Multiresolution Surface Modeling

Course Notes for SIGGRAPH '97
Los Angeles, California
5 August 1997

Course organizer:

Paul Heckbert

Speakers:

Paul Heckbert
Jarek Rossignac
Hugues Hoppe
William Schroeder
Marc Soucy
Amitabh Varshney

Preface

Course Summary

This course summarizes the best current techniques for simplifying complex polygonal surface models in order to accelerate rendering, speed network transmission, and conserve memory. The construction and use of multiresolution models that describe 3-D shapes at multiple levels of detail will be covered. Applications in CAD, Web publishing, geographic information systems, computer vision, and virtual reality will be discussed.

Course Objectives

Attendees will learn techniques for simplifying complex models and building multiresolution models. The algorithms covered include methods for terrains, methods for manifolds (simple 3-D surface models), and non-manifold surfaces (any set of polygons). Attendees will learn about free and commercial software, how the best algorithms work, and about open research problems.

Course Prerequisites

Understanding of 3-D geometry, simple polygon modeling techniques, and simple spatial data structures.

Intended Audience

Users, developers, and researchers working with complex polygonal models.

Level

Intermediate.

Course Notes

The papers include previously published and forthcoming papers and technical reports, and notes written specially for this course. Material is grouped by speaker, with slides following papers. The printed notes and SIGGRAPH '97 CD ROM contain identical material, except that some papers and slides on the CD ROM contain color images that appear in grayscale in the printed notes, and there is one set of slides on the CD ROM that does not appear in the printed notes.

Software

Other information (software, data, etc.) associated with this course that is not in these course notes is available on the World Web Web at http://www.cs.cmu.edu/~ph/mcourse97.html .

Contents

Paul Heckbert

- **Survey of Polygonal Surface Simplification Algorithms,**
  Paul S. Heckbert and Michael Garland, tech. report, CS Dept., Carnegie Mellon U., to appear.
- **Multiresolution Modeling for Fast Rendering,**
  Paul S. Heckbert and Michael Garland, Proc. Graphics Interface '94, Canadian Inf. Proc. Soc., Banff, May 1994, pp. 43–50.
- **Fast Triangular Approximation of Terrains and Height Fields,**
  Michael Garland and Paul S. Heckbert, submitted for publication.
- **Algorithms for Surface Simplification,**
  Paul Heckbert and Michael Garland, slides.

Jarek Rossignac

- **Geometric Simplification and Compression,**
  Jarek Rossignac.

Hugues Hoppe

- **Mesh Optimization,**
  Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle, SIGGRAPH '93 Proc., Aug. 1993, pp. 19–26.
- **Progressive Meshes,**
  Hugues Hoppe, SIGGRAPH '96 Proc., Aug. 1996, pp. 99–108.
- **View-Dependent Refinement of Progressive Meshes,**
  Hugues Hoppe, SIGGRAPH '97 Proc., Aug. 1997.
- **Progressive Simplicial Complexes,**
  Jovan Popović and Hugues Hoppe, SIGGRAPH '97 Proc., Aug. 1997.
- **Progressive Meshes and Recent Extensions,**
  Hugues Hoppe, slides, 6 per page.
- **Progressive Meshes and Recent Extensions,**
  Hugues Hoppe, slides, 1 per page, with notes. *This document is not in the printed notes.*

William Schroeder

- **Decimation of Triangle Meshes,**
  William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen, SIGGRAPH '92 Proc., July 1992, pp. 65–70.
- **A Compact Cell Structure for Scientific Visualization,**
  William J. Schroeder and Boris Yamrom. (From the book *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics*, Will Schroeder, Ken Martin, and Bill Lorensen, Prentice Hall, 1996.)
- **A Topology Modifying Progressive Decimation Algorithm,**
  William J. Schroeder, submitted for publication.
- **Decimation of Triangle Meshes,**
  William J. Schroeder, slides.

Marc Soucy

- **InnovMetric's Multiresolution Modeling Algorithms,**
  Marc Soucy.

Amitabh Varshney

- **A Hierarchy of Techniques for Simplifying Polygonal Models,**
  Amitabh Varshney.
- **Optimizing Triangle Strips for Fast Rendering,**
  F. Evans, S. Skiena, and A. Varshney, IEEE Visualization '96 Proc., Oct. 1996.
- **Simplification Envelopes,**
  J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. V. Wright, SIGGRAPH '96 Proc., Aug. 1996, pp. 119–128.
- **Controlled Topology Simplification,**
  T. He, L. Hong, A. Varshney, and S. Wang, IEEE Trans. on Visualization & Computer Graphics, 2(2), June 1996, pp. 171–184.
- **Adaptive Real-Time Level-of-detail-based Rendering for Polygonal Models,**
  J. Xia, J. El-Sana, and A. Varshney, IEEE Trans. on Visualization & Computer Graphics, June 1997.
- **A Hierarchy of Techniques for Simplifying Polygonal Models,**
  Amitabh Varshney, slides.

Speaker Biographies

Paul S. Heckbert
Assistant Professor
Computer Science Dept.
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh PA 15213-3891

Email: ph@cs.cmu.edu
Web: http://www.cs.cmu.edu/∼ph

Paul Heckbert is an Assistant Professor of Computer Science at Carnegie Mellon University. His research interests are computer graphics and rendering and modeling, specifically multiresolution surface modeling, radiosity, mesh generation, and texture mapping. Heckbert has a BS in Mathematics from MIT, and MS and PhD in Computer Science from the University of California at Berkeley. Previously he worked at the New York Institute of Technology Computer Graphics Lab and at Pixar, and he edited the book *Graphics Gems IV*.

Hugues Hoppe
Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Email: hhoppe@microsoft.com
Web: http://www.research.microsoft.com/research/graphics/hoppe/

Hugues Hoppe is a researcher in the Computer Graphics Group of Microsoft Research. His main area of interest is geometric modeling. Recently, his efforts have focused on level-of-detail (multiresolution) representations for storage, transmission, and rendering of complex polygonal models. He has also done research on the reconstruction of geometric models from 3D scanned data. He received a BS in electrical engineering in 1989 and a PhD in computer science and engineering in 1994 from the University of Washington.

Jarek Rossignac
Director of GVU Center, Professor in the College of Computing
Graphics, Visualization, and Usability Center
Georgia Institute of Technology, CoC 241
Atlanta, GA 30332-0280

Email: jarek@cc.gatech.edu
Web: http://www.cc.gatech.edu/gvu/people/jarek.rossignac

Jarek Rossignac is Professor in the College of Computing at Georgia Institute of Technology and the Director of GVU, Georgia Tech's Graphics, Visualization, and Usability Center. Prior to joining Georgia Tech, Jarek was the strategist for Visualization and the Senior Manager of the Visualization, Interaction, and Graphics department at IBM Research, where he managed research groups involved in 3D modeling, design review, scientific visualization, medical imaging, and VR. His research interests focus on 3D geometric modeling and on interactive and intuitive techniques for collaborative 3D design and inspection. Jarek co-invented simplification and 3D compression techniques currently used in IBM's 3D Interaction Accelerator, an interactive viewer for the collaborative inspection of highly complex 3D CAD models, which he managed, along with two other IBM visualization products. Jarek holds a PhD in EE from the University of Rochester, New York in the area of Solid Modeling.

William (Will) J. Schroeder
Computational Scientist
GE Corporate R&D Center, KW-C219
1 Research Circle
Niskayuna, NY 12309

Email: schroeder@crd.ge.com
Web: http://www.crd.ge.com/∼schroede/

Will Schroeder is a computational scientist at GE's Research & Development Center. He has designed the object-oriented VISAGE visualization system used throughout GE. Will's contributions to the visualization field include the decimation polygon reduction algorithm, the stream polygon for vector/tensor visualization, and swept surfaces for motion representation. Dr. Schroeder received a BS in mechanical engineering at the University of Maryland, and MS and PhD in applied mathematics at Rensselaer Polytechnic Institute.

Marc Soucy
President
InnovMetric Software Inc.
2065 Charest Ouest, Suite 218
Ste-Foy, Quebec
CANADA G1N 2G1

Email: msoucy@innovmetric.com
Web: http://www.innovmetric.com

Marc Soucy is President and R&D Director at InnovMetric Software Inc. He has designed and supervised the development of POLYWORKS, an integrated suite of software tools for building 3-D models from 3-D digitizer data. His research interests include registration and integration of 3-D data obtained from multiple viewpoints, decimation of large polygonal models, and the use of texture mapping for creating compact geometric representations that can be displayed in real-time. Marc Soucy received the BSc and PhD degrees in Electrical Engineering from Laval University, Quebec, Canada, in 1988 and 1992 respectively.

Amitabh Varshney
Assistant Professor
Dept. of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400, USA

Email: varshney@cs.sunysb.edu
Web: http://www.cs.sunysb.edu/∼varshney/

Amitabh Varshney is an Assistant Professor of Computer Science at the State University of New York at Stony Brook. Varshney's research focus is on exploring the applications of virtual reality in engineering, science, medicine, and commerce. His research interests are in three-dimensional interactive graphics, geometric modeling, molecular graphics, and scientific visualization. Varshney received a B. Tech. in Computer Science from the Indian Institute of Technology, Delhi in 1989 and a M.S. and Ph.D. in Computer Science from the University of North Carolina at Chapel Hill in 1991 and 1994.

Survey of Polygonal Surface Simplification Algorithms

Paul S. Heckbert and Michael Garland

1 May 1997

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Multiresolution Surface Modeling Course
SIGGRAPH '97

This is a draft of a Carnegie Mellon University technical report, to appear. See http://www.cs.cmu.edu/∼ph for final version.

Send comments or corrections to the authors at: {ph,garland}@cs.cmu.edu

Abstract

This paper surveys methods for simplifying and approximating polygonal surfaces. A polygonal surface is a piecewise-linear surface in 3-D defined by a set of polygons; typically a set of triangles. Methods from computer graphics, computer vision, cartography, computational geometry, and other fields are classified, summarized, and compared both practically and theoretically. The surface types range from height fields (bivariate functions), to manifolds, to non-manifold self-intersecting surfaces. Piecewise-linear curve simplification is also briefly surveyed.

This work was supported by ARPA contract F19628-93-C-0171 and NSF Young Investigator award CCR-9357763.

Keywords: multiresolution modeling, surface approximation, piecewise-linear surface, triangulated irregular network, mesh coarsening, decimation, non-manifold, cartographic generalization, curve simplification, level of detail, greedy insertion.

Contents

1 Introduction

The simplification of surfaces has become increasingly important as it has become possible in recent years to create models of greater and greater detail. Detailed surface models are generated in a number of disciplines. For example, in computer vision, range data is captured using scanners; in scientific visualization, isosurfaces are extracted from volume data with the "marching cubes" algorithm; in remote sensing, terrain data is acquired from satellite photographs; and in computer graphics and computer-aided geometric design, polygonal models are generated by subdivision of curved parametric surfaces. Each of these techniques can easily generate surface models consisting of millions of polygons.

Simplification is useful in order to make storage, transmission, computation, and display more efficient. A compact approximation of a shape can reduce disk and memory requirements and can speed network transmission. It can also accelerate a number of computations involving shape information, such as finite element analysis, collision detection, visibility testing, shape recognition, and display. Reducing the number of polygons in a model can make the difference between slow display and real time display.

A variety of methods for simplifying curves and surfaces have been explored over the years. Work on this topic is spread among a number of fields, making literature search quite challenging. These fields include: cartography, geographic information systems (GIS), virtual reality, computer vision, computer graphics, scientific visualization, computer-aided geometric design, finite element methods, approximation theory, and computational geometry.

Some prior surveys of related methods exist, notably a bibliography on approximation [45], a survey of spatial data structures for curves and surfaces [106], and surveys of triangulation methods with both theoretical [6] and scientific visualization [89] orientations. None of these surveys surface simplification in depth, however.

The present paper attempts to survey all previous work on surface simplification and place the algorithms in a taxonomy. In this taxonomy, we intermix algorithms from various fields, classifying algorithms not according to the application for which they were designed, but according to the technical problem they solve. By doing so, we find great similarities between algorithms from disparate fields. For example, we find common ground between methods for representing terrains developed in cartography, methods for approximating bivariate functions developed in computational geometry and approximation theory, and methods for approximating range data developed in computer vision. This is not too surprising, since these are fundamentally the same technical problem. By calling attention to these similarities, and to the past duplication of work, we hope to facilitate cross-fertilization between disciplines.

Our emphasis is on methods that take polygonal surfaces as input and produce polygonal surfaces as output, although we touch on curved parametric surface and volume techniques. Our polygons will typically be planar triangles. Although surface simplification is our primary interest, we also discuss curve simplification, because many surface methods are simple generalizations of curve methods.

1.1 Characterizing Algorithms

Methods for simplifying curves and surfaces vary in their generality and approach – among surface methods, some are limited to height fields, for example, while others are applicable to general surfaces in 3-D. To systematize our taxonomy, we will classify methods according to the problems that they solve and the algorithms they employ. Below is a list of the primary characteristics with which we will do so:

Problem Characteristics

Topology and Geometry of Input: For curves, the input can be a set of points, a function $y(x)$, a planar curve, or a space curve. For surfaces, the input can be a set of points, samples of a height field $z(x, y)$ in a regular grid or at scattered points, a manifold[1], a manifold with boundary, or a set of surfaces with arbitrary topology (e.g. a set of intersecting polygons).

Other Attributes of Input: Color, texture, and surface normals might be provided in addition to geometry.

Domain of Output Vertices: Vertices of the output can be restricted to be a subset of the input points, or they can come from the continuous domain.

Structure of Output Triangulation: Meshes can be regular grids, they can come from a hierarchical subdivision such as a quadtree, or they can be a general subdivision such as a Delaunay or data-dependent triangulation.

Approximating Elements: The approximating curve or surface elements can be piecewise-linear (polygonal), quadratic, cubic, high degree polynomial, or some other basis function.

Error Metric: The error of the approximation is typically measured and minimized with respect to L_2 or L_∞ error[2]. Distances can be measured in various ways, e.g., to the closest point on a given polygon, or closest point on the entire surface.

Constraints on Solution: One might request the most accurate approximation possible using a given number of elements (e.g. line segments or triangles), or one might request the solution using the minimum number of elements that satisfies a given error tolerance. Some algorithms give neither type of guarantee, but give the user only indirect control over the speed/quality tradeoff. Other possible constraints include limits on the time or memory available.

Algorithm Characteristics

Speed/Quality Tradeoff: Algorithms that are optimal (minimal error and size) are typically slow, while algorithms that generate lower quality or less compact approximations can generally be faster.

Refinement/Decimation: Many algorithms can be characterized as using either *refinement*, a coarse-to-fine approach starting with a minimal approximation and building up more and more accurate ones, or *decimation*, a fine-to-coarse approach starting with an exact fit, and discarding details to create less and less accurate approximations.

1.2 Background on Application Areas

The motivations for surface simplification differ from field to field. Terminology differs as well.

[1] A *manifold* is a surface for which the infinitesimal neighborhood of every point is topologically equivalent to a disk. In a triangulated manifold, each edge belongs to two triangles. In a triangulated *manifold with boundary*, each edge belongs to one or two triangles.

[2] In this paper, we use the following error metrics: We define the L_2 error between two n-vectors $\mathbf{u}$ and $\mathbf{v}$ as $||\mathbf{u}-\mathbf{v}||_2 = \left[\sum_{i=1}^{n}(u_i - v_i)^2\right]^{1/2}$. The L_∞ error, also called the *maximum error*, is $||\mathbf{u}-\mathbf{v}||_\infty = \max_{i=1}^{n}|u_i - v_i|$. We define the *squared error* to be the square of the L_2 error, and the *root mean square* or RMS error to be the L_2 error divided by $\sqrt{n}$. Optimization with respect to the L_2 and L_∞ metrics are called *least squares* and *minimax* optimization, and we call such solutions L_2–*optimal* and L_∞–*optimal*, respectively.

Cartography. In cartography, simplification is one method among many for the "generalization" of geographic information [86]. In that field, curve simplification is called "line generalization". It is used to simplify the representations of rivers, roads, coastlines, and other features when a map with large scale is produced. It is needed for several reasons: to remove unnecessary detail for aesthetic reasons, to save memory/disk space, and to reduce plotting/display time. The principal surface type simplified in cartography is, of course, the terrain. Map production was formerly a slow, off-line activity, but it is currently becoming more interactive, necessitating the development of better simplification algorithms.

The ideal error measures for cartographic simplification include considerations of geometric error, viewer interest, and data semantics. Treatment of the latter issues is beyond the scope of this study. The algorithms summarized here typically employ a geometric error measure based on Euclidean distance. The problem is thus to retain features larger than some size threshold, typically determined by the limits of the viewer's perception, the resolution of the display device, or the available time or memory.

Computer Vision. Range data acquired by stereo or structured light techniques (e.g. lasers) can easily produce millions of data points. It is desirable to simplify the surface models created from this data in order to remove redundancy, save space, and speed display and recognition tasks. The acquired data is often noisy, so tolerance of and smoothing of noise are important considerations here.

Computer Graphics. In computer graphics and the closely related fields of virtual reality, computer-aided geometric design, and scientific visualization, compact storage and fast display of shape information are vital. For interactive applications such as military flight simulators, video games, and computer-aided design, real time performance is a very important goal. For such applications, the geometry can be simplified to multiple levels of detail, and display can switch or blend between the appropriate levels of detail as a function of the screen size of each object [13, 52]. This technique is called *multiresolution modeling*. Redisplaying a static scene from a moving viewpoint is often called a *walkthrough*. For off-line, more realistic simulations such as special effects in entertainment, real time is not vital, but reasonable speed and storage are nevertheless important.

When 3-D shape models are transmitted, compression is very important. This applies whether the channel has very low bandwidth (e.g. a modem) or higher bandwidth (e.g. the Internet backbone). The rapid growth of the World Wide Web is spurring some of the current work in surface simplification.

Finite Element Analysis. Engineers use the finite element method for structural analysis of bridges, to simulate the air flow around airplanes, and to simulate electromagnetic fields, among other applications. A preprocess to simulation is a "mesh generation" step. In 2-D mesh generation, the domain, bounded by curves, is subdivided into triangles or quadrilaterals. In 3-D mesh generation, the domain is given by boundary surfaces. Surface meshes of triangles or quadrilaterals are first constructed, and then the volume is subdivided into tetrahedra or hexahedra. The criteria for a good mesh include both geometric fidelity and considerations of the physical phenomena being simulated (stress, flow, etc). To speed simulation, it is desirable to make the mesh as coarse as possible while still resolving the physical features of interest. In 3-D simulations, surface details such as bolt heads might be eliminated, for example, before meshing the volume. This community calls simplification "mesh coarsening".

Approximation Theory and Computational Geometry. What is called a terrain in cartography or a height field in computer graphics is called a bivariate function or a function of two variables in more theoretical fields. The goal in approximation theory is often to characterize the error in the limit as the mesh becomes infinitely fine. In computational geometry the goal is typically to find algorithms to generate approximations with optimal or near-optimal compactness, error, or speed or to prove bounds on these. Implementation of algorithms and low level practical optimizations receive less attention.

2 Curve Simplification

Curve simplification has been used in cartography, computer vision, computer graphics, and a number of other fields.

A basic curve simplification problem is to take a polygonized curve with n vertices (a chain of line segments or "polyline") as input and produce an approximating polygonized curve with m vertices as output. A closely related problem is to take a curve with n vertices and approximate it within a specified error tolerance.

Douglas-Peucker Algorithm. The most widely used high-quality curve simplification algorithm is probably the heuristic method commonly called the Douglas-Peucker[3] algorithm. It was independently invented by many people [99], [31], [30, p. 338], [5, p. 92], [125], [91, p. 176], [3]. At each step, the Douglas-Peucker algorithm attempts to approximate a sequence of points by a line segment from the first point to the last point. The point farthest from this line segment is found, and if the distance is below threshold, the approximation is accepted, otherwise the algorithm is recursively applied to the two subsequences before and after the chosen point. This algorithm, though not optimal, has generally been found to produce the highest subjective- and objective-quality approximations when compared with many other heuristic algorithms [85, 130]. Its best case time cost[4] is $\Omega(n)$, its worst case cost is $O(mn)$, and its expected time cost is about $\Theta(n \log m)$. The worst case behavior can be improved, with some sacrifice in the best case behavior, using a $\Theta(n \log n)$ algorithm employing convex hulls [54].

A variant of the Douglas-Peucker algorithm described by Ballard and Brown [4, p. 234] on each iteration splits at the point of highest error along the whole curve, instead of splitting recursively. This yields higher quality approximations for slightly more time. If this subdivision tree is saved, it is possible to dynamically build an approximation for any larger error tolerance very quickly [18].

A potential problem is that simplification can cause a simple polygon to become self-intersecting. This could be a problem in cartographic applications.

Faster or Higher Quality Algorithms. There are faster algorithms than Douglas-Peucker, but all of these are generally believed to have inferior quality [84]. One such algorithm is the trivial method of *regular subsampling* (known as the "nth-point algorithm" in cartography), which simply keeps every kth point of the input, for some k, discarding the rest. This algorithm is very fast, but will sometimes yield very poor quality approximations.

Least squares techniques are commonly used for curve fitting in pattern recognition and computer vision, but they do not appear to be widely used for that purpose in cartography.

Polygonal Boundary Reduction. While the Douglas-Peucker algorithm and its variants are refinement algorithms, curves can also be simplified using decimation methods. Boxer *et al.* [8] describe two such algorithms for simplifying 2-D polygons. The first, due to Leu and Chen [75], is a simple decimation algorithm. It considers boundary arcs of 2 and 3 edges. For each arc, it computes the maximum distance between the arc and the chord connecting its endpoints. It then selects an independent set of arcs whose deviation is less than some threshold, and replaces them by their chords. The second algorithm is an improvement of this basic algorithm which guarantees that the approximate curve is always within some bounded distance from the original. They state that the running time of the simple algorithm is $\Theta(n)$, while the bounded-error algorithm requires $O(n + r^2)$ time where r is the number of vertices removed.

Optimal Approximations. Algorithms for optimal curve simplification are much less common than heuristic methods, probably because they are slower and/or more complicated to implement. In a common form of optimal curve simplification, one searches for the approximation of a given size with minimum error, according to some

[3] Pronounced, and later spelled, due to name change, "Poiker".

[4] A function is in $O(f(n))$ if it is less than or equal to $cf(n)$ as $n \to \infty$, for some positive constant c. "O" is used for upper bounds.
A function is in $\Theta(f(n))$ if it is between $c_1 f(n)$ and $c_2 f(n)$ as $n \to \infty$, for some positive constants c_1, c_2.
A function is in $\Omega(f(n))$ if it is greater than or equal to $cf(n)$ as $n \to \infty$, for some positive constant c. "Ω" is used for lower bounds.

definition of "error". Typically the output vertices are restricted to be a subset of the input vertices. A naive, exhaustive algorithm would have exponential cost, since the number of subsets is exponential, but using dynamic programming and/or geometric properties, the cost can be reduced to polynomial. The L_2–optimal approximation to a function $y(x)$ can be found in $O(mn^2)$ time, worst case, using dynamic programming. Remarkably, a slight variation in the error metric permits a much faster algorithm: the L_∞–optimal approximation to a function can be found in $O(n)$ time [63], using visibility techniques (see also [123, 91]). When the problem is generalized from functions to planar curves, the complexity of the best L_∞–optimal algorithms we know of jumps to $O(n^2 \log n)$ [63]. These methods use shortest-path graph algorithms or convex hulls. For space curves (curves in 3-D), there are $O(n^3 \log m)$ L_∞–optimal algorithms [62].

Asymptotic Approximation. In related work, McClure and de Boor analyzed the error when approximating a highly continuous function $y(x)$ using piecewise-polynomials with variable knots [82, 21]. We discuss only the special case of piecewise-linear approximations. They analyzed the asymptotic behavior of the L_p error of approximation in the limit as m, the number of vertices (knots) of the approximation, goes to infinity. They showed that the asymptotic L_p error with regular subsampling is proportional to m^{-2}, for any p. The L_p–optimal approximation has the same asymptotic behavior, though with a smaller constant. McClure showed that the spacing of vertices in the optimal approximation is closely related to the function's second derivative. Specifically, he proved that as $m \to \infty$, the density of vertices at each point in the optimal L_2 approximation becomes proportional to $|y''(x)|^{2/5}$. For optimal L_∞ approximations, the density is proportional to $|y''(x)|^{1/2}$. Also, as $m \to \infty$, all intervals have equal error in an L_p–optimal approximation.

The density property and the balanced error property described above can be used as the basis for curve simplification algorithms [82]. Although adherence to neither property guarantees optimality for real simplification problems with finite m, iterative balanced error methods have been shown to generate good approximations in practice [91, p. 181]. Another caveat is that many curves in nature do not have continuous derivatives, but instead have some fractal characteristics [80]. Nevertheless, these theoretical results suggest the importance of the second derivative, and hence curvature, in the simplification of curves and surfaces.

Summary of Curve Simplification. The Douglas-Peucker algorithm is probably the most commonly used curve simplification algorithm. Most implementations have $O(mn)$ cost, worst case, but typical cost of $\Theta(n \log m)$. An optimization with worst case cost of $O(n \log n)$ is available, however. Optimal simplification typically has quadratic or cubic cost, making it impractical for large inputs.

3 Surface Simplification

Surfaces are more difficult to simplify than curves. In the flight simulator field, lower level of detail models have traditionally been prepared by hand [17]. The results can be excellent, but the process can take weeks. Automatic methods are preferable for large and dynamic databases, however.

If only a single level of detail is needed, then in some cases, simplification can be obviated by simply avoiding generation of redundant data in the first place. In scientific visualization, for example, the marching cubes algorithm [90] is widely used. It generates many tiny triangles without testing for coplanarity between neighbors. A more sophisticated alternative is adaptive polygonization that subdivides finely only where the surface is highly curved [7]. In computer aided geometric design, when polygonizing parametric surfaces, rather than subdivide and polygonize a surface with a regular (u, v) grid, better results are often possible by subdividing adaptively based on curvature [10].

When simplification is needed however, one of the algorithms summarized below can be used.

Taxonomy of Surface Simplification Algorithms. We categorize algorithms at the highest level according to the class of surfaces on which they operate:

- height fields and parametric surfaces,

- manifold surfaces, and
- non-manifold surfaces.

Within each surface class we often group algorithms according to whether they work by refinement or decimation. Within the subclasses, methods are generally listed chronologically. We have attempted to be fairly comprehensive, so consequently the good methods are described along with the bad. As we summarize algorithms, we list their computational complexities and quote empirical times, where known (of course, hardware, compilers, languages, and programming styles differ between individuals, so we must be careful when judging based on this information). Complexities are given in terms of n, the number of vertices in the input, and m, the number of vertices in the output. Typically, $m \ll n$.

3.1 Height Fields and Parametric Surfaces

Height fields and parametric surfaces are the simplest class of surfaces. Within this class of surfaces, we divide methods into the following six sub-classes: regular grid methods, hierarchical subdivision methods, feature methods, refinement methods, decimation methods, and optimal methods.

Regular grid methods are the simplest techniques, using a grid of samples equally and periodically spaced in x and y. The hierarchical subdivision methods are based on quadtree, k-d tree, and hierarchical triangulations using a divide and conquer strategy. They recursively subdivide the surface into regions, constructing a tree-structured hierarchy. The next four categories employ more general subdivision and triangulation methods, most commonly Delaunay triangulation. Feature methods select a set of important "feature" points in one pass and use them as the vertex set for triangulation. Refinement methods are essentially generalizations of the Douglas-Peucker algorithm from curves to surfaces, where intervals are replaced by triangles and splitting is replaced by retriangulating. They start with a minimal approximation and use multiple passes of point selection and retriangulation to build up the final triangulation. Decimation methods use an approach opposite that of refinement methods: they begin with a triangulation of all of the input points and iteratively delete vertices from the triangulation, gradually simplifying the approximation. Refinement methods thus work top-down, while decimation methods work bottom-up. The final category, "optimal methods" are distinguished more for their theoretical focus than for their method.

