Dealing with Shape Complexity for Internet Access and Graphic Applications

Jarek Rossignac
GVU Center and College of Computing
Georgia Institute of Technology
www.gvu.gatech.edu/~jarek

Abstract

Standard representations of 3D models are so verbose that only very simple models can be accessed over common communication links for immediate viewing. This situation is not likely to improve, since the need for more accurate 3D models and their deployment throughout a broader spectrum of industrial, scientific, and consumer application areas will outpace the improvements in transmission bandwidth to the office, home, or mobile worker or private user. Recently developed multi-resolution modeling technologies play an important role in addressing this bandwidth bottleneck, especially when combined with other approaches, such as intelligent culling, pre-fetching, and image-based rendering. This tutorial will discuss the details of compression, simplification, and progressive transmission techniques and of their interrelations.

Introduction

Objectives

In this tutorial I discuss the storage complexity of digital representations of 3D shapes. I focus on the details of compression, simplification, and progressive transmission techniques. My objective is to provide the reader with an in-depth expertise of the details of a selected subset of these techniques. You may ask "Why?". You are right, a technique popular today may be replaced tomorrow. So why teach a particular one? I believe that one should understand one approach in depth to be able to grasp the differences between approaches and appreciate the hidden subtleties. If you hear about or read about a new approach, you should be able identify its advantages and drawbacks, based on the experience you had with the one technique that you know in details. What kind of drawbacks am I talking about? Complexity of the algorithms in excess of what you would consider implementing, poor behavior for a type of models that are important for your application domain, limitations on the type of models that it is capable of handling.

Motivation for this research

3D models play an important role in manufacturing, architecture, petroleum, entertainment, training, engineering analysis and simulation, medicine, science, and so on. I believe that they will soon impact electronic commerce and replace the flat windows on our screens. Why is that?

Two reasons. First, 3D rendering is pervasive and practically free (all PCs come with a powerful 3D graphics board). Second, 3D is how we see our universe (in perspective and with parallax and motion queues). So why should we put up with other, less natural representations?

Some would say: "Because 3D does not provide any intrinsic benefits". This is a common mistake. 3D perspective provides you with the ability to see details within a broader context of their surrounding. Furthermore, this presentation is natural and thus does not come at the cost of an excess of cognitive load. I mean that we don't have to think about what the images mean if they look like the type of things that we have been used to see since childhood. We can tell what things are by the way they look. We can tell how they relate to each other by the way they touch or move.

Of course, one could suggest that our children are more familiar with 2D animated characters that appear in their video games than with the 3D world, and one could use this as an argument to point out that we can adapt to a 2D presentation and to the use of icons, like a trashcan on a desktop. Sure, we could. But why should we? Who wants to use a black&white TV or computer screen today? Yet, some of us can still hear the voices of those who believed that color monitors were unnecessary and will never take off.

So, convinced? You probably were before reading this, since you decided to read it. So let's move along.

For many of these applications, 3D data sets are increasingly accessed through the Internet. Why? Because the data you want is most often somewhere else. Maybe it is available form online 3D catalogs or being modified by your designer colleagues.

The number and complexity of these 3D models is growing rapidly. Why? Several reasons: improved design and model acquisition tools, wide spread acceptance of this technology, and incessant need for higher accuracy. If we want to use 3D models, we need them to be more than course replicas of the real thing.

In many of these applications, human productivity or satisfaction would be significantly enhanced by the possibility of an immediate access to remotely located 3D data sets for visual inspection or manipulation.

Do we always have to transmit the 3D shape? No, sometimes we could get away with imagery and still think that we are seeing a 3D environment. For example, like the backdrops in the movies, we could use images (textures) for the background and thus avoid sending geometry. We could also send one image and then transmit the minimum information required for modifying this image to produce the next one. For example when we pan, a portion of the image is simply shifted. Nevertheless, even when these and other image-based rendering techniques are used to reduce the fraction of the 3D representation that must be transferred at any given time, geometry transfer remains the bottle-neck.

Consequently, it is urgent to develop optimal bit-efficient formats and associated compression and fast decompression algorithms for 3D models.

Target audience

This tutorial is geared towards researchers and developers interested in inventing and implementing simple and robust transmission acceleration techniques for 3D models. It is not designed to provide leading researchers with a

comparative study of the subtleties of various approaches. They don't need a tutorial. Instead I focus on presenting practical solutions and offer my impressions of the field, stressing the opportunities for exploring other approaches.

Prerequisites

I would like this tutorial to be accessible to most readers who are familiar with only he most primitive concepts of 3D geometry, graphics, and algorithms. I would like to say that I expect you to know only what a point is, what a pixel is, and what an array is. Unfortunately, I also expect you to know a bit more about programming. I will assume that you know how to manage doubly inked lists and other graph structures. I will also assume that you are comfortable with simple geometric constructions and with the basic concepts used in ray-casting and projective rendering techniques.

Methodology

After this introduction, I will start with a detailed discussion of a simple solution to a simple problem. How to compress and decompress a simple triangle mesh. Then I will explain how to generalize these compression techniques to a broader, and more useful, domain…of coffee cups…and airplanes. In each chapter, instead of discussing and comparing various approaches, I will focus on one or two.

Outline of the tutorial

The tutorial is structured as follows.

Compression

I will first spend some time discussing why we want to focus on triangle meshes and on tetrahedra meshes. I will define what I call simple triangle meshes (STMs) and present a simple data structure and primitive operations for them. I will also discuss some of their properties that we will exploit later to estimate compression ratios or prove compression bounds. I will clarify the distinction between connectivity and geometry and analyze their relative costs in uncompressed representations.

I will introduce the Edgebreaker compression technique for STMs. I am rather proud of it, because the complete compression algorithm takes only a page of a low-level Fortran-like code and yet it is one of the best compression technique available to date. It bridges the gap between practical compression efforts for graphics and theoretical results on planar graphs encoding. In fact, I dare say that it has improved on prior art in both areas. Edgebreaker produces what I call the CLERS sequence of symbols, one per triangle, which can always be encoded with an average of less then 2 bits.

I will also teach an elegant (i.e., simple and fast) decompression technique that Andrzej Szymczak and I have discovered for the CLERS encoding produced by Edgebreaker.

I will also describe the Matchmaker approach that David Cardoze and I have developed for representing non-manifold solid models using data structures for manifold triangle meshes. I will then discuss extensions of these compression/decompression techniques to more general triangle meshes, which may have handles and holes. I will discuss a simple trick from Touma and Gotsman for dealing with the holes in triangle meshes. I will build upon the discussions that Gabriel Taubin and I have had on how to best explain the encoding of handles.

Edgebreaker, and many of the competing techniques focus on the compact encoding of the connectivity information. The connectivity specifies which sample points belong to which triangle, and thus also which triangles are neighbors. The encoding of the location and of other properties attached to the sample points, must also be addressed. The choice of a technique for compressing locations is often orthogonal to the choice of the method for compressing the connectivity. Most techniques are based on geometric estimators. I will describe a very simple and effective geometry compression technique based on a predictive scheme first proposed by Touma and Gotsman.

I will review several improvements of the Edgebreaker approach developed in collaboration with a PhD student Davis King or by other researchers in the US, Germany, and Israel.

Progressive transmission

I present two simplification techniques as a form of lossy compression. Simplification takes a triangle mesh and produces another one, which has fewer triangles, but still resembles the original. The simplification that I will describe first is often referred to as "vertex clustering". I developed it with my friend Paul Borrel at IBM Research many years ago. I will include it here because it is rather trivial to implement in a very fast and robust manner. It also can automatically simplify objects that have many handles or connected components into objects that have much fewer of those. This is an important advantage over other simplification approaches. Unfortunately, vertex clustering does not produce the best results. Other methods, although slower and often restricted to simple meshes, have been proposed to produce better looking simplified models for a given triangle budget. Nevertheless, vertex

clustering is still in use in its original form and has recently been combined with other techniques by several researchers. I will also describe one of these "better looking" techniques. It was developed with Remi Ronfard and is similar to a mesh simplification technique developed simultaneously by Hugues Hoppe. Both are inspired by Hugues prior work on edge-collapse operations.

At this point, we will have the tools to compute a more or less simplified model and to compress it for faster transmission. What if we transmit a very crude model first and then realize that we need a more accurate model? Can we use any of the previously received information and thus reduce the cost of transmitting the better model? That is called *progressive transmission.* I will start with a description of Hugues Hoppe's Progressive Mesh. This very simple solution is based on the encoding of a sequence of inverse edge-collapses, which in some sense undo the simplification steps in reverse order. It is simple and effective, but does not produce a very compact format. I will describe the improvements that Renato Pajariola and I have developed. We traded granularity of the progressive model for a better compression. This approach is called Compressed Progressive Meshes.

Compression

Simple triangle meshes (STMs)

Interpolating samples

The shape of a 3D object is defined by its boundary. What is the boundary? Let me for the sake of simplicity say only that it is the surface that separates the interior of the object from its exterior.

That surface may be sampled by generating a dense set of points on it. Yet, the sample points do not by themselves define the shape. For example, the set of points in Fig. 1 (left), do not specify a surface. A triangular interpolation of them (right) does. We need to understand how to interpolate between the samples. Why is that?

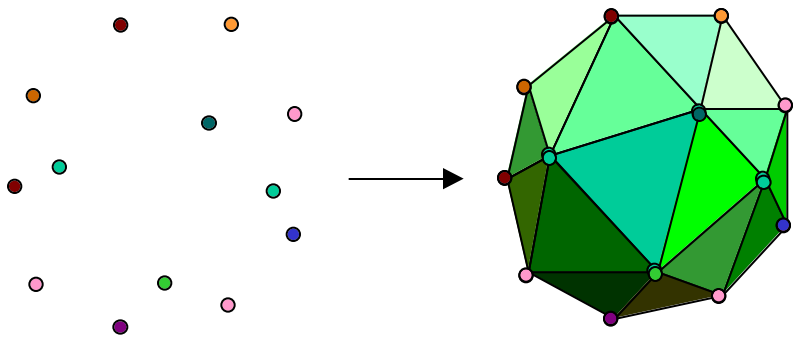

Fig 1: The interpolation of a set of sample points by a triangle mesh

Here is a simple answer to this question. Without the interpolation, we would not be able to compute the intersection of a ray with the object. Why? Because the ray would most probably miss all the sample points. So? Well, if we can't compute ray/surface intersections, how are we going to compute pictures of the object? After all, the ray-tracing technique used to produce realistic images of 3D scenes is based on computing ray-surface intersections.

What if we project the sample points on the screen instead? We would avoid the ray/surface intersection problem altogether. True, but two problems will occur.

First, we will not know how to compute the color and intensity of the reflected light, because we do not have any information about the orientation of the surface at a sample point. We could fix this problem by associating the surface normal to each sample point. (I call these sample points "**surfels**", short for surface-elements and counterpart of the term "pixel".)

Second, several surfels may project onto one pixel and none onto another pixel, that should have been covered by the projection of the object on the screen. We could fix this problem by making sure that there are enough surfels so that each pixel that should be covered is covered by at least one surfel. If we do not have an adaptive surfel model, this solution would prevent us from zooming too much. An alternative solution would be to replace surfels with balls (or adaptively oriented disks) so that there is no gap between the balls of neighboring surfels on the surface. Balls of the same radius would work if the surfels were uniformly distributed on the surface. The color of each ball would be computed from the surface normal at the associated surfel. Now some pixels may be covered by the projections of the balls of several surfels. Which color should be used? A weighted average of colors may provide a

good compromise if the weights are proportional to the distance between the surfels projection and the center of the pixel and if they take into account the relative depth of these balls.

So what is the conclusion? Do we need interpolation at all? If we took all sorts of precautions, such as having the surfels uniformly distributed and ensuring that the distance between any two surfels is smaller than the constrictions or other details of the surface, then we may be able to do graphics. But the price of these precautions is not worth the benefits. What do I mean by "price"? The required number of surfels will increase storage and transmission delays. Given that the whole purpose of this tutorial is to discuss how to reduce them, we will forgo the opportunity of an interpolation-free life and focus on the transmission of models that combine sample points (with or without normals) with an interpolation scheme. So why did I bother you with this discussion? Because I want you to realize that we have a choice of representation and that although we have made today the decision to use interpolating models, that decision may need to be revisited if the circumstances change.

Smoothness and connectivity

OK. So we need to know how to interpolate between the samples. There are two aspects of the interpolation: **smoothness** and **connectivity**. To better understand them, consider the simplest interpolation: a **triangle**. It is a planar facet defined by three sample points, called the **corners** of the triangle. A triangular interpolation may differ from the original shape, not only because it may not exactly match the shape, but because is may have drastically different smoothness or connectivity.

True, we could have opted for interpolations that have non-triangular connectivity. For example, samples are often arranged into quadrilaterals. It may be valuable to retain that original organization for several reasons, one being that they may be easier to compress than triangles. This was the topic of a study by Davis King in me and by several other researchers. Nevertheless, they can be easily triangulated and I will focus the rest of this tutorial on triangle connectivity. (You can only accomplish so much you your life.)

Clearly, if the original shape is a smooth surface, say a coffee cup, a triangle-interpolation of a set of sample points will not look like it unless a lot of samples are used or unless the interpolation, or at least its rendering is smoothened. So, the solution is to use higher order interpolating or approximating surfaces. You may have run into such things as B-splines, NURBS, or subdivision surfaces. I view them as filters that start with an interpolating triangle mesh and then apply a smoothing process that recursively chop corners or splits and bends faces and edges. The result is a smooth curved surface that either interpolates the samples or runs close to them. So, you should remember that even when one uses smooth surfaces, they are in general defined in terms of a triangular (or quadrilateral) interpolation of the sample points. Therefore it is important to compress such triangular interpolations, even when a smooth interpolation is ultimately produced.

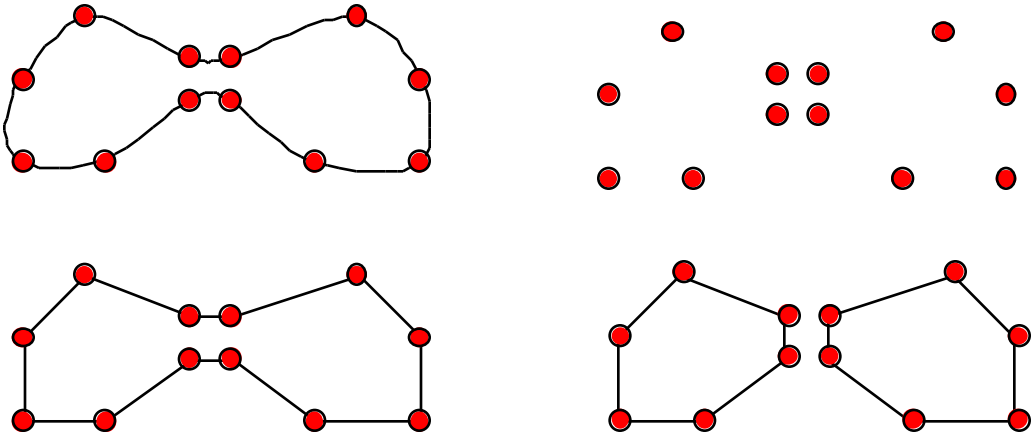

Fig 2: The interpolation of a set of sample points may have the wrong smoothness or connectivity.

The connectivity of the triangle interpolation (and thus of the corresponding smooth surface) may not always agree with the connectivity of the original surface. By connectivity, I mean the graph that maps each sample point into its neighbors. Figure 2 attempts to explain this in 2D. Instead of a surface, I show (top-left) a smooth curve upon which a set of sample points (large dots) were selected. These are shown without the curve (top-right) to stress the fact that the points alone do not define the interpolation. Here, in 2D, the interpolation corresponds to the way edges interpolate between consecutive sample points along the curve. These edges in 2D correspond to triangles in 3D. A correct interpolation (bottom-left) has the same connectivity as the original curve. If two samples were neighbors on the original curve, they are on the correctly interpolated approximation. Yet, although topologically correct, the

resulting shape is piecewise linear and is thus a poor approximation of the smooth shape. A much worse interpolation (bottom-right) corresponds to a different connectivity.

This example may also serve as an answer to a question that we have not yet asked, but that is at the heart of some compression decisions. Although we may want to know the connectivity, so that we can produce the desired linear or smooth interpolation, it is not clear that we need to encode it. Why not simply encode the sample points and transmit them. Let the decompression algorithm recover the connectivity automatically. Well, as shown in Figure 2 (bottom-right), this may not be easy, especially when the size of features is comparable to the distance between neighboring sample points. Basically, to avoid having to encode the connectivity and still be sure that the recovered connectivity is correct, we would have to ensure that the neighbors of each sample points are closer to it than other samples. And even that clause will not suffice. But with more precautions it can be done, simply because techniques for guessing the connectivity of a set of sample points exist and because we can agree on which technique will be used. Therefore, we can try to ensure that the sample points are chosen in such a way that the application of the chosen connectivity-recovering techniques will produce the correct connectivity. If it does not, we can insert additional sample points. This approach has been used for finite element mesh generation.

Because we do not always have control over the sampling process, because the connectivity-recovery algorithms are computationally expensive and rather challenging to debug, and especially because we do not want to transmit more sample points than necessary, we will assume that the connectivity has to be encoded and transmitted.

