K	311K	381K		**56,179**		114K	99,410
167,636	311K		617K	462K	617K		27.0M		290K	**112K**
167,584	813K								148K	3.07M
234,901,044	617K								125M	206M
469,381,488	924K									
atlas	139		28,705	28,701	28,701		45,998		**22,638**	29,923
5,496	28,701		30.6M	30.6M	30.6M		28.5M		64,354	**32,156**
38	58,281								37,469	246K
254,837,027	30.6M								94.8M	254M
507,512,682	61.2M									

Table 9.1: Layout and stream measures for the meshes used in our experiments. We report the skip, vertex span, and triangle width of the original vertex order, and the vertex width and triangle span of the original triangle order (which can be quite incoherent). Starting from the original layout, we report the front width and span of pre-order streaming meshes created by interleaving, vertex compaction, and triangle compaction, and include snapshots of these layouts. The rightmost columns highlight the improvements of vertex-compact streams obtained by reordering triangles and vertices using spectral sequencing, geometric sorting along the axis of maximum extent and along a z-order space-filling curve, and topological breadth- and depth-first traversals. We also list the genus and component, vertex, and triangle counts for each mesh.

no particular vertex order within each cluster. While the intra-cluster order can be improved, the reduction in width is bounded by the cluster size.

A few implementation details: To keep track of triangle clusters, we use a memory mapped array containing one 32-bit index per triangle. Ultimately this array will report the position of each triangle in the reordered mesh. Initially this array represents circular linked lists that join triangles into clusters. When an edge collapse joins two clusters, their corresponding lists can be merged in constant time. Each in-core cluster has an index to one of its triangles in this list. Since both the input mesh and the resulting clusters are fairly coherent, accesses to the memory mapped array will be too. Once the clusters have been ordered in-core, the triangle links are replaced with the final triangle positions, which is done one cluster at a time. To ensure evenly-sized and well-shaped clusters, we use the following metric: the cost of an edge collapse equals the perimeter (in number of primal edges) of the resulting cluster. To avoid high spans associated with non-manifold vertices, we "glue" the disjoint triangle loops around such vertices together.

mesh name	inter-leaving	com-paction	spatial sorting		spectral sequencing	
			ranking	total	ranking	total
buddha	0:05	0:09	0:02	0:13	0:27	0:33
lucy	4:36	5:11	1:03	11:33	3:41	6:59
st. matthew	1:41:29	2:08:36	21:18	4:09:52	45:42	2:31:47

Table 9.2: Timings (h:m:s) including compressed I/O on a 3.2 GHz PC.

9.5.3 Results

We have measured the performance of our mesh reordering tools on a 3.2 GHz Intel XEON PC running Linux with 2 GB of RAM. Table 9.2 summarizes the performance on a few example meshes. Interleaving takes gzipped PLY as input and writes a binary streaming mesh. We use this as input to compaction, which outputs a compressed streaming mesh for input to spatial sorting, and so on. Interleaving and compaction achieve an overall throughput of 60–100 thousand triangles per second (Ktps). Spatial sorting and spectral sequencing are broken down into the vertex ranking phase and the triangle compaction sorting phase. Spatial sorting is somewhat slower at 25–50 Ktps because it assumes the input layout is highly incoherent. Spectral is faster with at 70–100 Ktps because is takes advantage of coherent streaming input; hence the large speedup in the reordering phase. These timings include reading the original mesh from gzipped PLY input and writing the final output as a compressed streaming mesh.

9.6 Compressing Streaming Meshes

Current mesh compression schemes do not preserve the ordering of the vertices and triangles of a mesh. The impressive reductions in file size are mainly achieved by encoding mesh connectivity with on average only two bits per vertex. By comparison, standard indexed formats need at least $6 \log_2(v)$ bits per vertex for the connectivity of meshes with v vertices because because they store not only the mesh but also the particular ordering of its elements. Compression schemes completely disregard the original ordering and rearrange mesh elements in an order they see best fit for compression, which generally means as encountered during a deterministic traversal of the connectivity graph. Hence, for current compression schemes the layout of the compressed mesh is dictated by the traversal strategy employed by the compression scheme.

Compressing an initially incoherent mesh will generally improve its layout. In the resulting order subsequent triangles often share an edge and vertices appear in the order they are referenced by the triangles. After all, it was the element odering of the compression scheme from Chapter 7 that inspired this work. However, the traversal heuristic used by current compression schemes really aim at lowering the bit-rates with good output layouts being coincidental and not part of the design. Most compression schemes traverse meshes in depth-first order and thereby generate triangle orderings with maximal span. The layout artifacts of such compressors are shown in Figure 9.6. They also produce orderings of unnecessary high width—especially for meshes with many topological handles. While one-pass schemes (Touma and Gotsman, 1998; Gumhold and Strasser, 1998) can easily be modified to operate in a breadth-first manner, multi-pass schemes (Taubin and Rossignac, 1998; Rossignac, 1999; Isenburg and Snoeyink, 2000) cannot. So far nobody paid attention to what a compressor does to the layout of a mesh—maximum compression and algorithmic elegance were the sole design criteria.

9.6.1 Compressing in Stream Order

To preserve the ordering of mesh elements we have to depart from the traditional approaches to mesh compression and use a scheme that can encodes meshes in their particular stream order. Obviously such a scheme will not achieve the same rate of compression as schemes that are allowed to reorder the mesh as they please. However, the benefit of encoding in stream order is that streaming mesh output can be immediately compressed while it is written to disk or piped across a network. In contrast, previous schemes require loading the complete mesh and constructing a representation

that allows traversing its connectivity graph—before the compression process can even begin. In order to do this for large meshes that can not be loaded into memory they either need to cut the mesh into smaller pieces as suggested by (Ho et al., 2001) or need to build complex external memory data structures as we described in Chapter 7. Instead we now have a *streaming mesh writer* and a corresponding *reader* through which on-the-fly compressed streaming meshes can be written and read in increments of single vertices and triangles. An example API is outlined in Figure 9.7.

```
class SMwriter_smc {
  // specifies optional quantization
  bool open(FILE* file, int bits);

  // may optionally be set if known in advance
  void set_bounding_box(float* min, float* max);
  void set_num_verts(int nverts);
  void set_num_faces(int nfaces);

  bool write_vertex(float* v_pos);
  // finalize indices used for the last time
  bool write_triangle(int* t_idx, bool* t_final);

  bool close();
}

typedef Type enum {SM_VERTEX, SM_TRIANGLE};
```

```
class SMreader_smc {
  int bits;
  // only optionally known
  float *bb_min, *bb_max;
  int nverts, nfaces;

  bool open(FILE* file);
  Type read_element();
  bool close();

  // position of read vertex
  float* v_pos;
  // indices of read triangle ...
  int* t_idx;
  // ...  and their finalization
  bool* t_final;
}
```

Figure 9.7: An example API for reading and writing compressed streaming meshes.

For efficiency reasons, our compressor writes only vertex-compact pre-order meshes with immediate vertex finalization. In order to compress meshes that are not pre-order or that do immediate finalize vertices they only need to be piped through the appropriate converter. While the streaming mesh input does not need to be vertex-compact, it will be compressed in a vertex-compact manner. Out-of-order vertices are delayed and will not be compressed until actually referenced by a triangle. Whenever a vertex is written it is simply inserted into a hash using its index as key. Only when a triangle is written actual compression takes place. In this moment both the connectivity between this triangle and all previously written triangles and also the positions of all vertices that are referenced for the first time are output in compressed form.