For many height field simplification tasks, the input is a height field and the output is a general triangulation, called a *triangulated irregular network*, or TIN, in cartography. A TIN is a mesh of triangles where height is a function of x and y: $H(x, y)$. Examples of a height field and general triangulation are shown in Figures 1 and 2.

3.1.1 Triangulation

Most polygonal surface simplification methods employ triangles as their approximating elements when constructing a surface. For height fields and parametric surfaces, there is a natural 2-D parameterization of the surface. Basic triangulation methods are described in a 2-D domain, or in a 3-D domain where height z is a function of x and y.

In general, the topology of the triangulation can be chosen using only the xy projections of the input points, or it can be chosen using the heights of the input points as well. The latter approach is called data-dependent triangulation [32].

The most popular triangulation method that does not use height values is Delaunay triangulation; it is a purely two-dimensional method. Delaunay triangulation finds the triangulation that maximizes the minimum angle of all triangles, among all triangulations of a given point set. This helps to minimize the occurrence of very thin *sliver* triangles. Delaunay triangulations have a number of nice theoretical properties that make them very popular in computational geometry. In a Delaunay triangulation, the circumscribing circle (circumcircle) of each triangle contains no vertices in its interior [71]. Delaunay triangulations of m points can either be computed whole, using divide-and-conquer or sweepline algorithms, or incrementally, by inserting vertices one at a time, updating the triangulation after each insertion [48]. The former approach has cost $O(m \log m)$, while the latter, incremental method has worst case cost of $O(m^2)$. Typical costs for the incremental approach are much better than quadratic, however.

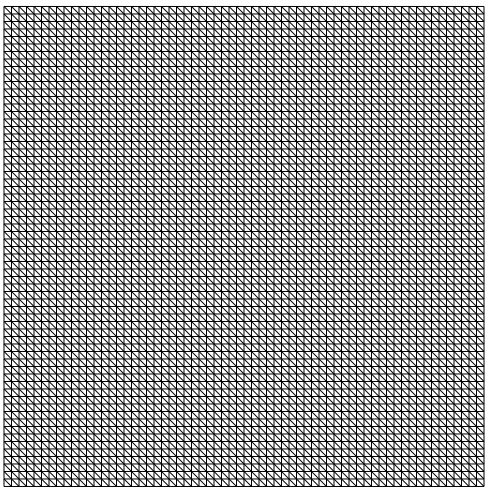


Figure 1: Top view of a regular grid triangulation of 65×65 height field.

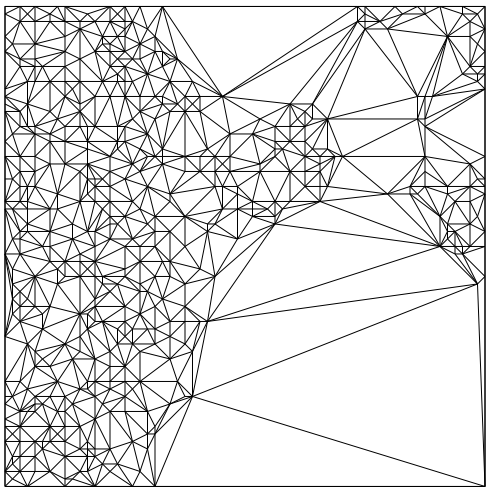


Figure 2: A triangulation using 512 vertices approximating the height field.

Sometimes equilateral triangles are not optimal, and maximization of the minimum angle is not the appropriate goal. Triangulation methods that attempt to optimize the approximation of z or other data associated with the triangulation are called *data-dependent triangulation* methods. Several researchers have shown that slivers can be good when the surface being approximated is highly curved in one direction, but not the other [102, 88, 20, 32]. Such slivers would not be generated by Delaunay triangulation, which minimizes slivers by tending to choose "fat" triangles.

3.1.2 Regular Grid Methods

The simplest method for approximation of surface grids is regular subsampling, in which the points in every kth row and column are kept and formed into a grid, and all other points are discarded. Regular grids are also known as uniform grids, and sometimes the term *DEM* (digital elevation model) is used in the specific sense of a regular grid terrain model. As with curves, regular subsampling is simple and fast, but low quality, since the points discarded might be the most important ones. The results are improved if a low pass filter [51] is run across the data before subsampling, but this still does not fix the basic problem with this method, its non-adaptive nature.

Kumler 94. An extensive comparison of regular grids (DEMs) and general triangulations (TINs), and the space/error tradeoffs between them, was done by Kumler [70]. He concluded, surprisingly, that for a given amount of storage space, regular grids approximate terrains better than general triangulations. His comparison seems biased against general triangulations, however, since he compares models of equal memory size, not equal rendering time, and the simplification algorithms he uses are not the best known [39]. Kumler assumes that general triangulations require three to ten times the memory of regular grids with the same number of vertices.

Pyramids. Regular subsampling can be done hierarchically, forming a pyramid of samples [131, 121]. Despite the wealth of research on hierarchical triangulations and TINs, pyramids are probably the most widely used type of multiresolution terrain model in the simulator commu-

nity [16, 17] and in the visualization/animation community [60], because of their simplicity and compactness.

3.1.3 Hierarchical Subdivision Methods

Hierarchical subdivision methods construct a triangulation by recursively subdividing a surface. They are the adaptive form of pyramids. The hierarchical pattern of subdivision, even if not stored explicitly in the data structure, forms a tree, each node of which has no more than one parent. With Delaunay triangulation and other general triangulation algorithms, the topology is not hierarchical, because a triangle might have multiple parents. Hierarchical subdivision methods are generally fast, simple, and they facilitate multiresolution modeling. In perspective scenes where nearby portions of the terrain require more detail than distant regions, the hierarchy facilitates rendering at adaptive levels of detail. Nearby portions are drawn at at a fine level, while distant regions are drawn at a coarse level. The penalty for their simplicity and speed is that hierarchical subdivision methods typically yield poorer quality approximations than more general triangulation methods.

Quadtrees and k-d Trees. Terrains and parametric surfaces are easily simplified using adaptive quadtree and k-d tree subdivision methods [106, 105]. DeHaemer and Zyda used quadtree and k-d tree splitting at the point of maximum error within each cell to approximate general 3-D surfaces described by a grid [27]. Taylor and Barrett described a similar method for terrains [122]. Von Herzen and Barr discuss a method for crack-free adaptive triangulation of parametric surfaces using quadtrees [128]. Gross, Gatti, and Staadt have used wavelets to construct quadtree approximations of height fields [44]. With an unoptimized implementation, they were able to simplify a 256×256 terrain in about 2 seconds on an SGI Indy.

Gómez-Guzmán 79. A *quaternary triangulation* method for height field approximation was proposed by Gómez and Guzmán [42]. In their method, each triangle is recursively subdivided into four subtriangles until a maximum error tolerance is met. To subdivide each triangle, a "significant" point near the midpoint of each edge is chosen (in some unspecified way), and the triangle is split into four nearly congruent triangles (Figure 3). Since the new vertices are not constrained to lie on the edges, however, the surface develops unsightly cracks, rendering the method unsuitable for most purposes.

De Floriani-Falcidieno-Nagy-Pienovi 84. In 1984, De Floriani *et al.* published a hierarchical *ternary triangulation* method in which points are inserted in triangle interiors and each triangle is split into three subtriangles by adding edges to its vertices [23]. No edge swapping is done (Figure 4). Consequently, all of the initial edges remain in the triangulation forever, most notably the diagonal across the entire grid rectangle, leading to spurious knife-edge ridges and valleys through the terrain. The flaws of this method make it unacceptable.

Schmitt 85. Schmitt and Gholizadeh simplified a grid with rectangular topology in 3-D using a triangulated surface [112]. Their method is similar to that of Faugeras *et al.* [34], described later. Having the input points in a grid allows the partition of points into triangles to be done in a two dimensional parametric space. The method begins with a small number of triangles and repeatedly splits those triangles whose associated input points are above the error tolerance. Triangles are subdivided into 2–4 subtriangles by splitting one, two, or three edges of the triangle. Triangle splitting is done in no particular order. They report that simplifying a grid of $n = 288 \times 360$ points down to about $m = 3,500$ points takes 1.5 hours on a DEC VAX 780. The computational complexity of their algorithm is $O(mn)$.

Scarlatos-Pavlidis 92a. The hierarchical triangulation algorithm for height fields developed by Scarlatos and Pavlidis employs a recursive triangulation approach [108, 107]. Their method begins with a minimal triangulation (typically two triangles) as level of detail 0. Error tolerances for each level of detail in the tree are specified by the user. To create level i from level $i-1$, the point of highest error along each triangle edge and in each triangle interior is found, those points with error above the threshold for level i are taken as new vertices, and each triangle is retriangulated using one of five simple subdivision templates (Figure 5). Passes of vertex selection and retriangulation

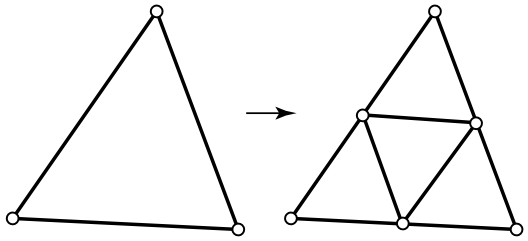


Figure 3: Quaternary triangulation.

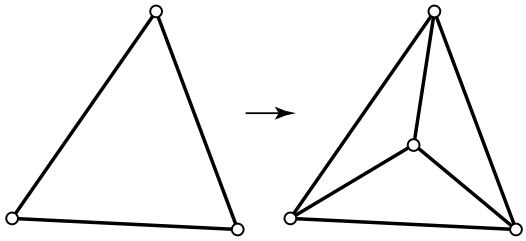


Figure 4: Ternary triangulation.

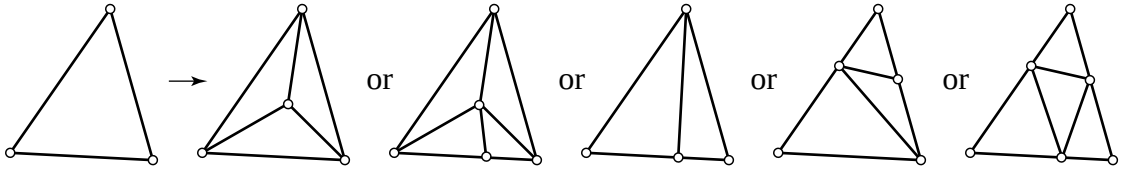


Figure 5: Subdivision templates for Scarlatos and Pavlidis' hierarchical triangulation.

for level i are repeated until no more candidates for that level are found. All levels of the hierarchy are retained in the data structure, facilitating adaptive display at any desired detail level. In their analysis, Scarlatos and Pavlidis suggest that the cost of the algorithm is $O(n \log n)$. Our analysis of their algorithm is that their expected case is $O(n \log m)$, but if the hierarchy is very unbalanced, the worst case cost is $O(mn)$.

De Floriani-Puppo 92. A similar method was developed by De Floriani and Puppo [26]. The triangle subdivision is more general, however. To subdivide a triangle for a given level in the hierarchy, a curve approximation algorithm [4] is used to add new vertices along the edges, then additional points are inserted in the interior of the triangle until the error threshold is met throughout the triangle, and the interior of the triangle is retriangulated using Delaunay triangulation. The method appears to have nearly identical flexibility and speed compared to Scarlatos and Pavlidis' method [108], but it will probably yield slightly better simplification for a given error threshold.

3.1.4 Feature Methods

A simple, intuitive approach to height field simplification is to make one pass over the input points, ranking each of them using some "importance" measure, to select the most important points as the vertex set, and construct a triangulation of these points. Typically, Delaunay triangulation is used. Feature methods are quite popular in cartography. Overall, our conclusion is that their quality relative to many of the other methods is inferior, so we only survey them briefly here.

Important points, also known as "features" or "critical points" and the edges between them, often known as "break lines", include such topographic features as peaks, pits, ridges, and valleys. The philosophy of many of the feature approaches is that some knowledge about the nature of terrains is essential for good simplification [129, 108]. In a feature approach, the chosen features become the vertex set, and the chosen break lines (if any) become edges in a constrained triangulation [6]. The most commonly used feature detectors are 2×2 and 3×3 linear or nonlinear filters, sometimes followed by a weeding process that discards features that are too close together,

such as a sequence of points along a ridge line. Such approaches were employed by Peucker-Douglas and Chen-Guevara [92, 12]. Some methods examine larger neighborhoods of points in an attempt to measure importance more globally.

Southard 91. One of the more interesting feature methods is Southard's [120]. He uses the Laplacian as a measure of curvature. The rank of each point's Laplacian is computed within a moving window, analogous to a median filter in image processing, and all points whose rank is below some threshold are selected. This is an improvement over the selection criteria of Peucker-Douglas and Chen-Guevara cited earlier, because it is less susceptible to noise and high frequency variations, but unfortunately, Southard's ranking approach tends to distribute points roughly uniformly across the domain, wasting points and leading to inferior approximations, in many cases. After computing the Delaunay triangulation of the selected points, Southard performs a data-dependent retriangulation, swapping edges where that would reduce the sum of the absolute errors along the edges in the triangulation.

3.1.5 Refinement Methods

Refinement methods are multi-pass algorithms that begin with a minimal initial approximation, on each pass they insert one or more points as vertices in the triangulation, and repeat until the desired error is achieved or the desired number of vertices is used. For input data in a rectangular grid, the minimal approximation is two triangles; for other topologies, the initial approximation might be more complex. Incremental methods are typically used to maintain the triangulation as refinement proceeds.

To choose points, importance measures much like those of the feature methods can be used. Whereas feature methods typically use importance measures that are independent of the approximation, in refinement algorithms, the importance of a given point is usually a measure of the error between it and the approximation. For a height field, the most common metric for the error is simply the maximum absolute value of the vertical error, the L_∞ norm. This is the error measure most closely related to the Douglas-Peucker algorithm.

Greedy Insertion. We call refinement algorithms that insert the point(s) of highest error on each pass *greedy insertion* algorithms, "greedy" because they make irrevocable decisions as they go [15], and "insertion" because on each pass they insert one or more vertices into the triangulation. Methods that insert a single point in each pass we call *sequential greedy insertion* and methods that insert multiple points in parallel on each pass we call *parallel greedy insertion*. The words "sequential" and "parallel" here refer to the selection and re-evaluation process, not to the architecture of the machine. Many variations on the greedy insertion algorithm have been explored over the years; apparently the algorithm has been reinvented many times.

Fowler-Little 79. In 1979, Fowler and Little published a hybrid algorithm that uses an initial pass of feature selection using 2×2 filters to "seed" the triangulation, followed by multiple passes of parallel greedy insertion [37]. On each of these latter passes, for each triangle, the point with highest error, or *candidate point*, is found, and all candidate points whose error is above the requested threshold are inserted into the triangulation. (When the point of highest error falls on an edge, they expand their search for the candidate to a sector of the triangle's circumcircle, a quirk unique to their algorithm.)

Fowler and Little discussed two methods for finding candidates. In their exhaustive search method, the error at each input point is computed and tested against the highest error seen so far for that triangle. In the initial passes of a greedy insertion method, the triangles are big, necessitating the testing of many points, but in later passes the triangles shrink and less testing per triangle is required. As a way to speed the selection of candidates, they propose an alternative method using hill-climbing, in which a test point is initialized to the center of the triangle, and it repeatedly steps to the neighboring input point of highest error until it reaches a local maximum, where it becomes the candidate. This latter method can be much faster, especially for the initial passes, but it would also yield poorer quality approximations in many cases, because the hill climbing might fail to find the global maximum within the

triangle. Unfortunately, Fowler and Little did not show a comparison of the results of the two methods, and did not analyze the speed of their algorithm. An approach similar to Fowler and Little's was very briefly described by Lee and Schachter [72].

De Floriani-Falcidieno-Pienovi 83. In 1983, De Floriani *et al.* presented a sequential greedy insertion algorithm [24, 25]. Their method is purer than Fowler and Little's: it does not seed the triangulation using feature points, and it inserts a single point on each pass, not multiple points. Consequently, the quality of its approximations can be higher than Fowler and Little's. The point inserted in each pass is the point of highest absolute error from the input point set. To find this point they apparently visit all input points on each pass, computing errors. Their paper says that their algorithm has worst case cost of $O(n^2)$, but too few details of the algorithm or its data structures are provided to verify this. We will refer to this paper and algorithm as "DeFloriani83".

De Floriani 89. In later work, De Floriani published an algorithm to build a "Delaunay pyramid" [22], a hierarchy of Delaunay triangulations, using a variant of her 1983 greedy insertion algorithm to construct each level of the pyramid. Her 1989 paper describes the greedy insertion algorithm in greater detail than her earlier papers ([24, 25]).

Each triangle stores the set of input points it contains and the error of its candidate point. On each pass, the set of triangles is scanned to find the candidate of highest error, this point is inserted using incremental Delaunay triangulation, and the candidates of all the triangles in the modified region are recomputed. Recomputing the candidate of a triangle requires calculating the error at each point in the triangle's point set.

De Floriani states that the worst case time cost to create a complete pyramid of all n points is $O(n^2)$. We believe that the expected time cost of her algorithm, to select and triangulate m points, is $O(n \log m + m^2)$ (compare to Algorithm III in [40]).

Because point set traversal is used, rather than triangle scan conversion [36], this algorithm is not limited to input points in a regular grid, as are most height field approximation algorithms. The price of this generality is speed; the inner loops of a set traversal method cannot be optimized as much as those of a scan conversion approach.

Heller 90. Heller explored a hybrid technique that he called "adaptive triangular mesh filtering" [53, p. 168]. This technique is much like Fowler and Little's. The principal difference is that the features are chosen not with a fixed-size local filter but by checking a variable-sized neighborhood to determine if each point is a local extremum within some height threshold. This feature selection method, while more expensive than Fowler and Little's, probably yields higher quality approximations.

His insertion method is sequential, like that of DeFloriani83. He optimizes the algorithm by storing the set of candidates, one candidate from each triangle, in a heap[5]. Below is an excerpt of Heller's brief explanation of his algorithm [53, p. 168]:

> The [insertion] of a point requires a local retriangulation which consists of swapping all necessary triangles, and readjusting the [importances] of all affected points. It is clear that the time for retriangulation is proportional to the number of readjusted points and the logarithm of the number of queued points. It is, therefore, advisable to start the process with as many [feature] points as possible.

Due to his optimizations, Heller's algorithm is probably faster than most others of comparable quality, such as DeFloriani83, but unfortunately, beyond the statements quoted above he does not analyze the speed of his algorithm theoretically or empirically. It appears that the expected complexity of the greedy insertion portion of his algorithm is $O((m+n)\log m)$, like Algorithm III in [40].

Schmitt-Chen 91. In order to segment computer vision range data into planar regions, Schmitt and Chen use a two stage process called *split-and-merge* [110, 91]. The

[5] Christoph Witzgall has also employed a heap. Personal communication. 1994.

splitting stage is a form of greedy insertion with Delaunay triangulation similar to DeFloriani83. The merging stage joins together adjacent regions with similar normals, in the process destroying the triangulation, but yielding a segmentation of the image. Their splitting stage approximated a height field with $n = 256^2$ points using about $m = 3,060$ vertices in 67 seconds on a DEC VAX 8550.

Scarlatos-Pavlidis 92a and De Floriani-Puppo 92. The hierarchical triangulation methods of Scarlatos-Pavlidis [108] and De Floriani-Puppo [26] discussed earlier are analogous to greedy insertion in many ways, although their triangulations are quite different. Their techniques will typically use more triangles to achieve a given error than sequential greedy insertion with Delaunay triangulation, but on the other hand, they have the advantage of a hierarchy.

Rippa 92. Rippa generalized the greedy insertion algorithm of DeFloriani83 to explore data-dependent triangulation and least squares fitting [101].

In place of incremental Delaunay triangulation, Rippa's algorithm computes a data-dependent triangulation using a version of Lawson's local optimization procedure [71], repeatedly swapping edges around a new vertex until the global error reaches a local minimum. He tested two definitions of global error. The first is a purely geometric measure: the sum of the absolute values of the angles between normals of all pairs of adjacent triangles in the triangulation, and the second is a simple L_2 measure: the sum of squares of absolute vertical errors over all input points.

From experiments with Delaunay and data-dependent triangulation on several smooth, synthetic functions, Rippa concluded that data-dependent triangulation usually yields more accurate approximations using a given number of vertices than Delaunay triangulation. The angle criterion performed well in most cases, so he mildly recommended it over both the L_2 criterion and Delaunay triangulation. Rippa observed that the L_2 criterion occasionally allowed long, extremely thin sliver triangles that did not fit the surface well to enter and remain in the triangulation. The algorithm failed to eliminate such triangles because they were so thin that they contained no input points, and hence they contributed zero error to the L_2 measure.

The angle criterion also made poor choices in some cases, so Rippa tried a hybrid scheme that on each pass compares the errors resulting from Delaunay triangulation and data-dependent triangulation with the angle criterion, and updates using the one with the smaller global error. The hybrid scheme generated high quality approximations more consistently than the other methods that Rippa tested. Unfortunately, the hybrid is less elegant, and it appears slower than the other methods. Margaliot and Gotsman reported an error measure yielding a better fit than the angle criterion [81].

Rippa also explored least squares methods that approximate the input points instead of interpolating them. The (x, y) coordinates of the vertices are frozen, but their heights are allowed to vary, and the combination of heights that minimizes the global sum of squared errors is found. This involves solving a large, sparse, $m \times m$ system of linear equations. He found that high quality results could be achieved fairly efficiently, on low-noise data, if the least-squares fitting was done as a post-process to greedy insertion. His empirical tests on simple functions showed that least squares fitting roughly halved the error of the standard interpolative methods. Overall, Rippa's methods appear expensive (data-dependent triangulation, particularly so) but the resulting approximations are higher quality than those of simpler sequential greedy insertion methods. The least squares technique appears to be particularly effective at improving the approximation.

Rippa tested his algorithm on rather small height fields and did not discuss computational costs of data-dependent triangulation much.

Polis-McKeown 93. Polis and McKeown explored a somewhat parallel variation of the greedy insertion method [95]. Their basic algorithm, in each pass, computes the absolute error at each input point. The set of points of maximal absolute error is found, and these are inserted into the triangulation, one at a time, rejecting any that are within a tolerance distance of vertices already in the triangulation (see paper for details). This method might insert multiple points per triangle, unlike the greedy insertion algorithms previously discussed. It would typically insert fewer points per pass than Fowler

and Little's algorithm, however.

Several practical issues in the creation of large terrain models for simulators are raised by Polis and McKeown. To facilitate dynamic loading of the terrain as a viewer roams, many display programs require that terrain databases be broken into small square blocks or "load modules". This necessitates extra care along block boundaries to avoid cracks between polygons. Polis and McKeown also proposed *selective fidelity*, in which regions of the terrain could be assigned error weights according to their visual importance, their likelihood of being seen, or some other criterion. Thus, for example, for a tank simulator, one might weight navigable valleys more than inaccessible mountain slopes.

Polis and McKeown tried a data-dependent triangulation method involving summing the squares of errors along all edges of the triangulation [94], much like Southard's method. They found Delaunay triangulation to be preferable to data-dependent triangulation, however, because the former was much faster [95].

Polis and McKeown's algorithm appears to have an expected cost of $O(mn)$ (like Algorithm I in [40]). They reported a compute time of 18 hours to select $m = 76,500$ points total from an $n = 1,979^2$ terrain broken into 36 tiles on a DECstation 5000. Speed was not the major issue for them, however, since they were creating their TINs offline. They later optimized their algorithm to select $m = 50,000$ points from a terrain of $n = 8,966,001$ points in 89 minutes on a DEC Alpha [93].

Franklin 93. Franklin has released code for a sequential greedy insertion algorithm (PL/I code from 1973, C code from 1993) [38]. His algorithm is quite similar to DeFloriani83, but optimized in a manner similar to De Floriani's Delaunay pyramid method ([22]). With each triangle, Franklin stores a candidate pointer, and he updates only the candidates of new or modified triangles on each pass. He stores an array of input points with each triangle, as in [22], so the algorithm is more general but typically slower than a comparable surface simplification algorithm limited to height fields.

Between his two implementations, Franklin has experimented with several triangulation methods: swapping an edge if it reduces the maximum error of the approximation, swapping an edge if it has shorter length, and Delaunay triangulation.

Unfortunately, Franklin has not published his results and conclusions. By comparison to De Floriani's Delaunay pyramid algorithm and Algorithm III of [40], we conclude that the expected cost of Franklin's algorithm is $O(n \log m + m^2)$. Franklin's program can select $m = 100$ points from an $n = 257^2$ height field in 7 seconds on an SGI Indy.

Puppo-Davis-DeMenthon-Teng 94. Puppo *et al.* explored terrain approximation algorithms for the Connection Machine that are parallel both in the computer architecture sense and also in the greedy insertion sense [98]. Their algorithm is much like that of DeFloriani83, except they insert all candidate points with error above the requested threshold on each pass, like Fowler and Little. They found that the number of points inserted on each pass grew exponentially, so the number of passes required to insert m points would typically be $\Theta(\log m)$. On a Thinking Machines CM-2 with 16,384 processors, they reported compute times of 8 seconds to select $m = 379$ points from an $n = 128^2$ terrain [98], or 86 seconds to select $m = 2,933$ points from an $n = 512^2$ terrain [97].

The algorithm was parallelized by assigning each input point to a different logical processor. Most of the parallelization was straightforward, but parallel incremental triangulation required the use of special mutual exclusion techniques to handle simultaneous topology changes in neighboring triangles.

Puppo *et al.* implemented both sequential and parallel greedy insertion and concluded, surprisingly, that the latter is better. Our own experiments have indicated otherwise [40].

Chen-Schmitt 93. Chen and Schmitt explored a hybrid feature/refinement approach for triangulation of computer vision range data [11]. To best approximate the step and slope discontinuities that are common in range data, they first use edge detection to identify significant discontinuity features. These then become constraint curves during greedy insertion of additional vertices, using either con-

strained Delaunay or data-dependent triangulation. Chen and Schmitt found that data-dependent triangulation simplified better on surfaces with a preferred direction, such as cylinders.

Silva-Mitchell-Kaufman 95. A rather different approach to height field triangulation was proposed by Silva *et al.* [117]. We classify it here as a refinement method, although it is different in spirit from the previous methods. Their method uses *greedy cuts*, triangulating the domain from the perimeter inward, on each pass "biting" out of the perimeter the triangle of largest area that fits the input data within a specified maximum error tolerance. The method is thus a generalization of greedy visibility techniques for curve simplification [123, 63], and also a form of data-dependent triangulation. In a comparison with Franklin's greedy insertion algorithm, their unoptimized program was about two to four times slower, but produced triangulations of a given quality using fewer vertices. They reported running times of about 8 seconds to select $m = 1,641$ points from grids of $n = 120^2$ points on a one-processor SGI Onyx.

Garland-Heckbert 95. Our own work in height field simplification has explored fast and accurate variations of the greedy insertion algorithm [40, 39].

We explored two optimizations of the most basic greedy insertion algorithm (as in DeFloriani83). First, we exploited the locality of mesh changes, and only recalculated the errors at input points for which the approximation changed, and second, we used a heap to permit the point of highest error to be found more quickly. When approximating an n point grid using an m vertex triangulated mesh, these optimizations sped up the algorithm from an expected time cost of $O(mn)$ to $O((m+n)\log m)$. We were able to approximate an $n = 1024^2$ grid to high quality using 1% of its points in about 21 seconds on a 150 MHz SGI Indigo2.