Simple Triangle Mesh (STM)

A simple triangle mesh is homeomorphic to a sphere. What does it mean to be homeomorphic to a sphere? Think about a sphere whose surface is covered by triangles that do not overlap. That's it? Well, not quite. The triangles do overlap, but only at their borders. Each triangle has 3 borders. I am carefully distinguishing **borders** from **edges**. Although not endorsed in the literature, this distinction will help us make things clear. Think about each triangle as a tile on the floor. It has three borders, which are straight line segments. Now, arrange the tiles so that each border of each triangle is either exactly aligned with the border of another triangle or is not touching any other triangle, except maybe at its two end-points. Such a flat arrangement of four triangles is shown on Fig. 3 (bottom-left). Each triangle has three **corners**. You may have noticed that I have refrained from calling them **vertices**. Why. Because I want to reserve the terms edges and vertices to the entities represented in the entire mesh, and distinguish them from the corresponding topological elements of a triangle. So, the small arrangement of Fig. 3 (bottom-left) has 4 triangles, 6 vertices, and 9 edges.

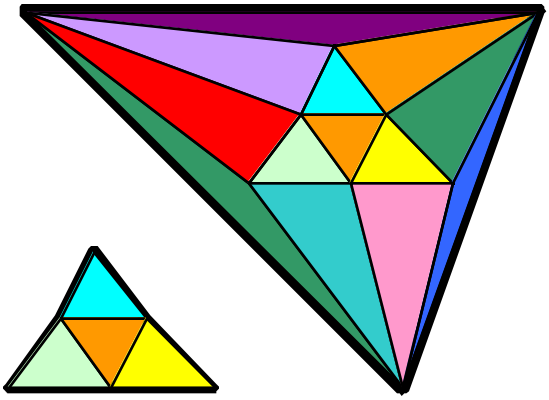

Fig 3: A simple triangle mesh is a planar triangle graph.

Three of these edges correspond to line segments where two borders overlap. I call them the **interior edges** of the mesh. Each interior edge corresponds to exactly two borders. The other six edges correspond to a single border each. I call them the **exterior edges** of the mesh. (Others have used "open edges" or even "borders".)

Three of the vertices correspond to a single corner. The others mark locations where more corners meet.

The connectivity information associates three vertices with each triangle. These specify the location of the triangle corners, and these define the shape, position, and orientation of the triangle.

Note that, although the tiles are initially arranged on the floor, we could move the vertices out of the plane.

For example, we could fold the small arrangement of Fig. 3 at its interior edges by moving the three single-corner vertices out of the plane. If we make them coincident and replace them with a single vertex where three corners meet, we have changed the connectivity and have produced a tetrahedron with 4 triangular faces, 6 internal edges, and four vertices. Such a tetrahedron is the simplest example of a Simple Triangle Mesh (STM). The folding

operation, which has identified several vertices and "zipped" together pairs of external edges is the essence of the Wrap&Zip approach that we will use for decompressing triangle meshes.

So, you may think of an STM as a planar triangular tiling of a floor that has been folded and zipped so that it does not have any external edges. That unfortunately is not a sufficiently restrictive definition. Why? Well, the devious mind could construct flat tillings, then fold them and zip them to obtain figures with separate components, handles (through holes), and even Klein bottles that are not orientable. So, to avoid such complications, we need top be more specific.

Let's arrange our triangles on the floor so that there are only and exactly 3 external edges (Fig. 3, top-right). The tiling has to be connected. The triangles cannot overlap.

Remember that, except for external edges, each border is matching exactly one other border. If this was not the case, we would have a T-junctions, as shown in Fig. 4 (left). Why do we wish to prevent such T-junctions? Because it is harder to represent them in a uniform and compact data structure and because the compression efforts have assumed simple data structures. If you did have T-junctions, you could always split some triangles and get rid of them, as illustrated in Fig. 4 (right) or think of triangles with T-junctions at their borders as polygons with flat corners.

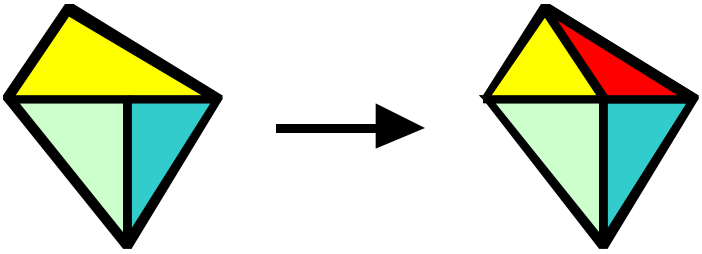

Fig 4: Splitting a triangle to remove a T-junction

Now take a big triangle whose three borders match exactly the exterior edges of this arrangement of tiles on the floor. Slide the triangle under the tiles without disturbing anything. (You have to do this with sudden push.) You made a two layer tiling, with a big triangle as the first layer. In this arrangement there are no more external edges.

Now gently, blow some air between the two layers and let the whole thing inflate a little by moving up vertices that are not corners of the bottom triangle. You have an example of an STM.

You may further deform it by moving vertices. No matter what you do to the vertices, if you don't tell anybody, they will still think that you have an STM. But of course, you may have inadvertently, or purposely, made the triangles intersect each other or made two vertices or two edges coincide, although the representation you have thinks that they are different vertices. We would say that you have the connectivity of a STM, but that its immersion in space (i.e. the actual position of the vertices) is inconsistent with that connectivity. We cheat in this way when representing non-manifold models using an STM structure.

To draw STMs, we will use the convention illustrated in Fig. 3 (right). You basically pick a back facing triangle and stretch it, pulling the rest of the mesh, until all other triangles are front-facing. Then you draw what you see. So, remember that the three edges that bound such a figure are not really external edges. They each correspond to a border of some front-facing triangle and to a border of the back triangle.

Testing whether a set of triangles forms an STM

Only few models are STMs. You want some examples of non-STMs? How about a cup? It has a handle. Some call that a through-hole. So a cup is not an STM. Now to be precise, a real cup is usually smooth, and therefore a triangulated surface will only be an approximation of it, STM or not STM. That observation is at the heart of the whole tutorial, because most compression techniques are lossy, and the fact that we accept in the first place to use an approximating triangulated model for a ball or cup justifies the fact that we can take some liberty with that model in order to compress it.

So, you should ask: "How do we know whether a model has STM connectivity or not?". The answer is not trivial. In addition to the fact that we do not want any T-junctions and that all edges are internal, we need to ensure that the surface has a single connected component, is orientable, and has no handles. We could start by making sure that triangles don't intersect one another. To specify this requirement more precisely, let us consider the triangles as being relatively open. By that I mean that they do not include their borders and corners. Let the edges also be relatively open, so that they do not include their bounding vertices. Now, a proper immersion of an STM has to following property. All the triangles, edges, and vertices are pair-wise disjoint. If this is the case, their union forms a tight shell that separates the interior of a volume from its exterior. Paint the exterior faces of the triangles in green

and the interior faces in red. If, from where you are, you see a green face, then it is front-facing. This process helps us to define outward pointing normals. These are important for graphics.

In practice, however we don't want to even consider geometry. Why? First, because vertex locations may have accuracy problems, in which case we may not be able to ensure that we are always making consistent decisions. Second, because testing for geometric intersections is expensive (quadratic cost, need for extended precision to avoid round-off problems). Third, because we may want to use an STM connectivity and the associated data structures to represent shapes that violate our non-intersection rule.

So, let us define STM connectivity in terms of adjacency relations, regardless of vertex locations.

Let us say that two corners are **coincident** if they point to the same vertex. Let us also say that two borders, B1 and B2 are **coincident** if each corner of B1 is coincident with a corner of B2.

We say that two triangles are **adjacent** when they have coincident borders. We will assume that two adjacent triangles have only one coincident border. That is, only one border of triangle A coincide with a border of triangle B.

Two adjacent triangles are **border-connected**. This border-connectivity relation is transitive. So if A is border-connected with B and B is border-connected with C then A is with C. To test whether a mesh is border-connected, you start with a triangle, paint it, and then recursively visit all adjacent triangles that have not yet been painted. If at the end you have reached them all, the mesh is border-connected. More formally, a set of triangles is border-connected if any pair of its triangles is border-connected by a chain of zero or more triangles.

Let us assume that each triangle is represented by its three corners and that each corner is simply an identifier of the corresponding vertex. So, several corners point to the same vertex. Assume that these 3 corners are arranged in a circular list for a given triangle. The list has no beginning: it is circular. So {a,b,c}, {b,c,a}, and {c,a,b} are the same list. A different list could be stored as {b,a,c}, {c,b,a}, and {a,c,b}. Consider that the Ids (a, b, and c) are integer indices or pointers. We could store the list of corners in a table of 3 entries so that the smallest of the three is the first one. The other two would be either in order or out of order. That choice may be used to define an orientation of the triangle. This orientation may be used to also orient the borders. So, the borders of triangle {a,b,c} would be the oriented segments ab, bc, and ca. We say that two adjacent triangles have **compatible orientations** if their common border have opposite orientations. For example {a,b,c} and {b,d,a} do not have compatible orientations. {a,b,c} and {b,a,d} do.

Here's simple process for testing whether the mesh is **orientable**. Pick an orientation for the first triangle, then visit the other triangles, as we did for testing connectivity. In addition to painting the new triangles, you switch their orientation if they are not compatible with the last triangle. You also test whether each the current triangle is compatible with its previously visited neighbors (adjacent triangles).

We say that a triangle is **incident** upon a vertex if one of its corners references the vertex.

A vertex is **manifold** if its incident triangles are border-connected. To visualize what it means, consider that a manifold vertex is surrounded by a chain of triangles, each one being border-connected with the next. A non-manifold vertex, on the other hand, has several of such chains incident upon it. Like for example when the apices of two cones meet.

Consider a set of triangles with the following properties:

1. Each border is coincident with exactly one border of another triangle.
2. Any two adjacent triangles have only one coincident border.
3. The mesh is border connected.
4. The mesh is orientable
5. All vertices are manifold

Such a set is an orientable manifold surface and represents the boundary of a solid (3D volume). However, the volume may have handles (i.e., through holes). How can we detect these? The topological answer is to check for the existence of closed loop cuts that would not split the surface. This is non-trivial. Later, when we discuss the extension of the Edgebreaker algorithm to meshes with handles, we will learn a practical method for dealing with handles (and of course for detecting if there are any). For the time being, we will invoke a simpler trick to test for the existence of handles. It is based on the following observation.

The number of handles in an orientable manifold mesh is equal to T/4-V/2+1 where t is the number of triangles and V is the number of vertices. This formula is derived by simple substitution in the Euler equation, V-E+T=2(S-H), where E is the number of edges and S the number of connected components. We simply replace E with 3T/2, because there are 3T borders and each pair of them is an edge.

So, a simple strategy for checking that there are no handles in a connected, orientable, manifold mesh is to count the number of triangles and vertices and to verify that T/4-V/2+1 is zero. For example, for a tetrahedron, we have T=4 and V=4, thus the formula yields 1-2+1, which is zero.

Data-structure for representing STMs

In the above discussions, we were implicitly assuming that the mesh is represented in some data structure that identifies the vertices through pointers or some ID to a table of vertex coordinates (and possibly other vertex properties). For simplicity, we will assume that each vertex is identified by an integer number. We were also implicitly assuming that the connectivity was identified by a table of triplets of corners, one per triangle. Indeed these two tables suffice to define the connectivity.

The variation of the Euler formula presented above yields for STMs: T=2V-4. Therefore, for complex STMs, we have roughly twice more triangles than vertices. If we were to represent each vertex coordinate with B bits, the vertex table would take 3BV bits, ignoring normals, colors, and other vertex data. Given that for each corner we would need to store log(V) bits (or more precisely the ceiling of log(V)), we would need a total storage of 3BV+6Vlog(V) bits. Which is 3V(B+2log(V)) bits. Note that as soon as 2log(V) becomes larger than B, the triangle table dominates the storage cost. Research on compression tries to balance the storage costs of vertex locations and of connectivity.

Before we attempt to compress the connectivity, let us represent the mesh in a different format, which will not only make it easy to retrieve the corner information, but will support a more effective traversal of the mesh from one triangle to an adjacent one. Why? Because compression techniques discussed here are based on this traversal.

We will base the traversal on operators that move from one border to another and that identify the triangle associated with the border and the vertex of that triangle that is not a vertex of the border. The semantics of these operators is best described visually (See Fig 5.) The bottom border, marked b, on the top-left triangle will be the starting entity that I will use to illustrate the operators. The border b is oriented, as we discussed earlier. For simplicity, think of b as an integer, which identifies a border in some table. We will discuss data structures in a minute.

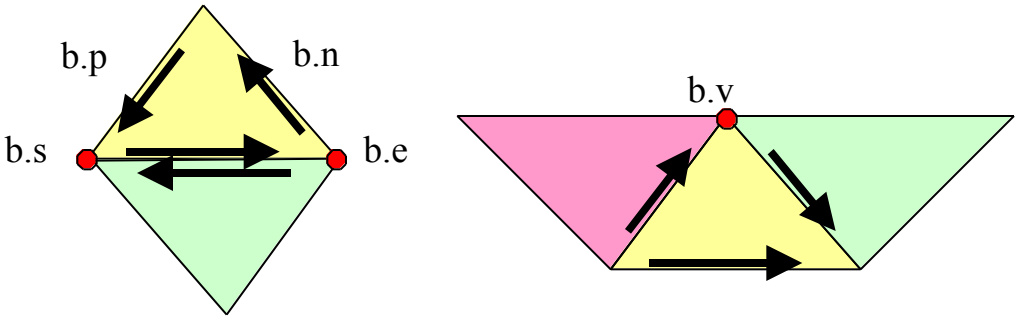

Fig 5: Local border operators.

Let b.t be the integer id of the triangle associated with b. I am using here a simplified object-oriented notation. A trivial way to implement it would be to have a triangle table T and to interpret b.t as T[b]. For simplicity, I will use b.x.y as a short for (b.x).y.

Although we do not need to store them explicitly, it is convenient to have names for the starting and ending vertices of b. Let me call them b.s and b.e, respectively. The third vertex will be denoted b.v (it is shown on the right of Fig. 4). All three are integer references to the vertex table.

Let b.p and b.n be integers that refer to the previous and next borders around b.t.

So far, we have only provided operators that were readily available from the simple listing of corners for each triangle. To support the traversal of the mesh from one triangle to an adjacent one, we define b.o, which refers to the other border that coincides with b. As explained below, b and b.o have opposite orientations. Of course b.o.o is b.

During the compression, we will use the references indicated on the right of Fig. 5. In addition to b.t and b.v, they include b.l and b.r. These will help us to access the left and right neighbors of b.t.

Note that we do not actually need to store the links represented by all these operators. For example, b.p is b.n.n. Also, b.r is b.n.o and b.l is b.p.o.

I like to store only b.o and b.v. This is an aesthetic choice. How do I get b.t and b.n?

I simply make the entries of the three borders of an given triangle consecutive in my data structure. So, b.t is b DIV 3, where DIV is an integer division. I make sure that I store them in the order {b, b.n, b.p} that is compatible with the triangle orientation. Then b.n is simply 3(b.t)+((b+1) MOD 3). And b.p is b.n.n.

If you were to implement these as tables, you would need only the O and the V tables, where b.o is O[b] and where b.v is V[b]. Actually, V[b] would return an integer that we could use as an index into the arrays that contain the vertex coordinates and other properties.

So, how would these two tables be used? For example, if you wanted to go to the right triangle, you would replace b by b.r. If you were using tables as data structures, you would say b:=O[N(b)]; where N(b) could be a macro defined as 3(b DIV 3)+((b+1) MOD 3).

What is the storage cost of this representation?

For each border b, we need to store two things: a log(V) bit reference b.v to the vertex table and a log(6V) bit reference b.o to the border table. We need log(6V) bits because there are 3T borders, three per triangle and because there are roughly twice as many triangles than vertices.

So the total storage cost of the connectivity using this data structure is 6V(log(V)+log(6V)), or 12Vlog(V)+6Vlog(6)). For large V, this takes twice more space than the simple table of triplets of corners, which requires 6Vlog(V) bits. So this is not a step in the right direction. We will only use this temporary representation for to support a simple compression or simplification algorithm.

The two tables, O and V, may be computed efficiently from the table of corner triplets. How is that done?

First, we can trivially fill in the V table for the triplet of consecutive borders of each triangle. We simply enter one triangle at a time. For each triangle, we append three borders. For each border, we enter the id of the opposite vertex. Thus, if the first triangle is {a,b,c}, the first border would be bc, the second ca, and the third ab. Remember that the starting and ending vertices of border bc are not stored explicitly. They may be accessed using the DIV and MOD operators as above, or though combination of the primitive operations (b.n.v and b.n.n.v).

To fill in the O table takes more work. We first make three versions of each triangle, by performing a circular permutation of its corners. So, triangle {b,a,c} would also yield {a,c,b) and {c,b,a}. With each copy, we associate the triangle ID. Then we rearrange each entry by swapping the first two corners if they are out of order. We mark these entries where the swap has occurred. Now each entry represents a border or its inverse. We can sort them efficiently using a hashing method. (We know all possible keys for the first corner.) Now, adjacent triangles will be consecutive in this sorted list, because the coincident borders were represented in the same way (one was inverted) and thus ended up being consecutive.

In conclusion, I have specified in this chapter what I mean by a simple triangle mesh, I have suggested a simple data structure for representing it and local operators for traversing it, and I have proposed simple and efficient algorithms for building the data structure and for implementing the operators. Now we are ready to compress connectivity of the mesh.

Edgebreaker compression for STMs

Edgebreaker produces a sequence of symbols taken from the set {C,L,E,R,S}. I call this sequence the ***clers*** stream. There is one symbol per triangle, although the two initial ones could be skipped, because they are always C. It also produces a sequence of vertex ids. I will call this sequence the ***order*** stream. Because Edgebreaker only appends entries into these streams, I will use them as logical names of output channels and use a WRITE command to append to them.

The entries in the *clers* stream will be encoded using a variable length binary coding scheme. I will propose a simple one and discuss improvements later.