The compressor maintains a set of *active vertices* and a set of *active half-edges*. The active vertices have been referenced by previously written triangles but have not yet been finalized. The active half-edges are oriented and connect two active vertices. They are part of a previously written triangle and their counterpart of opposite orientation has either not yet appeared or does not exist. With each active vertex the compressor keeps a list to all its incident active half-edges.

When a triangle is written the compressor checks whether any of the triangle's vertices or any of the counterparts of the triangle's half edges are already active. There are eight different configurations that can arise, namely $start_0$, $start_1$, $start_2$, $start_3$, *add*, *join*, *fill*, and *end*, which are illustrated in Figure 9.8. The compressor encodes the configuration of the current triangle with an arithmetic using four different symbols: START, ADD, JOIN, FILL_END. For reasons of efficiency it uses only one symbol for all four $start_i$ configurations as they are typically of infrequent occurance. The i is subsequently compressed with a separate contexts. The *fill* and *end* configurations only need one symbol because they can be distinguished at the decoding end.

Unless the current triangle is in the $start_0$ configuration, the appropriate active vertices are then referenced. This could be done using $\log_2(w)$ bits per vertex, where w is the current number of active vertices (i.e. the width). However, since subsequently written triangles often share vertices we first check whether the active vertex under consideration was either v_0, v_1, or v_2 of the previously written triangle. We use an arithmetic context to encode if this is indeed the case, which often saves us those $\log_2(w)$ bits that are the single most expensive item in our connectivity encoding.

In case of an *add*, *join*, *fill*, or *end* configuration the current triangle is also adjacent to one or more active half-edges. After having referenced the first active vertex (either with $\log_2(w)$ bits or as a vertex from the previous triangle) we can reference other active vertices using the list of half-edges maintained with each active vertex. Since this list often contains only one half-edge with the correct orientation, we usually avoid having to store any further information. Only vertex v_2 of a *join* configuration can obviously not be referenced this way, making them the most expensive configurations to encode.

The positions of newly introduced vertices are predicted with the parallelogram rule (Touma and Gotsman, 1998) in case of an *add* configuration or as a neighboring vertex in case of a $start_i$ configuration and only a corrective vector is stored. For the first vertex of a $start_0$ configuration there is no known neighbor. Here we simply use the most recent vertex that was compressed as the prediction.

Finalization information is encoded by specifying for all three vertices whether the current triangle finalizes them or not. These binary flags can be efficiently compressed with context-sensitive arithmetic coding. The context is chosen based on the number of triangles and active half-edges around this vertex. As most vertices are finalized when they are surrounded by a closed ring of triangles there is a strong correlation between the moment a vertex no longer has active half-edges and its finalization. Border vertices, which will still have one or two such edges but tend to be surrounded by a smaller

number of triangles. The triangle and half-edge counts are shown in Figure 9.8 in the small box associated with vertex.

The vertices are maintained in two data structures: a hash table and a dynamic vector. The hash table is used to look up the vertices by their index. A vertex is added to the hash when it is written, it is looked up in the hash when a triangle is written that references it, and it is removed from the hash when it is finalized. The dynamic vector is used to address previously encoded vertices with an index between 0 and $w-1$. A vertex is added to the dynamic vector when the triangle that references it for the first time is written. Subsequently the encoder looks up the current index that a vertex has in this dynamic vector whenever it needs to encode a reference to this vertex. These indices can then be encoded with $\log_2(w)$ bits. The dynamic vector implements constant time insertion and removal of vertices and constant time lookup for vertex indices simply by moving the last entry to a deleted position. This means that the indices with which vertices are addressed in that data structure will change over time, but they do this in a consistent manner at both encoder and decoder.

9.6.2 Bounding-box less quantization

To support quantization of floating-point geometry for streaming meshes whose bounding box is not known in advance, we use a scheme that quantizes conservatively using a bounding box that is learned as the mesh streams by. We perform predictions in floating-point and encode separate correctors for sign, exponent, and mantissa. For compressing them, we switch between multiple arithmetic contexts as the success of predictions in floating-point varies with the exponent. Although initial experiments indicate that this approach works well in practice, we still need to analyze its performance. In addition our streaming mesh writer supports lossless floating-point compression (Isenburg et al., 2005a), which will obviously be less efficient but allows the use of compression when quantization—for whichever reason—is not an option.

9.6.3 Results

In Table 9.3 reports results for compressing meshes in different stream-orders. We list what percentage of triangles is written in which configuration and what percentage of active vertices is referenced as a vertex from a previous triangle. Whenever this is the case we do not have to store an explicit reference to that active vertex. The detailed break-down of connectivity coding costs show that the references are, as expected, the single most expensive item, whereas the finalization information is basically free.