We also explored a data-dependent greedy insertion technique similar to Rippa's method. We found an algorithm that yielded, in a fairly representative test, a solution with 88% the error of Delaunay greedy insertion at a cost of about 3–4 times greater. Source code for these algorithms is available.

In that paper, we propose several ideas for future work that could improve the performance of the greedy insertion algorithm in the presence of cliff discontinuities, high frequencies, and noise.

Arc/Info Latticetin. The geographic information system Arc/Info sold by the Environmental Systems Research Institute (ESRI) can approximate terrain grids. Its "Latticetin" command employs a hybrid feature/refinement approach that starts with a regular grid of equilateral triangles and refines it with parallel greedy insertion [70, 95].

3.1.6 Decimation Methods

In contrast to refinement methods, the decimation approach to surface simplification starts with the entire input model and iteratively simplifies it, deleting vertices, triangles, or other geometric features on each pass. The decimation approach is not so common for height field simplification; we will see far more decimation methods in the section on manifold simplification.

Lee 89. A "drop heuristic" method for simplifying terrains was proposed by Lee [73]. We call it a vertex decimation approach because on each pass it deletes a vertex. The algorithm takes the height field grid as input and creates an initial triangulation in which each 2×2 square between neighboring input points is split into two triangles [73]. On each pass, the error at each vertex is computed and the vertex with lowest error is deleted. The error at a vertex is found by temporarily deleting the vertex from the triangulation, doing a local Delaunay retriangulation, and measuring the vertical distance from the vertex to its containing triangle. The process continues until the error exceeds the desired level, or the desired number of vertices is reached. Deletion in a Delaunay triangulation can be done incrementally to avoid excessive cost [68].

The drop heuristic method yields high quality approximations, but its computational cost and memory cost appear very high. When Lee compared his algorithm to Chen and Guevara's method and to De Floriani's ternary triangulation method [23], he found, not surprisingly, that his method yielded superior results [74]. The drop heuristic

method is expensive because of the need to visit each vertex on every pass. Its memory cost is high because a triangulation with n vertices must be created[6].

Scarlatos-Pavlidis 92b. Scarlatos and Pavlidis explored a method for adjusting a triangulation in order to equalize the curvature of the input data within each triangle [109], extending McClure's and Pavlidis' earlier work [82, 91, 83]. Their algorithm takes an initial triangulation and applies three passes: shrinking triangles with high curvature, merging adjacent coplanar triangles, and swapping edges to improve triangle shape and fit. In tests, the method achieved little improvement when applied to the output of their hierarchical triangulation algorithm [108, 107]: in most cases, the method reduced the number of triangles, but it also increased the maximum error unless explicit error tests were added [109]. Curvature equalization was more successful at improving regular subsampling meshes [107, p. 89]. No unshaded pictures of the resulting meshes were given, however, so it is difficult to compare the quality of the results to other methods.

Scarlatos 93. In addition to the recursive subdivision method described earlier, Scarlatos also developed a vertex decimation method for constructing hierarchical triangulations [107]. The method begins with an initial triangulation and, to generate each level of the hierarchy, computes the "significance" of each vertex and deletes vertices in increasing order of significance until no more can be deleted. Significance is an (unspecified) function of the error between a vertex and a weighted average of its neighbors, and the degree of a vertex. The method is similar to that of Schroeder *et al.*, discussed later, except that Scarlatos' method is limited to height fields, and it takes more precautions to minimize error accumulation. Scarlatos reported a running time of 7.75 minutes to build a complete hierarchy for about $n = 5,900$ points on a VAX 8530.

Hughes-Lastra-Saxe 96. The simplification algorithm described by Hughes, Lastra, and Saxe [59] is targeted towards simplifying global illumination meshes resulting from radiosity systems. Consequently, the algorithm must simplify both the mesh geometry and the color values associated with each mesh vertex. They rejected a greedy insertion algorithm because of its inability to deal well with sharp discontinuities (i.e., shadow borders). Instead, they chose a combination of local vertex decimation and simplification envelopes as in [126, 14]. Interestingly, they chose to select vertices for removal at random rather than in order of increasing error. They claim that this provides more uniform meshes, which they believe to be advantageous. Their method also uses higher-order elements (quadratic, cubic, etc.) for reconstructing the surface, a possibility which most simplification methods ignore.

[6] We find that storing a triangulation with n vertices uses 5 to 100 times the memory of a height field of n points because of the extra adjacency information required.

3.1.7 Optimal Methods

The error of an optimal piecewise-linear, triangulated approximation to a smooth function of two variables has been analyzed in the limit as the number of triangles goes to infinity. Nadler showed that the L_2–optimal approximation has L_2 error proportional to m^{-1} [88].

Finding the optimal approximation of a grid or surface using triangulations of a subset of the input points could be done by enumerating all possible subsets and all possible triangulations, but this would take exponential time, and it would clearly be impractical. As with curves, certain problems in optimal surface approximation are well understood, while others are not. It is known that L_∞–optimal polygonal approximation of convex surfaces is NP-hard (requires exponential time, in practice) [19, 9]. This implies, of course, that L_∞–optimal approximation of height fields and more general surfaces (in the space of all triangulations) is also NP-hard, since they are a superset of convex surfaces. We do not know if there are polynomial time algorithms for optimal surface simplification using any other error metric (such as L_2), or within a more restricted class of triangulations. Even if some form of this problem permits an optimal algorithm with polynomial time, it would be surprising if it were as fast as the heuristic methods we have summarized above.

Polynomial time algorithms are known, however, for sub-optimal solutions with provable size and quality bounds. If the optimal L_∞ solution for a given error tolerance has m_o vertices, there is an $O(n^7)$ algorithm

to find an approximation with the same error using $m = O(m_o \log m_o)$ vertices [87, 1], but this is far too slow to be practical for large problems.

3.2 Manifold Surfaces

We now turn our attention from height fields and parametric surfaces to manifolds and manifolds with boundary. In general, the manifold can have arbitrary genus and be non-orientable[7] unless stated otherwise. Manifolds are more difficult to simplify than height fields or parametric surfaces because there is no natural 2-D parameterization of the surface. Delaunay triangulation is thus less easily applied. We group manifold simplification methods into two classes: refinement methods and decimation methods.

3.2.1 Refinement Methods

Faugeras-Hebert-Mussi-Boissonnat 84. Faugeras *et al.* developed a technique somewhat similar to De Floriani's 1984 algorithm, but it does not have persistent long edges, and it is applicable to the simplification of any 3-D triangulated mesh of genus 0, not just height fields [34]. The method begins with a pancake-like two-triangle approximation defined by three vertices of the input mesh. Associated with each triangle of the approximation is a set of input points. In successive passes, for each triangle of the approximation, the input point farthest from the triangle is found, and if the distance is above threshold, the triangle is split into 3–6 subtriangles by inserting new vertices at the interior point of highest error. Edges common to two subdivided triangles are split at their points of highest error (Figure 6). Splitting in this way eliminates the long edges of ternary triangulation.

During subdivision, each triangle's point set must be partitioned into 3–6 subsets. In methods that are limited to height fields, the partition of input points to subtriangles is done with simple projection and linear splitting. To partition point sets on a surface in 3-D, Faugeras *et al.* instead split using the shortest path along edges of the input mesh.

[7] A manifold is *orientable* if its two sides can be consistently labeled as "inside" and "outside". A Möbius strip is non-orientable.

The method simplified an $n = 2,000$ point model in 1 minute on a Perkin Elmer computer. The approximations generated were sometimes poor, however, and the method had particular problems with concavities [96]. A later subdivision data structure, the "prism tree", addressed these problems by recursively subdividing surface points into truncated pyramidal volumes [96].

Delingette 94. A related method for the simplification of orientable manifolds was developed by Delingette [28]. He fits surfaces to sets of 3-D points by minimizing an energy function which is a sum of an error term, an edge length term, and a curvature term. The algorithm starts with a mesh that is the dual to a subdivided icosahedron. It then iteratively adjusts the geometry, attempting to minimize the global energy [29]. After a good initial fit is achieved with this fixed topology, the mesh is refined. Regions of the mesh with high curvature, high local fit error, or elongated faces are subdivided and vertices migrate to points of high curvature [28]. Delingette reports that it takes 2 to 7 minutes to approximate a set of $n = 260,000$ points with a mesh of $m = 1,700$ vertices on a DEC Alpha. The method is much faster than the related method of Hoppe *et al.* [58], but it does not achieve comparable simplification, and it has a number of parameters that appear to require careful tuning.

Lounsbery-Eck-*et al.* 95. A two-stage method for multiresolution wavelet modeling of arbitrary triangulated polyhedra was developed by Lounsbery, Eck, *et al.* [76, 33]. The method is not limited to height fields or even to triangulated meshes with spherical topology; it can be applied to any triangulated manifold with boundary. The approach first constructs a base mesh which is a triangulated polyhedron with the same topology as the input surface. Geodesic-like distance measures are used in this step, reminiscent of the method of Faugeras *et al.*. It then uses repeated quaternary subdivision of the base mesh to construct a new mesh that approximates the input surface very closely. A multiresolution model of the new mesh is then built using wavelet techniques, after which an approximation at any desired error tolerance can be quickly generated. Eck *et al.* simplified a model with about $n = 35,000$ vertices to $m = 5,400$ vertices in 22 minutes of resampling

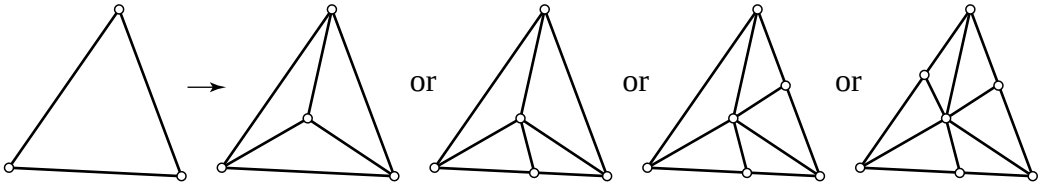


Figure 6: Subdivision pattern of Faugeras *et al.*.

plus 5 minutes of wavelet analysis/synthesis, on an SGI Onyx. The intermediate, approximating mesh had about twice as many vertices as the original.

While the approach is very attractive for interactive surface design and surface optimization, it may not be the best method for multiresolution modeling of static surfaces because of the cost of resampling. For the approximation of height fields, resampling is not needed, and simpler tensor product wavelet techniques could be used instead [79]. Another disadvantage is that the method does not resolve creases at arbitrary angles well, since the final mesh subdivides the triangles of the base mesh on a regular grid.

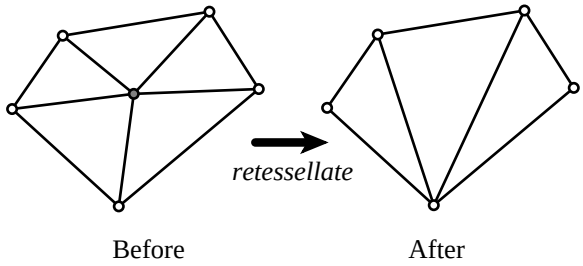


Figure 7: Vertex decimation. The target vertex and its adjacent triangles are removed. The resulting hole is then retessellated.

3.2.2 Decimation Methods

The next class of surface simplification algorithms we will consider is decimation methods: algorithms that start with a polygonization (typically a triangulation) and successively simplify it until the desired level of approximation is achieved. Most decimation algorithms fall into one of the following categories:

vertex decimation methods delete a vertex and retriangulate its neighborhood,

edge decimation methods delete one edge and two triangles, and merge two vertices,

triangle decimation methods delete one triangle and three edges, merge three vertices, and retriangulate the neighborhood, and

patch decimation methods delete several adjacent triangles and retriangulate their boundary.

Several variants of the decimation approach have been used for the problem of simplifying manifolds, particularly for thinning the output of isosurface polygonizers.

Kalvin 91. Kalvin *et al.* developed a two phase method to create surface models from medical data [65]. The first phase approximates a surface with tiny polygons using an algorithm similar to marching cubes [90], and the second phase then does patch decimation on the model by merging adjacent coplanar rectangles. Since it only merges precisely coplanar faces, the method does not allow control over the degree of simplification, so it is quite limited.

Schroeder-Zarge-Lorensen 92. Schroeder *et al.* developed a general vertex decimation algorithm primarily for use in scientific visualization [116]. Their method takes a triangulated surface as input, typically a manifold with boundary. The algorithm makes multiple passes over the data until the desired error is achieved. On each pass, all vertices that are not on a boundary or crease that have error below the threshold are deleted, and their surrounding polygons are retriangulated (see Figure 7). The error at

a vertex is the distance from the point to the approximating plane of the surrounding vertices. Note that errors are measured with respect to the previous approximation, not relative to the input points, so errors can accumulate (this flaw was fixed in later versions of the algorithm). Their paper demonstrated simplifications of models containing as many as 1,700,000 triangles. The computation time to simplify a model of $n = 400,000$ vertices to $m = 40,000$ vertices is about 14 minutes on an R4000 processor [115]. This method uses significant memory, like Lee's. To conserve memory, compact data structures were developed [115]. Source code for this algorithm is available [114].

Relative to Lee's method, the technique of Schroeder *et al.* is more general since it is not limited to height fields, it uses a less expensive and less accurate error measure, and it deletes multiple vertices per pass. Consequently, it is faster, but probably has lower quality.

Soucy and Laurendeau 92. To simplify manifolds with boundary, Soucy and Laurendeau also developed a vertex decimation algorithm [118, 119]. Their application was the construction of surface models from multiple range views. On each pass, the vertex with least error is deleted, and its neighborhood (the set of adjacent triangles) is retriangulated. The process stops when the error rises above a specified tolerance or the desired size of model is achieved.

To compute rigorous error bounds, a set of deleted vertices is stored with each triangle. We will call these points the *ancestors* of the triangle. To compute the error at a vertex, a temporary vertex deletion and retriangulation are done. The error of a vertex is a measure of the error that would result from its removal. More precisely, it is defined to be the maximum distance between either an ancestor from the neighborhood or the vertex itself to the retriangulated surface. Deletion of a non-boundary vertex is considered legal if the neighborhood triangles can be projected to 2-D without foldover.

To retriangulate, Soucy and Laurendeau first compute a constrained Delaunay triangulation in a 2-D projection, then this triangulation is improved using a version of Lawson's local optimization procedure [71] adapted to surfaces in 3-D. To update the data structures after retriangulation, first the ancestor lists are redistributed among the new triangles, then the error of each formerly neighboring vertex is updated.

We can relate the method to several of its precursors. Like Lee's method, this algorithm does vertex decimation by "one move lookahead", but unlike his technique, it is not limited to height fields. Like Faugeras *et al.* and De Floriani *et al.* (1989), it stores a point set with each triangle, but unlike those methods, it is a decimation algorithm, and it is more general: it can simplify any manifold with boundary.

Soucy and Laurendeau estimate the expected complexity of their algorithm to be $O\Big(n \log (n/(n-m))\Big)$. Their method appears to yield higher quality results than the method of Schroeder *et al.*, but it is slower and it uses more memory, since it maintains lists of all deleted points. A revised version of this algorithm is used in the *IMCompress* software sold by InnovMetric [64].

Turk 92. Another method for simplifying a manifold with boundary is due to Turk [124]. This algorithm is not a decimation method in the same sense as the previous methods, but we list it here because it also starts with a full triangulation and simplifies.

Turk's algorithm takes a triangulated surface as input, sprinkles a user-specified number of points on these triangles at random, and uses an iterative repulsion procedure to spread the points out nearly uniformly. The points remain on the surface as they move about. After these points are inserted into the original surface triangulation, the original vertices are deleted one by one, yielding a triangulation of the new vertices that has the same topology as the original surface. Turk also demonstrated an improved variant of this technique that groups points most densely where the surface is highly curved.

Turk's method appears to be best for smooth surfaces, since it tends to blur sharp features[8]. Overall, it appears that Turk's algorithm is quite complex and that it will yield results inferior in quality to the methods of Schroeder *et al.* or Soucy-Laurendeau.

[8] William Schroeder, SIGGRAPH '94 tutorial talk.

Hinker-Hansen 93. Hinker and Hansen developed a patch decimation algorithm for use in scientific visualization [55]. It is a one pass method that first finds patches of triangles with nearly parallel normal vectors, and then retriangulates each patch. The method has $O(nlogn)$ time cost in practice. A model with about $n = 510,000$ vertices was simplified to $m = 321,000$ vertices in 9 minutes on a CM-5. The method is "largely ineffective when faced with surfaces of high curvature", however [55]. It appears to work best on piecewise-ruled surfaces: those with zero curvature in at least one direction, such as cylinders, cones, and planes. Therefore the method is not as general as that of Schroeder *et al.* or Soucy-Laurendeau.

Hoppe-DeRose-Duchamp-McDonald-Stuetzle 93. Hoppe *et al.* developed an optimization-based algorithm for general 3-D surface simplification [58]. Their method takes a set of points and an initial, fine triangulated surface approximation to those points as input, and outputs a coarser triangulation of the points with the same topology as the input mesh. The method attempts to minimize a global energy measure consisting of three terms: a complexity term that penalizes meshes with many vertices, an error term that penalizes geometric distance of the surface from the input points, and a spring term that penalizes long edges in the triangulation. The method proceeds in three nested loops, the outermost one decreasing the spring constant, the middle one doing an optimization over mesh topologies, and the inner one doing an optimization over geometries. The topological optimization uses heuristics and random selection to pick an edge and either collapse it, split it, or swap it. The geometric optimization uses nonlinear optimization techniques to find the vertex positions that minimize the global error for a given topology. Topological changes are kept if they reduce the global error, otherwise they are discarded. In other words, the method makes repeated semi-random changes to the mesh, keeping those that allow better fit and/or a simpler mesh.

Unlike most general surface simplification methods, the method of Hoppe *et al.* does not constrain output vertices to be a subset of the input points. Their method appears to be less sensitive to noise in the input points than most other methods because of its freedom in choosing vertices and because the geometric error measure uses an L_2 norm, and not an L_∞ norm.

Their method is slow, but it is capable of very good simplifications. They simplified a mesh with $m_1 = 4,059$ vertices to $m_2 = 262$ vertices while fitting to $n = 16,864$ points in 46 minutes on a 1-processor DEC Alpha. They have released their code. Their algorithm yields higher quality approximations than that of Eck *et al.*, but it is slower [33].

Hamann 94. A triangle decimation method was explored by Hamann [49]. In this algorithm, triangles are deleted in increasing order of weight, where weight is the product of "equi-angularity" and curvature, roughly speaking. Thus, slivers and low curvature triangles are deleted first. The method appears rather complex, however, since second degree surface fitting is used to position the new vertices, and a number of geometric checks are required to prevent topological changes.

Kalvin 94. In later work, Kalvin and Taylor developed a patch decimation method called "superfaces" to simplify manifolds within a given error tolerance [66, 67]. The algorithm operates in a single pass. This pass consists of three phases. The first phase segments the surface into approximately planar patches. Each patch is found by picking a face at random and merging in adjacent faces until the patch's faces can no longer be fit by a plane within the error tolerance. Additional tests prevent degenerate or highly elongated patches from being created. The second phase simplifies the curves common to adjacent patches using the Douglas-Peucker algorithm. The third phase retriangulates the patches by subdividing them into star polygons and then triangulating each star polygon.

When a face is merged into a patch, the set of feasible approximating planes $ax + by + cz + 1 = 0$ of the patch must be updated. This set could be represented using linear programming, as a convex polytope in the 3-D (a, b, c) parameter space of planes, but the complexity of this data structure could grow quite large. Instead, Kalvin and Taylor use an ellipsoidal approximation that supports constant time updates and queries.

At a high level, this method is quite similar to Hinker-Hansen, in that it employs a single pass to find nearly coplanar sets and then retriangulates them. Hinker-

Hansen define patches based on angles between normal vectors, however, while Kalvin-Taylor define them based on distance-to-plane. Distance to plane is probably a better method for defining patches, since it is less sensitive to noise. Guéziec reports that Kalvin and Taylor's algorithm can simplify a model with about $n = 90,000$ vertices to $m = 5,000$ vertices in 3 to 5 minutes on an IBM RS6000.

Varshney 94. Using visibility techniques from computational geometry, Varshney developed a patch decimation algorithm for simplifying orientable triangulated manifolds with boundary [127, 126]. The method has bounded error. Instead of simplifying in a fast, greedy manner, as most other decimation methods do, it is much more brute force, exhaustively testing to find the largest triangle to insert on each pass.

First, the input surface is offset inwards and outwards by a tolerance distance ϵ to create two offset surfaces. All triangles defined by three vertices of the input surface are checked for validity by testing that they do not intersect either offset surface and that they do not overlap previously inserted triangles. On each pass of the algorithm, the valid triangle that "covers" the greatest number of previously uncovered input vertices is inserted, the old triangulation of this portion of the surface is deleted, and small triangles are added to fill the cracks between the old and the new. The algorithm generates good approximations when it works, but problems arise when the offset surfaces collide. So far, the method has not been demonstrated for simplifications below 30% of the input size, and it is very slow. The time costs of this algorithm and its variants range from $O(n^2)$ to $O(n^6)$[9]

Guéziec 95. Guéziec developed a method for simplifying orientable manifolds that employs edge decimation [46]. He defines the *edge collapse*, or edge contraction, operator to delete an edge and merge its two endpoints into a single vertex (Figure 8). Guéziec's algorithm orders edges by "importance" (in some unspecified way), and makes a single pass through the edges in increasing order of importance, doing edge collapses where legal.

[9]Personal communication, Pankaj K. Agarwal and Amitabh Varshney, 1995.

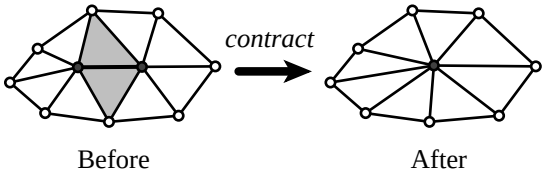


Figure 8: A simple edge contraction. The highlighted edge is contracted into a single point. The shaded triangles become degenerate and are removed during the contraction.

Testing legality entails most of the work required to do an edge collapse. The provisional new vertex is positioned to fit the old faces well and to preserve volume. During simplification, an error radius is associated with each vertex. By interpolating spheres with these radii across the surface, a error volume is defined. At any step during simplification, the error volume encloses the original surface. When an edge collapse is being considered, the error radius for the provisional new vertex is set so that the new error volume encloses the old error volume.

The collapse is considered legal if it meets four conditions: (1) the topology of the surface is preserved, (2) the normals of the modified faces change little, (3) the new triangles are well shaped (not slivers), and (4) the error radius for the new vertex is below an error threshold.

Use of the error volume could give the user local control of error tolerance at each vertex. No examples of this are shown in the paper, however.

Guéziec reports a time of 10 minutes to simplify a model with about $n = 90,000$ vertices to $m = 5,000$ vertices on an IBM RS6000 model 350. He says that Kalvin and Taylor's algorithm yields more compact approximations for small error tolerances, but that his algorithm performs better for large error tolerances, and that his triangles are better shaped. Closely related algorithms are illustrated with better pictures in another paper [47]. In that work, a model with about $n = 181,000$ vertices was simplified to $m = 26,000$ vertices in 53 minutes, on the same type of machine. The quality of the resulting meshes appears good.

Gourdon 95. Gourdon explored a method for simplifying orientable surface meshes resulting from surface reconstruction [43]. His algorithm differs from almost all other simplification algorithms in that it does not assume the surface mesh to be a triangulation. The algorithm is designed to preserve the Euler characteristic[10] of the model; this implies that the topology is preserved. Topological preservation is important for simplifying medical data, which is the focus of this technique. The algorithm iteratively removes edges based on an unspecified curvature criterion. Because the algorithm supports non-triangular facets, no retessellation is required after removing edges. Gourdon observes that simply removing a sequence of edges can lead to undesirable, irregular meshes. To control the regularity of the tessellation, he restricts the degree of vertices to be at most 6 and facet may have at most 12 edges. Following simplification, a "regularization" step is performed. Regularization attempts to improve the mesh by moving points to minimize an energy function, in this case the sum of squared edge lengths. A simple regularization step would move a vertex towards the barycenter of its neighbors. However, this can produce significant shrinkage of the surface. To avoid this, Gourdon uses a regularization step that moves the vertex towards the barycenter, but constrains the vertex to move parallel to the average plane of its neighbors.

Klein-Liebich-Strasser 96. The algorithm described by Klein, Liebich, and Strasser [69] is very similar to the method of Soucy and Laurendeau [119]. It simplifies an oriented manifold by iteratively removing a vertex a retriangulating the resulting hole using a constrained Delaunay triangulation. Each deleted vertex is linked to the closest face in the approximation. These links are used to compute the distance between the original and approximate surfaces. To select a vertex for removal, each vertex is tentatively removed and the additional error introduced by the removal is computed. The vertex which introduces the least error is selected for removal. After the vertex is removed, the links and projected additional errors within its neighborhood must be recomputed.

[10]The *Euler characteristic* of a model is defined as $\chi = F - E + V$ where F, E, and V are, respectively, the number of faces, edges, and vertices.

Algorri-Schmitt 96. Algorri and Schmitt developed an algorithm for simplifying closed, dense triangulations resulting from surface reconstruction [2]. Their algorithm begins with a pre-processing phase which smooths the initial mesh by swapping edges based on a G^1-continuity criterion as in [32]. After this initial smoothing, every edge whose dihedral angle exceeds some user-specified planarity threshold is classified as a *feature edge*. Each vertex is subsequently labeled according to its number of incident feature edges. An independent set of edges connecting "0" vertices is collected, and all the edges are collapsed simultaneously. This simplification phase is followed by a smoothing phase were all non-feature edges are considered for swapping based on the G^1-continuity criterion. If further simplification is desired, edges are reclassified and the process outlined above is repeated. Since only edges in mostly planar regions are selected for decimation, the basic step will not simplify "characteristic curves" (e.g., the edges of a cube) and there will always be a single vertex left in the midst of planar regions. Algorri and Schmitt describe additional iterative steps which simplify these cases separately from the basic step outlined above.

Ronfard-Rossignac 96. Another algorithm based on edge collapse was described by Ronfard and Rossignac [103]. The fundamental observation underlying their algorithm is that each vertex in the original model lies at the intersection of a set of planes, in particular, the planes of the faces that adjoin the vertex. They associate a set of planes with each vertex; they call this set the *zone* of the vertex. A vertex's zone is initialized to be the set of planes of the adjoining faces. The error at a vertex is measured by the maximum distance between the vertex and the planes in its zone. When contracting an edge, the zone of the resulting vertex is the union of the zones of the original endpoints. The error of this resulting vertex characterizes the cost of contracting the edge. At each iteration, the edge of lowest cost is selected and contracted. The complexity of this algorithm would seem to be $O(n \log n)$.

Hoppe 96. The simplification algorithm presented by Hoppe [56] for the construction of progressive meshes is a simplified version of the algorithm of Hoppe *et al.* [58]. Rather than performing a more general search, it simply

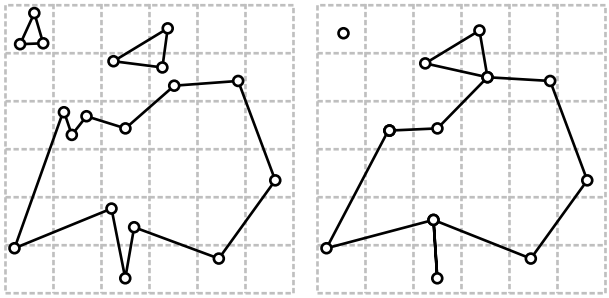


Figure 9: Uniform vertex clustering. Note the triangle which as collapsed to a single point, and the now dangling edge at the bottom. Also note how separate components have been joined together.