The *order* stream defines the order in which the vertices should be transmitted. More precisely, it is a dictionary. If the first number in the order stream is K, it means that our vertex number K will be labeled as vertex one by the decoder. Thus, we can simply send the vertices in the order in which the decoder would be expecting them. For example send vertex K first.

Note that as long as the sender and the decoder agree on a systematic perturbation, other orders may be chosen. We may for example chose to send one vertex out of ten first, and then send the rest. Why? Possibly to better distribute the early vertices and know more about the overall shape early on. This knowledge may help in a progressive scheme and may also help better predict the locations of the remaining vertices. The vertices will also be encoded as explained later.

Edgebreaker performs a systematic depth-first traversal of the mesh. It enters each new triangle from one side, tries first to go right and then left. It only visits each triangle once. It paints (marks) all the visited vertices and triangles. To implement the painting metaphor, we use the v.m and t.m operators which take a binary value in the set {*painted,virgin*}. They may be implemented using separate arrays, Mv[b.v] and Mt[b.t]. Let *clers* and *order* be initially empty and the paint tables be set to *virgin*.

Let border b denote the current triangle, Edgebreaker check whether the opposite vertex, b.v is *virgin* and if so, it appends the symbol C to *clers*. If b.v.m is *painted,* then Edgebreaker checks whether the left and right triangles have been painted (using b.l.t.m and b.r.t.m). The four possible combinations of these two variables correspond to the symbols, L, R, E, and S. The five situations are depicted on in Fig. 6.

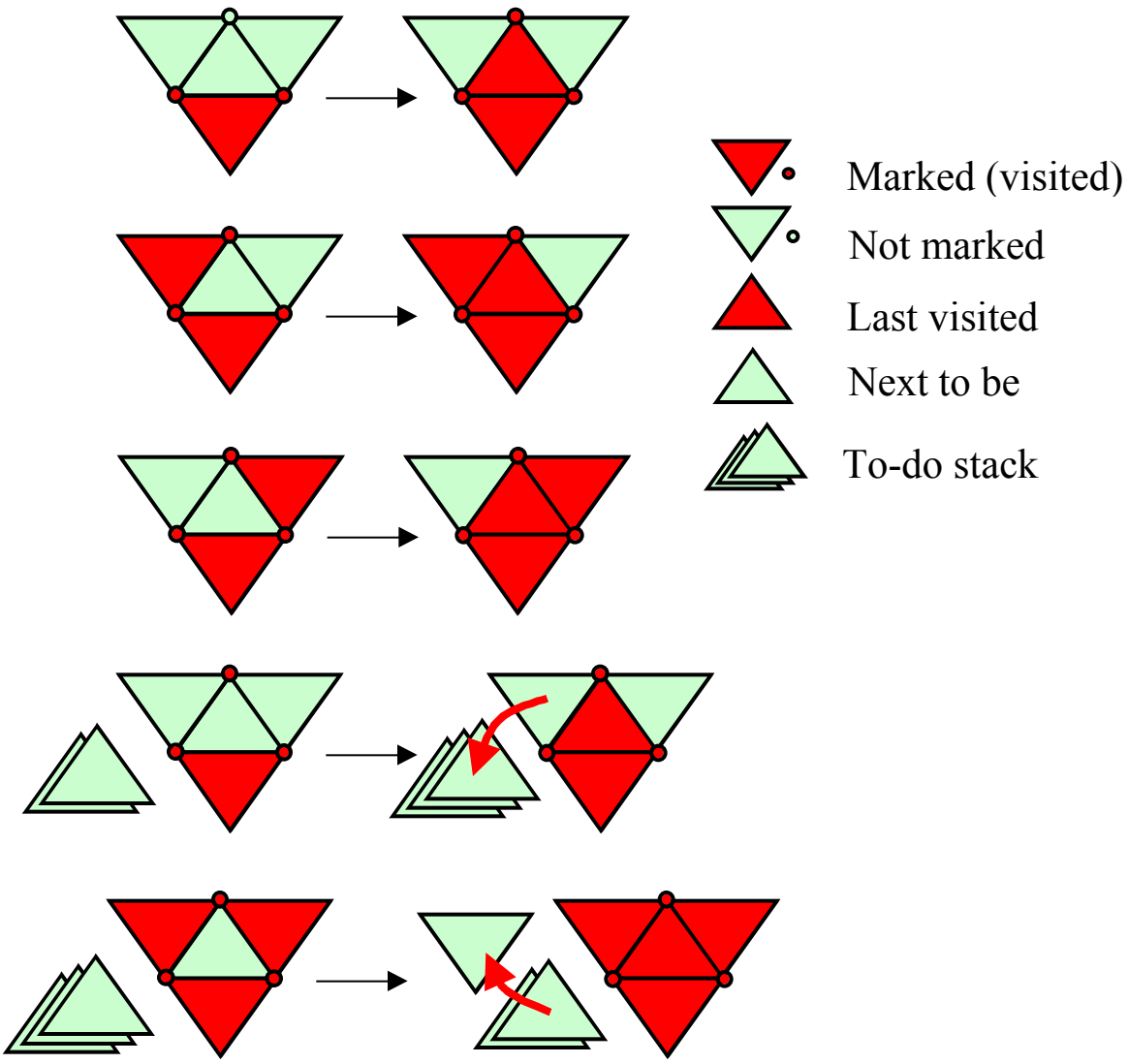

Fig 6: Edgebreaker CLERS states and labels.

The entire algorithm is included below. In our notation, the variables C, L, E, R, and S represent some binary encoding of the corresponding symbols.

To encode a mesh, we pick an arbitrary border b, paint its two vertices (b.n.v.m:=painted; b.n.n.v.m:=painted;), and append them to the *order* stream.
Then we call the procedure visit(b) presented below. It uses a recursive call to follow the right corridor when an S triangle is encountered.

```
RECURSIVE PROCEDURE visit (e)
    REPEAT
            BEGIN
            b.t.m:=painted                          # mark the triangle as visited
            IF b.v.m==virgin      # test whether tip vertex was visited
              THEN BEGIN                            # case C
                 WRITE(vertices, b.v);              # append index V[e] to vertex order
                 WRITE(clers, C)                    # append encoding of C to clers string
                 b.v.m:=painted;                    # mark tip vertex as visited
                 b:=b.r;
                 END
```

```
    ELSE IF b.r.t.m==painted          # test whether right triangle was visited
        THENIF b.l.t.m==painted       # test whether left triangle was visited
            THEN BEGIN
               WRITE(clers, E);       # append encoding of E to clers string
               RETURN;                # exit (or return from recursive call)
               END
            ELSE BEGIN
               WRITE(clers, R);       # append encoding of R to clers string
               b:=b.r;                # move to left triangle
               END
        ELSE IF b.l.t.m == painted    # test whether left triangle was visited
            THEN BEGIN
               WRITE(clers, L);       # append encoding of L to clers string
               b:=b.l                 # move to right triangle
               END
            ELSE BEGIN
               Visit(b.r);            # recursive call to visit right branch first
               b:=b.l;                # move to left triangle
               END;
END;
```

The typical beginning of a *clers* stream starts with a series of C symbols as Edgebreaker turns around one of the vertices of the initial border. Then the algorithm spirals out as shown in Fig. 7. The stream initially contains mostly C and R symbols.

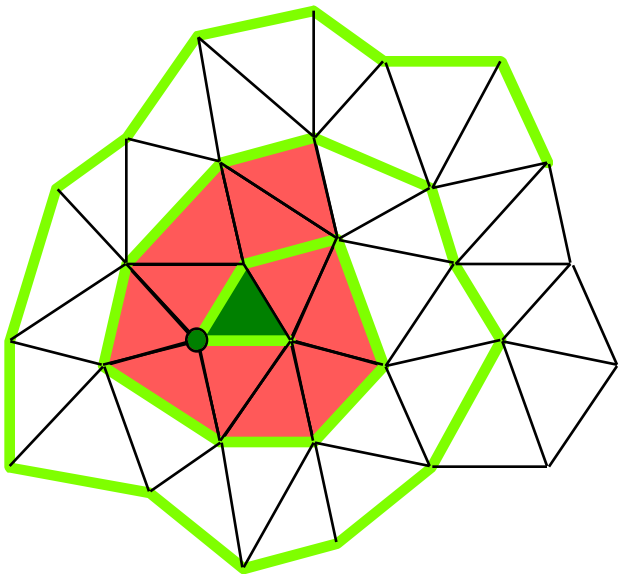

Fig 7: Typical starting Edgebreaker sequence, producing the *clers* stream CCCCCRCCRCRC

Fig. 8 shows a different spiral, which leads to an S triangle. The recursion will start another "visit", which will go to the E case, because both b.r.m and b.r.l are painted. Thus recursion will return after writing E. Edgebreaker will resume its traversal and proceed to the left neighbor of the S triangle.

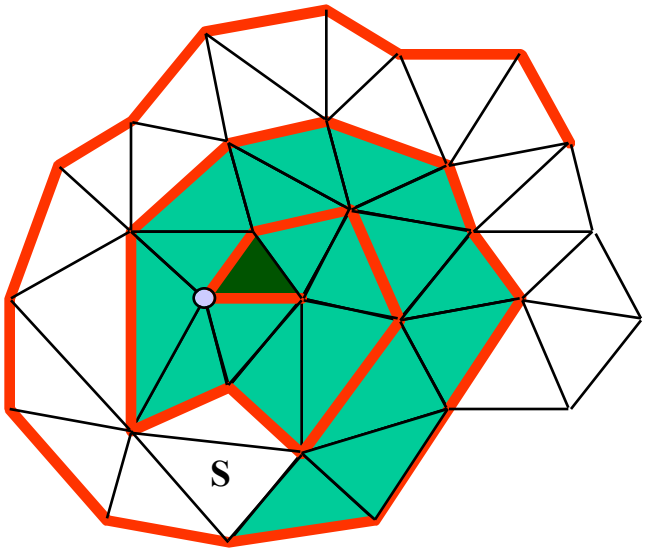

Fig 8: An S triangle early in the spiral.

A slightly more complex right branch of an S triangle is shown Fig 9. The encoding produces the *clers* stream: *CCCRCCCRCCCRCCCRRLCCCRCSLE.*

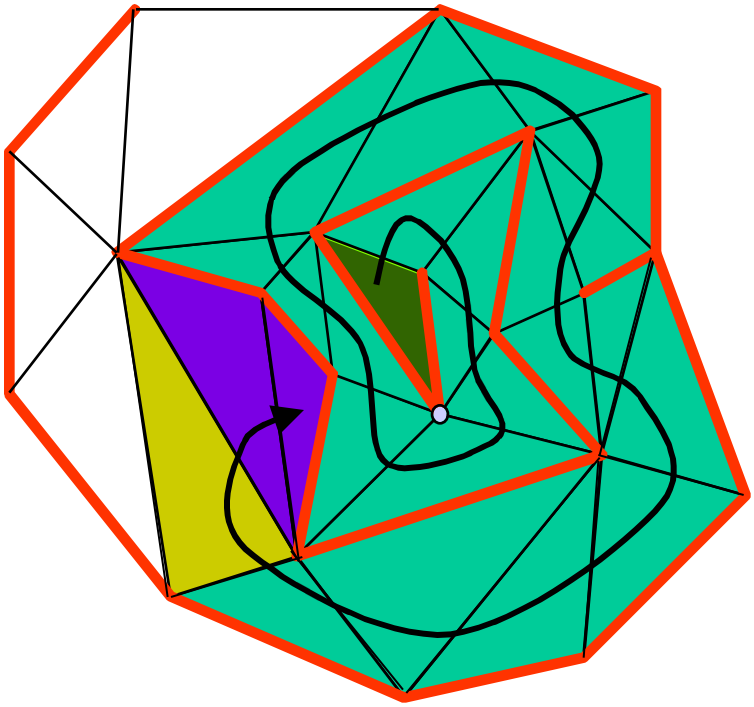

Fig 9: A more complex Edgebreaker beginning producing the *clers* stream
CCCRCCCRCCCRCCCRRLCCCRCSLE

Edgebreaker is always working to visit an edge-connected set of triangles, splitting it sometimes through an S operation in two parts and working on the one at a time. Fig. 10 illustrates a typical ending of the traversal of one such connected set. Triangles not yet visited are shown (left) in white. The upward arrow indicates where we are coming from, that is how we enter this set. The resulting triangle labels are shown (right). They are appended to *clers* in the following order: *CRSRLECRRRLE*. Notice that at the S triangle, the set is split and a recursive call visits the RLE triangles.

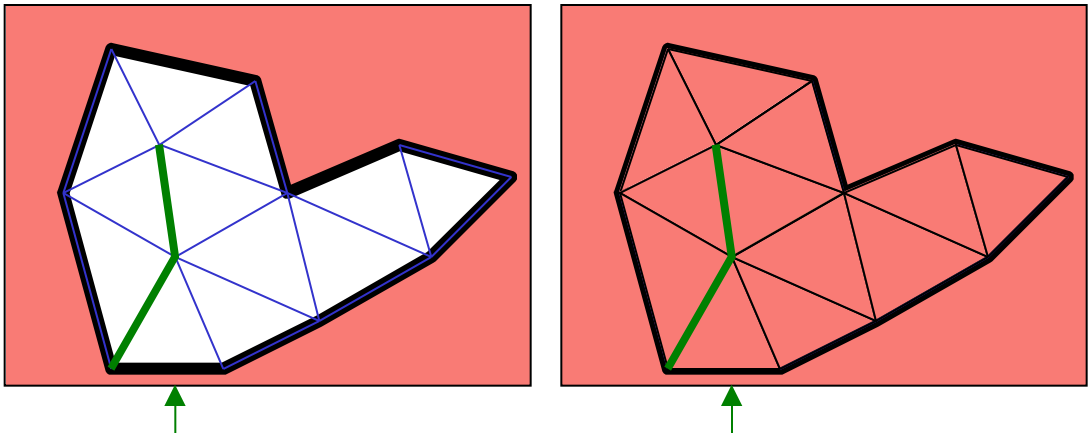

Fig 10: Typical ending Edgebreaker sequence, producing the *clers* stream CRSRLECRRRLE

I have marked the left borders of C triangles in thicker lines to indicate the corridors followed by EB. The S triangles corresponds to places where the corridors bifurcate (a Y junction for the traveler). The E triangles correspond to the endings of corridors. The other triangles define how a corridor is tiled.

Guaranteed 2T bit encoding of the CLERS stream

Except for the first two vertices, there is a one-to-one association between the vertices of the mesh and the triangles processed by C operations (remember that vertices are only painted when we encounter C triangles). Therefore, the number of Cs is V–2, which equals T/2, given that V=(T+4)/2. The total number of non-C operations, T–V+2, also equals T/2. Hence, if we use a 1-bit code for C and 3-bit codes for the other four operations, the total cost for storing the string with the above scheme would be exactly 2T bits.

Because the first two operations are Cs, they can be omitted from the string, yielding a total storage cost of 2T–2 bits for any STM.

This guarantee is important, because many compression techniques produce good compression ratios for very large meshes, but perform poorly on small meshes. So if you have a lot of small meshes to send, they are not much use.

Wrap&Zip decompression of the CLERS stream

The Wrap&Zip decompression algorithm, that Andrzej Szymczak and I have developed, receives a binary encoding of the *clers* string and reproduces a labeled planar triangle graph that is homeomorphic to the original graph and has its vertices labeled as discussed in the compression section. The process is very simple and has two phases: Wrapping and Zipping. (You may have already guessed this from the name.) They may be performed sequentially or simultaneously.

Furthermore, if inline decompression is desired, so that the receiver may build (and possibly render) the initial portions of the mesh before receiving the rest, the encoding of the vertices may be interleaved with the *clers* sequence.

The Wrap&Zip decompression works by first growing a triangle-tree. (They are very easy to grow in Georgia, where the climate is nice, i.e., hot and humid, through much of the summer.)

To initialize the process, we read the first symbol from the *clers* stream (it is a C of course), create one triangle, an number its vertices 0, 1, and 2. That was easy. Then we assign **orientations** to two of its borders: from 0 to 1 and from 1 to 2. (These orientations are discussed later and are called zippers.) We declare that the border bounded by vertex 0 and vertex 2 is **hot**. We will attach new triangles to hot borders and make new borders hot.

To execute the wrapping process, we keep reading symbols from the *clers* stream. For each symbol, we attach one triangle to the hot border and possibly make hot one or both free borders of this new triangle. Which border is hot is indicated by the exiting arrows of Fig. 11 for each symbol. An C and an L make the right border hot. An R triangle has its left border hot. An S triangle will first make its right border hot, maybe pursuing this right branch through a recursive call, and then when that branch is fully grown, it will make its left border hot and continue to grow that way. An E triangle has no hot border.

As we grow the triangles, you could partially fill in the O and V tables for an STM. I say paritally, because only the hot borders (and their coincident borders) have an opposite in the O table. (which is filled when we proceed to the next triangle and glue it to the hot border.) The other borders are free (i.e. we do not know what their opposite are).

Furthermore, only when we encounter C symbols, do we enter a b.v label for the border b that is coincident with the hot border. When I say that we enter a label, I mean that we store the next available integer at that corner. (We keep a vertex counter to know which integer to enter.)