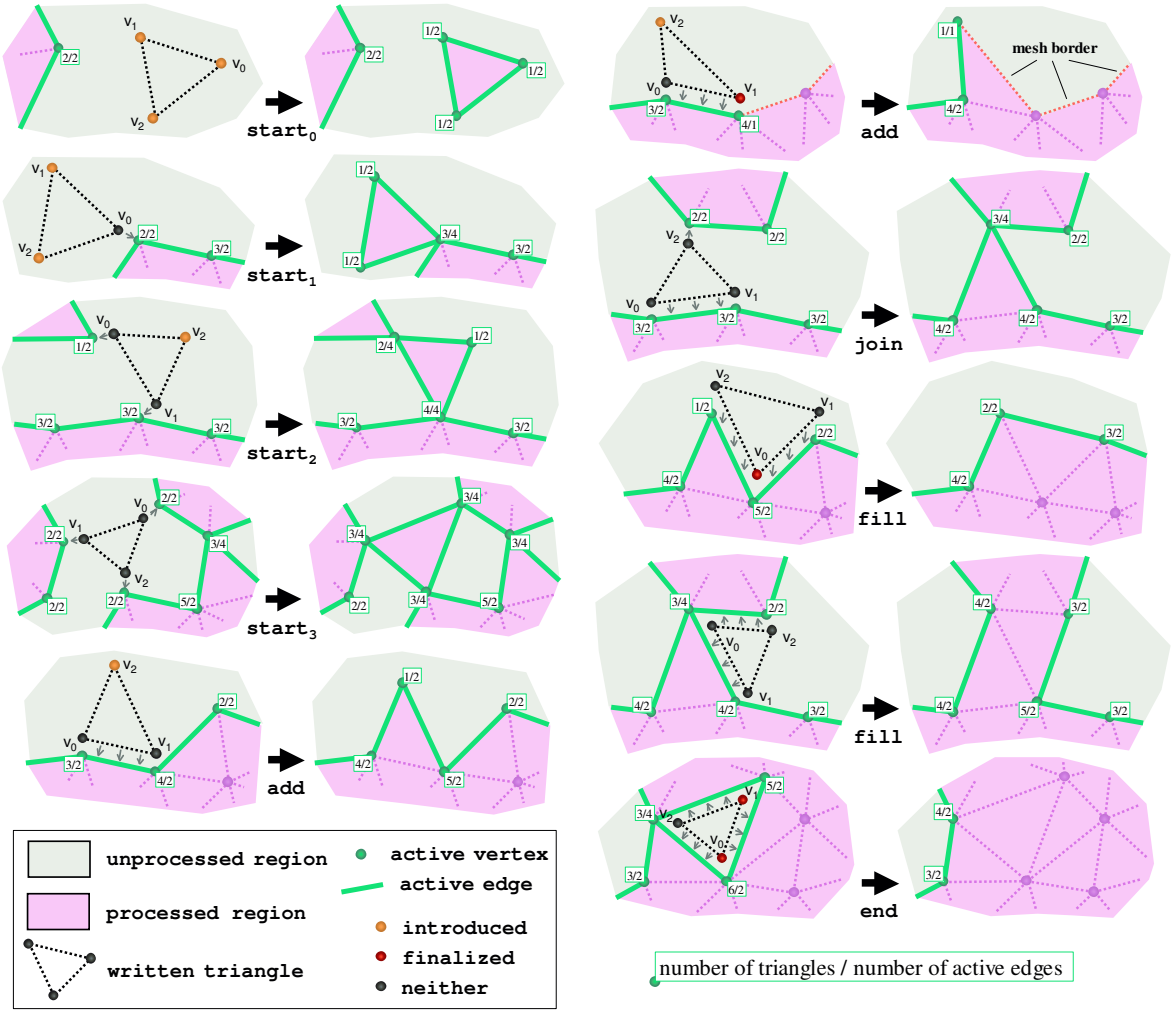


Figure 9.8: The different adjacency configurations that occur between the written triangle and the active vertices and edges maintained by the compressor: a **start** triangle is not adjacent to an active edge, but may be adjacent to one, two, or even three active vertices; an **add** triangle is only adjacent to one active edge with the third vertex being a newly introduced; for the similar **join** configuration this third vertex is already active; a **fill** triangles is adjacent to two edges and an **end** triangle is adjacent to three edges. Vertices are usually finalized by fill and end triangles, but for meshes with borders or non-manifold vertices this also happens in other configurations. Small boxes show counts for number of triangles and number of active half-edges.

Compared to the results of Table 7.4, we achieve almost the same geometry compression rates, while our connectivity compression rates are ususally much higher.

Most interesting is the drastically different performance of connectivity compression on the two (almost identical) topological orderings. While topo_{rnd} performs a slightly

mesh (ordering)	operations [%] s	a	j	f	e	use [%]	details for conn [bpv] fig	pre	exp	adj	fin	totals [bpv] conn	geom	time [sec]	mem [MB]
buddha															
(vcompact)	15	25	9.6	37	14	**49**	3.9	3.9	13.5	.82	.00	**22.08**	21.28	2.8	1.5
(spatial)	1.3	48	4.2	43	4.2	**56**	2.5	2.4	8.9	.15	.00	**13.65**	21.49	2.3	0.9
(topo$_{rnd}$)	6.8	37	2.3	50	3.3	**60**	2.8	2.7	8.8	.36	.00	**14.64**	21.63	2.8	1.0
(topo$_{ord}$)	.0	50	.3	49	.3	**97**	1.9	1.3	0.7	.02	.00	**3.98**	21.72	2.5	0.9
lucy															
(vcompact)	1.6	47	8.1	34	8.7	**77**	2.5	2.2	8.6	.35	.00	**13.60**	14.70	77	37
(spatial)	.5	49	2.1	46	2.1	**53**	1.9	1.9	11.3	.07	.00	**15.18**	14.58	65	1.6
(topo$_{rnd}$)	3.1	44	1.0	50	1.5	**62**	2.2	2.6	9.2	.15	.00	**14.22**	14.42	65	1.8
(topo$_{ord}$)	.0	50	.0	50	.0	**99**	1.7	0.8	0.3	.00	.00	**2.83**	14.59	61	1.8
david$_{1mm}$															
(vcompact)	12	28	5.8	44	9.4	**66**	2.8	2.6	9.9	.60	.02	**15.94**	10.95	126	4.8
(spatial)	.8	49	2.0	47	2.0	**67**	1.9	2.8	8.0	.07	.02	**12.71**	11.63	131	2.4
st. matthew															
(vcompact)	11	31	6.2	44	8.5	**67**	2.7	2.4	10.0	.53	.02	**15.62**	8.22	15 m	5.2
(spatial)	.9	48	2.2	46	2.3	**69**	1.9	2.9	8.6	.08	.02	**13.60**	8.97	15 m	4.0

Table 9.3: For compressing in stream order we report the percentages of **s**tart, **a**dd, **j**oin, **f**ill, and **e**nd configurations that occur and of subsequent triangles that re-**use** vertices. We give itemized coding costs for triangle con**fig**uration, **pre**vious and **exp**licit references, edge **adj**acency, and vertex **fin**alization. Total bit-rates for **conn**ectivity and **geom**etry (quantized at 16 bits) and both the **time** and the maximum **mem**ory footprint for reading, compressing, and writing the streaming meshes are listed.

randomized version of a breadth-first traversal that "jumps" around on the boundary, topo$_{ord}$ tries to maximize the edge-connectedness of subsequent triangles during the breadth-first traversal. The resulting high re-use of vertices between subsequent triangles almost always saves us from using explicit references. This already hints at a possible variation of this coder that can lead to *significantly* better compression rates: Instead of strictly following the original triangle order, we could allow the compressor to keep a small triangle buffer within which it could locally reorder triangles to bring them into a more edge-connected order without affecting the global stream quality.

Nevertheless, even following the dictated triangle order we get reasonable compression rates considering the high speeds and the low memory use. Compared to the out-of-core compressor, which required gigabytes of temporary disk space and about 7 hours time to create an 11 GB data structure on disk and then needed another 4 hours and 384 MB of main memory to produce the compressed mesh, we can now compress the compacted "St. Matthew" statue directly in only 15 minutes while using less than 6MB of main memory. The drawback is that we currently only achieve half

the overall compression rate and decompression is three times slower. However, in the meantime we have shown that we are able to improve compression by integrating the local reordering strategy we just mentioned. Using a *delay buffer* of a few hundred to a few thousand triangles within which the compressor greedily brings triangles into an edge-connected order leads to connectiviy compression of around 4 to 5 bits per vertex, nearly independent from the original input order (Isenburg et al., 2005b).

9.7 Summary

In this chapter we have identified a major headache in large mesh processing—poor mesh layouts—and suggested how to avoid this pain—keeping the mesh in a streamable layout. We have both established a theoretical framework that characterizes the quality of a layout and presented out-of-core tools that can improve poor layouts.

Compatible mesh layouts can be streamed by interleaving vertices and triangles in their original order and adding finalization information. Incompatible yet low-width layouts can be made streamable by *compacting*, meaning reordering either only the vertices or only the triangles. The width and span of a vertex-compacted mesh are proportional to the vertex width and triangle span of its layout. Vertex compaction can create low-span streams when only the vertex span is high (e.g. horse and dragon). The width and span of a triangle-compacted mesh are proportional to the triangle width and vertex span of its layout. Triangle compaction can create low-span streams for layouts where only the triangle span is high (e.g. the buddha). Layouts that are both high in width and span always require reordering both vertices and triangles.

Breadth-first traversals naturally are low in span—and hence width—since the "oldest" vertex introduced tends to be finalized first. Depth-first traversals, on the other hand, leave the oldest vertices hanging and therefore guarantee high-span layouts. The z-order layouts are by definition high in span, although of bounded length and frequency, which results in a lower width. Linear spatial sorts produce layouts that are sufficiently low in width and span for practial purposes. Spectral orderings can achieve the lowest width but often at the expense of a higher span. For the large statues, even the width suffers due to coarse granularity clustering (we used at most one million clusters), which leaves the front increasingly ragged as it winds around the clusters.

Documenting coherence in the file format makes processing large meshes considerably more efficient. It solves the main problem of dereferencing that complicates almost every mesh application, from rendering an initial image to get an idea of what data one is dealing with to the construction of complex hierarchical mesh structures.

While we presented simple backwards-compatible extensions to existing file formats, our streaming meshes are not limited to a particular mesh format. One may even stick to standard formats such as PLY or OBJ without explicit finalization information. As long as we are guaranteed that the triangle ordering has low span and comes with a compatible vertex ordering we can always "guess" finalization using a buffer that delays vertices by a duration proportional to the maximal span.