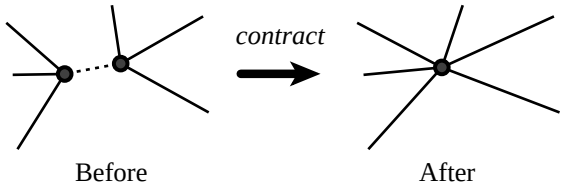


Figure 10: Pair contraction joining unconnected vertices. The dashed line indicates the two vertices being contracted together.

selects a sequence of edge contractions. The algorithm uses essentially the same error formulation of the earlier method, although it is augmented to handle surface attributes such as colors. Hoppe suggests that the resulting meshes are just as good, and perhaps even better, than the results of the more general mesh optimization algorithm.

3.3 Non-Manifold Surfaces

The most general class of surfaces is the non-manifold surface, which permits three or more triangles to share an edge, and permits arbitrary polygon intersections. Relatively few surface simplification algorithms can handle models of this generality.

Rossignac-Borrel 93. A very general technique for simplifying general 3-D triangulated models was described by Rossignac and Borrel [104]. They subdivide the object's bounding volume into a regular grid of boxes of user-specified size. All vertices are graded (or weighted) according to some scheme, and all vertices within each box are merged together into a new representative vertex. A simplified model is then synthesized from these representative vertices by forming triangles according to the original topology (see Figure 9). This method is extremely general, as it can operate on any set of triangles (not just manifolds), it can achieve arbitrary simplification levels, and it can even eliminate small objects or otherwise change the topology of a surface. Unfortunately, it does not preserve detail well [52]. When applied to height fields, it is roughly equivalent to blurring followed by regular subsampling. This software is being sold as part of IBM's "3D Interaction Accelerator" [61]. This method has been extended using octrees instead of regular grids [78].

Low-Tan 97. Low and Tan [77] developed a clustering algorithm that is intended to provide higher quality than the uniform clustering described by Rossignac and Borrel while maintaining its generality. Their first improvement was to suggest a better weighting criterion. More importantly, they replaced the uniform grid with a set of cluster cells. These cells can be any simple shape, such as cubes or spheres. Cells are centered around their vertex of highest weight. When a vertex falls within the intersection of multiple cells, it is placed in the cell whose center is closest. In addition to these algorithmic improvements, they improved the appearance of simplified models by rendering stray edges as thick lines whose area approximates the area of the original model in that region.

Garland-Heckbert 97. We have developed an algorithm for simplifying surfaces based on iterative vertex-pair contractions [41]. A pair contraction is a natural generalization of edge contraction (Figure 8) where the vertex pair need not be connected by an edge (see Figure 10). A 4×4 symmetric matrix $\mathbf{Q}_i$ is associated with each vertex $\mathbf{v}_i$. The error at the vertex is defined to be $\mathbf{v}^T\mathbf{Q}\mathbf{v}$, and when a pair is contracted, their matrices are added together to

form the matrix for the resulting vertex. We derive these matrices to calculate the sum of squared distances of the vertex to a set of planes (this is similar to the error metric of Ronfard and Rossignac [103]).

Our technique for tracking vertex error is quite efficient, and the algorithm is correspondingly fast. The quality of the approximations is similar to those of Ronfard and Rossignac, although the algorithm is more general in that it can join model components.

3.4 Related Techniques

We have focused on approximation of surfaces by polygons, but there has been related work in fitting curved surfaces to a set of points on a surface, and approximation of volumetric data. We include a partial survey.

Fitting a Curved Surface Model. Polygon models for curved surfaces can be bulky. More compact representations for surfaces are often possible using curved surface primitives such as piecewise-polynomial surfaces. The next class of models beyond piecewise-linear surfaces are surfaces with tangent continuity. Schmitt and others have developed adaptive refinement methods for fitting rectangular Bézier patches [113] and triangular Gregory patches [111] to a grid of points in 3-D. The latter method is superior to the former because it is better able to adapt to features at an angle to the grid. Another curved surface primitive, the subdivision surface, has been fit to points in 3-D by Hoppe *et al.*, with very nice results [57]. Piecewise quadratic surfaces have been fit to range data using least squares techniques [35].

Fitting to a Volume. A generalization of the feature approach to the approximation of volumes (scalar functions of three variables) was explored by Hamann and Chen [50]. They ranked points according to an estimate of the curvature of the function $f(x, y, z)$ at each point, and incrementally inserted vertices into a data-dependent tetrahedrization, in decreasing order of curvature, until a given error tolerance was met. The errors for data-dependent tetrahedrization were measured using L_2 or L_∞ norms on all points inside each tetrahedron. The surface decimation approach has also been generalized to tetrahedrizations [100].

4 Conclusions

Surface simplification is not as well understood as curve simplification. Whereas there appears to be fairly widespread agreement that one algorithm, Douglas-Peucker, does a high quality job of curve simplification at acceptable speeds, there is little agreement about the best approach for surface simplification. No thorough empirical comparison of surface simplification methods has been done analogous to the studies for curves ([85, 130]). Furthermore, surface simplification seems inherently much more difficult than curve simplification.

Why are surfaces so much harder? The biggest qualitative difference we observe is that curves inherently lend themselves to divide and conquer strategies like Douglas-Peucker, since splitting a curve at the point of highest error yields two curves, breaking the task into two smaller subtasks of the same type. Splitting a surface at the point of highest error is an ambiguous concept. Certain methods arbitrarily choose some way of splitting at a point, as with the hierarchical subdivision methods that split a triangle into three or more subtriangles; and other methods abandon the divide and conquer strategy and employ the more complex general triangulations.

Our purpose has been primarily to survey the existing methods, not to evaluate them, so we offer few conclusions here. Instead we hope that this survey, by collecting references and descriptions to the large body of work on this topic, will draw attention to similar lines of research in disparate fields, and facilitate future cross-fertilization.

5 Acknowledgements

We thank Frank Bossen, Marshall Bern, Jon Webb, and Anoop Bhattacharjya for their comments on drafts of this survey. The CMU Engineering & Science library has been very helpful in locating obscure papers.

6 References

[1] Pankaj K. Agarwal and Subhash Suri. Surface approximation and geometric partitions. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 24–33, 1994. (Also available as Duke U. CS tech report, ftp://ftp.cs.duke.edu/dist/techreport/1994/1994-21.ps.Z).

[2] María-Elena Algorri and Francis Schmitt. Mesh simplification. *Computer Graphics Forum*, 15(3), Aug. 1996. Proc. Eurographics '96.

[3] Dana H. Ballard. Strip trees: A hierarchical representation for curves. *Communications of the ACM*, 24(5):310–321, 1981.

[4] Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice Hall, Englewood Cliffs, NJ, 1982.

[5] Bruce G. Baumgart. *Geometric Modeling for Computer Vision.* PhD thesis, CS Dept, Stanford U., Oct. 1974. AIM-249, STAN-CS-74-463.

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

[7] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[8] Laurence Boxer, Chun-Shi Chang, Russ Miller, and Andrew Rau-Chaplin. Polygonal approximation by boundary reduction. *Pattern Recognition Letters*, 14(2):111–119, February 1993.

[9] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite VC-dimension. In *Proc. 10th Annual ACM Symp. on Computational Geometry*, pages 293–302, 1994.

[10] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, Dept. of CS, U. of Utah, Dec. 1974.

[11] Xin Chen and Francis Schmitt. Adaptive range data approximation by constrained surface triangulation. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 95–113. Springer-Verlag, Berlin, 1993.

[12] Zi-Tan Chen and J. Armando Guevara. Systematic selection of very important points (VIP) from digital terrain model for constructing triangular irregular networks. In N. Chrisman, editor, *Proc. of Auto-Carto 8 (Eighth Intl. Symp. on Computer-Assisted Cartography)*, pages 50–56, Baltimore, MD, 1987. American Congress of Surveying and Mapping.

[13] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, Oct. 1976.

[14] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *SIGGRAPH '96 Proc.*, pages 119–128, Aug. 1996. http://www.cs.unc.edu/∼geom/envelope.html.

[15] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[16] Michael A. Cosman, Allan E. Mathisen, and John A. Robinson. A new visual system to support advanced requirements. In *Proceedings of the 1990 Image V Conference*, pages 371–380. Image Society, Tempe, AZ, June 1990.

[17] Michael A. Cosman and Robert A. Schumacker. System strategies to optimize CIG image content. In *Proceedings of the Image II Conference*, pages 463–480. Image Society, Tempe, AZ, June 1981.

[18] Robert G. Cromley. Hierarchical methods of line simplification. *Cartography and Geographic Information Systems*, 18(2):125–131, 1991.

[19] Gautam Das and Michael T. Goodrich. On the complexity of approximating and illuminating three-dimensional convex polyhedra. In *Proc. 4th Workshop Algorithms Data Struct.*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[20] Eduardo F. D'Azevedo. Optimal triangular mesh generation by coordinate transformation. *SIAM J. Sci. Stat. Comput.*, 12(4):755–786, July 1991.

[21] Carl de Boor. *A Practical Guide to Splines.* Springer, Berlin, 1978.

[22] Leila De Floriani. A pyramidal data structure for triangle-based surface description. *IEEE Computer Graphics and Appl.*, 9(2):67–78, March 1989.

[23] Leila De Floriani, Bianca Falcidieno, George Nagy, and Caterina Pienovi. A hierarchical structure for surface approximation. *Computers and Graphics*, 8(2):183–193, 1984.

[24] Leila De Floriani, Bianca Falcidieno, and Caterina Pienovi. A Delaunay-based method for surface approximation. In *Eurographics '83*, pages 333–350. Elsevier Science, 1983.

[25] Leila De Floriani, Bianca Falcidieno, and Caterina Pienovi. Delaunay-based representation of surfaces defined over arbitrarily shaped domains. *Computer Vision, Graphics, and Image Processing*, 32:127–140, 1985.

[26] Leila De Floriani and Enrico Puppo. A hierarchical triangle-based model for terrain description. In A. U. Frank et al., editors, *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, pages 236–251, Berlin, 1992. Springer-Verlag.

[27] Michael DeHaemer, Jr. and Michael J. Zyda. Simplification of objects rendered by polygonal approximations. *Computers and Graphics*, 15(2):175–184, 1991.

[28] Hervé Delingette. Simplex meshes: a general representation for 3D shape reconstruction. Technical report, INRIA, Sophia Antipolis, France, Mar. 1994. No. 2214, http://zenon.inria.fr:8003/epidaure/personnel/delingette/delingette.html.

[29] Hervé Delingette, Martial Hebert, and Katsushi Ikeuchi. Shape representation and image segmentation using deformable surfaces. *Image and Vision Computing*, 10(3):132–144, Apr. 1992.

[30] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, Dec. 1973.

[31] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, 1973.

[32] Nira Dyn, David Levin, and Shmuel Rippa. Data dependent triangulations for piecewise linear interpolation. *IMA J. Numer. Anal.*, 10(1):137–154, Jan. 1990.

[33] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH '95 Proc.*, pages 173–182. ACM, Aug. 1995. http://www.cs.washington.edu/research/projects/grail2/www/pub/pub-author.html.

[34] Olivier Faugeras, Martial Hebert, P. Mussi, and Jean-Daniel Boissonnat. Polyhedral approximation of 3-D objects without holes. *Computer Vision, Graphics, and Image Processing*, 25:169–183, 1984.

[35] Olivier D. Faugeras, Martial Hebert, and E. Pauchon. Segmentation of range data into planar and quadratic patches. In *Proc. IEEE Intl. Conf. on Computer Vision and Pattern Recognition*, pages 8–13, June 1983.

[36] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, 2nd ed.* Addison-Wesley, Reading MA, 1990.

[37] Robert J. Fowler and James J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics (SIGGRAPH '79 Proc.)*, 13(2):199–207, Aug. 1979.

[38] W. Randolph Franklin. tin.c, 1993. C code, ftp://ftp.cs.rpi.edu/pub/franklin/tin.tar.gz.

[39] Michael Garland and Paul S. Heckbert. Fast triangular approximation of terrains and height fields. Submitted for publication.

[40] Michael Garland and Paul S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical report, CS Dept., Carnegie Mellon U., Sept. 1995. CMU-CS-95-181, http://www.cs.cmu.edu/∼garland/scape.

[41] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97 Proc.*, August 1997. To appear. http://www.cs.cmu.edu/∼garland/.

[42] Dora Gómez and Adolfo Guzmán. Digital model for three-dimensional surface representation. *Geo-Processing*, 1:53–70, 1979.

[43] Alexis Gourdon. Simplification of irregular surface meshes in 3D medical images. In *Computer Vision, Virtual Reality, and Robotics in Medicine (CVRMed '95)*, pages 413–419, Apr. 1995.

[44] Markus H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In *Proc. IEEE Visualization '95*, July 1995. (Also ETH Zürich CS tech report 230, http://www.inf.ethz.ch/publications/tr.html).

[45] Eric Grosse. Bibliography of approximation algorithms. ftp://netlib.att.com/netlib/master/readme.html, link="catalog".

[46] André Guéziec. Surface simplification with variable tolerance. In *Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery (MRCAS '95)*, pages 132–139, November 1995.

[47] André Guéziec and Robert Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Trans. on Visualization and Computer Graphics*, 1(4):328–342, 1995.

[48] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):75–123, 1985.

[49] Bernd Hamann. A data reduction scheme for triangulated surfaces. *Computer-Aided Geometric Design*, 11:197–214, 1994.

[50] Bernd Hamann and Jiann-Liang Chen. Data point selection for piecewise trilinear approximation. *Computer-Aided Geometric Design*, 11:477–489, 1994.

[51] Richard W. Hamming. *Digital Filters*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[52] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, Banff, Canada, May 1994. Canadian Inf. Proc. Soc. http://www.cs.cmu.edu/∼ph.

[53] Martin Heller. Triangulation algorithms for adaptive terrain modeling. In *Proc. 4th Intl. Symp. on Spatial Data Handling*, volume 1, pages 163–174, Zürich, 1990.

[54] John Hershberger and Jack Snoeyink. Speeding up the Douglas-Peucker line-simplification algorithm. In P. Bresnahan et al., editors, *Proc. 5th Intl. Symp. on Spatial Data Handling*, volume 1, pages 134–143, Charleston, SC, Aug. 1992. Also available as TR-92-07, CS Dept, U. of British Columbia, http://www.cs.ubc.ca/tr/1992/TR-92-07, code at http://www.cs.ubc.ca/spider/snoeyink/papers/papers.html.

[55] Paul Hinker and Charles Hansen. Geometric optimization. In *Proc. Visualization '93*, pages 189–195, San Jose, CA, October 1993. http://www.acl.lanl.gov/Viz/vis93_abstract.html.

[56] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96 Proc.*, pages 99–108, Aug. 1996. http://www.research.microsoft.com/research/graphics/hoppe/.

[57] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. In *SIGGRAPH '94 Proc.*, pages 295–302, July 1994. http://www.research.microsoft.com/research/graphics/hoppe/.

[58] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *SIGGRAPH '93 Proc.*, pages 19–26, Aug. 1993. http://www.research.microsoft.com/research/graphics/hoppe/.

[59] Merlin Hughes, Anselmo A. Lastra, and Edward Saxe. Simplification of global-illumination meshes. *Computer Graphics Forum*, 15(3):339–345, August 1996. Proc. Eurographics '96.

[60] Peter Hughes. Building a terrain renderer. *Computers in Physics*, pages 434–437, July/August 1991.

[61] IBM. IBM 3D Interaction Accelerator, 1995. Commercial software, http://www.research.ibm.com/3dix.

[62] Insung Ihm and Bruce Naylor. Piecewise linear approximations of digitized space curves with applications. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 545–569, Tokyo, 1991. Springer-Verlag.

[63] Hiroshi Imai and Masao Iri. Polygonal approximations of a curve – formulations and algorithms. In G. T. Toussaint, editor, *Computational Morphology*, pages 71–86. Elsevier Science, 1988.

[64] InnovMetric. Commercial software, http://www.innovmetric.com.

[65] Alan D. Kalvin, Court B. Cutting, B. Haddad, and M. E. Noz. Constructing topologically connected surfaces for the comprehensive analysis of 3D medical structures. In *Medical Imaging V: Image Processing*, volume 1445, pages 247–258. SPIE, Feb. 1991.

[66] Alan D. Kalvin and Russell H. Taylor. Superfaces: Polyhedral approximation with bounded error. In *Medical Imaging: Image Capture, Formatting, and Display*, volume 2164, pages 2–13. SPIE, Feb. 1994. (Also IBM Watson Research Center tech report RC 19135).

[67] Alan D. Kalvin and Russell H. Taylor. Superfaces:polygonal mesh simplification with bounded error. *IEEE Computer Graphics and Appl.*, 16(3), May 1996. http://www.computer.org/pubs/cg&a/articles/g30064.pdf.

[68] Thomas Kao, David M. Mount, and Alan Saalfeld. Dynamic maintenance of Delaunay triangulations. Technical report, CS Dept., U. of Maryland at College Park, Jan. 1991. CS-TR-2585.

[69] Reinhard Klein, Gunther Liebich, and W. Straßer. Mesh reduction with error control. In *Proceedings of Visualization '96*, pages 311–318, October 1996.

[70] Mark P. Kumler. An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica*, 31(2), Summer 1994. Monograph 45.

[71] Charles L. Lawson. Software for C^1 surface interpolation. In John R. Rice, editor, *Mathematical Software III*, pages 161–194. Academic Press, NY, 1977. (Proc. of symp., Madison, WI, Mar. 1977).

[72] D.T. Lee and Bruce J. Schachter. Two algorithms for constructing a Delaunay triangulation. *Intl. J. Computer and Information Sciences*, 9(3):219–242, 1980.

[73] Jay Lee. A drop heuristic conversion method for extracting irregular network for digital elevation models.

In *GIS/LIS '89 Proc.*, volume 1, pages 30–39. American Congress on Surveying and Mapping, Nov. 1989.

[74] Jay Lee. Comparison of existing methods for building triangular irregular network models of terrain from grid digital elevation models. *Intl. J. of Geographical Information Systems*, 5(3):267–285, July-Sept. 1991.

[75] J.-G. Leu and L. Chen. Polygonal approximation of 2-D shapes through boundary merging. *Pattern Recognition Letters*, 7(4):231–238, April 1988.

[76] Michael Lounsbery. *Multiresolution Analysis for Surfaces of Arbitrary Topological Type.* PhD thesis, Dept. of Computer Science and Engineering, U. of Washington, 1994. http://www.cs.washington.edu/research/projects/grail2/www/pub/pub-author.html.

[77] Kok-Lim Low and Tiow-Seng Tan. Model simplification using vertex-clustering. In *1997 Symposium on Interactive 3D Graphics*. ACM SIGGRAPH, 1997. To appear, http://www.iscs.nus.sg/∼tants/.

[78] David Luebke. Hierarchical structures for dynamic polygonal simplification. TR 96-006, Department of Computer Science, University of North Carolina at Chapel Hill, 1996.

[79] Stephane G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[80] Benoit Mandelbrot. *Fractals – form, chance, and dimension*. Freeman, San Francisco, 1977.

[81] Michael Margaliot and Craig Gotsman. Approximation of smooth surfaces and adaptive sampling by piecewise-linear interpolants. In Rae Earnshaw and John Vince, editors, *Computer Graphics: Developments in Virtual Environments*, pages 17–27. Academic Press, London, 1995.

[82] Donald E. McClure. Nonlinear segmented function approximation and analysis of line patterns. *Quarterly of Applied Math.*, 33(1):1–37, Apr. 1975.

[83] Donald E. McClure and S. C. Shwartz. A method of image representation based on bivariate splines. Technical report, Center for Intelligent Control Systems, MIT, Mar. 1989. CICS-P-113.

[84] Robert B. McMaster. Automated line generalization. *Cartographica*, 24(2):74–111, 1987.

[85] Robert B. McMaster. The geometric properties of numerical generalization. *Geographical Analysis*, 19(4):330–346, Oct. 1987.

[86] Robert B. McMaster and K. S. Shea. *Generalization in Digital Cartography*. Assoc. of American Geographers, Washington, D.C., 1992.

[87] Joseph S. B. Mitchell. Approximation algorithms for geometric separation problems. Technical report, Dept. of Applied Math. and Statistics, State U. of New York at Stony Brook, July 1993.

[88] Edmond Nadler. Piecewise linear best L_2 approximation on triangulations. In C. K. Chui et al., editors, *Approximation Theory V*, pages 499–502, Boston, 1986. Academic Press.

[89] Gregory M. Nielson. Tools for triangulations and tetrahedrizations and constructing functions defined over them. In Gregory M. Nielson, Hans Hagen, and Heinrich Mueller, editors, *Scientific Visualization: Overviews, Methodologies, and Techniques*. IEEE Comput. Soc. Press, 1997.

[90] Paul Ning and Jules Bloomenthal. An evaluation of implicit surface tilers. *Computer Graphics and Applications*, pages 33–41, Nov. 1993.

[91] Theodosios Pavlidis. *Structural Pattern Recognition.* Springer-Verlag, Berlin, 1977.

[92] Thomas K. Peucker and David H. Douglas. Detection of surface-specific points by local parallel processing of discrete terrain elevation data. *Computer Graphics and Image Processing*, 4:375–387, 1975.

[93] Michael F. Polis, Stephen J. Gifford, and David M. McKeown, Jr. Automating the construction of large-scale virtual worlds. *Computer*, pages 57–65, July 1995. http://www.cs.cmu.edu/∼MAPSLab.

[94] Michael F. Polis and David M. McKeown, Jr. Iterative TIN generation from digital elevation models. In *Conf. on Computer Vision and Pattern Recognition (CVPR '92)*, pages 787–790. IEEE Comput. Soc. Press, 1992. http://www.cs.cmu.edu/∼MAPSLab.

[95] Michael F. Polis and David M. McKeown, Jr. Issues in iterative TIN generation to support large scale simulations. In *Proc. of Auto-Carto 11 (Eleventh Intl. Symp. on Computer-Assisted Cartography)*, pages 267–277, November 1993. http://www.cs.cmu.edu/∼MAPSLab.

[96] Jean Ponce and Olivier Faugeras. An object centered hierarchical representation for 3D objects: The prism tree. *Computer Vision, Graphics, and Image Processing*, 38:1–28, 1987.

[97] Enrico Puppo, Larry Davis, Daniel DeMenthon, and Y. Ansel Teng. Parallel terrain triangulation using the

Connection Machine. Technical Report CAR-TR-561, CS-TR-2693, Center for Automation Research, University of Maryland, College Park, Maryland, June 1991.

[98] Enrico Puppo, Larry Davis, Daniel DeMenthon, and Y. Ansel Teng. Parallel terrain triangulation. *Intl. J. of Geographical Information Systems*, 8(2):105–128, 1994.

[99] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1:244–256, 1972.

[100] Kevin J. Renze and James H. Oliver. Generalized surface and volume decimation for unstructured tessellated domains. In *VRAIS '96 (IEEE Virtual Reality Annual Intl. Symp.)*, Mar. 1996. Submitted.

[101] Shmuel Rippa. Adaptive approximation by piecewise linear polynomials on triangulations of subsets of scattered data. *SIAM J. Sci. Stat. Comput.*, 13(5):1123–1141, Sept. 1992.

[102] Shmuel Rippa. Long and thin triangles can be good for linear interpolation. *SIAM J. Numer. Anal.*, 29(1):257–270, Feb. 1992.

[103] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3), Aug. 1996. Proc. Eurographics '96.

[104] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 455–465, Berlin, 1993. Springer-Verlag. Proc. of Conf., Genoa, Italy, June 1993. (Also available as IBM Research Report RC 17697, Feb. 1992, Yorktown Heights, NY 10598).

[105] Hanan Samet. *Applications of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[106] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[107] Lori Scarlatos. *Spatial Data Representations for Rapid Visualization and Analysis.* PhD thesis, CS Dept, State U. of New York at Stony Brook, 1993.

[108] Lori Scarlatos and Theo Pavlidis. Hierarchical triangulation using cartographic coherence. *CVGIP: Graphical Models and Image Processing*, 54(2):147–161, March 1992.

[109] Lori L. Scarlatos and Theo Pavlidis. Optimizing triangulations by curvature equalization. In *Proc. Visualization '92*, pages 333–339. IEEE Comput. Soc. Press, 1992.

[110] Francis Schmitt and Xin Chen. Fast segmentation of range images into planar regions. In *Conf. on Computer Vision and Pattern Recognition (CVPR '91)*, pages 710–711. IEEE Comput. Soc. Press, June 1991.

[111] Francis Schmitt and Xin Chen. Geometric modeling from range image data. In *Eurographics '91*, pages 317–328, Amsterdam, 1991. North-Holland.

[112] Francis Schmitt and Behrouz Gholizadeh. Adaptative polyhedral approximation of digitized surfaces. In *Computer Vision for Robots*, volume 595, pages 101–108. SPIE, 1985.

[113] Francis J. M. Schmitt, Brian A. Barsky, and Wen-Hui Du. An adaptive subdivision method for surface-fitting from sampled data. *Computer Graphics (SIGGRAPH '86 Proc.)*, 20(4):179–188, Aug. 1986.

[114] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics.* Prentice Hall, 1996. Code at http://www.cs.rpi.edu:80/~martink/.

[115] William J. Schroeder and Boris Yamrom. A compact cell structure for scientific visualization. In *SIGGRAPH '94 Course Notes CD-ROM, Course 4: Advanced Techniques for Scientific Visualization*, pages 53–59. ACM SIGGRAPH, July 1994.

[116] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proc.)*, 26(2):65–70, July 1992.

[117] Cláudio T. Silva, Joseph S. B. Mitchell, and Arie E. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. In *Proc. Visualization '95*. IEEE Comput. Soc. Press, 1995. http://www.cs.sunysb.edu:80/~csilva/claudio-papers.html.

[118] Marc Soucy and Denis Laurendeau. Multi-resolution surface modeling from multiple range views. In *Conf. on Computer Vision and Pattern Recognition (CVPR '92)*, pages 348–353, June 1992.

[119] Marc Soucy and Denis Laurendeau. Multiresolution surface modeling based on hierarchical triangulation. *Computer Vision and Image Understanding*, 63(1):1–14, 1996.

[120] David A. Southard. Piecewise planar surface models from sampled data. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 667–680, Tokyo, 1991. Springer-Verlag.

[121] Steven L. Tanimoto and Theo Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[122] David C. Taylor and William A. Barrett. An algorithm for continuous resolution polygonalizations of a discrete surface. In *Proc. Graphics Interface '94*, pages 33–42, Banff, Canada, May 1994. Canadian Inf. Proc. Soc.

[123] Ivan Tomek. Two algorithms for piecewise-linear continuous approximation of functions of one variable. *IEEE Trans. Computers*, C-23:445–448, Apr. 1974.

[124] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proc.)*, 26(2):55–64, July 1992.

[125] K. J. Turner. *Computer perception of curved objects using a television camera.* PhD thesis, U. of Edinburgh, Scotland, November 1974.

[126] Amitabh Varshney. *Hierarchical Geometric Approximations*. PhD thesis, Dept. of CS, U. of North Carolina, Chapel Hill, 1994. TR-050.

[127] Amitabh Varshney, Pankaj K. Agarwal, Frederick P. Brooks, Jr., William V. Wright, and Hans Weber. Generating levels of detail for large-scale polygonal models. Technical report, Dept. of CS, Duke U., Aug. 1995. CS-1995-20, http://www.cs.duke.edu/department.html#techrept.

[128] Brian Von Herzen and Alan H. Barr. Accurate triangulations of deformed, intersecting surfaces. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):103–110, July 1987.