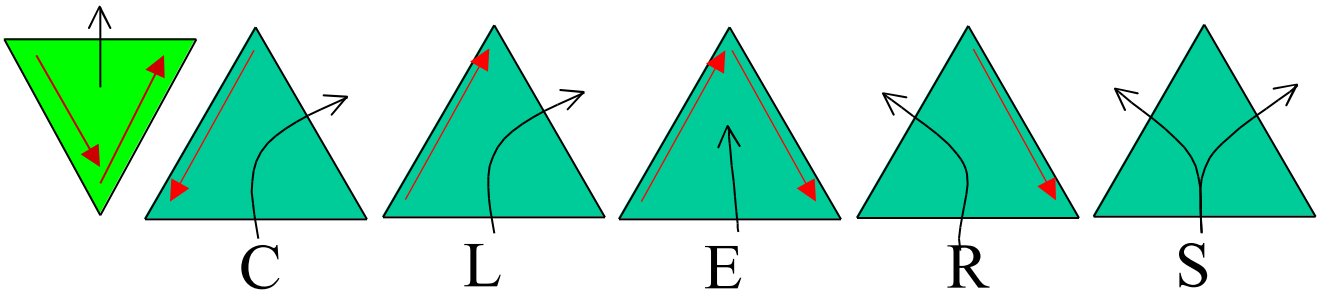

Fig 11: Free border orientation for Wrap&Zip. Initial triangle on the left.

Remember, each C triangle corresponds to a vertex. During compression, we have labeled these vertices in the order in which they were encountered by C operations. Now we are simply reassigning these labels.

Some of the vertices do not get labels. They are coincident with labeled vertices, but we don't know yet which ones.

Once the tree is grown, we have a triangulation of a simply connected polygon that has a single component and no holes. It has a boundary (so it is not an STM). Its boundary is a set of external edges. Note that all the vertices are connected to external edges. There are no interior vertices. One such flattened triangle-tree (which happens to be a single corridor), is shown Fig. 12 (left). It cam from the encoding of a dodecahedron (left). The arrows are shown on both the unfolded tree and on the initial triangles of the dodecahedron. During compression, Edgebreaker started visiting the dodecahedron from the central front triangle and went down to a C triangle and then turned right (i.e. to our left). The cracks on the dodecahedron indicate edges that were either on the initial triangle, or that were the left edges of a C triangle.

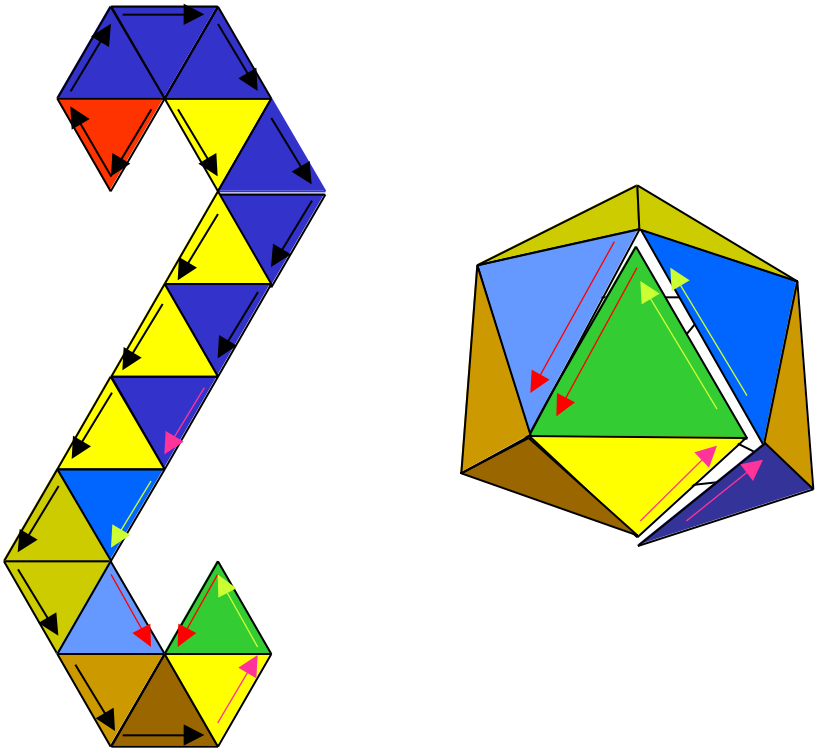

Fig 12: Zipping up the triangle tree.

Well… that was easy. We made a nice triangle tree. But this is not the STC that we were supposed to get.

Patience. We'll zip it up (and get back the dodecahedron on the right of Fig 12). However, before we do so, we need to install the zipper. It comes in little pieces. Each piece corresponds to an exterior edge of the triangle tree.

Each zipper piece is oriented as shown Fig. 11. Notice that all zippers are oriented clockwise, except for free borders of C triangles and except for the initial triangle. Of course, we don't need to wait until the whole tree is grown to install the oriented zippers. Each time we attach a triangle to a hot border, we know which of its borders will be attached to other triangles and which of them will be zippers. So we can install them right away. That means: orient the free borders properly. Let us allocate a bit to store that information.

Now all we have to do is zip. Anytime we find two external edges that are incident upon the same vertex and oriented to point away from it, we zip. (It is like attaching the two bottom parts of a zipper before zipping it up.) When I say zip, I mean that we make the two borders coincident. They already had one common vertex. Now we make their other vertices coincident. How? We copy the label of the corner that has one in the corner that does not. Andrzej Szymczak and I have proven that during the zipping operation, one of these two vertices has a label and the other one does not.

The process is illustrated Fig. 13. The previously reconstructed part of the triangle tree is darker. Its external edges are oriented and ready to be zipped (a). Then we build the triangle tree for the *clers* sequence CRSRLECRRRL and install zippers. When we are done installing the zipper for the last L triangle, we are in a situation where we can zip, but just one pair of borders (b) that bound triangles L and C. The corner at the end of the border of the C triangle had a label. The corner at the end of the L triangle did not. After the zipping, both point to the same vertex. The result of that zip and of the growth of the final E triangle is shown (c) where both corners point to the same vertex and thus are now shown as coincident. Then we can zip up the rest (d), staring from the borders of triangles E and C. Note that as we walk along the crease of the zip, the vertices on the left have labels, but the vertices on the right do not.

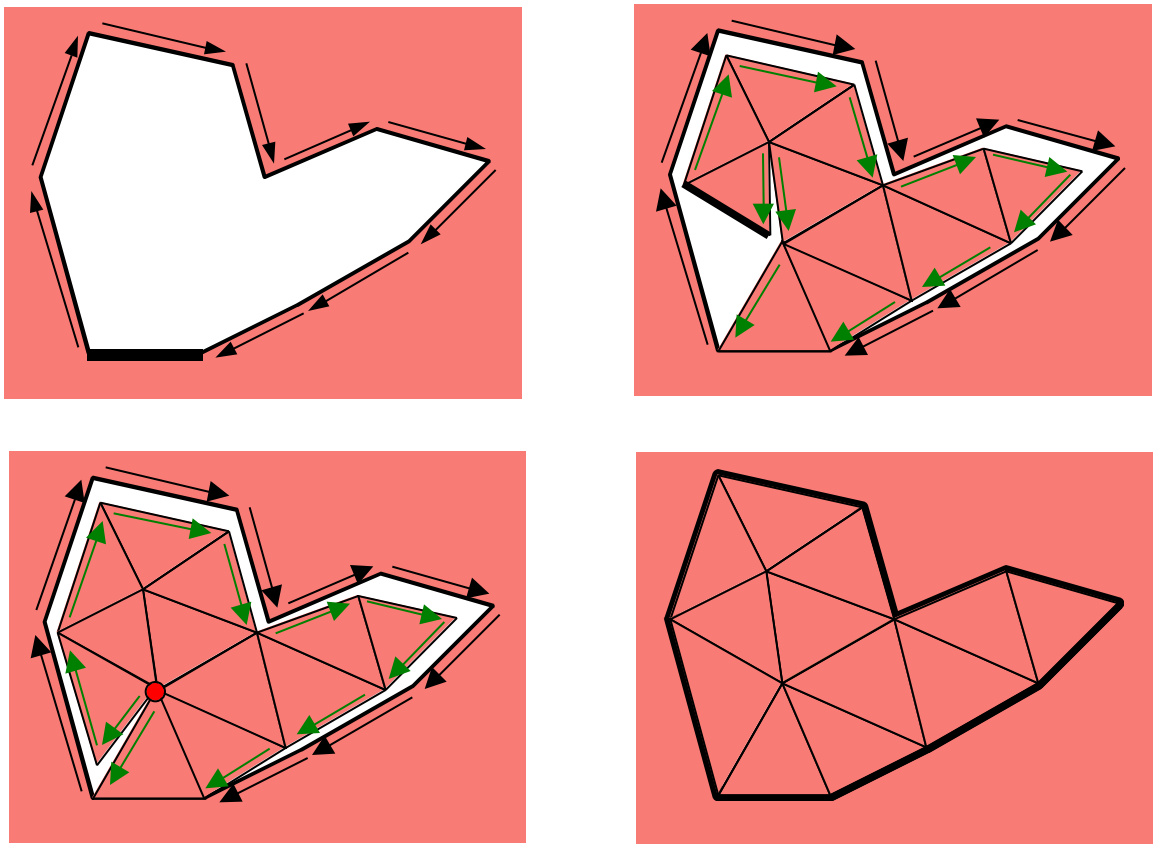

Fig 13: Zipping up the triangle tree.

The creases are the walls of the corridors. They have been produced as the left borders of C triangles. That is why their vertices are labeled.

So, should we keep checking for zipping opportunities everywhere?

No. We know exactly when to check. Note that no zipping may be initiated by the creation of a C triangle, because the matching edge has not yet been created. Also, note that no zipping may be initiated by the creation of an R triangle, because its arrow is pointing towards a vertex that bounds the gate (which is not oriented). Finally, S triangles have no arrows, since both their free edges will be used as gates. So we only need to watch out for zip opportunities with L and E triangles.

I suggest to use, during the tree-growing process, a doubly linked circular list. Each entry would point to an exterior edge and would also store the orientation of the zipper that is attached to it. In fact, the entry would point to the border that corresponds to that edge. One entry in that doubly linked list would be the hot border. Other hot borders may be piled up on the stack. As we grow the tree, we insert one new entry in the doubly linked list. Either just before, or just after the hot border.

When we grow E or L triangles, we use the list to check whether the new external edge that we have just added is adjacent to another external edge that also points away from their common vertex. (We use the doubly linked list to locate that neighboring edge.) If so, we zip and remove the two entries from the doubly linked list.

The decompression algorithm has linear time complexity. To prove this, notice that we start the recursive zipping procedure at most T times: once for each L and E operation. Consequently, we stop the zipping procedure the same number of times. (The zipping procedure only goes up the creases and does not bifurcate.) Therefore, the number of times we test a vertex and decide not to zip it is bounded by T. The number of successful zip operations equals the number of free borders divided by two, which is precisely V–1.
Both the Edgebreaker compression and the Wrap&Zip decompression algorithms have been tested on a variety of meshes. For example, running on a single processor SGI Power Challenge, compressing the bunny model with 69,674 triangles took 3.87 seconds and decompression took 0.38 seconds. This decompression rate of 184K triangles per second was achieved without any attempt to optimize performance.

Topological extensions

Holes

I will now discuss extensions of these compression/decompression techniques to more general triangle meshes, which may have handles, holes, and non-manifold singularities.

To clarify the distinction between ***handles***, ***holes***, and ***cavities***, consider that a torus has one ***handle*** (also, called through-hole), while a solid wooden ball with an empty small core has an enclosed ***cavity***, but no ***hole*** or ***handle*** (using our terminology). A spherical surface, from which one has cut out a disk, is no longer a closed boundary that separates space into disjoint components. We say that such a surface has a ***hole*** and is thus a two-manifold with boundary, having for boundary a single one-manifold curve. Of course, a surface may have multiple holes.

Let us address the hole-problem first. A triangle mesh with a triangular hole is illustrated in Fig. 17.

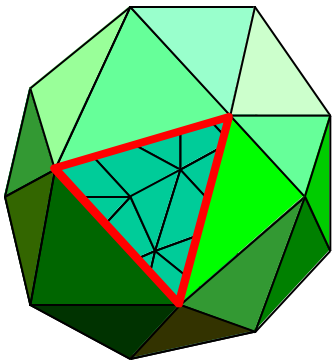

Fig 17: A triangle mesh with a hole

Although we are using a different technique for encoding such meshes with Edgebreaker, for sake of simplicity I will explain a very elegant approach that was first published by Touma and Gotsman. Consider the polygonal hole of Fig. 18 (left). Create a dummy vertex (for which you need not specify coordinates). Then use it as the tip of a fan of triangles whose border matches the boundary of the hole (right). We have filled the hole. Now it suffices to remember the id of that dummy vertex.

If we fill all holes this way, we can encode a mesh with holes using Edgebreaker and transmit the ids of the dummy vertices before the vertex stream so that the decoder can skip the dummy vertices. The decoder will first reconstruct the connectivity of the mesh with holes plugged in by fans. Then it will remove all triangles incident upon the dummy vertices. Finally, it will decode the vertices.

We have opted for a different solution, because the cost of encoding the ids of the dummy vertices and the unnecessary triangles can be somewhat reduced by using a different approach. Nevertheless, the approach outlined above is so simple that it may be preferred . From now on, we assume that our mesh has no holes.

A bit of caution is appropriate here. In pathological configurations where the union of all borders of unmatched triangles forms a non-manifold curve, the approach above will not work. Basically you do not know how to build the fans. In fact, there are configurations for which no fans can be built that would produce the boundary of a solid.

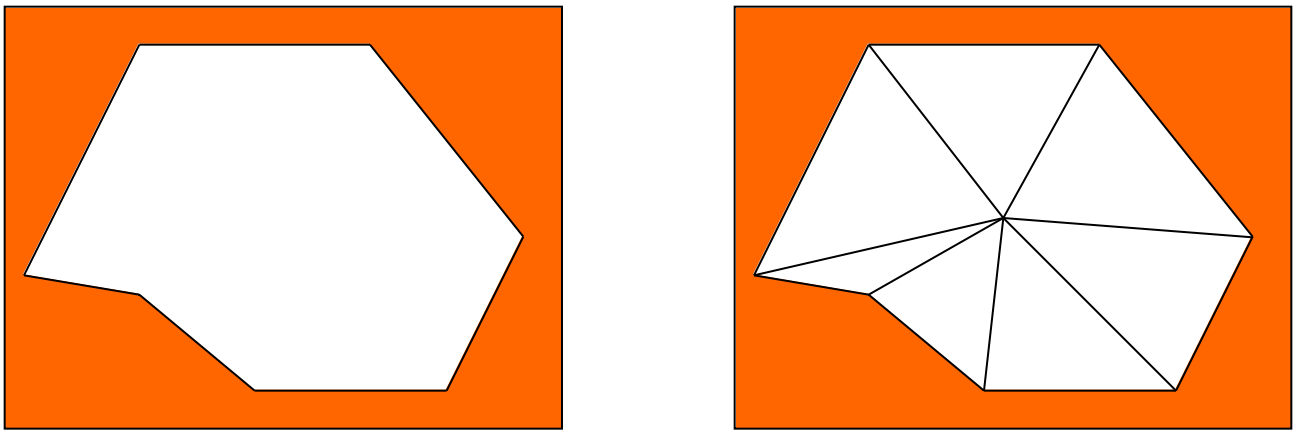

Fig 18: Filling the hole with a dummy vertex.

Non-manifold solids

We will consider here only triangle meshes where with edges that correspond to an even number of coincident borders. (We assume that all holes have been successfully removed.) Furthermore, we assume here that the triangle mesh is a valid boundary of a solid polyhedron that has a well defined interior and exterior. We can say that such a boundary forms a waterproof surface. If the mesh has several connected components, we treat them one at a time.

Such meshes may exhibit two types of non-manifold situations. **Non-manifold edges** and **non-manifold vertices.**

Non-manifold edges are edges where more than two borders meet. They have 2K incident triangles, where k>1. Such a model is shown Fig. 14 (left).

Fig 14: Non-manifold solid with a non-manifold edge (left) and vertex (right).

Although all vertices bounding non-manifold edges are themselves non-manifold in some sense, we will restrict the term of non-manifold vertex to non-manifold vertices (as defined earlier) whose incident edges are manifold. Such a vertex is shown Fig. 14 (right).

The approach that David Cardoze and I have chosen is in our Matchmaker technique is to split the borders of each non-manifold edge into pairs. We have decided to impose a constraints on this match-making.

To explain the constraint, consider that each pair of matched borders is an edge now. Consider also that we are allowed to bend the edges. The constraint is that there is a way of bending the edges so that the resulting shape would be a valid solid with all its edges manifold and no faces intersecting other faces. We call that an edge-manifold solid. Notice that an edge-manifold solid may still exhibit non-manifold vertices.

We say that the original solid is the limit of a family of edge-manifold solids obtained by a continuous straightening of the edges. Such solids preserve certain properties, which are important for the validity of certain algorithms. For example, we know that adjacent triangles of an edge-manifold solid have compatible orientations. Furthermore, we know that if we put a bug (a really tine one) on the exterior face of a triangle and let it crawl through the non-manifold edge to the exterior face of the adjacent triangle, as defined by our matching, the bug will stay outside of the solid and will be able to breathe. Less careful matching strategies may suffocate the adventurous bug.

Our objective is not only to satisfy the above constraint, but also to reduce the number of non-manifold vertices in the resulting edge-manifold solid. For example, the matching of Fig 15 removes two non-manifold edges, but leaves one non-manifold vertex (left). We then have to duplicate that vertex (right) to obtain a manifold mesh that can be encoded by our Edgebreaker algorithm. Remember that Edgebreaker likes to paint vertices. It would get confused if we tried it on a polyhedron with non-manifold vertices. We were successful in the solid of Fig. 15 to pair match the borders coincident with the non-manifold edge in the back so that the two vertices that bound the non-manifold edge become manifold. In this case we made a passage inside the solid through the gap between the (imaginary) bends of the two manifold edges that resulted from the matching. In most cases, we are able to match coincident borders without introducing non-manifold vertices.