We argue that stack-based approaches to large mesh compression are bad because they systematically create meshes of maximal span and because they cannot efficiently handle high-genus models where the stack can grow very deep—thereby also creating meshes with high width. Instead a mesh compressor should be designed to perform some sort of a breadth-first traversal that attempts to give vertices a similar "lifetime" on the compression boundary. This means that some of the most celebrated compression algorithms, such as the Edgebreaker scheme by (Rossignac, 1999) and the Topological Surgery method of (Taubin and Rossignac, 1998), are poor choices for compressing larger meshes because they can not be modified to breadth-first operation.

Finally we described a new scheme for compressing streaming meshes in their particular stream order. While it does not achieve state-of-the-art connectivity compression, the sacrifice in bit-rate is well spent because we can now write compressed meshes on-the-fly. This allows transparent integration of compressed mesh input and output for mesh consuming and producing applications, which makes compression a more usable feature in a typical mesh processing pipeline. Contrast this with previous schemes that first spend several hours of constructing external memory data structures that use gigabytes of auxiliary disk space—even when the mesh already had a nice layout.

Naturally, the question arises whether there are triangle orderings that are not only good for streaming, but also good for rendering. We believe that such sequences exist and that they will look similar to those generated by the scheme of (Hoppe, 1999). They would essentially be short strips with generous overlaps that are grown in a breadth-first manner such that no vertex is left hanging behind. For gigabyte sized data sets, rendering sequences are probably not useful for the sake of image generation. However, graphics hardware is more and more used for intensive numerical computations. Due to current restrictions these computations are mainly implemented as pixel programs and can not involve irregular connectivities. But as soon as there are mechanisms for vertex programs to access local mesh connectivity, such sequences may be the most band-width efficient way to stream large amounts of mesh data in a cache coherent way from the harddisk through RAM and CPU onwards to the buffers on the graphics unit.

In the future we would like to investigate concurrent streaming at multiple resolutions, multiplexing of streaming meshes for parallel processing, and extensions to volume meshes. We also envision that some sort of 'space finalization' would be useful for processing tasks that require a spatially—as opposed to a strictly topologically—coherent traversal, for example algorithms that check the mesh for self-intersections. For better compression, future versions of the streaming mesh writer will have the option to allow the compressor to do local reorderings (Isenburg et al., 2005b).

9.8 Hindsights

After this chapter was written, we realized that there was a large body of literature on heuristics on graph reorderings that seemed largely ignored in the graph-theoretical literature, namely algorithms for sparse matrix reordering to allow more efficient solving of linear systems. The pioneering works in this area were mainly published in engineering journals, usually accompanied by an implementation in FORTRAN. It is quite surprising how few cross references there are between the graph theory community and the engineering community. In the engineering papers is virtually no mention of the close relation between objectives in matrix reordering and the classic problem of computing a minimal linear arrangement for a graph. Vice-versa there is no mention in graph theory papers of the Sloan-like approaches, which are discussed below, that seem to be good heuristics for generating arrangements with small cut-width. We briefly survey the matrix reordering methods that are used in engineering so they can be considered in follow-up works on mesh reordering.

Traditionally, there have been two different objectives for reordering matrices. Either to create matrices with low bandwidth which are then used by fixed-band solvers, or to create matrices with low wavefront (also called frontwidth) and profile (basically the same as envelope), which are then used by profile methods (or variable-band or skyline) solvers or frontal method solvers. The frontal method is due to (Irons, 1970) and the profile method is due to (George, 1971).

The original reordering algorithms are based on level set structures from (Cuthill and McKee, 1969), (Gibbs et al., 1976), or (Lewis, 1982) and all concerned with bandwidth reduction, since at that time mainly fixed-band solvers were in use. In our terms, low bandwidth translates to low span. The first improvement over the level set methods for better wavefront reduction was the algorithm by (Sloan, 1986) that adds a global component to the priority function directing the traversal. The algorithm was then improved and integrated into the HSL package by (Duff et al., 1989). However, in

general this algorithm result in higher bandwidth, since the nodes are now longer traversed with breadth-first style like in (Cuthill and McKee, 1969) or (Gibbs et al., 1976). Since wavefront (or frontwidth) pretty much translates into what we call width, these results are in accordance with ours—attempts to minimize the wavefront (i.e. the width) inflate the bandwidth (i.e. the span).

Later, (Paulino et al., 1994) and (Barnard et al., 1995) realized that the order induced by the Fiedler vector of the Laplacian connectivity matrix results is a good heuristic for reducing the profile and the wavefront. In contrast to previous approaches that use only the two end nodes of a pseudodiameter of the graph as the global information for directing the traversal, these spectral approaches exclusively use the global information of the Fiedler vector to position the nodes in the sequence. This is more or less what we did in our spectral sequencing approach for reordering the mesh vertices.

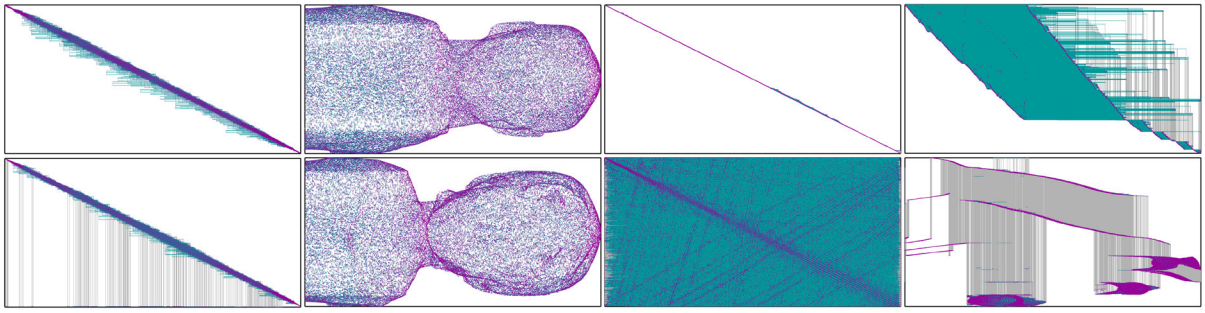
The state of the art combines the globally good information of the Fiedler vector used by (Barnard et al., 1995) with the locally optimal decisions of the improved Sloan method by (Duff et al., 1989). This is due to (Kumfert and Pothen, 1997) who use a Sloan type algorithm that incorporates the Fielder vector as the global component of its priority function. This spectral/Sloan hybrid, which is also called "Fast Sloan", almost always outperforms the individual algorithms. Again, this hybrid algorithm was later improved and included into the HSL package by (Reid and Scott, 1999; Scott, 1999). It seems that a similar hybrid strategy could also be used to further improve on the orderings created by our spectral sequencing approach.

In order to speed up the reordering of large matrices, there have been two multi-level approaches (Boman and Hendrickson, 1996; Hu and Scott, 2001) that get close to the performance of the hybrid Sloan algorithm by (Kumfert and Pothen, 1997). In particular, they do not require the expensive computation of a Fiedler vector which is a major obstacle for using the hybrid Sloan on large matrices. In contrast, our spectral sequencing uses an out-of-core version of (Koren et al., 2002) to efficiently compute an approximation of the Fiedler vector with a multi-level method.

Finally, there is the exchange method by (Hager, 2002) that can further reduce the profile of matrices already reordered with, for example "Fast Sloan". While the original version of this algorithm seemed prohibitively slow, (Reid and Scott, 2002) have shown how to implement it such that becomes practical. Other approaches we should mention include simulated annealing which was investigated for bandwidth, profile and wavefront reduction by (Armstrong, 1985) and for profile and fill reduction by (Lewis, 1993), but these methods are too slow for practical purposes.