[129] Robert Weibel. Models and experiments for adaptive computer-assisted terrain generalization. *Cartography and Geographic Information Systems*, 19(3):133–153, 1992.

[130] Ellen R. White. Assessment of line-generalization algorithms using characteristic points. *The American Cartographer*, 12(1):17–27, 1985.

[131] Lance Williams. Pyramidal parametrics. *Computer Graphics (SIGGRAPH '83 Proc.)*, 17(3):1–11, July 1983.

Multiresolution Modeling for Fast Rendering

Paul S. Heckbert and Michael Garland
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania, 15213-3891, USA
ph@cs.cmu.edu, garland@cs.cmu.edu

Abstract

Three dimensional scenes are typically modeled using a single, fixed resolution model of each geometric object. Renderings of such a model are often either slow or crude, however: slow for distant objects, where the chosen detail level is excessive, and crude for nearby objects, where the detail level is insufficient. What is needed is a *multiresolution model* that represents objects at multiple levels of detail. With a multiresolution model, a rendering program can choose the level of detail appropriate for the object's screen size so that less time is wasted drawing insignificant detail. The principal challenge is the development of algorithms that take a detailed model as input and automatically simplify it, while preserving appearance. Multiresolution techniques can be used to speed many applications, including real time rendering for architectural and terrain simulators, and slower, higher quality rendering for entertainment and radiosity. This paper surveys existing multiresolution modeling techniques and speculates about what might be possible in the future.

Keywords: multiresolution model, level of detail, rendering, simplification, approximation.

The first section of this paper discusses the goals and computational cost of complex scene rendering and explains why the use of a model with the appropriate level of detail is important. Section 2 summarizes the goals of multiresolution modeling and section 3 summarizes several data structures for achieving these goals. The paper concludes with a comparison of these data structures.

1 Rendering Complex Scenes

In computer graphics, we would like to render complex scenes such as terrains, cities, building interiors, molecules, and biological structures as quickly as possible. These might contain millions of geometric primitives (polygons, spheres, etc.).

1.1 Optimized Rendering

Early rendering algorithms performed transformation, clipping, sorting, and hidden line or hidden surface removal. For a scene of n primitives, such algorithms had costs of $O(n^2)$ or $O(n \log n)$. In the late 70's and early 80's, flight simulators capable of displaying several thousand polygons in real time became available, but at a price of several million dollars. Advances in graphics hardware and memory technology have allowed brute force algorithms such as the z-buffer to supplant the earlier, sorting-based algorithms. These advances have sped rendering dramatically, so that it is now possible to draw ten thousand shaded polygons in real time on a $40,000 workstation.

The time cost of rendering a scene of n primitives with a total screen area of a_c[1] with a simple z-buffer algorithm is $\Theta(n+a_c)$[2]. The cost is linear in the number of primitives because of transforming and clipping, and linear in the screen area due to scan conversion and shading. When the scene is very detailed and most surfaces project to an area smaller than a pixel, the transformation and clipping cost (n) dominates. Algorithms with costs that are linear in n are an improvement over earlier algorithms, but they are still slower than necessary.

In highly complex scenes, the user cannot see all of the primitives at one time because many of them are either off-screen, occluded, or too small to be seen. If a simple z-buffer algorithm is used, as described above, then each of the primitives in the scene must be transformed and clipped, even if it is way off screen, and each on-screen primitive must be scan converted and (potentially) shaded, even if it is occluded. Off-screen primitives can be culled quickly using hierarchical bounding volumes [4], octrees,

[1] a_c is the sum of screen areas of clipped primitives, ignoring occlusion. It could be much greater than the number of pixels in the image if the depth complexity is high.

[2] Asymptotic complexity notation: $O(f(n))$ is an upper bound, $\Theta(f(n))$ is both an upper bound and lower bound. We discuss only worst case cost in this paper.

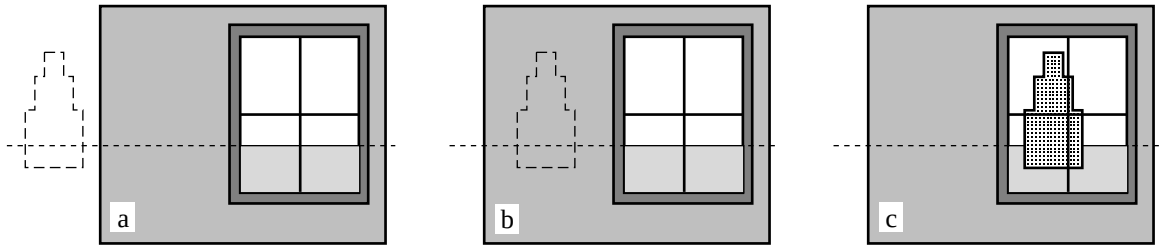


Figure 1: Three views out a window. a) Building is off-screen. b) Building is on-screen, but occluded by wall. c) Building is on-screen, and visible through window.

or other spatial data structures.

Occluded objects can be weeded out either with sophisticated visibility analysis techniques [24] or more brute force z-buffer pyramids [10]. Together, these optimizations would reduce the cost of rendering n primitives with clipped screen area of a_c from $\Theta(n+a_c)$ to $\Theta(n_v+a_v)$ not counting preprocessing, where n_v is the number of visible primitives, and a_v is their total screen area. Note that $0 \leq n_v \leq n$ and $0 \leq a_v \leq a_c$, but the relative sizes of these variables are very scene-dependent.

The third optimization for complex scenes, speedier handling of small objects, is employed by few existing rendering systems. It is the focus of this paper.

Figure 1 shows three views that illustrate the differences between these optimizations of the z-buffer algorithm. The scene is a building that is (sometimes) visible through a window in a wall. Suppose the visible facade of the building is modeled using n polygons, the wall and ground plane are modeled using a small, bounded number of polygons, and the number of pixels in the image is a. The costs of various z-buffer-based algorithms on these three views are:

ALGORITHM	Fig. 1a	Fig. 1b	Fig. 1c
unoptimized	$\Theta(n+a)$	$\Theta(n+a)$	$\Theta(n+a)$
with off-screen culling	$\Theta(a)$	$\Theta(n+a)$	$\Theta(n+a)$
with off-screen & visibility culling	$\Theta(a)$	$\Theta(a)$	$\Theta(n+a)$
with off-screen & visibility culling, & multiresolution modeling (projection)	$\Theta(a)$	$\Theta(a)$	$\Theta(a)$

Figure 1a, where the building is off-screen, is optimized by simple off-screen culling. Figure 1b is more difficult to optimize; visibility culling is required to generate this picture quickly. Figure 1c is the most difficult of all. In the case where $n \gg a$, there are many more polygons than necessary to model the building, and they're almost all visible, so neither off-screen nor visibility culling will render this view efficiently.

What is needed are methods for simplifying an object that has been modeled with excessive detail so that arbitrary views can be rendered quickly, ideally with a cost (not counting preprocessing) proportional to the number of pixels, but independent of scene complexity. Such an algorithm would be optimal on a computer where writing a pixels takes $\Theta(a)$ time[3]. For any particular view, this goal is trivially achievable, since the scene could be modeled as a plane tiled with one rectangular polygon per screen pixel. The challenge is to find a model that works for any viewpoint, while remaining fast and compact.

2 Multiresolution Modeling

Geometric models typically describe each shape in a scene with a single representation. The scale or *level of detail* at which each object is modeled is fixed. Such a model is called a *fixed resolution model*. When rendering with a perspective projection, distant objects project to a small screen area and nearby objects project to a large area. In a complex scene, the dynamic range of these screen areas can be very great. Rendering such a scene using a fixed resolution model can be very inefficient, as seen with figure 1c.

The best method for optimizing the rendering of small and distant objects is *multiresolution modeling*: the description of geometry and surface attributes such as color and texture at a variety of scales. Depending on the screen size of a given object or cluster of objects, the appropriate level of detail within the model would be chosen [4]. The appropriate level for a given view is the coarsest level that looks the same as the finest level. Thus, nearby objects would be rendered using a detailed model, while distant objects would be rendered using a coarse model.

2.1 Applications

Multiresolution modeling has many applications. The primary one is fast display, both for real-time rendering and for high quality images that might take minutes or hours

[3] On a parallel machine with $\Theta(a)$ processors, we might be able to render in constant time.

of compute time. Architectural walkthroughs, flight simulators, scientific visualization, computer-aided design, movie special effects, and virtual reality are natural applications.

In the past, the person who models a 3-D scene has often been the one who runs the renderer. Since this person knows what the camera will see, he or she can model the scene appropriately, including only those details that will be seen.

For 3-D animation, objects that are seen at widely varying scales are often modeled at two or more levels of detail: a "fine" model for closeups and a "coarse" model for distant shots. As the object recedes into the distance during animation, the scene description will often be manually altered to switch from the fine model to the coarse model. This procedure can be automated by including both the coarse and fine models in the scene description, and using a measure of screen size, such as the area of the projected bounding volume of the object, to choose between the levels of detail.

In flight simulators and virtual reality, the person preparing the model is not the person choosing the viewpoints (i.e. the pilot). For these applications, the modeler must include enough detail that the user can move through the whole scene without losing the illusion of reality.

Multiresolution modeling is also useful in radiosity and other global illumination algorithms. In rendering, we determine the visible surfaces within a viewing pyramid and create a picture of them, while in radiosity, we determine the visible surfaces within a hemisphere and integrate them. These tasks are very similar.

Radiosity algorithms subdivide each input polygon into many *elements*. Early radiosity algorithms had a cost that was quadratic in the number of elements because they used a fixed subdivision and they calculated the amount of light reflected between each pair of elements. These algorithms wasted most of their time computing insignificant light transfers between distant objects. The hierarchical radiosity algorithm uses adaptive subdivision instead: when gathering light into each element, it subdivides distant polygons coarsely and nearby polygons finely [11]. With this improvement, the algorithm's cost is linear in the number of elements, but it is still quadratic in the number of polygons, since pairwise subdivision starts with the given polygons. Thus, the algorithm is fast only for simple scenes consisting of a few large polygons. This is unacceptable for complex scenes.

The quadratic cost term of hierarchical radiosity could be eliminated, yielding an algorithm whose complexity would be linear in the number of polygons or better, if multiresolution modeling were used, clustering distant objects and treating them as a single unit. Rushmeier et al. have recently employed multiresolution models for radiosity, but their model creation system was not automated [20].

2.2 Model Use

Selection of the appropriate level of detail during rendering is easy, requiring only hierarchical bounding volumes and fast estimates of screen area. If levels of detail are selected discretely, however, this will cause visible artifacts in the spatial or temporal continuity of images. Experience has shown that consistency is often more important than correctness in computer graphics. Level-switching artifacts can be eliminated by smoothing the transitions using linear interpolation of geometry and color.

2.3 Model Creation

Creation of a multiresolution model is quite difficult. Although multiresolution modeling is an old idea, most existing such databases have been created by hand. In flight simulator and architectural walkthrough systems that employ multiresolution modeling, laborious manual database preparation is still required, to the best of our knowledge [29, 9, 8]. Renderman can render multiresolution models but it supplies no automatic tools for generating them [26].

The principal challenge of multiresolution modeling is to find a set of algorithms that can take a complex scene description as input, including both geometry and surface attributes such as color and texture, and automatically generate data structures that allow rapid rendering of the scene from any viewpoint. For greatest flexibility, the system should allow arbitrary input (e.g. a set of polygons with no topological information) and not assume that the input comes with a hierarchy. It is most important that the rendering be fast and the appearance of the scene be preserved, but it is also desirable that the preprocessing time and memory requirements be low.

2.4 Preservation of Appearance

Quantifying the "preservation of appearance" objective can aid in the development of algorithms. The real measure of appearance is the raster image output by the renderer. This is more important than precise preservation of topology or geometry. We would therefore like an *image error metric* that measures the overall difference between two images. This metric should measure the difference between an image $f(x, y)$ rendered using the fully detailed input model and an image $\hat{f}(x, y)$ rendered using the multiresolution model (the approximation). We'd like the two images to be indistinguishable. Ultimately, human viewers are the judges, so the best error metric would entail a model of the human visual system, a very complex

topic. Useful results can be obtained with much simpler error metrics, however. These can be viewed as crude approximations to human perception.

A simple starting point is the sum of squared distances in RGB color space between corresponding pixels:

$$E(f,\hat{f}) = \sum_{x,y} \left\| f(x,y) - \hat{f}(x,y) \right\|^2$$

This error metric can be improved by adding differential weighting for the color channels, nonlinear sensitivity to radiance, and spatial filtering. Any multiresolution modeling data structure that is developed should be validated either with perceptual tests using human viewers, or with a good image error metric.

3 Multiresolution Data Structures

For rendering, a model can be regarded as an abstract data type that supports queries of the form:

> **what does this object look like when viewed from a given viewpoint, with a given resolution?**

Any fast, compact data structure for such queries would suffice as a multiresolution representation. We discuss the following six data structures as possible candidates:

1. image pyramids,
2. volume pyramids,
3. texture and reflectance,
4. pictures from multiple angles,
5. ray space, and
6. polygonal models.

Several of these are rather speculative.

3.1 Image Pyramids

In two dimensions, the most natural multiresolution model is the image pyramid. Image pyramids are ubiquitous in image processing and computer vision [18], and are also widely used to optimize texture mapping [28, 12]. Image pyramids are an attractive multiresolution model because they are so easily resampled. Unfortunately, they are limited to 2-D.

3.2 Volume Pyramids

More natural as a 3-D multiresolution modeling data structure is the 3-D volume pyramid. Volume pyramids are very helpful for fast volume rendering [21], but as a surface representation, they are bulky and crude.

3.3 Texture and Reflectance

Texture and reflectance models are a form of multiresolution modeling. They model the visible effects of fine-scale variation in geometry and surface attributes that are too small to be modeled using geometry. Texture mapping is commonly used to model features whose geometry is smaller than a pixel but whose visible patterns are bigger than a pixel, and reflectance models describe features whose patterns are much smaller than a pixel.

In pictures or animation encompassing a wide range of scales, the choice of representation should be allowed to vary from frame to frame and from pixel to pixel. When flying over a terrain, for example, mountains in the far distance are best modeled as a textured plane, and the appearance of the trees on the mountain are best modeled statistically, in the reflectance model. In the near distance, what was texture (the mountain) should become geometry, and some of the larger features influencing reflectance (the trees) should become texture. Finally, in a closeup, the trees become geometry.

This idea has been proposed by Perlin [16], Kajiya [15], and others, but has never been implemented in a general way. The best progress along these lines has been made in generating bidirectional reflectance distribution functions (BRDF's) from geometry [2, 7, 27] and in smoothing the transitions between BRDF's, bump mapping, and displacement mapping [1].

3.4 Pictures from Multiple Angles

In architecture, initial design is typically done by sketching a building from multiple viewpoints. When we watch film or video, we are seeing a sequence of still images of objects from different viewpoints. Such representations suffice, in a practical sense, to define a 3-D shape. Hence the idea to represent an object not with a set of surface primitives, but with a set of pictures.

This approach has the obvious advantages that the representation is of the same form as the output of a renderer (a picture) and that image pyramids could be used, allowing quick extraction of an image of the desired resolution. The major disadvantage is that the appearance of the object from arbitrary viewpoints is not directly available; in the process of generating the pictures, information is lost.

Intermediate views can differ from the chosen views because of either occlusion or specular reflection. In a scene where sunlight shines directly through a tunnel, for example, only certain views see the sun, so if those views were not chosen, the system would have difficulty generating accurate intermediate views. And in a scene containing mirrors, only certain views see a reflection of the light sources, so again, intermediate views would be difficult to interpolate correctly.

Approximate intermediate views can be generated automatically if the correspondence between pixels of the chosen views is known, and the correspondences can be derived if the z-buffers of the chosen views are available [3]. This technique does not solve the complications caused by occlusion and specular reflection, however, so there are large unsolved problems to make this approach viable as a general multiresolution modeling data structure.

3.5 Ray Space

Another approach to multiresolution modeling is to treat an object's appearance in terms of ray queries, the type of queries performed in a ray tracing algorithm. A ray query takes a ray and returns the color traveling backward along that ray. Existing data structures for fast ray queries require huge memories, so this approach does not seem as promising as the use of textures and polygons. This approach is attractive, however, because it provides a unified, high level abstraction that allows us to blur the distinction between geometry and surface attributes such as BRDF's.

3.6 Polygonal Models

The final approach to multiresolution modeling that we consider, polygonal models, has received the most work, so we discuss it in the greatest detail.

The principal challenge when using a polygonal model for multiresolution modeling is *simplification*: automatically converting a detailed model into a simpler one that faithfully represents the underlying object. We seek algorithms that will minimize both the number of polygons in the simplified model and the error of the approximation.

Simplification algorithms differ greatly depending on the topology of the polygonal model. The simplest are curve models, consisting of a sequence of points or line segments (not really polygons at all). Next in complexity are mesh models, which consist of a network of polygons forming a single, continuous surface. The most general class of polygonal models are *polyhedral models*, where arbitrary topology is allowed. The latter class is the most relevant to multiresolution modeling.

3.6.1 Curve Simplification

Numerous algorithms for approximating one piecewise linear curve with another have been developed [6]. It is possible to find the least squares optimal m-segment approximation to an n-segment curve in time $O(n^3 log m)$ using dynamic programming [14]. Unfortunately, this is too slow for use on complex curves, and it does not appear to generalize to surfaces. Curve simplification algorithms may be of some guidance in our search for surface simplification methods, however.

3.6.2 Mesh Simplification

The aspect of polygonal simplification that has received the most attention is the simplification of surface meshes. Such models are commonly generated from digital sampling of real world objects. The data tend to be dense and redundant, so they can typically be drastically simplified without significant loss of fidelity. We consider grids with rectangular topology first, then height fields, and finally general meshes.

If the mesh is a grid with rectangular topology then a natural simplification technique is to low pass filter the data and then discard every other row and every other column from the grid, performing what is called "decimation by 2" in signal processing. Williams proposed this as a multiresolution modeling technique both to reduce the time needed to transform polygons and to reduce the need for antialiasing [28].

Another area of research is the generation of compact triangulations from digital terrain data and other height fields [17, 22]. Given a regular grid of height samples, the task is to construct a triangle mesh that closely approximates the actual surface with a small number of vertices. Typically, these algorithms are constructive; they begin with a minimal set of points and then add new points until the error of the approximation is below some threshold. Various criteria are used to select which points to add. Some of these algorithms are quite slow. For example, Polis and McKeown's algorithm required 18 hours to achieve a 40-to-1 simplification of a 4,000,000 point terrain. These applications compute the simplified model off-line, however, so for them, preprocessing speed was much less important than accuracy and simplification.

A broader problem is the simplification of general meshes. The typical goal here is to digitize a real world object and construct a compact surface description of it. In [5], DeHaemer and Zyda present an adaptive subdivision algorithm that fits polygons to a set of samples. This algorithm combines surface reconstruction and simplification; it constructs a simple surface directly from the data.

Other algorithms require a mesh as input. Schroeder, Zarge, and Lorensen [23] propose a decimation algorithm. They iteratively remove unimportant points from the mesh, performing local retriangulations to preserve the surface. Turk [25] describes a related approach. He selects a set of points on the surface that will become the vertices of a new mesh and uses point repulsion to achieve good coverage of the surface. A new triangulated mesh is generated by combining the old and new vertices. The

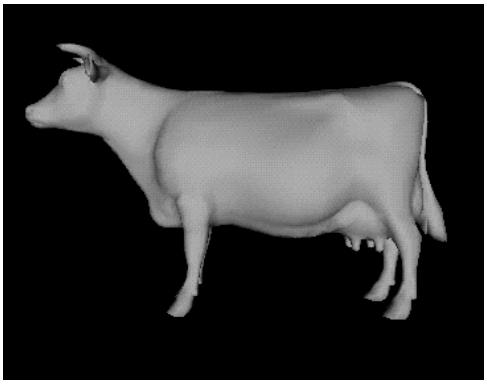


Figure 2: Original cow (5804 triangles)

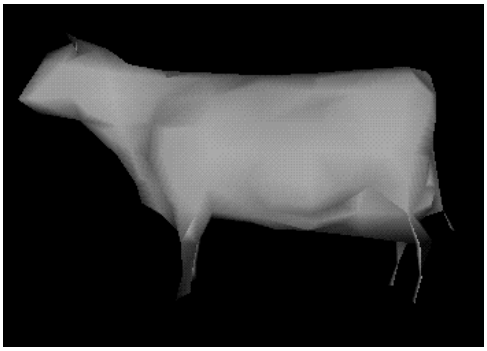


Figure 3: Simplified cow (658 triangles)

Figure 4: Original Beethoven (4998 triangles)

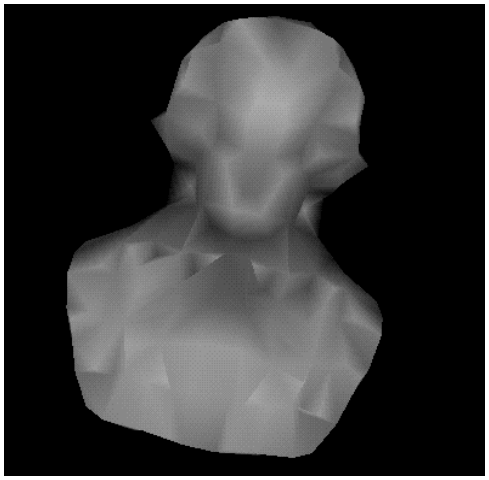


Figure 5: Simplified Beethoven (652 triangles)

old vertices are then iteratively deleted, using local retriangulations to preserve the topology of the surface. Most recently, Hoppe et al. [13] present an algorithm for optimizing fairly general surface meshes. They cast the problem in terms of minimizing an energy function that captures the conflicting goals of mesh simplification and error minimization.

3.6.3 Polyhedral Simplification

Rossignac and Borrel [19] have made one of the few efforts to address the simplification of general, polyhedral models with arbitrary topology. Their motivation is to speed interactive viewing of complex objects, so they seek a minimal set of polygons and lines that suggests a shape to the user. Given a polyhedral model that has been triangulated, they subdivide its bounding volume into a grid of boxes. All vertices within each box are merged together into a new representative vertex. A simplified model is then synthesized from these representative vertices by forming triangles according to the original topology.

This is essentially a signal processing approach: the model is filtered, resampled, and reconstructed. As with all sampling algorithms, aliasing can arise. One weakness of the algorithm is that averaging vertices removes high-frequency details that might have significant importance (features on a face, for example). Another weakness is that the results are not invariant to rigid body motion of the input model; if the model is rotated or translated, the output model ripples like a point-sampled image.

Figures 2–5 show results from an algorithm based on Rossignac and Borrel's. Figure 2 shows the original model of a cow and figure 3 is the result of simplification. With this model, the results are good; viewed from a distance the models are fairly similar. The second example is a bust of Beethoven, figure 4, whose simplified version, figure 5, illustrates the loss of important detail. A better algorithm would use more polygons in areas of high surface curvature and fewer polygons in areas of low curvature.

4 Conclusions

Most current rendering algorithms are inefficient when rendering very complex scenes. Their cost is linear in scene complexity, and this is unacceptable when the complexity is very high. When given a scene with many more surface primitives than pixels, z-buffer algorithms waste a lot of time transforming and clipping objects smaller than a pixel that have negligible impact on the final picture. Using multiresolution modeling it may be possible to render scenes in time proportional to screen area but independent of scene complexity.

The six data structures for multiresolution modeling that we have discussed are evaluated below:

Image Pyramids. Image pyramids are very good for planar and smoothly curved surfaces, but they do not represent real 3-D features well.

Volume Pyramids. Total brute force. The results will look blurry or blocky unless a huge memory is available.

Texture and Reflectance Models. Texture and reflectance don't represent geometry, but they are excellent, compact representations for fine detail, so they would be important components of any complete multiresolution modeling system. Much work remains to be done to derive textures from geometric models, however.

Pictures from Multiple Angles. This approach is intriguing, but can the problems of specular objects and occlusion be solved? Perhaps it should be used primarily for diffuse, nearly-convex objects.

Ray Space. Also intriguing, but the memory requirements may be extreme.

Polygonal Models. Polygonal models will probably form the core of a successful multiresolution modeling system, since they are the simplest, most versatile representation for geometry.

The polygonal simplification methods we discussed were developed with different goals in mind. Many of the simplification algorithms are limited to meshes, they are slow, and they consider shape only when doing their simplification, not material attributes such as color, specularity, or texture. Further work is needed to adapt them to the goals of multiresolution modeling.

Rossignac and Borrel's simplification algorithm is the most general, since it accepts polyhedral models with arbitrary topology as input. It can achieve greater simplification since it is free to change the topology of models. On the negative side, this algorithm shows artifacts of the clustering grid and it does not preserve detail as well as might be possible. Nevertheless, this algorithm is a good starting point for future research. With additional work on preservation of appearance, a simplification algorithm well suited for fast rendering could be developed.

A full multiresolution modeling system would need to combine several of these data structures in order to represent objects using a combination of geometry, texture, and reflectance, and it would need to smooth the transitions between representations during rendering.

Acknowledgments

Thanks to Tom Funkhouser, Jon Webb, Andy Witkin, Dave McKeown, and Jarek Rossignac for valuable discussions. This work was supported by ARPA contract F19628-93-C-0171.

References

[1] Barry G. Becker and Nelson L. Max. Smooth transitions between bump rendering algorithms. In *SIGGRAPH '93 Proceedings*, pages 183–189. ACM, 1993.

[2] Brian Cabral, Nelson Max, and Rebecca Springmeyer. Bidirectional reflection functions from surface bump maps. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):273–281, July 1987.

[3] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *SIGGRAPH '93 Proceedings*, pages 279–288. ACM, 1993.

[4] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, Oct. 1976.

[5] Michael DeHaemer, Jr. and Michael J. Zyda. Simplification of objects rendered by polygonal approximations. *Computers and Graphics*, 15(2):175–184, 1991.

[6] James George Dunham. Optimum uniform piecewise linear approximation of planar curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(1):67–75, January 1986.

[7] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, May 1992.

[8] Thomas A. Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, CS Division, UC Berkeley, 1993.

[9] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *1992 Symposium on Interactive 3D Graphics*, pages 11–20, 1992. Special issue of Computer Graphics.

[10] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *SIGGRAPH '93 Proceedings*, pages 231–238. ACM, 1993.

[11] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):197–206, July 1991.

[12] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, Nov. 1986.

[13] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *SIGGRAPH '93 Proceedings*, pages 19–26, Aug. 1993.

[14] Insung Ihm and Bruce Naylor. Piecewise linear approximations of digitized space curves with applications. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 545–569, Tokyo, 1991. Springer-Verlag.

[15] James T. Kajiya. Anisotropic reflection models. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):15–21, July 1985.

[16] Ken Perlin. A unified texture/reflectance model. In *SIGGRAPH '84 Advanced Image Synthesis seminar notes*, July 1984.

[17] Michael F. Polis and David M. McKeown, Jr. Issues in iterative TIN generation to support large scale simulations. In *Proc. of 11th Intl. Symp. on Computer Assisted Cartography (AUTOCARTO 11)*, pages 267–277, November 1993.

[18] Azriel Rosenfeld, editor. *Multiresolution Image Processing and Analysis*. Springer, Berlin, 1984. Leesberg, VA, July 1982.

[19] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. Technical report, Yorktown Heights, NY 10598, February 1992. IBM Research Report RC 17697 (#77951). Also appeared in the *IFIP TC 5.WG 5.10 II Conference on Geometric Modeling in Computer Graphics*, Genova, Italy, 1993.

[20] Holly E. Rushmeier, Charles Patterson, and Aravindan Veerasamy. Geometric simplification for indirect illumination calculations. In *Proc. Graphics Interface '93*, pages 227–236, Toronto, Ontario, May 1993. Canadian Inf. Proc. Soc.