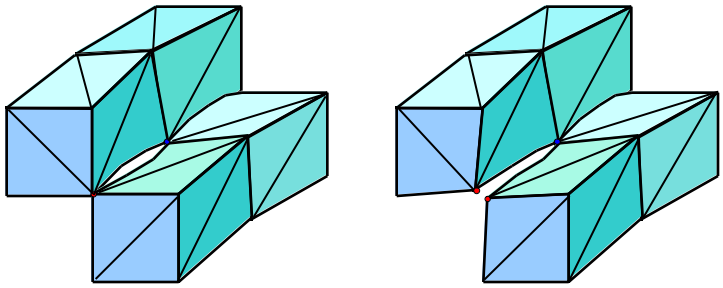

Fig 15: A non-manifold solid.

We picked this example, however, to illustrate that his is not always possible. In order to keep the central vertex manifold, we need to match the four borders of the front non-manifold edge in a different manner, which results in a non-manifold vertex. We need therefore to duplicate that vertex in our representation (right), and we hate having to do that.

How do we do the matching? You may ask. Our approach is based on the property shown Fig. 16, that we have discovered and proven. Each connected component of the boundary of a 2D region may be represented by a single closed curve that may intersect itself at its non-manifold vertices but does not cross itself. Again, the fact that it does not cross itself may be formulated more precisely by stating that the non-manifold curve is the limit of a series of manifold curves. To do the matching of the edges at the non-manifold vertices of the 2D case, we perform two traversal of the edges of each connected component of the boundary. First, we form natural circuits by always taking the rightmost edge that has not yet been visited (left). The labels of each circuit are painted in a different color. Then we visit the edges again. But this time, if we reach a non-manifold vertex with edges of a new color (that we have not yet seen), we take the leftmost of these. Otherwise, we continue our boring round taking the rightmost edge of the current color. Notice that, in the example of Fig. 16 (right), each connected component of the boundary of the polygon is represented by a single closed curve that has the desired property.

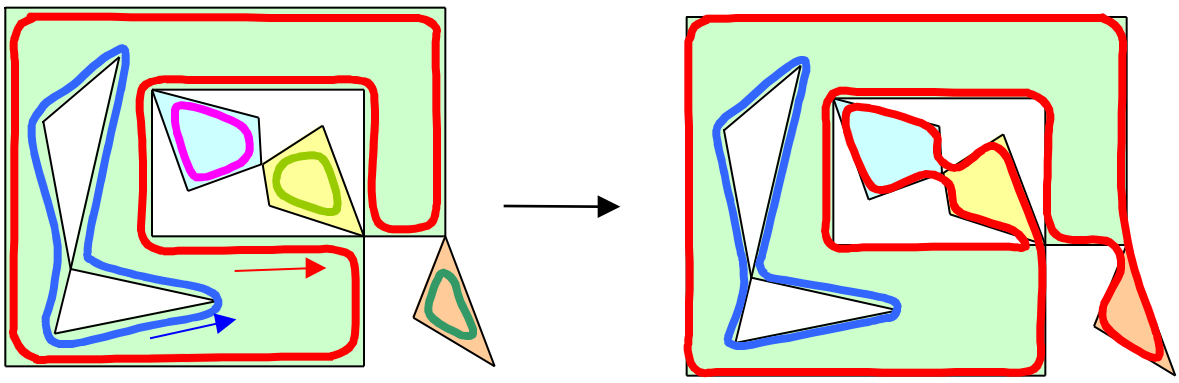

Fig 16: A non-manifold solid.

You may wonder what all this has to do with our problem of matching borders of non-manifold edges. Well, if you were to take all the triangles incident upon a vertex that is bounding one or more non-manifold edges and look at the

external edges (boundary) of the union of these triangles, you would get a polygonal (3D) curve. Each edge of that curve corresponds to a triangle. Non-manifold edges incident upon our vertex are mapped by this process into non-manifold vertices of that curve. Although the curve is 3D, we can inherit from the triangles the notion of leftmost and rightmost. Therefore, we can execute our little walk strategy. The resulting matching of the pairs of edges of that 3D curve that are incident upon a non-manifold vertex define how we match the triangles.

Handles

For clarity, we first describe how Wrap&Zip uses an extended CLERS code to reconstruct the connectivity of meshes with handles. Then, we suggest a format for storing this extended code. Finally, we explain how the extended code is generated by a modified version of Edgebreaker's compression algorithm.

The extended decompression algorithm reads the sequence of op-codes in the CLERS string and builds the triangle tree. However, it now must handle 6 types of triangles: the five C, L, E, R, and S types described above, plus the new S* type.

As it is for S triangles, the triangle-tree is grown from the right edges of the S* triangles, but not from their left edges, which temporarily become bounding edges. The extended CLERS string associates, with each S* triangle, an integer identifying a matching bounding edge that is glued to the left edge of the S triangle prior to zipping.

Fig. 19 shows a typical process of encoding a handle. Before it is processed by the compression algorithm, a handle (left) has a simply connected bounding loop. A fist S* triangle breaks the mesh into a topological polygon with one hole (second image from the right). A second S* triangle unifies the outer loops with the hole-bounding loop (bottom left). With each S* triangle is associated a reference to the corresponding edge (bottom-right).

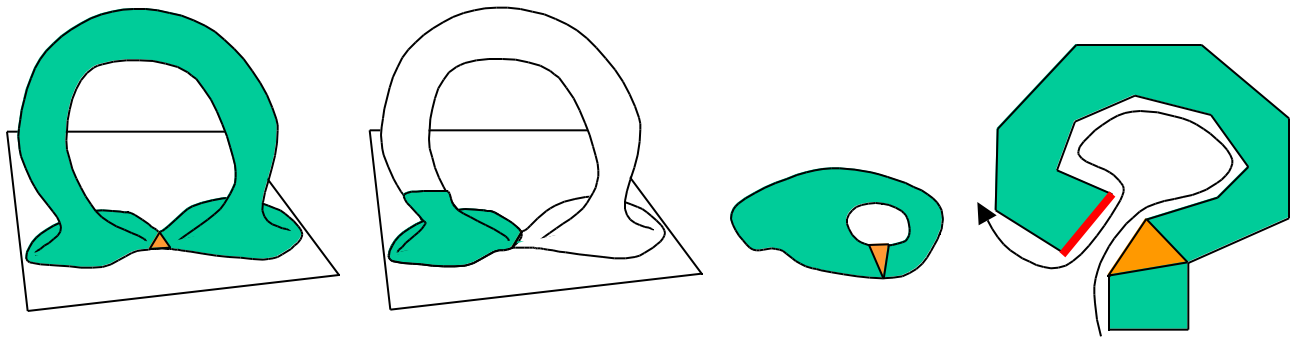

Fig 19: Discovering handles when returning to an S triangle.

Wrap&Zip decodes the CLERS sequence as described previously, but marking the left edges of S*-triangles as glue-edges. It builds the triangulated polygon and performs zipping operations whenever possible.

Once the entire triangle-tree topological polygon is built, Wrap&Zip traverses its boundary and builds an array of edge-pointers indexed by the integer edge ID incremented as we visit the consecutive edges along the bounding loop. This array will speed up the identification of the matching edges for the glue-edges of each S* triangle.

Then, Wrap&Zip traverses the triangle-tree again and glues the left edge of each S* triangle with the edge identified by an integer associated with the triangle. (That integer is computed during compression and stored in a separate table. It is used as an index into the array of edge-pointers.) Each glue operation stitches the mesh, merging two edges into one and merging their four vertices into two. Note that for orientable surfaces, there is no ambiguity as to the relative orientation of the two edges being glued.

Also, note that changing the order of the gluing operations does not affect the topology of the result because the matching edges are identified independently of each other.

Furthermore, note that each handle in the original mesh produces two S* triangles. The gluing operation associated with one of them will split the boundary of P into two disjoint loops. The second one will merge these two loops back into a single loop because the left edge of its S* triangle and the edge that it should be glued to are in separate loops (see Fig. 19). Consequently, executing the $2h$ gluing operations, where h is the number of handles, restores a single bounding loop.

The topological model that results from the gluing operations represents a shape, which no longer is a simple polygon. Instead, it is topologically equivalent to a surface obtained by cutting a small disk out of the original surface. The boundary of that disk is a single loop of edges. Each edge in the loop coincides physically with another edge of the loop. Wrap&Zip does not need to identify this correspondence through global gluing operations nor through geometric coincidence tests. Instead, it applies the "zipping" process described earlier and restores the

original manifold or pseudo-manifold surface with handles by a series of local zipping operations that glue pairs of adjacent edges that point towards their common vertex (using the orientation of the free edges derived from the op-code associated with each triangle).

The generalization of the CLERS format must contain the information necessary to identify the S* operations and the integer edge-identifier associated with each S* triangle. One could add one bit to the Edgebreaker code used for the S triangles in order to distinguish S triangles from S* triangles. When the number of handles, h, is much smaller than the number of S triangles, it is more compact to use the same code for S and S* operations and to distinguish them by storing a table of integer counts. Each count indicates how many of these S-or-S* triangles that precede the current S* triangle were actually of type S. This count could be the total number of S triangles preceding the current S* or just their count from the previous S* (or from the beginning for the first S* in the CLERS code). With each count in the table we also associate the edge identifier.

There are T+2 free edges bounding P. Therefore we need $2h\log_2(T+2)$ bits to store all the edge identifiers. If the table is decoded after the CLERS string, we know s, the total count of S or S* triangles, and need only $2h\log_2(s)$ bits to identify the $2h$ triangles of type S*. In practice, s is about T/20 or less. The number of handles varies from model to model, but is typically mush smaller then s.

Let us now describe how Edgebreaker's compression algorithm may be adapted to produce the table for handles when generating the modified CLERS code for supporting meshes with handles.

Compression proceeds as before labeling the encountered triangles as C, L, E, R, or S. Some of the S triangles are temporarily mislabeled during this process and will be turned later into S* triangles.

When compression reaches an S triangle, it starts a recursion to follow the right corridor. At the end of this corridor (and all its children) it returns from recursion. Before moving left, while still on the border of the S triangle, we test whether the left neighbor is still virgin. If so, we are really on an S triangle and a new (left) branch of the triangle-tree will be constructed. If however, the opposite triangle is marked as already visited, then the S triangle is relabeled as a triangle of type S*.

Once all the triangles are labeled, we traverse the loop of bounding (free) borders and number them using increasing integers. The ID is initialized to zero and we start the process at the first free border of the first triangle of the triangle-tree. Each time we encounter a border that has an opposite triangle of type S*, we store with that triangle the ID of the border.

Finally, we traverse the border a second time exploiting the triangle labels to avoid the initial tests that define the traversal. We keep track of the counts of S and S* triangles; store the triangle labels in the CLERS string (replacing each S* with an S); and produce the appropriate entries into our table, which identifies the S* triangles and the associated glue-edges.

Geometry compression

Most vertex-compression techniques are based on a vertex estimation. We start by **normalizing** the representation of vertex coordinates. We simply build a tight enclosing box around the model, chose a unit along each direction of the box, and express the vertex coordinates as integers. If we chose a larger unit, we will have more error, but smaller integers. We select the unit and the origin so that all coordinates are integers in the interval $[0,2^k-1]$. For most applications, sufficient accuracy may be provided with k between 6 and 14.

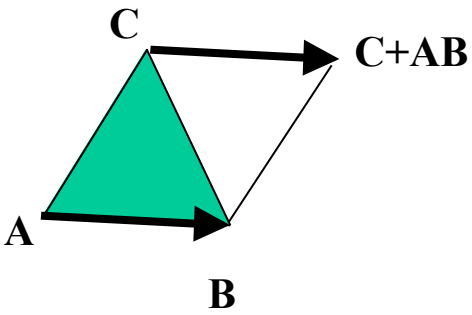

Fig 20: Parallelogram used for predicting a vertex.

Then we predict the vertices using the location of previously decoded vertices and the connectivity information, which tells us who our neighbors are. Touma and Gotsman have proposed a very simple predictor. If the last triangle was bounded by vertices A, B, and C, with the hot border being BC, then they use C+AB prediction for the

next vertex (see Fig. 20). To give them full credit, they also use a bend for the parallelogram, computed using model dependent statistics.

Compression uses the predictors and encodes only the difference between the predicted and the correct vertex location. Decompression predicts the vertex using the same technique, decodes the difference vector, and adds it to the predicted location to obtain the correct vertex location.

If the predictions are good, the corrective vectors are short and their coordinates are small integers. Because the distribution of these integers is not uniform, variable length entropy or arithmetic coding technique are used to compress the corrective vectors. The amount of compression depends of course on the smoothness of the surface, its level of tessellation, and the quantization factor k. For k=10 or 11, it is not uncommon to compress the location data to about 12 to 14 bits per vertex.

In opposition to the connectivity compression, which was lossless, geometry compression is lossy,because it involves the quatization step (conversion of vertex coordinate to fixed length integers).

Improvements to the encoding of *clers* streams

We first present a few improvements to the encoding of the *clers* stream produced by Edgebreaker.

Expected 1.7T bit Edgebreaker code never exceeding 2T bits

CL and CE combinations in the *clers* stram are impossible. Why? Because they would force the two triangles to have two coincident borders, and thus three coincident corners. And we have excluded this possibility for STMs.

To take advantage of this constraint, we can use a shorter code for S and R symbols that follow a C, thus we have two modes: “after a C” and “not after a C”.

In the “after a C” mode, there are only 3 possible symbols: C (coded as 0), R (coded as 11) and S (coded 10). In the “not after a C” mode, we use the following codes for the five possible symbols: 0 for C, 110 for L, 101 for R, 100 for E, and 111 for S. Experimental results with this code (which show a ratio of 36% R operations, almost half of which follow a C) consistently result in a storage cost of 1.7T bits.

This code will never exceed 2T bits.

Guaranteed 1.84t bit code

A slightly more complex code, that Davis King and I have proposed, is guaranteed to store any valid CLERS sequence in less than 1.84T bits. It encodes C as 0, S as 10 and R as 11, when they do not follow a C. Symbols that follow a C are encoded using one of three possible codes.

Code I: C is 0, S is 100, R is 101, L is 110, E is 111.

Code II: C is 00, S is 111, R is 10, L is 110, E is 01.

Code III: C is 00, S is 010, R is 011, L is 10 and E is 11.

We have proven that, for any given simple mesh, one of these three codes is guaranteed to yield a total storage of less than (1+5/6)T bits. We simply produce all three in parallel and store the shortest, preceded by a 2-bit identifier that will tell the decoder which code was used for the particular mesh.

Expected average code between 1.3T and 1.6T bit

By exploiting the relative frequencies of the various operations in large meshes, Andrzej Szymczak and I have devised a slightly different code that works better in practice, but no longer guarantees never to exceed 2T bits.

We encode CC, CS, and CR pairs as single symbols. We break the CLERS sequence into symbols of one or two letters each. Each symbol is one of the seven *words* listed in the table below. For each word, we suggest a binary *code* and indicate the *number of letters* that would be part of such words in a 100 letters sub-string generated for a typical model (we used the 69,674 triangle Stanford Bunny). The right column indicates the bit-cost associated with the occurrences of the word in a 100 letter sub-string.

Word	code	# of letters	Cost
CR (after even Cs)	01	53.6	53.6 bits
CC (after even Cs)	00	22.4	22.4 bits
CS (after even Cs)	1101	1.2	2.4 bits
R (after even Cs)	10	19.6	39.2 bits
E	1100	1.6	6.4 bits
S (after even Cs)	1111	0.9	3.6 bits
L	1110	0.3	1.2 bits
TOTAL		100	128.8 bits

We have explored a variety of models. The compressed file size varies between 1.3T bits (for a typical models such as the Stanford Bunny) and 1.6T bits (for 2D Delaunay triangulations).

Custom entropy codes between 0.91T to 1.26T bit

For large meshes, Andrzej Szymczak and I have devised a technique that constructs a Huffman code as follows. We introduce a space after each non-C symbol that is followed by a C. The resulting words start with one or more C symbols, which are followed by one or more non-C symbols.

Experimenting with Delaunay triangulations of 200,000 triangles, we found only 1,400 words. A Huffman encoding of these words yields 1.26T bits. The table of codes takes about 32,000 bits (0.16T bits for meshes with 200,00 triangles), but a part of it corresponding to most frequent words can be preloaded and kept constant for all large meshes.

For more realistic (i.e., smaller) models (such as the 69,674 triangle Stanford Bunny) the dictionary had only 173 words and the cost of Huffman encoding is 0.*85*T bits. The total cost, including the dictionary is 0.91T bits.

We use the 69,674 triangle Stanford Bunny to compare the performance of the Edgebreaker compression with respect to other general purpose compression techniques.

We have found that, Edgebreaker provides roughly a 50-to-1 compression ratio for gzipped files representing the uncompressed connectivity of triangle meshes, which is usually the main storage factor for uncompressed representations. Further attempts to gzip the best CLERS format result in a less than 2% improvement.

Alternative compression techniques

Deering's vertex buffer

Michael Deering's approach is a compromise between a standard triangle strip and a general scheme for referencing any previously decoded vertex. Deering uses a 16 registers cache to store temporarily 16 of the previously decoded vertices for subsequent uses. He suggests to use one bit per vertex to indicate whether each newly decoded vertex should be saved in the cache. Two bits per triangle are used to indicate how to form the next triangle. One bit per triangle indicates whether the next vertex should be read from the input stream or retrieved from the cache. 4 bits of address are used for randomly selecting a vertex from the stack-buffer, each time an old vertex is reused. Assuming at best that a ratio of 14/16 vertices are reused from the cache and that 2/16 vertices must be reused while they are not in the cache, the total cost of Deering's approach, when combined with a random access to previously decoded vertices would be 4.5+0.125log(V). Note that it may prove difficult to build such generalized strips while maintaining a low count of vertex replication.