Chapter 10

Conclusion



Layout diagrams for various meshes. Guess which is which. From top left to bottom right these are ahddub yppah, elam, edalb, woc, nogard, elamef, ollidamra, and esroh.

In this dissertation I have shown two things. First, that polygon meshes can be encoded more efficiently than triangle meshes by avoiding the initial triangulation step and by operating directly on the polygonal connectivity. Second, that a coherent and—more importantly—documented ordering of mesh elements gives IO-efficient access to large polygon meshes that enables the design of new out-of-core algorithms that organize their operations to follow a stream-based paradigm.

10.1 Contributions

The main results of my dissertation work are as follows.

- I have shown that both the classic works on graph coding as well as recent connectivity compression schemes represent mesh connectivity as an encoding of two dual spanning trees. I have given intuitive illustrations that explain how being more and more particular about the used pair of spanning trees allows recent schemes to improve their worst-case bounds. (Chapter 2)

- I have organized the existing body of compression schemes into *one-pass* and *multi-pass* schemes. This is of practical importance for designing compression engines as multi-pass schemes do not scale with increasing model size. (Chapter 2)
- I have designed an edge-based method for encoding mesh connectivity that is similar to Edgebreaker because also avoids explicit split offsets but different because it stores one label per edge instead of per triangle. This scheme has a straight-forward extension to polygonal mesh connectivity and also allows including pre-computed triangle strip information into the encoding. (Chapter 3)
- I have generalized the degree-based coder of (Touma and Gotsman, 1998) to polygonal meshes and thereby achieved the currently lowest reported bit-rates for polygonal connectivity. I have also given a simple proof that adaptive traversals cannot guarantee to avoid split operations and have disproved the long suspected redundancy of the split offsets used in degree coding. (Chapter 4)
- I have extended the most successful compression techniques for polygonal surface meshes to hexahedral volume meshes. The presented coder is the first encode hexahedral meshes directly and achieves bit-rates for connectivity and geometry that are clearly superior to those reported for tetrahedral meshes. This is not surprising. Hexahedral meshes are naturally more regular than tetrahedral meshes because only hexahedral elements allow a regular tiling of space. (Chapter 6)
- I have shown how to compress gigantic polygon models that are much larger than the available main memory on standard PCs. I have described an external memory data structure that provides efficient out-of-core mesh access, how to construct this data structure using only limited memory, and a one-pass compressor that accesses this data structure as coherently and as infrequently as possible. This allowed me to compress the largest models that were available to me in one piece while achieving state-of-the-art compression rates. (Chapter 7)
- I have demonstrated that the particular mesh access provided by our compressed format allows the implementation of highly IO and memory efficient simplification algorithms. I have defined two processing abstractions, namely boundary-based and buffer-based processing, that are supported by the order in which mesh elements are decompressed and by the availability of vertex finalization information. I have adapted out-of-core simplification methods to each abstraction and shown that this leads to improvements in in terms of more efficient execution, smaller memory footprints, and even improved quality. (Chapter 8)

- I have extracted what made our compressed format so useful to design a new streaming format to replace traditional mesh formats that are difficult to work with when meshes are large. As an added benefit, a streaming mesh format allows some mesh processing tasks (but not all) to perform in less time and with less memory and disk overhead by adapting a stream-based approach. This is an attractive alternative in situations where operating on polygon soup is insufficient and where building external memory data structures is an overkill. (Chapter 9)
- I have described a compression scheme that can compress a streaming mesh in its particular stream order. While this is obviously less efficient as a compression schemes that pick their own traversal order, it makes compression more useful in an actual mesh processing pipeline, as meshes can be compressed on-the-fly as they are written out to disk or streamed across a network. (Chapter 9)

10.2 Limitations

The one big concern that I have heard from both paper reviewers and committee members is in respect to my claim of a streaming mesh format being a *better* format for large meshes. The usual argument is that a streaming format is no universal solution to all problems in out-of-core mesh processing. That is a true statement but a somewhat unfair criticism. My supervisor tried to console me, saying "Give a bicycle to someone who has been walking all his life and he will come back complaining that it is not a jetplane". I never claimed that a streaming mesh format is a suitable on-disk representation for every type of out-of-core processing. The main intention of our streaming mesh format is to replace traditional mesh formats (e.g. indexed meshes like PLY and OBJ) that are cumbersome to work with when the meshes are large. In this respect, streaming meshes are merely a more appropriate representation for storing large meshes. But it does solve the problem of de-referencing, which is the first problem that every out-of-core mesh processing must face with current formats. It also gives a vocabulary and framework to analyze streaming processing as a potential alternative.

Streaming meshes neither solves nor attempt to solve the problem of accessing large data sets for the purpose of efficient out-of-core visualization. For this various techniques have been proposed (Lindstrom, 2003; Cignoni et al., 2004; Yoon et al., 2004) that arrange the data based on some sort of spatial clustering usually at multiple resolutions. These approaches try to ensure that the on disk storage of the data set reflects its spatial distribution or more importantly the anticipated access pattern. But

visualization application are an inherently interactive application that require an *online* processing paradigm that is quite different from our *stream* processing paradigm.

10.3 Future Work

So far we only have shown how to change compression and simplification algorithms to work on streaming data. But also algorithms for surface reconstruction, remeshing, parameterization, etc. seem suited to operate on streaming meshes. We hope that experts in their respective research area will consider whether their algorithms can be adapted to streaming processing. By defining both attributes and limitations within which streaming approaches can operate with high efficiency, we create a potential for inspirations to design new algorithms that can work within these parameters.

The presented streaming mesh format is just as flexible as traditional indexed formats that rewards coherence in the ordering of the mesh elements without imposing rigid constraints on it. But some algorithms may not only require topologic coherence but also spatial coherence if they are to be adapted to streaming processing. To check a mesh for self-intersections, for example, a stream-based algorithm would need a spatially streaming mesh. Our reordering results in Table 9.1 illustrate that spatially sorting a mesh along one axis usually gives sufficient topological coherence. But such an algorithm would also need knowledge about when a mesh will no longer intersect a piece of space. It remains to be investigated if some kind of "space finalization" would be useful and how it could be realized without overly constraining the format.

Finally, extending the streaming paradigm to other types of geometric data such as points clouds, scalar fields, and irregular as well as regular volume meshes seem obvious, but useful extensions. Streaming isosurface extraction, for example, requires the volume mesh that interpolates the function of interest to be arranged in a streamable layout. For the case of regular volume grids we have already investigated one particular way to arrange the grid cells that allows IO-efficient streaming extraction (Mascarenhas et al., 2004). Another example is streaming surface reconstruction from point clouds where we try to compute an approximation of a surface for which we have points that were scanned by a laser range finder. If we had a "streaming Delaunay triangulator" this should be a simple matter to implement. However, whether adapting a Delaunay tessellator to streaming out-of-core operation is possible and in which order the points should be streamed are interesting challenges.

Bibliography

Alliez, P. and Desbrun, M. (2001a). Progressive encoding for lossless transmission of 3D meshes. In *SIGGRAPH'01 Conference Proceedings*, pages 198–205.

Alliez, P. and Desbrun, M. (2001b). Valence-driven connectivity encoding for 3D meshes. In *Eurographics'01 Conference Proceedings*, pages 480–489.

ANN (v 0.2). A library for approximate nearest neighbor searching by D. Mount and S. Arya. *University of Maryland*.

Armstrong, B. (1985). Near-minimal matrix profiles and wavefronts for testing nodal resequencing algorithms. *International Journal for Numerical Methods in Engineering*, 21:1785–1790.