[21] Georgios Sakas and Matthias Gerth. Sampling and anti-aliasing of discrete 3-D volume density textures. In *Eurographics '91*, pages 87–102, 527, Amsterdam, 1991. North-Holland.

[22] Lori Scarlatos and Theo Pavlidis. Hierarchical triangulation using cartographic coherence. *CVGIP: Graphical Models and Image Processing*, 54(2):147–161, March 1992.

[23] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):65–70, July 1992.

[24] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments.* PhD thesis, CS Division, UC Berkeley, October 1992. Tech. Report UCB/CSD-92-708.

[25] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):55–64, July 1992.

[26] Steve Upstill. *The Renderman Companion*. Addison Wesley, Reading, MA, 1990.

[27] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting reflectance functions from complex surfaces. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(4):255–264, July 1992.

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

[29] Michael J. Zyda. Course notes, book 10. Technical report, Graphics & Video Laboratory, Dept. of Computer Science, Naval Postgraduate School, Monterey, CA, November 1991.

Fast Triangular Approximation of Terrains and Height Fields

Michael Garland and Paul S. Heckbert
Carnegie Mellon University*

May 2, 1997

Abstract

We present efficient algorithms for approximating a height field using a piecewise-linear triangulated surface. The algorithms attempt to minimize both the error and the number of triangles in the approximation. The methods we examine are variants of the *greedy insertion algorithm*. This method begins with a simple triangulation of the domain as an initial approximation. It then iteratively finds the input point with highest error in the current approximation and inserts it as a vertex in the triangulation. We describe optimized algorithms using both Delaunay and data-dependent triangulation criteria. The algorithms have typical costs of $O((m+n)\log m)$, where n is the number of points in the input height field and m is the number of vertices in the final approximation. We also present empirical comparisons of several variants of the algorithms on large digital elevation models. We have made a C++ implementation of our algorithms publicly available.

Keywords: surface simplification, surface approximation, Delaunay triangulation, data-dependent triangulation, triangulated irregular network, greedy insertion.

1 Introduction

A *height field* is a set of data values sampled at points in a planar domain. Terrain data, a common type of height field, is used in many applications, including flight simulators, ground vehicle simulators, video games, and in computer graphics for entertainment. Computer vision uses height fields to represent range data acquired by stereo and laser range finders. In all of these applications, an efficient data structure for representing and displaying the height field is desirable.

Our primary motivation is to render height field data rapidly and with high fidelity. Since almost all graphics hardware uses the polygon as the fundamental building block for object description, it seems natural to represent the terrain as a mesh of polygonal elements. The raw sample data can be trivially converted into polygons by placing edges between each pair of neighboring samples (see Figure 6). However, for terrains of any significant size, rendering this full model is prohibitively expensive. For example, the 2,000,000 triangles in a $1,000 \times 1,000$ grid take about seven seconds to render on current mid-range graphics workstations, which can display roughly 10,000 triangles in real time (every 30th of a second). Even as the fastest graphics workstations speed up in coming years, typical workstations and personal computers will remain far slower. More fundamentally, the detail of the full model is highly redundant when it is viewed from a distance, and its use in such cases is unnecessary and wasteful. Many terrains have large, nearly planar regions which are well approximated by large polygons. Ideally, we would like to render models of arbitrary height fields with just enough detail for visual accuracy. Additionally, in systems which are highly constrained, we would like to use a less detailed model in order to conserve memory, disk space, or network bandwidth.

To render a height field quickly, we can use multiresolution modeling, preprocessing it to construct approximations of the surface at various levels of detail [2, 15]. When rendering the height field, we can choose an approximation with an appropriate level of detail and use it in place of the original.

In some applications, such as flight simulators, the speed of simplification is unimportant, because database preparation is done off-line, once, while rendering of the simplified terrain is done thousands of times. However, in more general computer graphics and computer animation applications, the scene being simplified might be changing, so a slow simplification method might be useless. Finding a simplification algorithm that is fast is therefore quite important to us.

Our focus in this paper will be to generate simplified

*Computer Science Dept., Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh PA 15213-3891, USA. {garland,ph}@cs.cmu.edu http://www.cs.cmu.edu/~garland/scape

models of a height field from the original model. The simplified model should accurately approximate the original model, it should be as compact as possible, and the process of simplification should be as rapid as possible. We measure the compactness of a model in terms of the number of triangles (or, proportionally, the number of vertices), since that is the major factor influencing rendering speed, and fast rendering with accurate approximations is our main goal.

We do not attempt to survey other methods for surface simplification here; for a taxonomy and comparison of polygonal surface simplification methods, see our survey paper [16].

The remainder of this paper contains the following sections: We begin by stating the problem we are solving. Next we describe the greedy insertion algorithm and present three simple optimizations that speed it up dramatically. We explore the use of both Delaunay triangulation and data-dependent triangulation. The paper concludes with a discussion of empirical results, ideas for future work, and a summary.

2 Background

A height field is a function of two variables, $H(x, y)$, which represents a surface. The input to our system is a height field represented by a set of sample points in the plane and a set of data values associated with those points. For now, we will assume this to be a set of height samples, but later we will consider generalized values. We will assume that the set of sample points is arranged on a rectangular grid at integral coordinates, but the methods we will describe are easily generalized to scattered data points. This discrete representation can be transformed into a continuous surface by triangulating its set of points and defining a piecewise-linear function over this triangulation. An approximation of H can be constructed by triangulating a subset of the sample points of H such that the triangulation covers the entire domain of H. Such an approximation is often referred to as a *triangulated irregular network*, or TIN, in the cartography literature.

Our goal is to find an approximation $S(x, y)$ of $H(x, y)$ which approximates H as accurately as possible using as few points as possible, and to compute its triangulation as quickly as possible. We will let n be the number of input points in H and m be the number of vertices in the approximation S.

Delaunay and Data-Dependent Triangulation. In this paper, we explore the use of both Delaunay and data-dependent triangulations for constructing the approximate surface. *Delaunay triangulation* is a purely two-dimensional method; it uses only the *xy* projections of the input points. It finds a triangulation that maximizes the minimum angle of all triangles, among all triangulations of a given point set [12, 20]. This helps to minimize the occurrence of very thin *sliver* triangles. *Data-dependent triangulation*, in contrast, uses the heights of points in addition to their x and y coordinates [7, 26]. It can achieve lower error approximations than Delaunay triangulation, but it generates more slivers.

The incremental Delaunay triangulation algorithm starts with a simple, initial triangulation and inserts the points as vertices in the triangulation one-by-one, performing local topological updates after each insertion [12, 20]. The procedure to insert a single vertex is illustrated in Figure 1, and works as follows: To insert a vertex A, locate its containing triangle, or, if it lies on an edge, delete that edge and find its containing quadrilateral. Add "spoke" edges from A to the vertices of this containing polygon. All perimeter edges of the containing polygon are suspect and their validity must be checked. An edge is valid iff it passes the circle test: if A lies outside the circumcircle of the triangle that is on the opposite side of the edge from A. All invalid edges must be swapped with the other diagonal of the quadrilateral containing them, at which point the containing polygon acquires two new suspect edges. The process continues until no suspect edges remain. The resulting triangulation is Delaunay.

Refinement Methods. Refinement is one of the general approaches to surface simplification [16]. Refinement methods are multi-pass algorithms that begin with an initial approximation and iteratively add new vertices to the triangulation until some goal is achieved. This goal is typically stated in terms of a desired error threshold or a maximum number of vertices. In order to choose which points to add to the approximation, refinement methods rank the available input points using some *importance measure*.

In choosing an importance measure, we reject those that make use of implicit knowledge about the nature of terrains, such as the existence of ridge lines. We would like our algorithms to apply to general height fields, where assumptions that are valid for terrains might fail. Even if we were to constrain ourselves to terrains alone, we are not aware of conclusive evidence suggesting that high fidelity results (as measured by an objective L_2 or L_∞ metric[1]) require high level knowledge of terrains. After experiment-

[1] In this paper, we use the following error metrics: We define the L_2 error between two n-vectors $\mathbf{u}$ and $\mathbf{v}$ as $||\mathbf{u}-\mathbf{v}||_2 = \left[\sum_{i=1}^{n}(u_i - v_i)^2\right]^{1/2}$. The L_∞ error, also called the *maximum error*, is $||\mathbf{u}-\mathbf{v}||_\infty = \max_{i=1}^{n}|u_i - v_i|$. We define the *squared error* to be the square of the L_2 error, and the *root mean square* or RMS error to be the L_2 error divided by $\sqrt{n}$. Optimization with respect to the L_2 and L_∞ metrics are called *least squares* and *minimax* optimization, and we call such solutions L_2–*optimal* and L_∞–*optimal*, respectively.

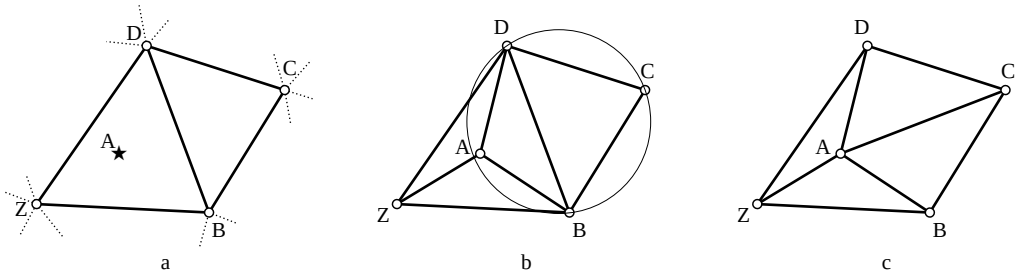


Figure 1: Incremental Delaunay triangulation: a) Point A is about to be inserted. Spoke edges from A to the containing polygon ZBD are added. b) The quadrilateral around suspect edge BD is checked using the circle test. The circumcircle of BCD contains A, so edge BD is invalid. c) After swapping edge BD for AC, edges BC and CD become suspect. The polygon ZBCD is the only area of the mesh that has changed.

ing with more complex alternatives [11], we chose one of the simplest importance measures, the local approximation error: $|H(x,y) - S(x,y)|$, which is simply the vertical distance at a point between the input data and the approximation.

3 Greedy Insertion

We call refinement algorithms that insert the point(s) of highest error on each pass *greedy insertion* algorithms, "greedy" because they make irrevocable decisions as they go, and "insertion" because on each pass they insert one or more vertices into the triangulation.

3.1 Previous Work

Many variations on the greedy insertion algorithm have been explored over the years; apparently the algorithm has been reinvented many times. However, in the previous work, analysis and testing on large problems were not always done.

In early work, Fowler and Little used a hybrid of feature methods and refinement methods to approximate a terrain [9]. A purer greedy insertion algorithm using Delaunay triangulation, with no optimizations described, was given by De Floriani *et al.* in 1983 [4, 5]. Others presented similar algorithms, demonstrating them on larger input data [24, 25]. In 1989 De Floriani described a more optimized version of the algorithm [3] with a typical cost, by our analysis, of $O(n \log m + m^2)$. Others independently developed essentially the same algorithm [10, 8]. Variations of this algorithm using least squares fitting, and using data-dependent triangulation instead of Delaunay triangulation, were given by Rippa [26]. Both variants typically improve the fit. The basic algorithm was optimized further by Heller, using a heap to reduce the typical cost, by our analysis, to $O((m+n)\log m)$ [17, p. 168].

Our contributions build on this work. We describe the greedy insertion approach in greater detail than it has been covered previously, optimize it, provide theoretical analysis of both worst case and typical complexity, and include extensive empirical testing on large terrain data sets. We have compared our greedy insertion method to several of the methods summarized above and found ours to be faster and/or more accurate.

3.2 Naive Algorithm

First, we consider simple and unoptimized greedy insertion [4, 5, 26, 24, 25]. We build an initial approximation of two triangles using the corner points of H. In repeated passes, we scan the points to find the one with the largest error and insert it into the current approximation using incremental Delaunay triangulation.

Cost Analysis of Naive Algorithm. Within each pass, we classify costs into three categories:

selection to pick the point of highest error,

insertion to insert a vertex into the mesh, and

recalculation to recalculate errors at grid points.

Recall that n is the number of points in the input grid and m is the number of points in the final approximation. Let L denote the time to locate one point in the Delaunay mesh and I be the time to insert a vertex into the mesh. Note that both I and L are mesh-dependent, and hence iteration-dependent.

For point location, we use the simple "walking method" described by Guibas-Stolfi [12, p. 121]. This algorithm starts on an edge of the mesh, and walks through the mesh toward the target point until it arrives at the target. Our variant of the algorithm, which generalizes it to a broader class of triangulations, is described in [11]. Logarithmic-time point location algorithms are known [13], of course, but it turns out that we will not need them.

In this simple implementation of greedy insertion, the costs per pass are: $O(n)$ to scan the points and select the

point of highest error, I for insertion, and $O(nL)$ to recalculate the errors at all of the n points. Recalculation, in this unoptimized algorithm, requires a point location of cost L to find the enclosing triangle, and an interpolation within the triangle of cost $O(1)$ for each mesh point.

In the worst case, a single location or insertion takes $O(i)$ time on pass i, and thus $L = I = O(i)$. This would yield an overall complexity of $\sum_{i=1}^{m} O(ni) = O(m^2 n)$ for this algorithm. Fortunately, the worst case behavior is very unlikely.

We will use the term "typical cost" to describe costs observed in practice (we do not use the term "expected cost", since we don't have a probabilistic model of the input). The typical cost of point location is only $L = O(1)$, since successive location sites are mostly in scanline order [12, 11]. Insertion requires point location, and our location method typically costs $O(\sqrt{i})$ in a triangulation with $O(i)$ vertices [12, 11]. Thus the typical insertion cost is $I = O(\sqrt{i})$. Therefore, the typical total cost is dominated by recalculation: $\sum_{i=1}^{m} O(n) = O(mn)$.

3.3 Optimized Algorithm

The algorithm above yields high quality results. However, even our typical time complexity estimate of $O(mn)$ is expensive, and the worst case complexity of $O(m^2 n)$ is exorbitant. Fortunately, we can improve upon our naive algorithm with three optimizations: (1) faster recalculation, (2) faster selection, and (3) the elimination of point location.

Faster Recalculation. After a Delaunay insertion, the point inserted will have edges emanating from it to the corners of a surrounding polygon [12]. This polygon defines the area in which the triangulation has been altered, and hence, it defines the area in which the approximation has changed (see Figure 1c). We call this polygon the *update region*. Such coherence permits our first significant optimization: we will cache the error values and only recompute errors within this update region. This can be done with polygon scan conversion (rasterization).

We can also use scan conversion to improve the efficiency of recalculation. With the naive algorithm, recalculation involved an interpolation within the enclosing triangle of each point. During scan conversion of a triangle, we can precompute a plane equation once and interpolate to that plane incrementally. This cuts the cost of recalculation by a significant constant factor.

Faster Selection. The next critical observation is that selection can be done more quickly with a heap or other fast priority queue. We define the *candidate point* of a triangle to be a grid point within the triangle that has the highest error in the current approximation. Each triangle can have at most one candidate point. Most triangles have one candidate, but if the maximum error inside the triangle is negligible, or there are no input points inside the triangle, then the triangle has none. We compute the candidate for each triangle as we scan convert it. These candidates are maintained in a heap keyed on their errors. During each pass, we simply extract the best candidate from the top of the heap.

Elimination of Point Location. In the naive algorithm, recalculation required a point location to find the enclosing triangle of each point. Insertion also required a point location. We can eliminate point location altogether by recording with each candidate a pointer to its containing triangle.

Data Structures. Our algorithm, listed below, has the following primary data structures: planes, height fields, triangulations, and heaps. A plane structure simply stores the coefficients a, b, and c for a plane equation $z = ax + by + c$. The height field consists of a rectangular array of points, each of which contains a height value $H(x, y)$, and a bit to record if the input point has been used by the triangulation. For the triangulation, we use a slight modification to Guibas and Stolfi's quad-edge data structure [12], which tracks triangles as well as 2-D points and directed edges. Triangles track their candidate's position (*candpos*), their location in the heap (*heapptr*), and their an error measurement (*err*). The heap contains triangles keyed on the error of their candidate point.

We state the conditions for termination abstractly as a function GOAL_MET(); they would typically be based on the number of points selected, the maximum error of the approximation, or the squared error of the approximation.

Delaunay Greedy Insertion:

```
HeapNode HEAP_CHANGE(HeapNode h, float key, Triangle T):
  % Set the key for heap node h to key,
  % set its triangle pointer to T, and adjust heap.
  % Return (possibly new) heap node.
  if h ≠ nil then
    if key > 0 then
      % update existing heap node
      HEAP_UPDATE(h, key)
      return h
    else
      % delete obsolete heap node
      HEAP_DELETE(h)
      return nil
  else
    if key > 0 then
      % insert new heap node
      return HEAP_INSERT(key, T)
    else
      return nil

MESH_INSERT(Point p, Triangle T):
  Insert a new vertex in triangle T
  Update the Delaunay mesh,
    calling HEAP_DELETE on candidates of deleted triangles
    and setting heapptr ← nil on new triangles

SCAN_TRIANGLE(Triangle T):
  plane ← FIND_TRIANGLE_PLANE(T)
  best ← nil
  maxerr ← 0
  forall points p inside triangle T do
    err ← |H(p) − INTERPOLATE_TO_PLANE(p, plane)|
    if err > maxerr then
      maxerr ← err
      best ← p
  T.heapptr ← HEAP_CHANGE(T.heapptr, maxerr, T)
  T.candpos ← best

INSERT(Point p, Triangle T):
  mark p as used
  MESH_INSERT(p, T)
  forall triangles U adjacent to p do
    SCAN_TRIANGLE(U)

GREEDY_INSERT():
  initialize mesh to two triangles formed by grid corners
  forall initial triangles T do
    SCAN_TRIANGLE(T)
  while not GOAL_MET() do
    T ← HEAP_DELETE_MAX()
    INSERT(T.candpos, T)
```

3.3.1 Cost Analysis of Optimized Algorithm

The three optimizations we have made speed up the algorithm significantly, both in theory and in practice.

In the optimized algorithm, time for selection is spent in three places: heap insertion, heap extraction, and heap updates. The growth of the heap per pass is bounded by the net growth in the number of triangles, which is 2. However, the heap does not always grow this fast. In particular, as triangles become so small or so well fit to the height field as to have no candidate points, they will be removed from the heap, and eventually the heap will shrink. Typically, the approximations that we wish to produce are much smaller than the original height fields. Consequently, the algorithm will rarely realize any significant benefit from shrinking heap sizes. To be conservative, we will assume that the size of the heap on pass i is $O(i)$, and thus, that individual heap operations require $O(\log i)$ time.

The number of changes made to the heap per pass is 3 plus the number of edge flips performed during mesh insertion. We assume that this is a small constant number. This is empirically confirmed on our sample data; in practice, the number of calls to HEAP_CHANGE per pass is roughly 3–5. Given this assumption, the total heap cost, and hence selection cost, is $O(\log i)$ per pass.

The other two tasks, insertion and recalculation, are also cheaper now, since neither performs locations, and recalculation also exploits locality and uses cached plane equations. The cost of recalculation has dropped from $O(nL)$ to $O(A)$, where A is the area of the update region.

Worst Case Time Cost. In the worst case, the insertion time is $I = O(i)$ and the update area is $A = O(n)$, so the costs per pass are: $O(\log i)$ for selection, $O(i)$ for insertion, and $O(n)$ for recalculation. The asymptotically dominant term is recalculation, as before, but it is much smaller now; the total worst case cost is only $\sum_{i=1}^{m} O(n) = O(mn)$.

Typical Case Time Cost. The typical number of edge flips is constant, so the cost of insertion is $I = O(1)$ and the size of the update region is $A = O(n/i)$. The costs per pass are thus: $O(\log i)$ for selection, $O(1)$ for insertion, and $O(n/i)$ for recalculation. The selection cost grows as the passes progress, while the recalculation cost shrinks. These two are the dominant terms. The total cost in the typical case is thus: $\sum_{i=1}^{m} O(\log i + n/i) = O((m+n)\log m)$.

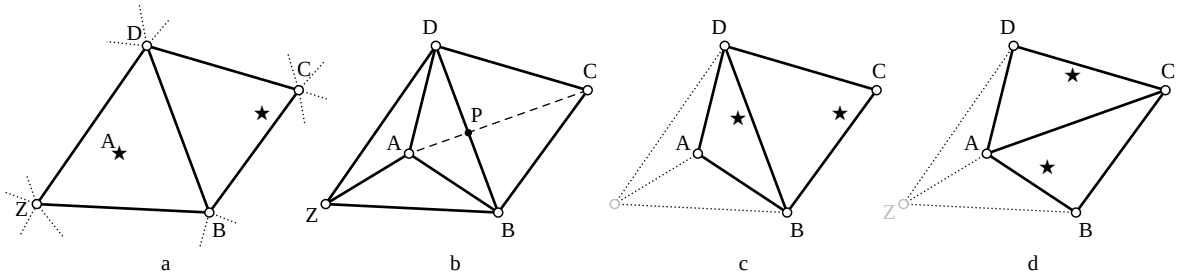


Figure 2: Data-dependent triangulation. a) Point A, the candidate of triangle ZBD, is about to be inserted. Stars denote candidate points. Spoke edges from A to the containing polygon ZBD are added. b) The quadrilateral ABCD around suspect edge BD is checked. ABCD can be triangulated in two ways, using diagonals BD or AC, which intersect at point P. c) If BD yields the lowest error, then we have the new triangle ABD and the old triangle CDB. d) If AC has lowest error, then we have the new triangles DAC and BCA, the containing polygon expands to ZBCD, and edges BC and CD become suspect.

3.4 Data-Dependent Triangulation

So far we have used Delaunay triangulation, which employs only two-dimensional (xy) information, and strives to create well-shaped triangles in 2-D. More accurate approximation is possible using data-dependent triangulation, where the topology of the triangulation is chosen based on the three-dimensional fit of the approximating surface to the input points.

Data-dependent variants of the greedy insertion algorithms described can be created by replacing Delaunay triangulation with data-dependent triangulation, as discussed by Rippa and Hamann-Chen [26, 14]. The vertex to insert in the triangulation on each pass is chosen as before, but the triangulation is done differently.

The incremental Delaunay triangulation algorithm used above tested suspect edges with a purely two-dimensional geometric test involving circumcircles. A generalization of this approach, Lawson's local optimization procedure [20], uses other tests. For data-dependent triangulation, instead of checking the validity of an edge with the circle test, the rule we adopt is that an edge is swapped if the change decreases the error of the approximation. We defer the definition of "error" until later. When used with the circle test, the local optimization procedure finds a global optimum, the Delaunay triangulation, but when used with more general tests, it is only guaranteed to find a local optimum.

Figure 2 illustrates our local optimization procedure for the data-dependent triangulation algorithm. Figure 2a: suppose that point A has the highest error of all candidates. It will be the next vertex inserted in the triangulation. Figure 2b: spokes are added connecting it to the containing polygon (a triangle, if A falls inside a triangle; a quadrilateral, if A falls on an edge). Each edge of the containing polygon is suspect, and must be tested. In some cases, the quadrilateral containing the edge will be concave, and can only be triangulated one way, but in most cases, the quadrilateral will be convex, and the other diagonal must be tested to see if it yields lower error.

The most straightforward way to test validity of an edge BD would be the following recursive procedure: Test both ways of triangulating the quadrilateral ABCD containing the edge. If edge BD yields lower global error (Figure 2c), then no new suspect edges are added, and we stop recursing. If swapping edge BD for edge AC would reduce the global error of the approximation (Figure 2d), then swap the edge to AC, and recurse on the four new suspect edges, BC, CD, AB, and AD.

Edges AB and AD are not suspect with the Delaunay criterion, but they are suspect when using a data-dependent criterion in Lawson's local optimization procedure, since swapping them might decrease the error. In our implementation, we do not test spokes such as AB and AD for swapping. This rule guarantees that the update region is exactly the containing polygon. This simplifies the implementation, but may sacrifice some quality.

When all suspect edges have been tested, it is then necessary to update the candidates for all the triangles in the containing polygon. This straightforward approach requires scan converting most of the triangles in the local neighborhood twice: once to test for swapping and once to find candidates.

A faster alternative is to scan convert once, computing both the global error and the candidate in one pass. This is about twice as fast. To do this, we split the quadrilateral ABCD with its two diagonals into four subtriangles: PDA, PAB, PBC, and PCD, where P is the intersection point of the two diagonals (Figure 2b). This splitting is conceptual; it is not a change to the data structures. As each of the four

subtriangles is scan converted, two piecewise-planar approximations are tested. For subtriangle ABP, for example, the planes defined by ABD and BCA are both considered. The other subtriangles have different plane pairs. During scan conversion of each subtriangle, for each of its two planes, the contribution to the triangulation's total error is calculated, and the best candidate point and its error is calculated. After scan conversion, the subtriangles' errors and candidates are combined pairwise to determine the error and candidates for each of the two pairs of triangles: ABD and CDB, versus BCA and DAC. Note that, with one exception, the triangle CDB is an old triangle, so its error and candidate have previously been computed, and need not be recomputed. The sole exception to this rule is during initialization when the first two triangles are being created.

Data Structures. The algorithm uses all of the previous data structures plus a new one, the *FitPlane*. A *FitPlane* is a temporary data structure that stores an approximation plane and other information. During scan conversion of the four subtriangles, it accumulates information about the error and candidate for a triangle approximated by a plane. Specifically, it contains the coefficients for the planar approximation function *plane*, the candidate's position *candpos* and error *canderr*, the error over the triangle *err*, and a *done* bit recording whether the triangle was previously scanned.

The *Triangle* and *Heap* data structures cache information about candidates and errors that is re-used during data-dependent insertion. A *FitPlane* can be initialized from this information with the subroutine FITPLANE_EXTRACT(*Triangle T*), which also marks the *FitPlane* as *done*. When a new triangle is being tested, the call FITPLANE_INIT(a, b, c) will initialize a *FitPlane* to a plane through the three points a, b, and c, with errors set to 0, and *done* = nil. One or more subsequent calls to SCAN_TRIANGLE_DATADEP are made to accumulate error and candidate information in the *FitPlane*. If this approximation plane turns out to be the best one, the heap is updated and the error and candidate information is saved for later use with a call to SET_CANDIDATE (listed below).

The routines LEFT_TRIANGLE and RIGHT_TRIANGLE return the triangles to the left and the right of a directed edge, respectively. The keyword "var" marks call-by-reference parameters.