Hoppe's vertex insertion

Hugues Hoppe's Progressive Meshes permit to transfer a 3D mesh progressively, starting from a coarse mesh and then inserting new vertices one by one. Hoppe applies a vertex insertion that is the inverse of the edge collapse operation popular in mesh simplification techniques (as discussed in the next section). This insertion process is illustrated Fig 21. Two adjacent edges are marked with a thick line (left). They define a slipt and their common vertex is replicated (center). Two new triangles are inserted (right).

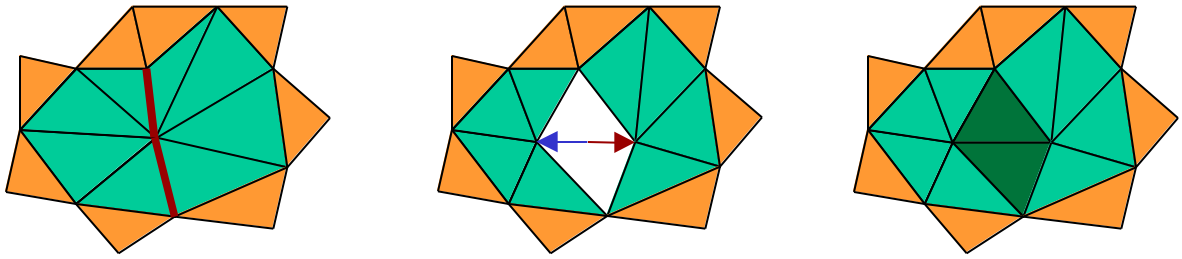

Fig 21: Vertex insertion (the inverse of an edge collapse).

A vertex insertion identifies a vertex **v** and two of its incident edges. It cuts the mesh open at these edges and fills the hole with two triangles. **v** is thus split into two vertices.

Each vertex is transferred only once in Hoppe's scheme. The cost of G for each vertex is the identification of one of the previously transferred vertices (on average more than 0.5log(V)) plus the cost of identifying two of the incident

edges (5 bits are sufficient if no vertex is bounding more than 32 edges). Thus, the cost per triangle would be more than 2.5+0.25log(V).

Turan

Turan has shown that the structure of a labeled planar graph may be encoded using slightly less than 6T bits. Having a constant number of bits per triangle has a significant advantage over the previous approaches, which all include a log(V) factor, especially for highly complex meshes.

Turan builds a vertex spanning tree and uses it to represent the boundary of a topological polygon of 2V-2 edges. The structure of this tree is encoded using 4V−4 bits. There are at most 2V-5 edges that do not belong to the vertex spanning tree. These may be encoded using 4 bits each. The overall cost is thus, 12V-24 bits.

Taubin and Rossignac

The Topological Surgery method developed by Gabriel Taubin and me also builds a vertex spanning tree of T that splits the surface of the mesh into a binary tree of corridors (generalized triangle strips). They encode both trees using a run length code, which for highly complex meshes yields an average of less than two bits per triangle. In addition, they use one bit per triangle to indicate whether the next triangle in a corridor is attached to the left or the right edge of the previous one. The compactness of the encoding of both trees comes from the fact that, by construction, both trees tend to have very few nodes with more than one child. Sequences of consecutive nodes with a single child are grouped into runs and encoded by simply storing their length, using a number of bits that is the ceiling of the log of the length of the longest run.

For pathological cases, with a non-negligible proportion of multi-child nodes, their approach does no longer guarantee a constant number of bits per triangle.

The vertices are stored in the depth-first traversal order of the vertex spanning tree. The entire mesh is represented by the list of vertex coordinates, an encoding of the sparse vertex and corridor trees, and the string of left/right bits. This technique has been applied to the compression of VRML files and has been integrated in the MPEG-4 standard.

Denny and Sohler Vertex permutation

Denny and Sohler have proposed a technique for encoding T for planar triangulations of sufficiently large size as a permutation of the vertices in V. They show that there are less than $2^{8.2|V| + O(\log|V|)}$ valid triangulations of a planar set of *v* points, and that for sufficiently large |V|, each triangulation may be associated with a different permutations of these points (there are approximately $2^{|V| \ln(|V|)}$ such permutations). Their approach requires transmitting a triangle that contains the set and the vertices of V in a suitable order, computed by the compression algorithm. The decoding process sorts V lexicographically and then sweeps over the crude triangulation left to right. At each vertex of V, the enclosing triangle is identified and the vertex is inserted according to the incidence relation derived from the permutation. The vertices of V are transmitted progressively in batches. The successive batches are constructed through repetitive plane-sweeps, during which all vertices with degree at most 6 are removed incrementally and the resulting holes re-triangulated. For each point, the information needed to reconstruct that triangulation is encoded in the permutation of the vertices of the batch. The batches are decompressed in inverse order. Although for sufficiently complex models the cost of storing G is null, the unstructured order in which the vertices are received and the absence of the incidence graph during their decompression makes it difficult to use predictive techniques for vertex encoding.

Summary of storage requirements

The table below compares the storage cost of the previously described techniques for a simple mesh with a negligible number of bounding vertices. The cost is expressed as the number of bits per triangle.

Vertex table	$3\log(\|V_I\|)$	
Independent triangles	$2.5\log(\|V_I\|)+0.5$	
Bar-Yehuda	$1.25\log(\|V_I\|)+9.25$	Require complex algorithms to compute sequence.
Triangle strips	$0.5\log(\|V_I\|)+1$	
Deering	$0.125\log(\|V_I\|)+4.5$	Assuming that we can build general strips with minimal vertex repetition.
Hoppe	$0.25\log(\|V_I\|)+2.5$	Assuming a constant bound on the number of edges per vertex
Turan	6	Works for general planar graphs
Taubin, Rossignac	1.5 - 3.5	Only when trees have long runs.
Denny and Sohler	0	Only for sufficiently complex 2D triangulations. Permutes vertices.

Mesh simplification

We focus in this section on 3D model simplification, a preprocessing step that generates a series of 3D models (sometimes called "impostors" or "impersonators"), which trade resemblance to the original model for fidelity.

Simplification techniques are a form of lossy compression. The compute a different model, which as fewer triangles and vertices, but resembles the original model. Of course, as the number of triangle is reduced, the fidelity of the corresponding approximation drops. The various approaches to simplification make different tradeoffs between speed, quality, and complexity of the implementation.

Rossignac and Borrel's vertex quantization

The vertex clustering technique that Paul Borrel and I have invented, is the simplest to implement and most efficient simplification approach. It groups vertices into clusters by coordinate quantization (round-off), computes a representative vertex for each cluster, and removes degenerate triangles which have at least two of their vertices in the same cluster. All vertices whose coordinates round-off to the same value are merged. The accuracy of the simplification is thus controlled by the quantization parameters (see Fig 22). The vertices of the original triangular mesh (far left) are quantized, which amounts to associating each vertex with a single cell of a regular subdivision of a box then encloses the object. Cells which contain one or more vertices are marked with a circle (center left). All the vertices that lie in a given cell form a cluster. A representative vertex is chosen for each cluster. It is indicated with a filled dot. The other vertices of each cluster are collapsed into one and placed at the representative vertex for the cluster (far right). Triangles having more than one vertex in a cluster collapse during this simplification process (shaded triangles center right). They will be removed to accelerate the rendering of approximated versions of the object while other neighboring triangles may expand. The resulting model (far right) has fewer vertices and triangles, but has a different topology. Notice that a thin area collapsed into a single line, while a gap was bridged by a different line segment.

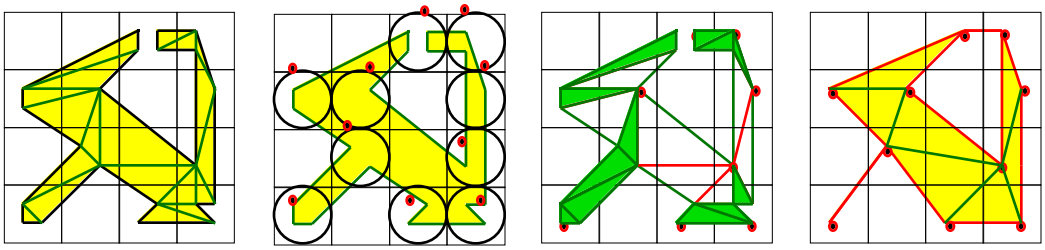

Fig 22: Vertex clustering.

It is aimed at very complex and fairly irregular CAD models of mechanical parts. It operates on boundary representations of an arbitrary polyhedron and generates a series of simplified models with a decreasing number of faces and vertices. The resulting models do not necessarily form valid boundaries of 3D regions--for example, an elongated solid may be approximated by a curve segment. However, the error introduced by the simplification is bounded (in the Hausdorff distance sense) by a user-controlled accuracy factor and the resulting shapes exhibit a remarkable visual fidelity considering the data-reduction ratios, the simplicity of the approach, and the performance and robustness of the implementation.

The original model of each object is represented by a vertex table containing vertex coordinates and a face table containing references to the vertex table, sorted and organized according to the edge-loops bounding the face. The simplification involves the following processing steps:

1. grading of vertices (assigning weights) as to their visual importance
2. triangulation of faces
3. clustering of vertices
4. synthesis of the representative vertex for each cluster
5. elimination of degenerate triangles and of redundant edges and vertices
6. adjustment of normals
7. construction of new triangle strips for faster graphics performance

Grading

A weight is computed for each vertex. The weight defines the subjective perceptual importance of the vertex. We favor vertices that have a higher probability of lying on the object's silhouettes from an arbitrary viewing direction

and vertices that bound large faces that should not be affected by the removal of small details. The first factor may be efficiently estimated using the inverse of the maximum angle between all pairs of incident edges on the candidate vertex. The second factor may be estimated using the face area.

Note that in both cases, these inexpensive estimations are dependent on the particular tessellation. For example, subdividing the faces incident upon a vertex will alter its weight although the actual shape remains constant. Similarly, replacing a sharp vertex with a very small rounded sphere will reduce the weight of the corresponding vertices, although the global shape has not changed. A better approach would be to consider the local morphology of the model (estimate of the curvature near the vertex and estimate of the area of flat faces incident upon the vertex (see [Gross95] for recent progress in this direction).

Triangulation

Each face is decomposed into triangles supported by its original vertices. Because CAD models typically contain faces bounded by a large number of edges, a very efficient, yet simple triangulation technique is required. The resulting table of triangles contains 3 vertex-indices per triangle.

Note that attempting to simplify non-triangulated faces will most probably result in non-flat polygons. On the other hand, it may be beneficial to remove very small internal edge loops from large faces prior to triangulation. This approach will significantly reduce the triangle count in mechanical CAD models, where holes for fasteners are responsible for a major part of the model's complexity. Simplifying triangulated models without removing holes will not create cracks in the surface of the solid and will not separate connected components, Removing holes prior to simplification may result in separation of connected components or in the creation of visible cracks.

Clustering

The vertices are grouped into clusters, based on geometric proximity. With each vertex, we associate the corresponding cluster's id. Although a variety of clustering techniques was envisioned, we have opted for a simple clustering process based on the truncation (quantization) of vertex coordinates. A box, or other bound, containing the object is uniformly subdivided into cells. After truncation of coordinates, the vertices falling within a cell will have equal coordinates. A cell, and hence its cluster is uniquely identified by its three coordinates. The clustering procedure takes as parameters the box in which the clustering should occur and the maximum number of cells along each dimension. The solid's bounding box or a common box for the entire scene may be used. The number of cells in each dimension is computed so as to achieve the desired level of simplification. A particular choice may take into account the geometric complexity of the object, its size relative size and importance in the scene, and the desired reduction in triangle count. The result of this computation is a table (parallel to the vertex table) which associates vertices with cluster indices (computed by concatenating cluster integer coordinates).

This approach does not permit to select the precise triangle reduction ratio. Instead, we use a non-linear estimator and an adaptive approach to achieve the desired complexity reduction ratios. For instance, given the size and complexity of a particular solid relative to the entire scene, we estimate the cell size that would yield the desired number of triangles, we run the simplification, and if the result is far from our estimate, we use it for a different level of detail and adjust the cell size appropriately for the next simplification level.

Synthesis

The vertex/cluster association is used to compute a vertex representative for each cluster. A good choice is the vertex closest to the weighted the average of the vertices of the cluster, where the results of grading are used as weights. Less ambitious choices, permits to compute the cluster's representative vertices without reading the input data twice, which leads to important performance improvements when the input vertex table is too large to fit in memory. Vertex/cluster correspondence yields a correspondence between the original vertices and the representative vertices of the simplified object. Thus, each triangle of the original object references three original vertices, which in turn reference three representative vertices. (Note that representative vertices are a subset of the original vertices, although a simple variation of this approach will support an optimization step that would compute new locations of the representative vertices.) The representative vertices define the geometry of the triangle in the simplified object.

The explicit association between the original vertices and the simplified ones permits to smoothly interpolate between the original model and the simplified one. The levels of detail may be computed in sequence, starting from the original and generating the first simplification, then starting with this simplified model and generating the next (more simplified) model and so on. This process will produce a hierarchy of vertex clusters, which may be used to smoothly interpolate between the transitions from one level to the next, and hence to avoid a distracting popping effect. We have experimented with such smooth transitions and concluded that, although visually pleasant, they benefit did not justify the additional interpolation and book-keeping costs. Indeed, during transition phases, the faces of a more detailed simplification must be used when the lower level of detail may suffice to meet the desired accuracy. For example, consider that simplification 2 contains 1000 triangles and corresponds to an error of 0.020, and that simplification 3 contains 100 triangles and corresponds to an error of 0.100. If the viewing conditions impose an error cap of 0.081, we could use simplification 2 alone and display only 100 triangles. If however we

chose to use a smooth interpolation between consecutive levels in the transition zone for errors between 0.080 and 1.020, we would have not only to compute a new position for 500 vertices as a linear combination of two vertices, but we will have to display 1000 triangles. Consequently, the smooth interpolation will result in significant runtime processing costs and in an order of magnitude performance drop for this solid. Assuming uniform distribution, this penalty will be averaged amongst the various instances (only 40% of instances would be penalized at a given time). The total performance is degraded by a factor of 4.6.

Also note that in order to prevent the accumulation of errors, when a level of detail is computed by simplifying another level of details, the cells for the two simplification processes should be aligned and the finer cells should be proper subdivisions of the coarser cells.

Elimination

Many triangles may have collapsed as the result of using representative vertices rather than the original ones. When, for a given triangle, all three representative vertices are equal, the triangle degenerates (collapses) into a point. When exactly two representative vertices are equal the triangle degenerates into an edge. Such edges and points, when they bound a triangle in the simplified object are eliminated. Otherwise, they are added to the geometry associated with the simplified model. Duplicated, triangles, edges, and vertices are eliminated during that process. Efficient techniques may be invoked, which use the best compromise between space and performance. When the number of vertices in the simplified model is small, a simple hashing scheme will yield an almost linear performance. When the number of vertices in the simplified model is large, duplicated geometries may be eliminated at the cost of sorting the various elements.

Adjustment of normals

This step computes new normals for all the triangles coordinates. It uses a heuristic to establish which edges are smooth. The process also computes triangle meshes. We use a face clustering heuristics which builds clusters of adjacent and nearly coplanar faces amongst all the incident faces of each vertex. An average normal is associated with the vertex-use for all the faces of a cluster.

Generation of new triangle strips

Because each simplification reduces the model significantly, it is not practical to exploit triangle strips computed on the original model. Instead, we re-compute new triangle strips for each simplified model.

Runtime level selection

Several levels of detail may be pre-computed for each object and used whenever appropriate to speed up graphics. In selecting the particular simplification level for a given object, it is important to take into accounts the architecture of the rendering subsystem so as not to oversimplify in situations where the rendering process is pixel bound. For example, the cost of rendering in software and in a large window an object that has a relatively low complexity but fills most of the screen is dominated by R. Consequently, simplification will have very little performance impact, and may reduce the image fidelity without benefit. On the other hand, displaying a scene with small, yet complex objects, via a software geometric processing on a fast hardware rasterizer will be significantly improved by simplification before the effects of using simplified models become noticeable.

Isolated edges, that result from the collapsing of some triangles may be displayed as simple edges whose width is adjusted taking into account their distance to the viewer.

Advantages and implementation

The process described above has several advantages over other simplification methods:

The computation of the simplification does not require the construction of a topological adjacency graph between faces, edges, and vertices. It works of a simple array of vertices and of an array of triangles, each defined in terms of three vertex-indices.

The algorithm for computing the simplification is very time efficient. In its simplest form, it needs to traverse the input data (vertex and triangle tables) only once.

The tolerance (i.e. bound on the Hausdorff distance between the original and simplified model) may be arbitrarily increased reducing the triangle count by several orders of magnitude.

To further reduce the triangle count, the simplification algorithm may produce non-regularized models. Particularly, when using the appropriate tolerance, thin plates may be simplified to dangling faces, long objects to isolated edges, and (groups of) small solids into isolated points.

The approach is not restricted by topological adjacency constraints and may merge features that are geometrically close, but are not topologically adjacent. Particularly, an arbitrary number of small neighboring isolated objects may be merged and simplified into a single point.

The simplification algorithm was combined with the data import modules of IBM's 3D Interaction Accelerator and exercised on hundreds of thousands of models of various complexity. It exhibits a remarkable performance characteristics, making it faster to re-compute simplifications than to read the equivalent ASCII files from disk. The algorithm pre-computes several levels of detail, which are then used at run time to accelerate graphics during interaction with models comprising millions of triangles. The particular simplification level of a given object is computed so as to match a user specified performance or quality target while allocating more geometric complexity (and thus more rendering cost) to objects which a higher visual importance.