Bajaj, C., Pascucci, V., and Zhuang, G. (1999). Single resolution compression of arbitrary triangular meshes with properties. In *Data Compression Conference'99 Conference Proceedings*, pages 247–256.

Balmelli, L., Taubin, G., and Bernardini, F. (2002). Space-optimized texture maps. In *Eurographics'02 Conference Proceedings*, pages 411–420.

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

Barnard, S., Pothen, A., and Simon, H. (1995). A spectral algorithm for envelope reduction of sparse matrices. *Numerical Linear Algebra with Applications*, 2(4):317–334.

Bernardini, F., Martin, I., Mittleman, J., Rushmeier, H., and Taubin, G. (2002). Building a digital model of Michelangelo's Florentine Pieta. *IEEE Computer Graphics and Applications*, 22(1):59–67.

Bogomjakov, A. and Gotsman, C. (2001). Universal rendering sequences for transparent vertex caching of progressive meshes. In *Graphics Interface'01 Conference Proceedings*, pages 81–90.

Boman, E. and Hendrickson, B. (1996). A multilevel algorithm for reducing the envelope of sparse matrices. Technical Report SCCM-96-14, Stanford University.

Brehm, E. (2000). 3-orientations and Schnyder 3-tree-decompositions. Technical Report Diploma Thesis, Freie Universitt Berlin.

Brodsky, D. and Watson, B. (2000). Model simplification through refinement. In *Graphics Interface'00 Conference Proceedings*, pages 221–228.

Bunyk, P., Kaufmann, A., and Silva, C. (2000). Simple, fast, and robust ray casting of irregular grids. In *Proceedings of Dagstuhl'97*, pages 30–36.

Chiang, Y. and Silva, C. (1997). I/O optimal isosurface extraction. In *Visualization'97 Proceedings*, pages 293–300.

Chiang, Y.-T., Lin, C.-C., and Lu, H.-I. (2001). Orderly spanning trees with applications to graph encoding and graph drawing. In *Proceedings of Symposium on Discrete Algorithms (SODA)*, pages 506–515.

Choudhury, P. and Watson, B. (2002). Completely adaptive simplification of massive meshes. Technical Report CS–02–09, Northwestern University.

Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. (2004). Adaptive tetrapuzzles: Efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In *SIGGRAPH'04 Conference Proceedings*. to appear.

Cignoni, P., Montani, C., Rocchini, C., and Scopigno, R. (2003). External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9:525–537.

Cohen-Or, D., Cohen, R., and Irony, R. (2002). Multi-way geometry encoding. Technical report, Department of Computer Science, Tel Aviv University.

Cohen-Or, D., Levin, D., and Remez, O. (1999). Progressive compression of arbitrary triangular meshes. In *Visualization'99 Conference Proceedings*, pages 67–72.

Cuthill, E. and McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference of the ACM*, pages 157–172.

Deering, M. (1995). Geometry compression. In *SIGGRAPH'95 Conference Proceedings*, pages 13–20.

Denny, M. and Sohler, C. (1997). Encoding a triangulation as a permutation of its point set. In *Proceedings of 9th Canadian Conference on Computational Geometry*, pages 39–43.

Desbrun, M., Meyer, M., and Alliez, P. (2002). Intrinsic parameterizations of surface meshes. In *Eurographics'02 Conference Proceedings*, pages 209–218.

Devillers, O. and Gandoin, P.-M. (2002). Progressive and lossless compression of arbitrary simplicial complexes. In *SIGGRAPH'02 Conference Proceedings*, pages 372–379.

Díaz, J., Petit, J., and Serna, M. (2002). A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356.

Duff, I., Reid, J., and Scott, J. (1989). The use of profile reduction algorithms with a frontal code. *International Journal for Numerical Methods in Engineering*, 28:2555–2568.

Eppstein, D. (1999). Linear complexity hexahedral mesh generation. *Computational Geometry Theory and Applications*, 12:3–16.

Evans, F., Skiena, S. S., and Varshney, A. (1996a). Completing sequential triangulations is hard. Technical report, Department of Computer Science, State University of New York at Stony Brook.

Evans, F., Skiena, S. S., and Varshney, A. (1996b). Optimizing triangle strips for fast rendering. In *Visualization'96 Conference Proceedings*, pages 319–326.

Farias, R., Mitchell, J., and Silva, C. (2000). Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of Volume Visualization Symposium'00*, pages 91–99.

Farias, R. and Silva, C. (2001). Out-of-core rendering of large unstructured grids. *IEEE Computer Graphics and Applications*, 21(4):42–50.

Fusy, E., Poulalhon, D., and Schaeffer, G. (2005). Dissections and trees, with applications to optimal mesh encoding and to random sampling. In *Proceedings of Symposium on Discrete Algorithms (SODA)*. to appear.

Garland, M. and Heckbert, P. (1997). Surface simplification using quadric error metrics. In *SIGGRAPH'97 Conference Proceedings*, pages 209–216.

Garland, M. and Shaffer, E. (2002). A multiphase approach to efficient surface simplification. In *Visualization'02 Conference Proceedings*, pages 117–124.

Garrity, M. (1990). Raytracing irregular volume data. *Computer Graphics*, 24(5):35–40.

George, A. (1971). Computer implementation of the finite element method. Technical Report Dissertation, Stanford University.

Gibbs, N., Poole, W., and Stockmeyer, P. (1976). An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal of Numerical Analysis*, 13:236–250.

Gotsman, C. (2003). On the optimality of valence-based connectivity coding. *Computer Graphics Forum*, 22(1):99–102.

Gu, X., Gortler, S., and Hoppe, H. (2002). Geometry images. In *SIGGRAPH'02 Conference Proceedings*, pages 355–361.

Guéziec, A., Bossen, F., Taubin, G., and Silva, C. (1999). Efficient compression of non-manifold polygonal meshes. In *Visualization'99 Conference Proceedings*, pages 73–80.

Guéziec, A., Taubin, G., Lazarus, F., and Horn, W. (1998). Converting sets of polygons to manifold surfaces by cutting and stitching. In *Visualization'98 Conference Proceedings*, pages 383–390.

Guibas, L. and Stolfi, J. (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi Diagrams. *ACM Transactions on Graphics*, 4(2):74–123.

Gumhold, S. (2000). New bounds on the encoding of planar triangulations. Technical Report WSI–2000–1, Wilhelm-Schikard-Institut für Informatik, Tübingen.

Gumhold, S. (2005). Optimizing Markov models with applications to triangular connectivity coding. In *Proceedings of Symposium on Discrete Algorithms (SODA)*. to appear.

Gumhold, S., Guthe, S., and Strasser, W. (1999). Tetrahedral mesh compression with the cut-border machine. In *Visualization'99 Conference Proceedings*, pages 51–58.

Gumhold, S. and Strasser, W. (1998). Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Conference Proceedings*, pages 133–140.

Hager, W. (2002). Minimizing the profile of a symmetric matrix. *SIAM Journal on Scientific Computing*, 23(5):1799–1816.

He, X., Kao, M.-Y., and Lu, H. (1999). Linear-time succint encodings of planar graphs via canonical orderings. *Discrete Applied Mathematics*, 12(3):317–325.

Ho, J., Lee, K., and Kriegman, D. (2001). Compressing large polygonal models. In *Visualization'01 Conference Proceedings*, pages 357–362.

Hoppe, H. (1996). Progressive meshes. In *SIGGRAPH'96 Conference Proceedings*, pages 99–108.

Hoppe, H. (1998). Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Visualization'98 Conference Proceedings*, pages 35–42.

Hoppe, H. (1999). Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH'99 Conference Proceedings*, pages 269–276.