Data-Dependent Greedy Insertion:

```
SET_CANDIDATE(var Triangle T, FitPlane fit):
  T.heapptr ← HEAP_CHANGE(T.heapptr, fit.canderr, T)
  T.candpos ← fit.candpos
  T.err ← fit.err
```

```
SCAN_POINT(Point x, var FitPlane fit):
  err ← |H(x) − INTERPOLATE_TO_PLANE(x, fit.plane)|
  fit.err ← ERROR_ACCUM(fit.err, err)
  if err > fit.err then
    fit.canderr ← err
    fit.candpos ← x
```

```
SCAN_TRIANGLE_DATADEP(Point p, Point q, Point r,
    var FitPlane u, var FitPlane v):
  % Scan convert triangle pqr,
  % updating error and candidate for planes u and v.
  % Plane u might be nonexistent or already done.
  forall points x inside triangle pqr do
    if u ≠ nil and not u.done then
      SCAN_POINT(x, u)
    SCAN_POINT(x, v)
```

```
FIRST_BETTER(float q1, float q2, float e1, float e2):
  % Return true iff edge 1 yields better triangulation of a
  % quadrilateral than edge 2, according to shape and fit.
  % q1 and q2 are "shape quality", and e1 and e2 are
  % fit error of the corresponding triangulations.
  qratio ← MIN(q1, q2) / MAX(q1, q2)
  % Use shape criterion if shape of one triangulation
  % is much better than the other, otherwise use fit.
  if qratio ≤ qthresh then
    return (q1 ≥ q2)          % shape criterion
  else
    return (e1 ≤ e2)          % fit error criterion
```

```
CHECK_SWAP(DirectedEdge e, FitPlane abd):
  % Checks edge e, swapping it if that reduces error,
  % updating triangulation and heap.
  % Error and candidate for the triangle to the left of e
  % is passed in in abd, if available.
  % Points a, b, c, d, and p are as shown in figure 2b,
  % and e is edge from b to d.
  if abd = nil then
    abd ← FITPLANE_INIT(a, b, d)
  if edge e is on boundary of input grid
      or quadrilateral abcd is concave then
    % Edge bd is good and edge ac is bad.
    if not abd.done then
      SCAN_TRIANGLE_DATADEP(a, b, d, nil, abd)
    SET_CANDIDATE(LEFT_TRIANGLE(e), abd)
  else
    % Check whether diagonal bd or ac has lower error.
    FitPlane cdb ← FITPLANE_EXTRACT(RIGHT_TRIANGLE(e))
    FitPlane dac ← FITPLANE_INIT(d, a, c)
    FitPlane bca ← FITPLANE_INIT(b, c, a)
    % scan convert the four subtriangles
    SCAN_TRIANGLE_DATADEP(p, d, a, abd, dac)
    SCAN_TRIANGLE_DATADEP(p, a, b, abd, bca)
    SCAN_TRIANGLE_DATADEP(p, b, c, cdb, bca)
    SCAN_TRIANGLE_DATADEP(p, c, d, cdb, dac)
    if FIRST_BETTER(SHAPE_QUALITY(a, b, c, d),
        SHAPE_QUALITY(b, c, d, a),
        ERROR_COMBINE(abd.err, cdb.err),
        ERROR_COMBINE(dac.err, bca.err)) then
      % keep edge bd
      SET_CANDIDATE(LEFT_TRIANGLE(e), abd)
      if not cdb.done then
        SET_CANDIDATE(RIGHT_TRIANGLE(e), cdb)
    else
      Swap edge e from bd to ac.
      dac.done ← bca.done ← true
      CHECK_SWAP(DirectedEdge cd, dac)  % recurse
      CHECK_SWAP(DirectedEdge bc, bca)

INSERT_DATADEP(Point a, Triangle T):
  Mark input point at a as used.
  In triangulation, add spoke edges connecting a to vertices
    of containing polygon (T and possibly a neighbor of T).
  forall counterclockwise perimeter edges e
      of containing polygon do
    CHECK_SWAP(e, nil)

GREEDY_INSERT_DATADEP():
  Initialize mesh to two triangles spanning height field.
  e ← (either directed edge along diagonal)
  CHECK_SWAP(e, nil)
  while not GOAL_MET() do
    T ← HEAP_DELETE_MAX()
    INSERT_DATADEP(T.candpos, T)
```

3.4.1 Data-Dependent Criterion

The routines ERROR_ACCUM and ERROR_COMBINE are used to accumulate the error over a subtriangle, and to total the error of a pair of triangles, respectively. These can be defined in various ways. For an L_2 error measure, they should be defined:

```
float ERROR_ACCUM(float accum, float x):
  return accum+x*x
float ERROR_COMBINE(float err1, float err2):
  return err1+err2
```

and for an L_∞ error measure, they should be defined:

```
float ERROR_ACCUM(float accum, float x):
  return MAX(accum, x)
float ERROR_COMBINE(float err1, float err2):
  return MAX(err1, err2)
```

The data-dependent-based method described above is slower than the Delaunay-based algorithm because it requires about twice as many error recalculations during scan conversion. However, the asymptotic complexities are identical to the Delaunay-based algorithm. Thus, the worst case cost is $O(mn)$ and its typical cost is $O((m+n)\log m)$.

3.4.2 Combating Slivers

Pure data-dependent triangulation, which makes swapping decisions based exclusively on fit error, will sometimes generate very thin sliver triangles. If the triangles fit the data well, and the surface is being displayed in shaded (not vector) form, then slivers by themselves are not a problem. But sometimes these slivers do not fit the data well, and lead to globally inaccurate approximations. After experiments with several sliver-avoidance schemes (see [11, 26]), we adopted a hybrid of data-dependent and Delaunay triangulation.

The pseudocode above implements this hybrid. The procedure SHAPE_QUALITY(a, b, c, d) returns a numerical rating of the shape of the triangles when quadrilateral *abcd* is split by edge *bd*. This rating should be constructed so that higher values indicate "better" shape. The parameter *qthresh* used in FIRST_BETTER is a quality threshold. When set to 0, pure data-dependent triangulation results, when set to 1, pure shape-dependent triangulation results, and when set in between, a hybrid results. If SHAPE_QUALITY returns the minimum angle of the triangles *abd* and *cdb*, then this shape-dependent triangulation will in fact be Delaunay triangulation. The hybrid method typically yielded lower error approximations than pure shape- or pure data-dependent triangulation.

Name	Dimensions	Location
West US	1024×1024	Idaho/Wyoming border
NTC	1024×1024	Tiefort Mtns., California
Ozark	369×462	Ozark, Missouri
Crater	336×459	Crater Lake, Oregon
Ashby	346×452	Ashby Gap, Virginia

Table 1: DEM datasets used for testing the simplification algorithms.

3.5 Extended Height Fields

So far, we have developed algorithms for simplifying basic height fields, and we have described techniques for making them faster without sacrificing quality. The greedy insertion algorithm can also be used to simplify data other than scalar height fields.

Consider the case in which our data specifies more than just height. For instance, the grid might contain measurements for some material property of the surface such as color, expressed as an RGB triple. Our algorithm can be easily adapted to support such extended height fields. Up to now, we have considered the simple case of surfaces of the form $H(x, y) = z$. An *extended height field* is one where the data values are tuples rather than single numbers. For example, we might model a color-textured terrain as a surface $H(x, y) = (z, r, g, b)$. We can think of this as sampling a set of distinct surfaces, one in *xyz*-space, one in *xyr*-space, and so on. We see here another reason to reject triangulation schemes that attempt to fit specific surface characteristics; we now have 4 distinct surfaces which need have no features in common. Given data for a generic set of surfaces, we can apply the importance measure to each surface separately and then compute some kind of average of these values. But when we know the precise interpretation of the data (i.e. the values represent height and color), we can construct a more informed measure. Our old measure was simply $|\Delta z|$; a reasonable extension to deal with color is $|\Delta z| + \frac{M}{3}(|\Delta r| + |\Delta g| + |\Delta b|)$. Here, M is the z-range of H; the $\frac{M}{3}$ term scales the total color difference to fit the range of the total height difference (here we assume that color values are between 0 and 1). In order to achieve greater flexibility, we can also add a color emphasis parameter, w, controlling the relative importance of height difference and color difference. The final error formula would be: $(1-w)|\Delta z| + w\frac{M}{3}(|\Delta r| + |\Delta g| + |\Delta b|)$.

To implement these changes, we simply added fields to the height field to record r, g, and b, modified the *FitPlane* to retain planar approximations to these three additional surfaces, and changed the error procedure to use the extended formula above.

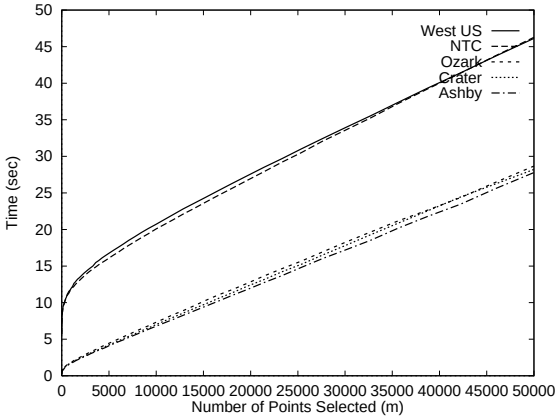


Figure 3: Running time of Delaunay greedy insertion on several DEM datasets.

These extensions allow our algorithm to be used to simplify terrains with color texture or planar color images [27]. The output of such a simplification is a triangulated surface with linear interpolation of color across each triangle. Such models are ideally suited for hardware-assisted Gouraud shading on most graphics workstations, and are a possible substitute for texture mapping when that is not supported by hardware.

4 Results

Our combined implementation of the Delaunay and data-dependent algorithms consists of about 5,200 lines of C++. The incremental Delaunay triangulation module is adapted from Lischinski's code [22].

Figures 4–11 are a demonstration of greedy insertion based approximation of a digital elevation model (DEM) for the western half of Crater Lake. Figure 4 shows the full DEM dataset (a rectangular grid with each quadrilateral split into two triangles). Our first approximation, shown in Figure 5, shows an approximation using 1% of the total points. This approximation has captured the major features of the terrain. However, it is still clearly different from the original. Figure 8 is an approximation using 5% of the original points. This model contains most of the features of the original. Thus, using only a fraction of the original data points, we can build high fidelity approximations. In a multiresolution database, we would produce a series of approximations, for use at varying distances.

Color Figures 16–18 illustrate the application of height field simplification methods to the approximation of planar color images by Gouraud shaded triangles. Figure 17 shows approximation by uniform subsampling, and Figure 18 shows approximation by data-dependent greedy in-

sertion, both using the same number of vertices. In both cases, the best results (shown) were achieved by low pass filtering the input before approximating. Clearly, greedy insertion yields a much better approximation.

4.1 Speed of the Algorithms

We have tested the performance of our simplification algorithms on a Silicon Graphics Indigo2 with a 150 MHz MIPS R4400 processor and 64 megabytes of main memory. For our timing tests we have used several digital elevation models. They are summarized in Table 1.

Figure 3 shows the running time of the Delaunay-based algorithm on the various DEM datasets as a function of points selected. In all cases, it was able to select 50,000 points in under one minute. In Figures 3, 12, 13, and 15, n is fixed (although it varies between datasets of different sizes) and the horizontal axis is m. All of the data points in Figure 3 are fit by the function

$$\text{time}(m,n) = .000001303\, n \log m - .0000259\, m \log m + .00000326\, n + .000845\, m - 0.178 \log m + .1 \text{ sec.}$$

with a maximum error of 1.7 seconds, supporting our $O((m+n)\log m)$ typical cost formula.

To quantify the improvement in efficiency due to our optimizations, we also implemented a naive greedy insertion algorithm. The optimizations proved to be very significant. In the time it takes our optimized algorithm to select 50,000 points from a $1{,}024 \times 1{,}024$ terrain (46 seconds), the naive algorithm can only manage to select a few hundred points from a 65×65 terrain [11]. These speedups were achieved without sacrificing quality; our optimizations increase processing speed with no meaningful change in the points selected or the approximation[2].

4.2 Memory Use

The optimized algorithm uses memory for three main purposes: the height field, the mesh, and the heap. The height field uses space proportional to the number of grid points n, and the mesh and heap use space proportional to the number of vertices in the mesh, m. Asymptotically, the memory cost is thus $O(m+n)$.

We now detail the memory requirements of our current implementation. For every point in the height field, we store one 2-byte integer for the z value, and a 1-byte Boolean indicating whether this point has been used. Thus, these arrays consume $3n$ bytes. In the mesh, 16 bytes are used to store each vertex's position, 68 bytes are used per edge, and 24 bytes per triangle, so assuming there are about $3m$ edges and $2m$ triangles, the memory required for a mesh with m vertices is $268m$ bytes. The heap uses 12 bytes per node, so heap memory requirements are about $12 \cdot 2m = 24m$ bytes.

[2] In the case of ties between candidates of equal importance, implementation details might cause a difference in selection order.

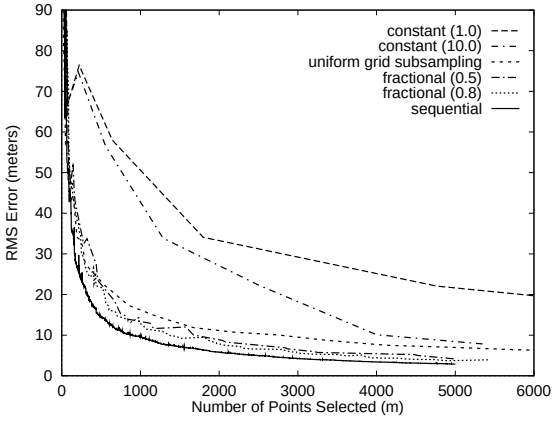


Figure 12: RMS error of approximation as vertices are added to the mesh, for Crater Lake DEM, for uniform grid subsampling and several variants of Delaunay greedy insertion: sequential insertion, fractional threshold parallel insertion with two different fractions α, and constant threshold parallel insertion with two different thresholds ϵ.

Total memory requirements of the data structures in our implementation are therefore $3n + 292m$ bytes. Thus, for example, we estimate that selecting $m = 10,000$ (1% of total) points from an $n = 1,024^2$ DEM would require about 6 megabytes of memory.

Our current implementation stores floating point numbers using double precision (8 bytes), and uses the quad-edge data structure to store the mesh. The quad-edge structure is less compact than some triangulation data structures, and double precision floating point may not always be necessary. So the program's memory requirements could probably be cut significantly.

4.3 Quality of the Approximations

We believe that the greedy insertion algorithm yields good results on most reasonably smooth height fields. This can be verified both visually and with objective error metrics. Figure 12 shows the RMS error as an approximation for the Crater Lake DEM is built one point at a time. This figure also shows the error behavior for some variant insertion policies, but we will ignore all but the "sequential" curve for the moment.

At a coarse level, the RMS error initially decreases quite rapidly and then slowly approaches 0. In the early phases

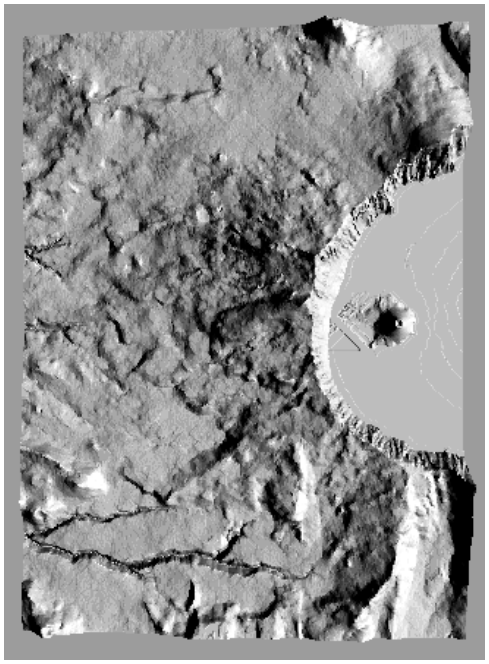


Figure 4: Original DEM data for west end of Crater Lake (154,224 vertices). Note the island in the lake.

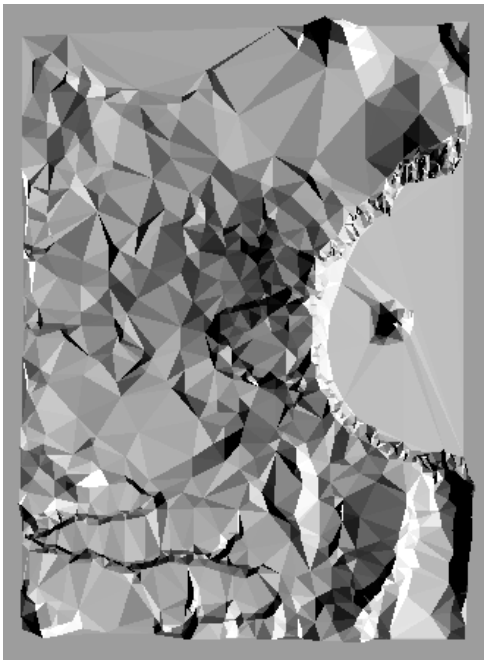


Figure 5: Delaunay approximation using 1% of the input points (1,542 vertices).

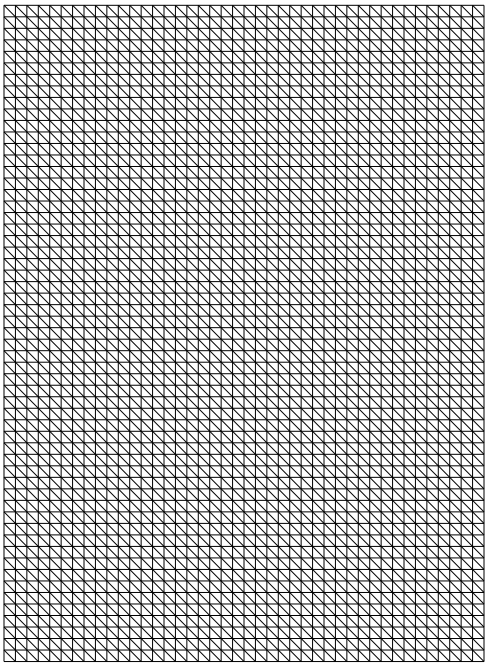


Figure 6: Mesh identical to the one used for the DEM picture above, except this mesh uses only every eighth point in x and y, for clarity.

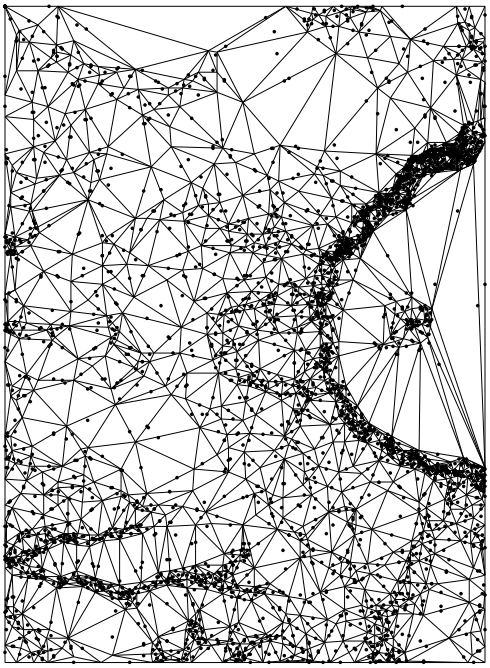


Figure 7: Delaunay mesh for the approximation above. Candidates are shown with dots. It is interesting to note that most candidates fall near edges.

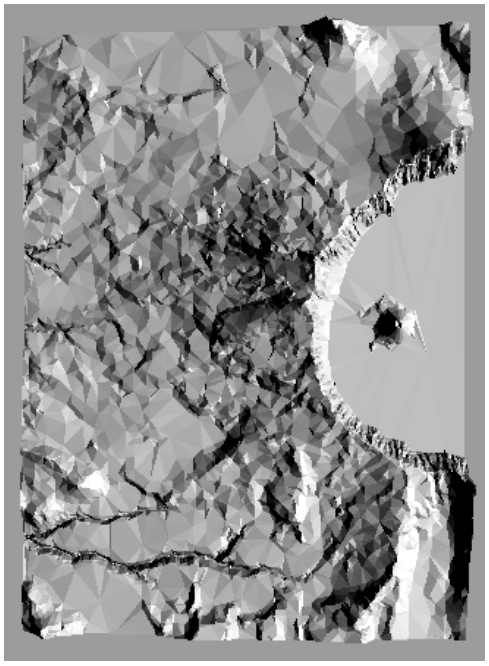


Figure 8: Delaunay approximation using 5% of the input points (7,711 vertices).

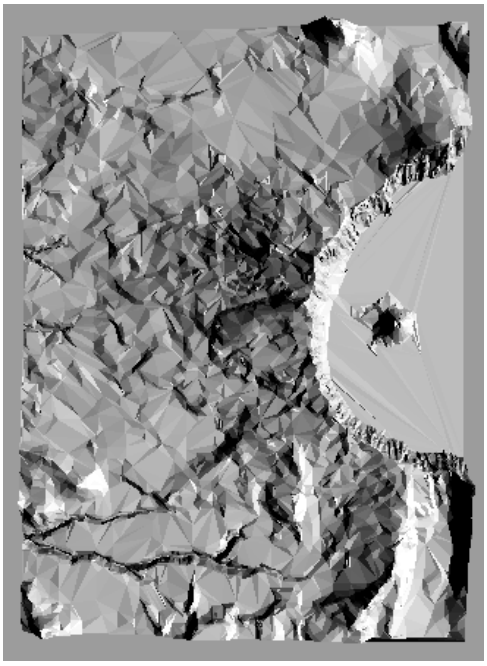


Figure 9: Data-dependent approximation using 5% of the input points (7,711 vertices).

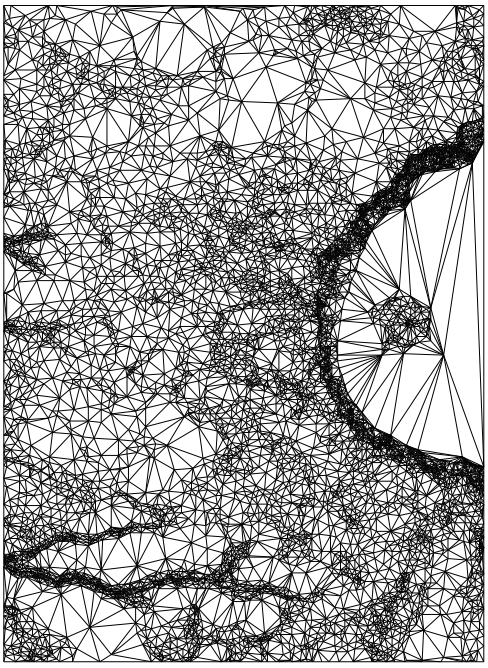


Figure 10: Delaunay mesh for the approximation above.

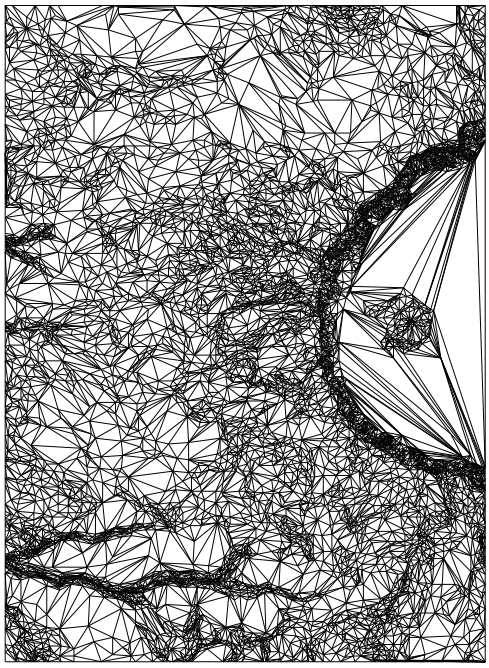


Figure 11: Data-dependent mesh for the approximation above.

of the algorithm, the error fluctuates rather chaotically, but it settles into a more stable decline. Theoretically, in the limit as $m \to \infty$, the error of the L_2-optimal triangulation converges as m^{-1} [23], but this empirical data is better fit by the function $m^{-.7}$. While we only show the error curve for a single terrain, we have tested the error behavior on several terrains, and the curves all share the same basic characteristics.

Data-dependent greedy insertion yielded the lowest error overall. For our SHAPE_QUALITY measure, we employed a simple formula which is the product of the areas of the two triangles divided by the product of their approximate diameters. While this does not yield Delaunay triangulation when $qthresh = 1$, we believe the difference is negligible. A shape quality threshold of $qthresh = .5$ gave the best results in most cases. By varying the ERROR_ACCUM and ERROR_COMBINE procedures as described in §3.4.1, several different error measures can be tested. The error differences were slight, but empirical tests showed that the lowest error approximations typically resulted when ERROR_ACCUM used the MAX function while ERROR_COMBINE used addition – thus, a combination of L_∞ and L_2 measures. We call this the sum-max criterion.

We also tested the angle between normals (ABN) criterion proposed by Dyn *et al.* [7]. This criterion swaps edges to minimize the angle between normals of adjacent triangles. The ABN criterion is thus data-dependent in the sense that it depends on the heights at the vertices, but unlike the L_∞ and L_2-based error measures our algorithms employ, the error measure is independent of the unselected input points. In our experiments, with $qthresh = .5$, the ABN criterion gave errors that were higher than the sum-max criterion in all cases, and often higher than the Delaunay criterion as well. (We did not test the more complex ABN hybrid described by Rippa [26, p. 1136]).

Delaunay greedy insertion is compared to data-dependent greedy insertion in Figures 13 and 14. The first shows that, for a given number of points, data-dependent triangulation with our sum-max criterion finds a slightly more accurate approximation than Delaunay triangulation. The ratio of data-dependent RMS error to Delaunay RMS error is about .8 to .9 for this height field. The second figure shows the time/quality tradeoff very clearly. With either algorithm, as the number of points selected increases, the error decreases while the time cost increases. To achieve a given error threshold, data-dependent greedy insertion takes about 3–4 times as long as Delaunay greedy insertion, but it generates a smaller mesh, which will display faster.

Data-dependent triangulation does dramatically better than Delaunay triangulation on certain surfaces [7]. The

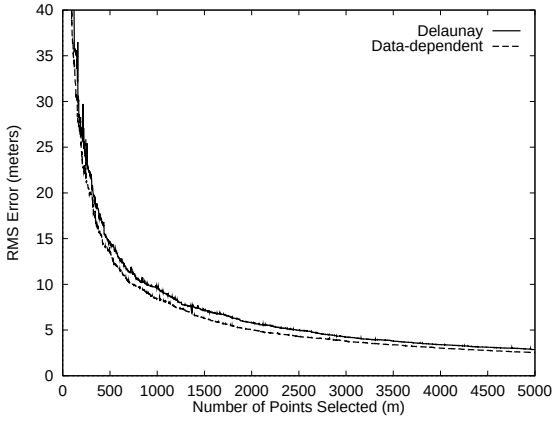


Figure 13: RMS error of approximation as vertices are added to the mesh, for Crater Lake DEM, comparing Delaunay triangulation (top curve) to data-dependent triangulation (bottom).

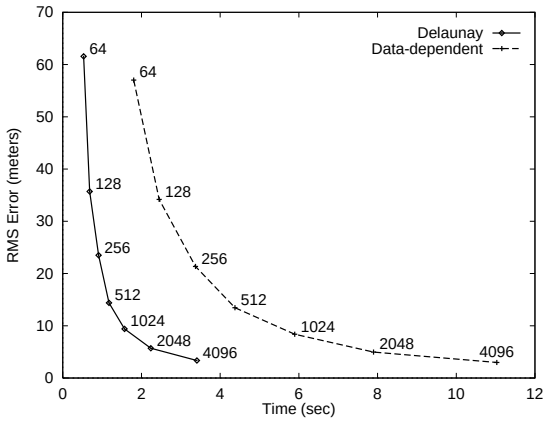


Figure 14: Time versus error plot for Delaunay and data-dependent triangulation on Crater Lake DEM. Data-dependent triangulation is slower, but higher quality. Data points are marked with *m*, the number of points selected.

optimal case for data-dependent triangulation is a ruled surface with zero curvature in one direction and nonzero curvature in another. Examples are cylinders, cones, and height fields of the form $H(x, y) = f(x) + ay$. On such a surface, if a Delaunay-triangulated approximation uses *m* roughly uniformly distributed vertices, then data-dependent triangulation could achieve the same error with about $2\sqrt{m}$ points using sliver triangles that span the rectangular domain.

From our empirical tests, it seems that the surfaces for which data-dependent triangulation excels are statistically uncommon among natural terrains. We conjecture that on natural terrains, data-dependent triangulation yields approximations that are only slightly higher quality than Delaunay triangulation, in general.