Our experience shows that typical CAD models of mechanical assemblies comprise dozens of thousands of objects. The relative size and complexity of the objects may vary greatly. A typical object may have a thousand triangles in its original boundary. The simplification process described here may be used to automatically reduce the triangle count by an average factor of 5 without impacting the overall shape and without hindering the users ability to identify the important features. Further simplifications lead to further reduction of the triangle count, all the way down to a single digit, while still preserving the overall shape of the object and making it recognizable in the scene. This simplification process has been further improved to yield a better fidelity/complexity ratio by incorporating topological and curvature considerations in the clustering process. However, these improvements only lead to additional computational costs and more complex code.

Ronfard and Rossignac's Edge collapse

Remi Ronfard and I have devised a different approach, which permits greater movement of vertices in areas where the surface is approximately flat. It is based on the edge-collapse operation of Fig 21. The distinction of this approach to similar edge-collapse approaches by Hugues Hoppe and others is the way in which we evaluate a bound on the total error resulting from each edge collapse. We need that bound to order the edge collapses, so that we execute the least damaging first.

Vertex displacement (as used in the vertex clustering technique) provides a rather pessimistic error bound. Imagine for example a displacement of a vertex in a dense triangulation of a flat portion of a terrain. When vertices move along the plane of the terrain, the represented geometry remains flat, and therefore the Hausdorf error is zero, although the vertex displacements may be large.

Ronfard and I have introduced an error estimator which measures the displacement of vertices in the direction orthogonal to their incident faces. Their estimator computes the maximum distance from the new location P of the vertex to all the supporting planes of the vertices that have collapsed into P. This estimator works well for flat and nearly flat regions, but may not provide an upper bound close to sharp edges or vertices, as shown below. In the cases of sharp corners, Ronfard and I introduce an additional plane that is orthogonal to the average normal to the incident triangles and suffices to guarantee a precise upper bound on the error.

Progressive transmission

Rather than waiting for the detailed 3D model to arrive, the remote user may wish to download an initial crude model first. This model may be visualized to provide the initial feedback and guide early navigation decisions. Then it would be automatically upgraded to a more accurate model. Note that, if, based on the approximated model, the user decides early on to turn around or to switch to a different 3D model, the accurate model needs not be downloaded at all. The client would simply interrupt its download.

Notice also that when a 3D model appears small on the screen, a low resolution approximation, as described below may be used for rendering. If the viewer never approaches that object, its detailed representation will never be needed by the client.

In conclusion, the bandwidth requirements may be reduced considerably by first transmitting a low resolution model and then a more accurate one, if at all necessary.

So, if you never need the higher resolution model, we can put together the results of the previous two sections: simplify the model first, then compress it and transmit it.

Now, suppose that you did this, and then you realize that the client does need the higher accuracy model. Well, we can pre-compute and store a compressed version of the high accuracy model and transmit it when needed. This is a viable approach. When both the crude and then the full resolution models are needed, the total transmission cost is the sum of the two. If the crude model is really simple, then the overhead of this approach over transmitting only the full resolution model is small. Furthermore, if we factor in the probability of not needing the full resolution model for a random object, this is a compelling solution.

So, why not stop here?

The problem with the "two-level" approach is that either you have to work with a very crude model, or you have to wait a long time to see the fully accurate one. Wouldn't we rather get a crude model and then see it improve progressively as we wait or as we approach it? Several techniques were developed to support this.

How do you compare them? Consider the graph of Fig. 23. It shows the approximation error of the model currently available to the client as a function of time (i.e., number of bits transmitted). Note that for a little while you have nothing (the error is very large). Then you receive and decompress the crudest model. The elapsed time is the time to first picture. (Although it includes the request, transmission, decompression, and rendering, we will only discuss the transmission cost here.) Then, if you need more resolution, with each fraction of bits received, the accuracy of the model is upgrade.

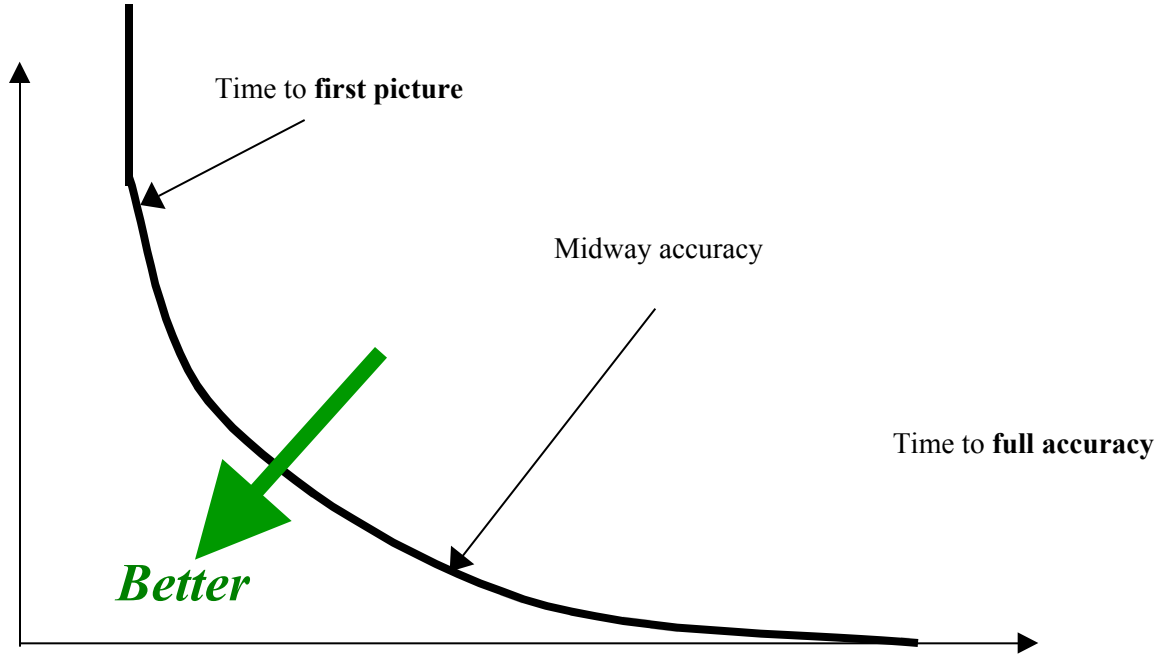

Fig 23: Error/time evolution.

How do we compare two progressive schemes? The smaller the area below that curve, the better a scheme. Typically we want to reduce the error quickly (want the curve to be low). We also want to limit the amount of time we wait for the whole resolution model (the curve should be as much left as possible).

Of course, this curve cannot be this smooth, because you need to receive more than one bit to improve the accuracy of the model. So, in reality, the bits are transmitted in batches and the curve becomes a set of horizontal steps. You have to wait between steps and after each step the error is decreased by some amount.

The main decision in selecting a progressive approach is the size of the steps. In the two step approach discussed above, the size of the steps is huge.

Hugues Hoppe's progressive mesh is the other extreme. Each step corresponds to the inverse of an edge-collapse, and thus inserts one vertex and two triangles. Because these operations are transmitted in inverse order of the simplification operations, the error decreases with each step.

The cost of each step was discussed in the compression section. The bulk of it lies in the need for encoding the ids of two adjacent edges at each step. This cost is proportional to the log of the number of edges (which, in practice, grows linearly with the number of vertices and triangles).

To reduce this overhead, Renato Pajarola and I have decided to group these vertex-split operations into batches, which define bigger steps. One such step is illustrated in Fig. 24.

The economy of scale, which results from the grouping, is exploited using the following approach. Instead of encoding the ids of all the edges that will have to be split, we encode one bit for each vertex of the current mesh. A zero means, leave it alone. A one means split it. For each vertex that has to be split, we also encode the information necessary to select two of its incident edges. Since the encoder and the decoder know how many edges there are, we can encode this in a compact manner.

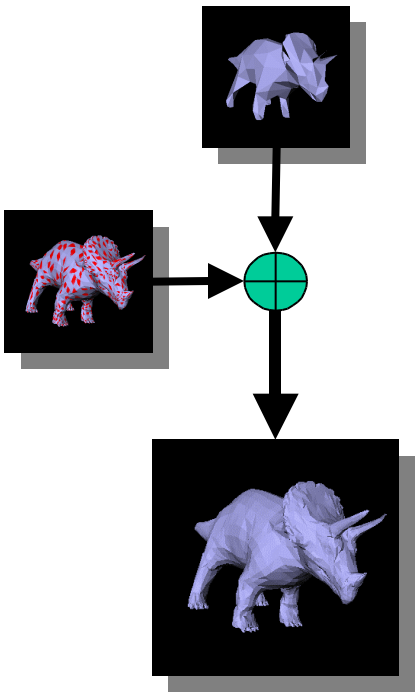

Fig 24: Progressive transmission (crude model plus upgrades)

How do we know the association between bits and vertices? We simply use the same mesh traversal as the one used for Edgebreaker. We transmit the bits in the order in which the corresponding vertices are visited by C operations. The set of triangle-pairs inserted in a given step are illustrated in Fig. 25.

Fig 25: Triangles inserted in one batch

At first, this system seems to be rather wasteful. A vertex in the crude model may be associated with a zero. Then in the second step, it may be associated with a zero again, and so on. So the total cost per vertex seems to be significant. Not quite. There are very few vertices in the crude model. So that first set of zeros is amortized over all the vertices in the entire model. Subsequent zeros are more numerous, but there are fewer and fewer of them associated with a vertex on average. On average, the amortized cost of encoding that vertex selection bit and the bits for identifying two if its incident edges is about 3.6 bits per triangle.

The vertex location is encoded using a predictor that takes into account the location of the split vertex and of its neighbors.

The total number of triangles, the number of step (i.e. of levels of detail), and the total number of bits per triangle are given for five models, which represent the typical shape used in entertainment and engineering applications. The shapes are illustrated in Fig. 26 from left to right. The total bit-cost per triangle is split into the cost of encoding the connectivity (both the bits identifying the split-vertices and the bits identifying their incident edges). These costs are indicated in parentheses.

1. Bunny: 9666 triangles, 10 LODs, 11.3 t bits (3.6 connectivity + 7.7 geometry)
2. Horse: 21622 triangles, 9 LODs, 10.6 t bits (3.5 + 7.1)
3. Skull: 21904 triangles, 7 LODs, 10.9 t bits (3.4 + 7.5)
4. Fohe: 7240 triangles, 7 LODs, 13.7 t bits (3.5 + 10.1)
5. Fandisk: 12950 triangles, 9 LODs, 11.4 t bits (3.7 + 7.7)

Fig 26: Models used to test our progressive transmission

Conclusions

Triangle meshes are the simplest interpolation of sample points in 3D. The vertex location and the connectivity of these meshes must be encoded when we wish to transmit them. Vertex location are best encoded as corrective vectors between a predicted location and the correct one. Good predictions lead to short corrective vectors whose coordinates may be encoded with variable length compression schemes.

To encode the connectivity, we have discussed the Edgebreaker scheme. It visits the mesh in a spiral and paints vertices and triangles. It also assigns symbols in the {C,L,E,R,S} set to the triangles and consecutive integers top the vertices. The sequence of symbols captures the connectivity if the vertices are sent in the order indicated by their integer labels. The cost of the connectivity is guaranteed to be below 2 bits per triangle and in practice averages one bit per triangle. The cost of transmitting vertex locations depends on the desired accuracy, but in practice fluctuates between 6 and 8 bits per triangle.

When compression is not sufficient, we simplify the model by clustering vertices or collapsing edges. This is a form of lossy compression.

A crude (simplified) model may be refined. This approach is cheaper than transmitting the finer model from scratch. The total cost of transmitting the connectivity in such a progressive manner is about 3.5 bits per vertex. For the small 1.5 bit per vertex overhead, the typical model is received in 7 to 10 progressive refinement steps. The savings come from the fact that more accurate models are available sooner to the user and that the transmission of the fully accurate models may often be avoided because the model remain small on the screen or because the viewer turns away.

References

[Airey90] J. Airey, J. Rohlf, and F. Brooks, Towards image realism with interactive update rates in complex virtual building environments, ACM Proc. Symposium on Interactive 3D Graphics, 24(2):41-50, 1990.

[Alexandroff61] P. Alexandroff, Elementary Concepts of Topology, Dover Publications, New York, NY, 1961.

[Algorri96] M.-E Algorri, F. Schmitt, Surface reconstruction from unstructured 3D data, Computer Graphics Forum, 15(1):4760, March 1996.

[Andujar96] C. Andujar, D. Ayala, P. Brunet, R. Joan-Arinyo, J. Sole, Automatic generation of multi-resolution boundary representations, Computer-Graphics Forum (Proceedings of Eurographics'96), 15(3):87-96, 1996.

[Banerjee96] R. Banerjee and J. Rossignac, Topologically exact evaluation of polyhedra defined in CSG with loose primitives, to appear in Computers Graphics Forum, Vol. 15, No. 4, pp. 205-217, 1996.

[Bar-Yehuda96] R. Bar-Yehuda and C. Grotsman, Time/space tradeoffs for polygon mesh rendering. ACM Transactions on Graphics, 15(2):141-152, April 1996.

[Baumgart72] Winged Edge Polyhedron Representation, B. Baumgart, AIM-79, Stanford University Report STAN-CS-320, 1972.

[Beigbeder91] M. Beigbeder and G. Jahami, Managing levels of detail with textured polygons, Compugraphics'91, Sesimbra, Portugal, pp. 479-489, 16-20 September, 1991.

[Bergman86] L. Bergman, H. Fuchs, E. Grant and S. Spach, Image Rendering by Adaptive Refinement, Computer Graphics (Proc. Siggraph'86), 20(4):29-37, Aug. 1986.

[Blake87] E. Blake, A Metric for Computing Adaptive Detail in Animated Scenes using Object-Oriented Programming, Proc. Eurographics`87, 295-307, Amsterdam, August1987.

[Borrel95] P. Borrel, K.S. Cheng, P. Darmon, P. Kirchner, J. Lipscomb, J. Menon, J. Mittleman, J. Rossignac, B.O. Schneider, and B. Wolfe, The IBM 3D Interaction Accelerator (3DIX), RC 20302, IBM Research, 1995.

[Brown82] C. Brown, PADL-2: A Technical Summary, IEEE Computer Graphics and Applications, 2(2):69-84, March 1982.

[Carey97] R. Carey, G. Bell, C. Martin, The Virtual Reality Modeling Language ISO/IEC DIS 14772-1, April 1997, http://www.vrml.org/Specifications.VRML97/DIS.

[Chen93] E. Chen and L. Williams, View interpolation for image synthesis, Proc. ACM Siggraph. pp. 279-288, 1993.

[Cignoni95] P. Cignoni, E. Puppo and R. Scopigno, Representation and Visualization of Terrain Surfaces at Variable Resolution, Scientific Visualization 95, World Scientific, 50-68, 1995. http://miles.cnuce.cnr.it/cg/multiresTerrain.html#paper25.

[Clark76] J. Clark, Hierarchical geometric models for visible surface algorithms, Communications of the ACM, 19(10):547-554, October 1976.

[Cohen96] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agrawal, F. Brooks, W. Wright, Simplification Envelopes, Proc. ACM Siggraph'96, pp. 119-128, August 1996.

[Crosnire99] "Tribox-based simplification of three-dimensional objects", A. Crosnier and J. Rossignac. **Computers&Graphics**, March 1999.

[Crow82] F. Crow, A more flexible image generation environment, Computer Graphics, 16(3):9-18, July 1982.

[Darsa97] L. Darsa, B. Costa Silva, A. Varshney, Navigating static environments using image-space simplification and morphing, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 7-16, April 1997.

[Deering95] M. Deering, Geometry Compression, Computer Graphics, Proceedings Siggraph'95, 13-20, August 1995.

[DeFloriani92] L. De Floriani and E. Puppo, A hierarchical triangle-based model for terrain description, in Theories and Methods of Spatio-Temporal Reasoning in Geographic Space, Ed. A. Frank, Springer-Verlag, Berlin, pp. 36--251, 1992.

[DeHaemer91] M. DeHaemer and M. Zyda, Simplification of objects rendered by polygonal approximations, Computers and Graphics, 15(2):175-184, 1991.

[DeRose92] T, DeRose, H, Hoppe, J. McDonald, and W, Stuetzle, Fitting of surfaces to scattered data, In J. Warren, editor, SPIE Proc. Curves and Surfaces in Computer Vision and Graphics III, 1830:212-220, November 16-18, 1992.

[Durand97] F. Durand, G. Drettakis, and C. Puech, The Visibility Skeleton: A powerful and efficient multi-purpose global visibility tool, Computer Graphics, Proceedings Siggraph'97, pp. 89-100, August 1997.

[Eck95] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery and W. Stuetzle, Multiresolution Analysis of Arbitrary Meshes, Proc. ACM SIGGRAPH'95, pp. 173-182, Aug. 1995.

[Erikson96] C. Erikson, Polygonal Simplification: An Overview, UNC Tech Report TR96-016, http://www.cs.unc.edu/~eriksonc/papers.html

[Evans96] F. Evans, S. Skiena, and A. Varshney, Optimizing Triangle Strips for Fast Rendering, Proceedings, IEEE Vizualization'96, pp. 319--326, 1996.

[Farin90] G. Farin, Curves and Surfaces for Computer-Aided Geometric Design, Second edition, Computer Science and Scientific Computing series, Academic Press, 1990.

[Funkhouser93] T, Funkhouser, Database and Display Algorithms for Interactive Visualization of Architectural Models, PhD Thesis, CS Division, UC Berkeley, 1993.