Hu, Y. and Scott, J. (2001). A multilevel algorithm for wavefront reduction. *SIAM Journal on Scientific Computing*, 23(4):1352–1375.

Irons, B. M. (1970). A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2:5–32.

Isenburg, M. (2000). Triangle Strip Compression. In *Graphics Interface'00 Conference Proceedings*, pages 197–204.

Isenburg, M. (2001). Triangle Strip Compression. *Computer Graphics Forum*, 20(2):91–101.

Isenburg, M. (2002). Compressing polygon mesh connectivity with degree duality prediction. In *Graphics Interface'02 Conference Proceedings*, pages 161–170.

Isenburg, M. and Alliez, P. (2002a). Compressing hexahedral volume meshes. In *Pacific Graphics'02 Conference Proceedings*, pages 284–293.

Isenburg, M. and Alliez, P. (2002b). Compressing polygon mesh geometry with parallelogram prediction. In *Visualization'02 Conference Proceedings*, pages 141–146.

Isenburg, M. and Alliez, P. (2003). Compressing hexahedral volume meshes. *Graphical Models*, 65(4):239–257.

Isenburg, M. and Gumhold, S. (2003). Out-of-core compression for gigantic polygon meshes. In *SIGGRAPH'03 Conference Proceedings*, pages 935–942.

Isenburg, M., Lindstrom, P., Gumhold, S., and Snoeyink, J. (2003). Large mesh simplification using processing sequences. In *Visualization'03 Conference Proceedings*, pages 465–472.

Isenburg, M., Lindstrom, P., and Snoeyink, J. (2005a). Lossless compression of predicted floating-point geometry. *Computer-Aided Design*, 37(8):869–877.

Isenburg, M., Lindstrom, P., and Snoeyink, J. (2005b). Streaming compression of triangle meshes. submitted for publication.

Isenburg, M. and Snoeyink, J. (2000). Face Fixer: Compressing polygon meshes with properties. In *SIGGRAPH'00 Conference Proceedings*, pages 263–270.

Isenburg, M. and Snoeyink, J. (2001a). Compressing the property mapping of polygon meshes. In *Pacific Graphics'01 Conference Proceedings*, pages 4–11.

Isenburg, M. and Snoeyink, J. (2001b). Spirale reversi: Reverse decoding of the Edgebreaker encoding. *Computational Geometry: Theory and Applications*, 20(1-2):39–52.

Isenburg, M. and Snoeyink, J. (2002). Compressing the property mapping of polygon meshes. *Graphical Models*, 64(2):114–127.

Isenburg, M. and Snoeyink, J. (2003). Compressing texture coordinates with selective linear predictions. In *Proceedings of Computer Graphics International'03*, pages 126–131.

Isenburg, M. and Snoeyink, J. (2005a). Early-split coding of triangle mesh connectivity. pages 1–8. submitted for publication.

Isenburg, M. and Snoeyink, J. (2005b). On the non-redundancy of split offsets in degree coding. pages 1–8. submitted for publication.

Itai, A. and Rodeh, M. (1982). Representation of graphs. *Acta Informatica*, 17:215–219.

Karni, Z. and Gotsman, C. (2000). Spectral compression of mesh geometry. In *SIGGRAPH'00 Conference Proceedings*, pages 279–286.

Keeler, K. and Westbrook, J. (1995). Short encodings of planar graphs and maps. In *Discrete Applied Mathematics*, pages 239–252.

Khodakovsky, A., Alliez, P., Desbrun, M., and Schroeder, P. (2002). Near-optimal connectivity encoding of 2-manifold polygon meshes. *Graphical Models*, 64(3-4):147–168.

Khodakovsky, A. and Guskov, I. (2004). Compression of normal meshes. *Geometric Modeling for Scientific Visualization*, pages 189–206.

Khodakovsky, A., Schroeder, P., and Sweldens, W. (2000). Progressive geometry compression. In *SIGGRAPH'00 Conference Proceedings*, pages 271–278.

King, D. and Rossignac, J. (1999). Guaranteed 3.67v bit encoding of planar triangle graphs. In *Proceedings of 11th Canadian Conference on Computational Geometry*, pages 146–149.

King, D., Rossignac, J., and Szymczak, A. (1999). Connectivity compression for irregular quadrilateral meshes. Technical Report TR–99–36, GVU Center, Georgia Tech.

Kirkpatrick, D. G. (1983). Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12(1):28–35.

Koren, Y., Carmel, L., and Harel, D. (2002). ACE: A fast multiscale eigenvector computation for drawing huge graphs. In *IEEE Information Visualization*, pages 137–144.

Kronrod, B. and Gotsman, C. (2000). Efficient coding of non-triangular meshes. In *Proceedings of Pacific Graphics*, pages 235–242.

Kronrod, B. and Gotsman, C. (2001). Efficient coding of non-triangular meshes. *Graphical Models*, 63(4):263–275.

Kronrod, B. and Gotsman, C. (2002). Optimized compression of triangle mesh geometry using prediction trees. In *International Symposium on 3D Data Processing Visualization and Transmission*, pages 602–608.

Kumfert, G. and Pothen, A. (1997). Two improved algorithms for envelope and wavefront reduction. *BIT*, 37(3):1–32.

Lee, E. and Ko, H. (2000). Vertex data compression for triangular meshes. In *Proceedings of Pacific Graphics*, pages 225–234.

Lee, H., Alliez, P., and Desbrun, M. (2002). Angle-analyzer: A triangle-quad mesh codec. In *Eurographics'02 Conference Proceedings*, pages 198–205.

Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., and Fulk, D. (2000). The digital michelangelo project. In *SIGGRAPH'00 Conference Proceedings*, pages 131–144.

Levy, B., Caumon, G., Conreaux, S., and Cavin, X. (2001). Circular incident edge list: A data structure for rendering complex unstrucutred grids. In *Visualization'01 Conference Proceedings*, pages 191–198.

Levy, B., Petitjean, S., Ray, N., and Maillot, J. (2002). Least squares conformal maps for automatic texture atlas generation. In *SIGGRAPH'02 Conference Proceedings*, pages 362–371.

Lewis, B. (1993). Simulated annealing for profile and fill reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, 37:905–926.

Lewis, J. (1982). Implementation of the Gibbs-Poole-Stockmeyer and Gibbs-King algorithms. *ACM Transactions on Mathematical Software*, 9(2):180–189.

Li, J. and Kuo, C. C. (1998). A dual graph approach to 3D triangular mesh compression. In *Proceedings of ICIP'98*, pages 891–894.

Lindstrom, P. (2000). Out-of-core simplification of large polygonal models. In *SIGGRAPH'00 Conference Proceedings*, pages 259–262.

Lindstrom, P. (2003). Out-of-core construction and visualization of multiresolution surfaces. In *Symposium on Interactive 3D Graphics*, pages 93–102.

Lindstrom, P. and Silva, C. (2001). A memory insensitive technique for large model simplification. In *Visualization'01 Conference Proceedings*, pages 121–126.

Maillot, J., Yahia, H., and Verroust, A. (1993). Interactive texture mapping. In *SIGGRAPH'93 Conference Proceedings*, pages 27–34.

Mantyla, M. (1988). *An Introduction to Solid Modeling*. Computer Science Press.

Mascarenhas, A., Isenburg, M., Pascucci, V., and Snoeyink, J. (2004). Encoding volumetric grids for streaming isosurface extraction. In *Proceedings of 2nd International Symposium on 3D Data Processing, Visualization, and Transmission*, pages 665–672.

McMains, S., Hellerstein, J., and Sequin, C. (2001). Out-of-core build of a topological data structure from polygon soup. In *Proceedings of the 6th ACM Symposium on Solid Modeling and Applications*, pages 171–182.

MeTiS (v 4.0). A software package for partitioning unstructured graphs by G. Karypis and V. Kumar. *University of Minnesota*.

Mitra, T. and Chiueh, T. (1998). A breadth-first approach to efficient mesh traversal. In *Proceedings of Eurographics Workshop on Graphics Hardware*, pages 31–38.

Mueller-Hannemann, M. (2001). Shelling hexahedral complexes for mesh generation. *Journal of Graph Algorithms and Applications*, 5(5):59–91.

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

Pajarola, R., Rossignac, J., and Szymczak, A. (1999). Implant sprays: Compression of progressive tetrahedral mesh connectivity. In *Visualization'99 Conference Proceedings*, pages 299–306.

Paulino, G., Menezes, I., Gattas, M., and Mukerjee, S. (1994). Node and element resequencing using the laplacian of a finite element graph. *International Journal for Numerical Methods in Engineering*, 37:1511–1555.

Poulalhon, D. and Schaeffer, G. (2003). Optimal coding and sampling of triangulations. In *30th International Colloquium on Automata, Languages and Programming (ICAZLP)*, pages 1080–1094.

Prince, C. (2000). Progressive meshes for large models of arbitrary topology. Technical Report Master Thesis, University of Washington.

Reid, J. and Scott, J. (1999). Ordering symmetric sparse matrices for small profile and wavefront. *International Journal for Numerical Methods in Engineering*, 45:1737–1755.

Reid, J. and Scott, J. (2002). Implementing Hagers exchange methods for matrix profile reduction. *ACM Transactions on Mathematical Software*, 28(4):377–391.

Rossignac, J. (1998). Just-in-time upgrades for triangle meshes. In *3D Geometry Compression, Course Notes 21, SIGGRAPH'98*, pages 18–24.

Rossignac, J. (1999). Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61.

Rossignac, J. and Borrel, P. (1993). Multi-resolution 3d approximation for rendering complex scenes. In *Modeling in Computer Graphics*, pages 455–465.

Rossignac, J. and Szymczak, A. (1999). Wrap&zip: Linear decoding of planar triangle graphs. *The Journal of Computational Geometry, Theory and Applications*.

Sander, P., Snyder, J., Gortler, S., and Hoppe, H. (2001). Texture mapping progressive meshes. In *SIGGRAPH'01 Conference Proceedings*, pages 409–416.

Schneider, R., Schindler, R., and Weiler, F. (1996). Octree-based generation of hexahedral element meshes. In *Proceedings of the 5th International Meshing Roundtable*, pages 205–215.

Schnyder, W. (1990). Embedding planar graphs on the grid. In *Proceedings of Symposium on Discrete Algorithms (SODA)*, pages 138–148.

Scott, J. (1999). On ordering elements for a frontal solver. *Communications in Numerical Methods in Engineering*, 15:309–323.

Shaffer, E. and Garland, M. (2001). Efficient adaptive simplification of massive meshes. In *Visualization'01 Conference Proceedings*, pages 127–134.

Sheffer, A., Etzion, M., Rappoport, A., and Bercovier, M. (1998). Hexahedral mesh generation using the embedded voronoi graph. In *Proceedings of the 7th International Meshing Roundtable*, pages 347–364.

Shikhare, D., Bhakar, S., and Mudur, S. (2001). Compression of 3D engineering models using automatic discovery of repeating geometric features. In *Proceedings of Vision Modeling and Visualization'01*, pages 233 – 240.

Silva, C., Chiang, Y., El-Sana, J., and Lindstrom, P. (2002). Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization'02 Course Notes*.

Sloan, S. (1986). An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, 23:1315–1324.

Snoeyink, J. and van Kreveld, M. (1997). Linear-time reconstruction of Delaunay triangulations with applications. In *Proceedings of 5th European Symposium on Algorithms*, pages 459–471.

Sorkine, O., Cohen-Or, D., Goldenthal, R., and Lischinski, D. (2002). Bounded-distortion piecewise mesh parametrization. In *Visualization'02 Conference Proceedings*, pages 355–362.

Sorkine, O., Cohen-Or, D., and Toledo, S. (2003). High-pass quantization for mesh encoding. In *Proceedings of Symposium on Geometry Processing'03*, pages 42–51.

Speckmann, B. and Snoeyink, J. (1997). Easy triangle strips for TIN terrain models. In *Proceedings of 9th Canadian Conference on Computational Geometry*, pages 239–244.

Staadt, O. and Gross, M. (1998). Progressive tetrahedralizations. In *Visualization'98 Conference Proceedings*, pages 397–402.

Szymczak, A. (2002). Optimized edgebreaker encoding for large and regular meshes. In *Data Compression Conference'02*, page 472.

Szymczak, A., King, D., and Rossignac, J. (2000). An Edgebreaker-based efficient compression scheme for connectivity of regular meshes. In *Proceedings of 12th Canadian Conference on Computational Geometry*, pages 257–264.

Szymczak, A. and Rossignac, J. (1999). Grow & fold: Compression of tetrahedral meshes. In *Proceedings of the 5th ACM Symposium on Solid Modeling and Applications*, pages 54–64.

Szymczak, A., Rossignac, J., and King, D. (2002). Piecewise regular meshes: Construction and compression. *Graphical Models*, 64(3-4):183–198.

Taubin, G., Guéziec, A., Horn, W., and Lazarus, F. (1998a). Progressive forest split compression. In *SIGGRAPH'98 Conference Proceedings*, pages 123–132.

Taubin, G., Horn, W., Lazarus, F., and Rossignac, J. (1998b). Geometry coding and VRML. *Proceedings of the IEEE*, 86(6):1228–1243.

Taubin, G. and Rossignac, J. (1998). Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115.

Tautges, T. and Mitchell, S. (1995). Whisker weaving: A connectivity-based based method for constructing all-hexahedral finite element meshes. In *Proceedings of the 4th International Meshing Roundtable*, pages 115–127.

Touma, C. and Gotsman, C. (1998). Triangle mesh compression. In *Graphics Interface'98 Conference Proceedings*, pages 26–34.

Trotts, I., Hamann, B., Joy, K., and Wiley, D. (1998). Simplification of tetrahedral meshes. In *Visualization'98 Conference Proceedings*, pages 287–295.

Turan, G. (1984). Succinct representations of graphs. *Discrete Applied Mathematics*, 8:289–294.

Tutte, W. (1962). A census of planar triangulations. *Canadian Journal of Mathematics*, 14:21–38.

Tutte, W. (1963). A census of planar maps. *Canadian Journal of Mathematics*, 15:249–271.

Wilhelms, J., Gelder, A. V., Tarantino, P., and Gibbs, J. (1996). Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *Visualization'96 Conference Proceedings*, pages 57–64.

Witten, I. H., Neal, R. M., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Woo, M., Neider, J., and Davis, T. (1996). *Open GL Programming Guide*. Addison Wesley.

Wu, J. and Kobbelt, L. (2003). A stream algorithm for the decimation of massive meshes. In *Graphics Interface'03 Conference Proceedings*, pages 185–192.

Xiang, X., Held, M., and Mitchell, J. (1999). Fast and efficient stripification of polygonal surface models. In *Proceedings of Interactive 3D Graphics*, pages 71–78.

Yang, C., Mitra, T., and Chiueh, T. (2000). On-the-fly rendering of losslessly compressed irregular volume data. In *Visualization'00 Conference Proceedings*, pages 101–108.

Yoon, S., Salomon, B., Gayle, R., and Manocha, D. (2004). Quick-VDR: Interactive view-dependent rendering of massive models. In *Visualization'04 Conference Proceedings*. to appear.