[Funkhouser93b] T. Funkhouser, C. Sequin, Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments, Computer Graphics (Proc. SIGGRAPH '93), 247-254, August1993.

[Garland95] M. Garland and P. Heckbert, Fast Polygonal Approximation of Terrains and Height Fields, Research Report from CS Dept, Carnegie Mellon U, CMU-CS-95-181. {http://www.cs.cmu.edu/~garland/scape). Sept.1995.

[Garland97] M. Garland and P. Heckbert, Surface simplification using quadric error metrics, Proc. ACM SIGGRAPH'97. pp. 209-216. 1997.

[Greene93] N, Greene, M, Kass, and G, Miller, Hierarchical z-buffer visibility, Proceedings, pp:231-238, 1993.

[Gross95] M. Gross, R. Gatti and O. Staadt, Fast Multi-resolution surface meshing, Proc. IEEE Visualization'95, pp. 135-142, 1995.

[Gueziec96] A, Gueziec, Surface Simplification inside a tolerance volume, IBM Research Report RC20440, Mars 1996.

[Haralick77] R. Haralick and L. Shapiro, Decomposition of polygonal shapes by clustering, IEEE Comput. Soc. Conf. Pattern Recognition Image Process, pp. 183-190, 1977.

[He96] T. He, A. Varshney, and S. Wang, Controlled topology simplification, IEEE Transactions on Visualization and Computer Graphics, 1996.

[Heckbert94] P. Heckbert and M. Garland, Multiresolution modeling for fast rendering, Proc Graphics Interface'94, pp:43-50, May 1994.

[Heckbert97] P. Heckbert and M. Garland, Survey of Polygonal Surface Simplification Algorithms, in *Multiresolution Surface Modeling Course,* ACM Siggraph Course notes, 1997.

[Hoppe92] H. Hoppe, T, DeRose, T, Duchamp, J. McDonald, and W, Stuetzle, Surface reconstruction from unorganized points, Computer Graphics (Proceedings SIGGRAPH'93), 26(2):71-78, July 1992.

[Hoppe96] H. Hoppe, Progressive Meshes, Proceedings ACM SIGGRAPH'96, pp. 99-108, August 1996.

[Hoppe97] H. Hoppe, View Dependent Refinement of Progressive Meshes, Proceedings ACM SIGGRAPH'97, August 1997.

[Jacobson89] G. Jacobson, Succinct Static Data Structures, PhD Thesis, Carnegie-Mellon, Tech Rep CMU-CS-89-112, January 1989.

[Kajiya85] J. Kajiya, Anisotropic reflection models, Computer Graphics, Proc. Siggraph, 19(3):15-21, July 1985.

[Kalvin91] A, Kalvin, C, Cutting, B. Haddad, and M, Noz, Constructing topologically connected surfaces for the comprehensive analysis of 3D medical structures, SPIE Image Processing, 1445:247-258, 1991.

[Kalvin96] A.D. Kalvin, R.H. Taylor, Superfaces: Polyhedral Approximation with Bounded Error, IEEE Computer Graphics & Applications, 16(3):64-77, May 1996.

[King99] "Optimal Bit Allocation in Compressed 3D Models", D. King and J. Rossignac, Journal of Computational Geometry, Theory and Applications. Volume 14, Issue 1-3, pp. 91-118. November 1999.

[King99b] "Guaranteed 3.67V bit encoding of planar triangle graphs" D. King and J. Rossignac, 11th Canadian Conference on Computational Geometry (CCCG'99), pp. 146-149, Vancouver, CA, August 15-18, 1999.

[King99c] "Connectivity Compression for Irregular Quadrilateral Meshes" D. King, J. Rossignac, submitted for publication, Dec 1999.

[King2000] "An Edgebreaker-based efficient compression scheme for regular meshes", D. King, J. Rossignac, A. Szymczak. Accepted to the 12th Canadian Conference on Computational Geometry, Fredericton, New Brunswick, August 16-19, 2000.

[Klein96] R. Klein and W. Strasser, Generation of multiresolution models from CAD data for realtime rendering. In W. Strasser, R. Klein, R. Rau, Editors. Theory and practice of Geometric Modeling. Springer-Verlag, 1996.

[Lawson72] C. Lawson, Transforming Triangulations, Discrete Math. 3:365-372, 1972.

[Lee77] D.T. Lee and F.P. Preparata, Location of a point in a planar subdivision and its applications. SIAM J. on Computers, 6:594-606, 1977.

[Lindstrom96] P. Lindstrom, D. Koller and W. Ribarsky and L. Hodges and N. Faust G. Turner, Real-Time, Continuous Level of Detail Rendering of Height Fields, SIGGRAPH '96, 109--118, Aug. 1996.

[Lounsbery94] M. Lounsbery, Multiresolution Analysis for Surfaces of Arbitrary Topological Type, PhD. Dissertation, Dept. of Computer Science and Engineering, U. of Washington, 1994.

[Low97] K-L. Low and T-S. Tan, Model Simplification using Vertex-Clustering, Proc. 3D Symposium on Interactive 3D Graphics, pp. 75-81, Providence, April 1997.

[Luebke95] D, Luebke, C, George, Portals and Mirrors: Simple, fast evaluation of potentially visible sets, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 105-106, April 1995.

[Luebke96] D. Luebke, Hierarchical structures for dynamic polygonal simplifications, TR 96-006, Dept. of Computer Science, University of North Carolina at Chapel Hill, 1996.

[Luebke97] D. Luebke and C. Erikson, View-dependent simplification of arbitrary polygonal environments, Proc. Siggraph. pp. 199-208, Los Angeles, 1997.

[Maciel95] P, Maciel, P, Shirley, Visual Navigation of Large Environments Using Textured Clusters, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 95-102, April 1995.

[Mann97] Y. Mann and D. Cohen-Or, Selective Pixel Transmission for Navigation in Remote Environments, Proc. Eurographics'97, Budapest, Hungary, September 1997.

[Mark97] W. Mark, L. McMillan, and G. Bishop, Post-rendering 3D warping, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 7-16, April 1997.

[Mitchell95] J.S.B, Mitchell, and S. Suri, Separation and approximation of polyhedral objects, Computational Geometry: Theory and Applications, 5(2), pp. 95-114, September 1995.

[Naylor95] B, Naylor, Interactive Playing with Large Synthetic Environments, 1995 Symposium on Interactive 3D Graphics, ACM Press, pp. 107-108, April 1995.

[Neider93] J. Neider, T. Davis, and M. Woo, OpenGL Programming Guide, Addison-Wesley, 1993.

[Optimizer97] SGI OpenGL Optimizer, http://www.sgi.com/Technology/openGL/optmizer_wp.html. 1997.

[Pajarola99] "Implant Sprays: Compression of Progressive Tetrahedral Mesh Connectivity", R. Pajarola, J. Rossignac, and A. Szymczak, IEEE Visualization 1999, San Francisco, October 24-29, 1999.

[Pajarola2000] "Compressed Progressive Meshes", R. Pajarola and J. Rossignac, IEEE Transactions on Visualization and Computer Graphics, Volume 6, No. 1, pp. 79-93, January-March 2000.

[Pajarola2000b] "Squeeze: Fast and Progressive Decompression of Triangle Meshes", R. Pajarola and J. Rossignac, accepted to the Computer Graphics International conference, Switzerland, June 2000.

[Pennebaker93] B. Pennebaker and J. Mitchell, JPEG, Still Image Compression Standard, Van Nostrand Reinhold, 1993.

[Perlin84] K. Perlin, A unified texture/reflectance model. Siggraph'84, Advanced Image Synthesis Sminar, July 1984.

[Popovic97] J. Popovic and H. Hoppe, Progressive Simplicial Complexes, Proceedings *ACM Siggraph'97*, pp. 217-224, August 1997.

[Puppo96] E. Puppo, Variable resolution terrain surfaces, Proceedings of the *Eight Canadian Conference on Computational Geometry,* Ottawa, Canada, pp. 202-210, August 12-15, 1996.

[Puppo97] E. Puppo and R. Scopigno, Simplification, LOD and multiresolution: Principles and applications, Tutorial at the Eurographics'97 conference, Budapest, Hungary, September 1997.

[Ralston83] A. Ralston and E. Reilly, Editors, Encyclopedia od Computer Science and Engineering, second Edition, van Nostrand Reinhold Co., New York, pp .97-102, 1983.

[Regan94] M. Regan and R. Pose, Priority rendering with a virtual reality address recalculation pipeline, Proceedings ACM SIGGRAPH'94, pp. 155-162, August 1994.

[Rockwood89] A, Rockwood, K Heaton, and T, Davis, Real-time Rendering of Trimmed Surfaces, Computer Graphics, 23(3):107-116, 1989.

[Ronfard94] R Ronfard, and J, Rossignac, Triangulating multiply-connected polygons: A simple, yet efficient algorithm, Proc. Eurographics'94, Oslo, Norway, Computer Graphics Forum, 13(3):C281-C292, 1994.

[Ronfard96] "Full-range approximations of triangulated polyhedra", Remi Ronfard and Jarek Rossignac, Proceedings of Eurographics'96, Computer Graphics Forum, pp. C-67, Vol. 15, No. 3, August 1996.

[Rossignac84] J. Rossignac and A. Requicha, Constant-Radius Blending in Solid Modeling, ASME Computers in Mechanical Engineering (CIME), Vol. 3, pp. 65-73, 1984.

[Rossignac86] J. Rossignac and A. Requicha, Depth Buffering Display Techniques for Constructive Solid Geometry, IEEE Computer Graphics and Applications, 6(9):29-39, September 1986.

[Rossignac86b] J. Rossignac and A. Requicha, Offsetting Operations in Solid Modelling, Computer-Aided Geometric Design, Vol. 3, pp. 129-148, 1986.

[Rossignac89] J. Rossignac and M. O'Connor, SGC: A Dimension-independent Model for Pointsets with Internal Structures and Incomplete Boundaries, in Geometric Modeling for Product Engineering, Eds. M. Wosny, J. Turner, K. Preiss, North-Holland, pp. 145-180, 1989.

[Rossignac93] "Representing Solids and Geometric Structures", J. Rossignac, in Geometry and Optimization Techniques for Structural Design, pp. 1-44, Eds. S. Kodiyalam, M. Saxena, Computational Mechanics Publications, Southhampton, 1993,

[Rossignac93b] "Representation, Design, and Visualization of Solid and of Geometric Structures", J. Rossignac. Tutorial notes, Eurographics, September, 1993.

[Rossignac93c] J. Rossignac and P. Borrel, Multi-resolution 3D approximations for rendering complex scenes, pp. 455-465, in Geometric Modeling in Computer Graphics, Springer Verlag, Eds. B. Falcidieno and T.L. Kunii, Genova, Italy, June 28-July 2, 1993.

[Rossignac94] "Representing and visualizing complex continuous geometric models", J. Rossignac. in Scientific Visualization: Advances and Challenges, Academic Press. Ed. L Rosenblum. pp. 337-348, 1994.

[Rossignac94b] "Specification, Representation, and Construction of non-manifold geometric structures", J. Rossignac. Lecture at the ACM SIGGRAPH 1994.

[Rossignac94c] J. Rossignac and M. Novak, Research Issues in Model-based Visualization of Complex Data Sets, IEEE Computer Graphics and Applications, 14(2):83-85, March 1994.

[Rossignac94d] J. Rossignac, Through the cracks of the solid modeling milestone, in *From Object Modelling to Advanced Visualization*, Eds. S. Coquillart, W. Strasser, P. Stucki, Springer Verlag, pp. 1-75, 1994.

[Rossignac95] J. Rossignac, Geometric Simplification, in Interactive Walkthrough of Large Geometric Databases (ACM Siggraph Course Notes 32), pp. D1-D11, Los Angeles, 1995.

[Rossignac97] J. Rossignac, Structured Topological Complexes: A feature-based API for non-manifold topologies", Proceedings of the ACM Symposium on Solid Modeing 97, pp.1-12, May 13-15, Atlanta, 1997.

[Rossignac97b] J. Rossignac, Geometric Simplification and Compression, in *Multiresolution Surface Modeling Course,* ACM Siggraph Course notes 25, Los Angeles, 1997.

[Rossignac97c] J. Rossignac, The 3D revolution: CAD access for all, International Conference on Shape Modeling and Applications, Aizu-Wakamatsu, Japan, IEEE Computer Society Press, pp. 64-70, 3-6 March 1997.

[Rossignac97d] J. Rossignac, Simplification and Compression of 3D Scenes, Eurographics Tutorial, 1997.

[Rossignac98] "Compression of 3D Models" J. Rossignac. Lecture at the ACM SIGGRAPH conference 1998.

[Rossignac98b] "Interactive exploration of distributed 3D datasets over the Internet", J. Rossignac. Computer Graphics International Congress (CGI '98 congress), Hanover, pp. 324-335, June 24-26, 1998.

[Rossignac99] "3D Compression" J. Rossignac. Lecture at the ACM SIGGRAPH conference 1999.

[Rossignac99b] "Edgebreaker: Connectivity compression for triangle meshes", J. Rossignac. IEEE Transactions on Visualization and Computer Graphics, Vol. 5, No. 1, pp. 47-61, January - March 1999.

[Rossignac99c] "Solid Modeling", J. Rossignac and A. Requicha, in the Encyclopedia of Electrical and Electronics Engineering, Ed. J. Webster, John Wiley & Sons. 1999.

[Rossignac99d] "Matchmaker: Manifold BReps for non-manifold r-sets", J. Rossignac and D. Cardoze. Proceedings of the ACM Symposium on Solid Modeling, pp. 31-41, June 1999.

[Rossignac99e] "Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker", J. Rossignac and A. Szymczak, Journal of Computational Geometry, Theory and Applications, Volume 14, Issue 1-3, pp. 119-135, November 1999.

[Rossignac2000] "3D Compression and progressive transmission" J. Rossignac. Lecture at the ACM SIGGRAPH conference July 2-28, 2000.

[Samet90] H. Samet, Applications of Spatial Data Structures, Reading, MA, Addison-Wesley, 1990.

[Scarlatos92] L. Scarlatos, and T, Pavlidis, Hierarchical triangulation using cartographic coherence, CVGIP: Graphical Models and Image Processing, 54(2):147-161, 1992.

[Schmitt86] F, Schmitt, B. Barsky, and W, Du, An adaptive subdivision method for surface-fitting from sampled data, Computer Graphics, 20(4):179-188, 1986.

[Schneider95] "BRUSH as a Walkthrough System for Architectural Models", B.-O. Schneider, P. Borrel, J. Menon, J. Mittleman, J. Rossignac, Proc. 5th Eurographics Workshop on Rendering, Darmstadt (Germany), June 1994. In Rendering Techniques'95, Springer-Verlag, 389-399, New York, 1995.

[Schroeder92] W, Schroeder, J. Zarge, and W, Lorensen, Decimation of triangle meshes, Computer Graphics, 26(2):65-70, July 1992.

[Schroeder97] W. Schroeder, A topology modifying Progressive Decimation Algorithm, in *Multiresolution Surface Modeling Course,* ACM Siggraph Course notes 25, Los Angeles, 1997.

[Sillion97] F. Sillion, G. Drettakis, and B. Bodelet, Efficient impostor manipulation for realtime visualization of urban scenery, Proceedings, Eurographics'97, Computer Graphics Forum, vol. 16, No. 3, pp. C207-C218, September 1997.

[Szymczak99] "Grow&Fold: Compression of Tetrahedral Meshes", A. Szymczak and J. Rossignac. Proc. ACM Symposium on Solid Modeling, June 1999, pp. 54-64

[Szymczak2000] "Grow&Fold: Compressing the connectivity of tetrahedral meshes", A. Szymczak and J. Rossignac. Computer-Aided Design. 32(8/9), 527-538, July/August, 2000.

[Taubin96] G. Taubin and J. Rossignac, Geometric Compression through Topological Surgery, IBM Research Report RC-20340. January 1996. (http://www.watson.ibm.com:8080/PS/7990.ps.gz).

[Taubin98] "Geometric Compression through Topological Surgery", G. Taubin and J. Rossignac. ACM Transactions on Graphics, Volume 17, Number 2, pp. 84-115, April 1998.

[Taubin98b] "Geometry coding and VRML", G. Taubin, W. Horn, F. Lazarus, and J. Rossignac, Proceedings of the IEEE, pp. 1228-1243, vol. 96, no. 6, June 1998.

[Teller91] S.J. Teller and C.H, Sequin, Visibility Preprocessing for interactive walkthroughs, Computer Graphics, 25(4):61-69, July 1991.

[Teller92] S, Teller, Visibility Computations in Densely Occluded Polyhedral Environments, PhD Thesis, UCB/CSD-92-708, CS Division, UC Berkeley, October 1992.

[Turan84] G. Turan, Succinct representations of graphs, Discrete Applied Math, 8: 289-294, 1984.

[Turk92] G. Turk, Re-tiling polygonal surfaces, Computer Graphics, 26(2):55-64, July 1992.

[Upstill90] S. Upstill, *The Renderman Companion.* Addison Wesley, Reading, MA, 1990.

[Varshney94] A, Varshney, Hierarchical Geometric Approximations, PhD Thesis, Dept. Computer Science, University of North Carolina at Chapel Hill, 1994.

[Weiler87] K. Weiler, Non-Manifold Geometric Boundary Modeling, ACM Siggraph, Tutorial on Advanced Solid Modeling, Anaheim, California, July 1987.

[Xia96] J. Xia and A. Varshney, Dynamic view-dependent simplification for polygonal models, Proc. Vis'96, pp. 327-334, 1996.

[Zhang97] H. Zhang, D. Manocha, T. Hudson, K. Hoff, Visibility culling using hierarchical occlusion maps, Computer Graphics, Proceedings Siggraph'97, pp. 77-88, August 1997.