4.4 Sequential versus Parallel Greedy Insertion

Others have employed variants of the greedy insertion algorithm that insert more than one point on each pass. Methods that insert a single point on each pass we call *sequential greedy insertion* and methods that insert multiple points in parallel on each pass we call *parallel greedy insertion.* The words "sequential" and "parallel" here refer to the selection and re-evaluation process, not to the architecture of the machine.

Puppo *et al.* showed statistics that suggest that parallel methods are better than sequential methods, saying: "…we show the results obtained by the sequential and the parallel algorithm …Because of the more even refinement of the TIN, which is due to the introduction of many points before the Delaunay optimization, our [parallel] approach needs considerably fewer points to achieve the same level of precision" [25, p. 123].

We tested this claim by comparing our sequential greedy insertion algorithm against two variants of parallel insertion. Both variants select and insert all candidate points p such that $\text{ERROR}(p) \geq \epsilon$, where ϵ is a threshold value.

The first insertion variant, which we call *fractional threshold* parallel insertion, selects all candidate points such that $\text{ERROR}(p) \geq \alpha e_{\max}$, where $e_{\max}$ is the maximum error of all candidates. This is an obvious generalization of sequential insertion, which selects a single point such that $\text{ERROR}(p) = e_{\max}$. Fractional thresholding with $\alpha = 1$ is almost identical to sequential insertion; it differs only in that it may select multiple points with the same error value (this is closely related to the approach of Polis and McKeown [24]). If $\alpha = 0$, fractional thresholding becomes highly aggressive and selects every triangle candidate. Looking at the error graphs in Figure 12, we can see that as α increases towards 1, the approximations become more accurate and converge to sequential insertion.

The second insertion variant is the rule used by Fowler-Little and Puppo *et al.* [9, 25]. We call it *constant threshold* parallel insertion. In this case, ϵ is the constant error threshold provided by the user. Thus, on each pass we select and insert all candidate points that do not meet the error tolerance. An algorithm very similar to this [19], called Latticetin, is used by the Arc/Info geographic information system which is sold by the Environmental Systems Research Institute (ESRI).

The error curves for the constant threshold method in Figure 12 show it performing much worse than sequential insertion or fractional thresholding. Similarly, Polis and McKeown found their algorithm (a form of fractional thresholding) superior to Latticetin (a form of constant thresholding) [24].

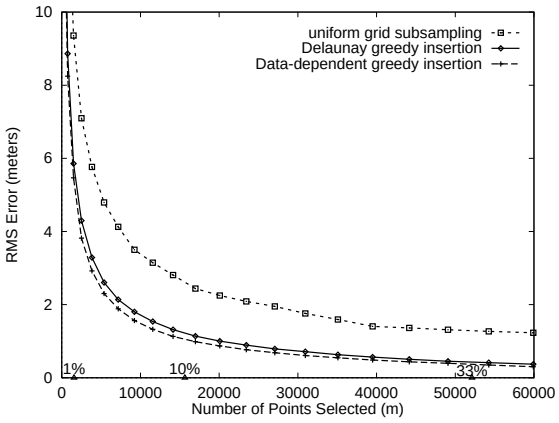


Figure 15: RMS error of approximation as vertices are added to the mesh, for two forms of greedy insertion and uniform grid subsampling, run on Ashby DEM.

When an insertion causes a small triangle to be created, it leads to a local change in the density of candidates. With the sequential method, smaller triangles are statistically less likely to have their candidates selected, because they will typically have smaller errors. In the parallel method, if the small triangles' candidate is over threshold, it will be selected, causing even more excessive subdivision in that area. Even on a simple surface like a paraboloid, which is optimally approximated by a uniform grid, the sequential method is better. On all tests we have run, sequential greedy insertion yields better approximations than parallel greedy insertion.

De Floriani seems to have reached a similar conclusion. While comparing her sequential insertion algorithm to a form of constant thresholding in which the selected points are not limited to one per triangle, she said: "parallel application of such an algorithm by a contemporaneous insertion of all points which have an associated search error greater than the tolerance and belong to the same search region, could lead to the insertion of points which are not meaningful for an improvement in the accuracy of the model" [4, p. 342].

4.5 Greedy Insertion versus Uniform Grids

We also compared greedy insertion with the simplest surface approximation method, uniform grids. In a large study, Kumler found that uniform grids (DEMs) were better than general triangulations (TINs), at least for the TIN generation methods he tested [19, p. 41]. We agree with this qualified conclusion, and go farther to point out certain aspects of his experiments that exaggerated the benefits of DEMs.

One should not conclude from Kumler's results that

DEMs are better than TINs in general, since the TIN algorithms he tested do not appear to be very good. The best DEM-to-TIN algorithm he tested was Latticetin, and we have found similar algorithms to be inferior to sequential greedy insertion (compare top three curves of Figure 12).

In addition, Kumler handicapped the TIN algorithms somewhat by comparing DEMs with n vertices to TINs with $n/3$ or $n/10$ vertices. These ratios are based on his assumption that storage size is the principal concern, and that TINs require 3–10 times the memory of DEMs. While these storage size comparisons are valid for most implementations, general triangulations can be compressed by a factor of 6–10 with very little error [6]. Also, to better understand the behavior of a simplification algorithm, we believe it is necessary to test it for a wide range of values of m/n. More importantly, we believe that, in many applications rendering speed is of much greater importance than storage space, so methods should be compared based primarily on error and speed, not on memory use.

Figures 12 and 15 show our own comparison of the errors of approximations created by sequential greedy insertion and subsampling on a uniform grid. The first figure shows coarse approximations (small values of m/n); the second shows finer approximations (larger values of m/n). Numerous comparisons of greedy insertion with subsampling were computed for the Ashby, Crater Lake, NTC, and WestUS datasets. Figure 15 shows the most representative of these four. Either form of greedy insertion is seen to generate significantly better approximations than a uniform grid. If storage size is the primary concern, and we assume that general triangulations require three times the memory of a uniform grid with the same number of vertices, then for $m/n < 1\%$ or so, uniform grids are probably best, but for larger values of m/n, a good adaptive triangulation scheme, such as sequential greedy insertion, appears better. If compression is used, then general triangulations are probably preferable in all cases.

5 Ideas for Future Research

Our experimentation has suggested several avenues for further research. These are discussed at greater length in our technical report [11].

Using Extra Grid Information. The algorithms we have described could easily be generalized to make use of additional information about the height field. Ridge lines, roads, block boundaries, and discontinuities could be preinserted, for example. The user could also be given control over point selection by allowing a weight to be assigned to each vertex [24].

Combating Slivers. Pure data-dependent triangulation generates too many slivers. It would be nice to find a more elegant solution to the sliver problem to replace the hybrid algorithm. Our current hypothesis to explain the inferiority of pure data-dependent triangulation in our algorithms is that it is caused by the short-sightedness of the greedy insertion algorithm. Sometimes more than one edge swap is required to correct a sliver problem, but our data-dependent greedy insertion algorithm never looks more than one move ahead, so it often gets stuck in local minima. Simulated annealing is one (expensive) remedy.

Discontinuities. The sequential greedy insertion algorithm does poorly when the height field contains a step discontinuity or "cliff". With the greedy insertion algorithm, a linear cliff of length k between two planar surfaces can use up about k vertices when, in fact, 4 would suffice. Cliffs similar to this arise when computer vision range data is approximated with this algorithm. This problem is caused by the short-sightedness of greedy insertion. One, somewhat *ad hoc*, solution is to find all cliffs and constrain the triangulation to follow them [1].

Dealing With Noise and High Frequencies. The greedy insertion algorithms we have described will work on noisy or high frequency data, but they will not do a very good job, as observed by us and others [8]. There are two causes of this problem. One is the simple-minded selection technique, which picks the point of highest error. Such an approach is very vulnerable to outliers. Finding a better strategy for point selection in the presence of noise appears quite difficult. A second cause is that triangles are not chosen to be the best fit to their enclosed data, but are constrained to interpolate their three vertices. Least squares fitting would solve this latter problem [26].

Better handling of noise and discontinuities would make these algorithms well suited to the simplification of computer vision range data.

A Hybrid Refinement/Decimation Approach. A technique that might permit better approximations of cliffs and better selection in the presence of noise is to alternate refinement and decimation passes, inserting several vertices with the greedy insertion approach, and then deleting a few vertices that appear the least important, using Lee's drop heuristic approach [21]. Although such a hybrid of refinement and decimation ideas resembles the algorithm of Hoppe *et al.* [18], it should be quite a bit faster since we already know how to do many of the steps quickly.

Generalization to Other Geometries. The algorithms presented here could easily be generalized from height field grids to scattered data approximation. That is, the *xy* projection of the input points need not form a rectangular grid, but could be any finite point set. This change would require each triangle to store a set of points [3, 17, 10, 8]. During re-triangulation, these sets would be merged and split. Instead of scan converting a triangle, one would visit all the points in that triangle's point set.

Generalization of these techniques from piecewise-planar approximations of functions of two variables to curved surface approximations and higher dimensional spaces [14] is fairly straightforward.

6 Summary

We have presented variants of the greedy insertion algorithm in significant detail, optimized them, analyzed their worst case and typical complexity, and presented empirical tests and comparisons.

Speed. Beginning with a very simple implementation of the greedy insertion algorithm, we optimized it in three ways: by only recalculating where necessary, by using a heap to find points of highest error, and by eliminating point location.

When approximating an n point grid using an m vertex triangulated mesh, these optimizations sped up the algorithm from a typical time cost of $O(mn)$ to $O((m+n)\log m)$. This speedup is significant in practice as well as theory. For example, we can approximate a 1024×1024 grid to high quality using 1% of its points in about 21 seconds on a 150 MHz processor. The memory requirements of the algorithm are $O(m+n)$.

Quality. Delaunay and data-dependent triangulation methods were compared. The latter is capable of higher quality approximations because it chooses the triangulation based on quality of data fit, not on the shape of a triangle's *xy* projection.

Data-dependent triangulation can be relatively fast. Had we used the straightforward algorithm, data-dependent triangulation would have been many times slower than Delaunay triangulation, since it would scan convert about twice as many input points, doing more work at each point, and it would visit each of these points twice, once for swap testing and once for candidate selection. We described a new, faster data-dependent triangulation algorithm that merges swap testing and candidate selection into one pass, saving a factor of two in cost.

With our implementation, we found that data-dependent triangulation takes about 3–4 times as long as Delaunay triangulation, and yields slightly higher quality on typical terrains. In applications where simplification speed is critical, Delaunay triangulation would be preferred, but if the quality of the approximation is primary, and the height field will be rendered many times after simplification, then the simplification cost is less important, and the data-dependent method is recommended.

Generality. The greedy insertion algorithm is quite flexible. It makes no assumptions that limit its usage to terrains. For example, it can be generalized to approximate color raster images by a set of Gouraud shaded polygons, or approximate computer vision range data with triangulated surfaces.

Comparison to Other Methods. Our Delaunay greedy insertion algorithm should produce nearly identical approximations to several previously published methods [4, 5, 26, 10, 8], but from the information available, it appears that our algorithm is the fastest both in theory and in practice.

Part of the adaptive triangular mesh filtering technique briefly described by Heller [17, p. 168] appears nearly identical in quality and asymptotic complexity to our Delaunay-based algorithm. Because the initial pass of his algorithm uses feature selection, we suspect, however, that his could be faster but that our method will produce somewhat higher quality approximations.

We have tested our sequential greedy insertion algorithm against parallel greedy insertion algorithms similar to those used by Fowler-Little and Puppo *et al.* [9, 25], and found that sequential greedy insertion yields superior approximations in all cases tested. We compared our error-based data-dependent triangulation criterion to a version of the normal-based criterion recommended by Rippa and Dyn *et al.* [26, 7], and found ours to be superior. We also compared the approximations of our algorithm against uniform grids (DEMs), and found that sequential greedy insertion generates more accurate approximations with a given number of vertices than uniform grids, contradicting some previously published conclusions [19].

Code. Portable C++ code for our Delaunay and data-dependent greedy insertion algorithms, and test data, is available by World Wide Web from http://www.cs.cmu.edu/~garland/scape or by anonymous FTP from ftp.cs.cmu.edu in /afs/cs/user/garland/public/scape.

7 Acknowledgements

We thank Michael Polis, Stephen Gifford, and Dave McKeown for exchanging algorithmic ideas with us and for sharing DEM data, and Anoop Bhattacharjya and Jon Webb for their thoughts on the application of these techniques to computer vision range data. The CMU Engineering & Science library has been very helpful in locating obscure papers. This work was supported by ARPA contract F19628-93-C-0171 and NSF Young Investigator award CCR-9357763.

References

[1] Xin Chen and Francis Schmitt. Adaptive range data approximation by constrained surface triangulation. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 95–113. Springer-Verlag, Berlin, 1993.

[2] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, Oct. 1976.

[3] Leila De Floriani. A pyramidal data structure for triangle-based surface description. *IEEE Computer Graphics and Appl.*, 9(2):67–78, March 1989.

[4] Leila De Floriani, Bianca Falcidieno, and Caterina Pienovi. A Delaunay-based method for surface approximation. In *Eurographics '83*, pages 333–350. Elsevier Science, 1983.

[5] Leila De Floriani, Bianca Falcidieno, and Caterina Pienovi. Delaunay-based representation of surfaces defined over arbitrarily shaped domains. *Computer Vision, Graphics, and Image Processing*, 32:127–140, 1985.

[6] Michael Deering. Geometry compression. In *SIGGRAPH '95 Proc.*, pages 13–20. ACM, Aug. 1995.

[7] Nira Dyn, David Levin, and Shmuel Rippa. Data dependent triangulations for piecewise linear interpolation. *IMA J. Numer. Anal.*, 10(1):137–154, Jan. 1990.

[8] Per-Olof Fjällström. Evaluation of a Delaunay-based method for surface approximation. *Computer-Aided Design*, 25(11):711–719, 1993.

[9] Robert J. Fowler and James J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics (SIGGRAPH '79 Proc.)*, 13(2):199–207, Aug. 1979.

[10] W. Randolph Franklin. tin.c, 1993. C code, ftp://ftp.cs.rpi.edu/pub/franklin/tin.tar.gz.

[11] Michael Garland and Paul S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical report, CS Dept., Carnegie Mellon U., Sept. 1995. CMU-CS-95-181, http://www.cs.cmu.edu/~garland/scape.

[12] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):75–123, 1985.

[13] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992. Also in Proc. 17th Intl. Colloq. — Automata, Languages, and Programming, Springer-Verlag, 1990, pp. 414–431.

[14] Bernd Hamann and Jiann-Liang Chen. Data point selection for piecewise trilinear approximation. *Computer-Aided Geometric Design*, 11:477–489, 1994.

[15] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, Banff, Canada, May 1994. Canadian Inf. Proc. Soc. http://www.cs.cmu.edu/~ph.

[16] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report, CS Dept., Carnegie Mellon U., to appear. http://www.cs.cmu.edu/~ph.

[17] Martin Heller. Triangulation algorithms for adaptive terrain modeling. In *Proc. 4th Intl. Symp. on Spatial Data Handling*, volume 1, pages 163–174, Zürich, 1990.

[18] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *SIGGRAPH '93 Proc.*, pages 19–26, Aug. 1993. http://www.research.microsoft.com/research/graphics/hoppe/.

[19] Mark P. Kumler. An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica*, 31(2), Summer 1994. Monograph 45.

[20] Charles L. Lawson. Software for C^1 surface interpolation. In John R. Rice, editor, *Mathematical Software III*, pages 161–194. Academic Press, NY, 1977. (Proc. of symp., Madison, WI, Mar. 1977).

[21] Jay Lee. A drop heuristic conversion method for extracting irregular network for digital elevation models. In *GIS/LIS '89 Proc.*, volume 1, pages 30–39. American Congress on Surveying and Mapping, Nov. 1989.

[22] Dani Lischinski. Incremental Delaunay triangulation. In Paul Heckbert, editor, *Graphics Gems IV*, pages 47–59. Academic Press, Boston, 1994.

[23] Edmond Nadler. Piecewise linear best L_2 approximation on triangulations. In C. K. Chui et al., editors, *Approximation Theory V*, pages 499–502, Boston, 1986. Academic Press.

[24] Michael F. Polis and David M. McKeown, Jr. Issues in iterative TIN generation to support large scale simulations. In *Proc. of Auto-Carto 11 (Eleventh Intl. Symp. on Computer-Assisted Cartography)*, pages 267–277, November 1993. http://www.cs.cmu.edu/~MAPSLab.

[25] Enrico Puppo, Larry Davis, Daniel DeMenthon, and Y. Ansel Teng. Parallel terrain triangulation. *Intl. J. of Geographical Information Systems*, 8(2):105–128, 1994.

[26] Shmuel Rippa. Adaptive approximation by piecewise linear polynomials on triangulations of subsets of scattered data. *SIAM J. Sci. Stat. Comput.*, 13(5):1123–1141, Sept. 1992.

[27] David A. Southard. Piecewise planar surface models from sampled data. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 667–680, Tokyo, 1991. Springer-Verlag.

Figure 16: Mandrill original, a 200×200 raster image.

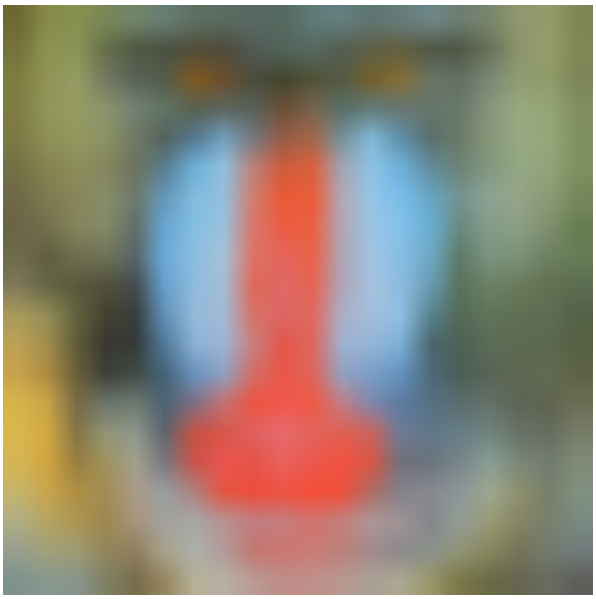


Figure 17: Mandrill approximated with Gouraud shaded triangles created by subsampling on a uniform 20×20 grid (400 vertices).

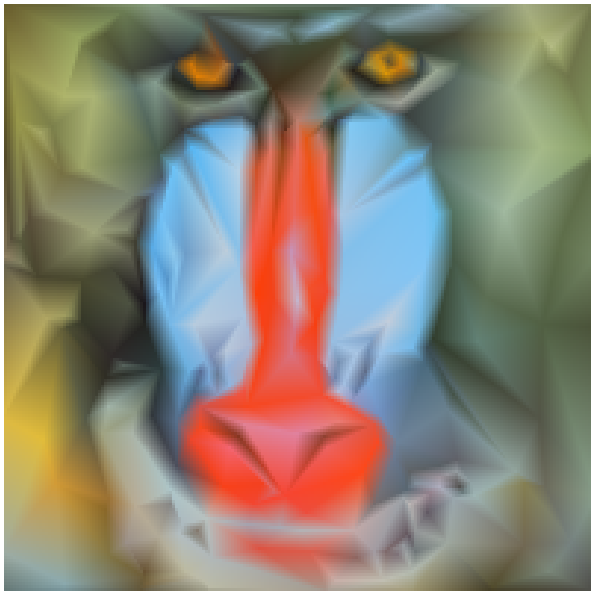


Figure 18: Mandrill approximated with Gouraud shaded triangles created by data-dependent greedy insertion (400 vertices).

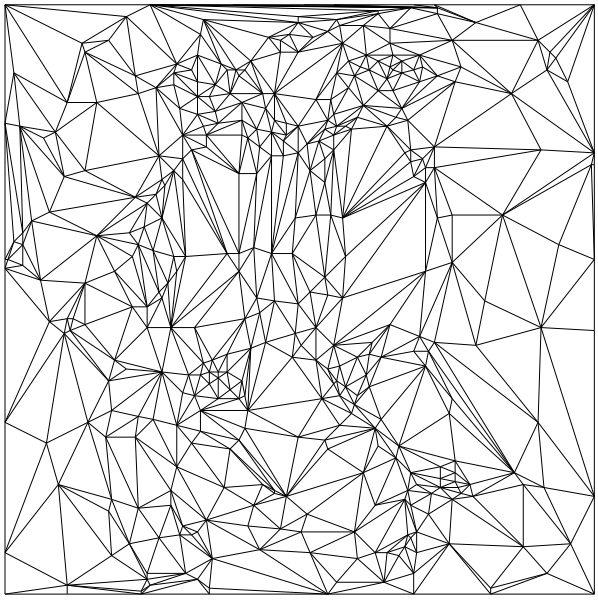


Figure 19: Mesh for the image to the left.

Algorithms for Surface Simplification

Paul Heckbert & Michael Garland
Computer Science Dept.
Carnegie Mellon University

Typical Problems

Typical Curve Simplification Problem:

Given curve with n vertices, find an accurate approximation using m vertices.

Given curve with n vertices, find a compact approximation with error $< \varepsilon$.

Typical Surface Simplification Problem:

Given surface with n vertices, find accurate approximation using m vertices.

Given surface with n vertices, find a compact approximation with error $< \varepsilon$.

Who Does Curve & Surface Simplification?

Cartography, Geographic Information Systems - "generalization" for aesthetics, compact storage. Curves, mostly.

Simulators - military applications, virtual reality, terrain, break into blocks, multiresolution models (level of detail), smooth transitions.

Computer Graphics - real time rendering, animation, compact for storage & transmission.

Scientific Visualization - clean up "marching cubes" isosurfaces.

Computer Vision - acquired range data (noisy), model fitting.

Computational Geometry - worst case complexity of optimal approximation.

Computer-Aided Geometric Design - curved surface modeling.

Approximation Theory - error analysis of asymptotic approximation.

Simplification Problem Characteristics

What problem do you want to solve?

- **topology of output**: curve or surface.
- **topology & geometry of input**: points, function *f(x)*, curve, height field *f(x,y)*, manifold, surface.
- **other attributes**: color, texture.
- **domain of output**: subset of input vertices?
- **topology of triangulation**: uniform, hierarchical, general
- **approximating elements**: linear, quadratic, cubic, ..., other.
- **error metric**: L_2 = sum of squared, L_∞ = maximum.
- **constraints**:
  - most accurate using a given number of elements or amount of memory
  - smallest satisfying a given error tolerance.

Simplification Algorithm Characteristics

How do you want to solve the problem?

- **speed/quality tradeoff**: optimal (& slow) or sub-optimal (& fast)?
- **refinement/decimation**: top down or bottom up?
- **number of passes**: one pass or multiple passes?
- **triangulation**: hierarchical triangulation, Delaunay triangulation, data-dependent triangulation, or other?

Curve Simplification

Douglas-Peucker algorithm:

```
douglas_peucker(curve[1..n], ε)
    i := index of vertex of curve farthest from chord 1..n
    if dist(i) < ε then return chord 1..n
    else
        douglas_peucker(curve[1..i], ε)
        douglas_peucker(curve[i..n], ε)
```

Cost: If *m* output vertices, *O(mn)* worst case, *O(n* log *m)* expected.
O(n log *n)* worst case using convex hulls.

Best known *optimal* solutions cost *O(n)* to *O(n^3)*, depending on problem.

Taxonomy of Surface Simplification Methods

(we list some of the better methods)

Height Field / Parametric Simplification

subsampling, pyramid, quadtree methods	many
greedy insertion	Garland-Heckbert, ...

Manifold Simplification

vertex decimation	Schroeder
vertex decimation with point lists	Soucy
wavelet	Eck, Lounsbery, ...
edge collapse	Ronfard-Rossignac, ...

Non-Manifold Simplification

vertex clustering	Rossignac-Borrel

Pyramids and Quadtrees

Input: height field or parametric surface with uniform mesh.

Preprocessing:

Build power-of-two pyramid/quadtree.

Display Algorithm:

At each node of tree, decide how much resolution is needed, descend tree.
Do careful triangulation to avoid cracks.

Advantages: Compact storage (z only; x & y are implicit). Simple. Fast.

Disadvantages: Pyramids unnecessarily bulky on smooth surfaces. Poor quality approximations. Crack prevention can be tricky.

Widely used in flight simulators.

Greedy Insertion [Garland-Heckbert95]

Input: height field or parametric surface.

Preprocessing:

Start with two big triangles.

Until error small enough or approximation too big

Find point of highest error (approximation-original).

Insert it in triangulation (Delaunay or data-dependent).

Display Algorithm:

draw it.

Advantages: Fast: $O((n+m) \log m)$.

Disadvantages: Moderate quality approximations. Susceptible to noise, discontinuities. Limited to height fields/parametrics.

Mesh Decimation [Schroeder et al. 92]

Input: polygonized manifold with boundary.

Preprocessing:

$\varepsilon = \varepsilon 0$

Until error too high or approximation small enough

For all vertices

if dist(vertex, neighbors' approximating plane) < ε then

delete vertex

$\varepsilon = \varepsilon + \Delta\varepsilon$

Display Algorithm:

draw it.

Advantages: Fast. Accepts fairly general inputs.

Disadvantages: Moderate quality approximations. Choice of $\Delta\varepsilon$. Big memory.

Wavelet Approximations [Eck95,Lounsbery94]

Input: manifold surface.

Preprocessing:

Simplify mesh topology to find base mesh.

Resample: subdivide base mesh to approximate input surface.

Do multiresolution analysis (wavelet transform) on resampled surface.

Display Algorithm:

At each node of tree, decide how much resolution is needed.

Reconstruct surface using multiresolution synthesis.

Do careful triangulation to avoid cracks.

Advantages: Fast display.

Disadvantages: Requires resampling. Approximates creases poorly.

Mesh Optimization [Hoppe et al. 93]

Input: triangulated manifold with boundary.

Preprocessing:

Loop, decreasing spring constant.

Loop, semi-randomly perturbing topology (delete vertex, swap edge, etc.)

Loop, optimizing geometry to fit input points.

Display Algorithm:

draw it.

Advantages: High quality. Accepts fairly general inputs.

Disadvantages: Slow. Requires some tuning.

Edge Collapse
[Ronfard-Rossignac96, Hoppe96, Gueziec95, Garland97]

Input: triangulated manifold with boundary.

Preprocessing:

Until error too high or approximation small enough
- find edge whose collapse would introduce the least error
- collapse edge into vertex
- position new vertex carefully
- save sequence of edge collapses

Display Algorithm:

replay sequence of edge collapses/vertex splits (progressive mesh)
or geomorph (faster)

Advantages: High quality. Continuous levels of detail. Accepts fairly general inputs. Error bounds (approximate).

Disadvantages: Not the fastest. Error bounds not the best.

3-D Grid Method [Rossignac-Borrel93]

Input: set of triangles (non-manifold).

Preprocessing:

Divide space into a uniform grid of boxes.
Merge all vertices within each box into their centroid.
Eliminate degenerate/redundant triangles.

Display Algorithm:

draw it.

Advantages: Very general - runs on any polygonal model. Fast.

Disadvantages: Output is grid-dependent. Non-adaptive. Low quality approximations.

IBM sells this software in their “Interaction Accelerator”

Conclusions & Recommendations

With current algorithms:

Poor quality, fast, **non-manifold**:
Rossignac-Borrel's 3-D grid method.

Top quality, moderate speed, manifolds:
Ronfard-Rossignac/Hoppe/Garland edge collapse with geomorph.

Moderate quality, **fast**, height fields:
Garland-Heckbert greedy insertion.

But this field is developing fast, so stay tuned.

Geometric simplification and compression

Jarek Rossignac

GVU Center and College of Computing
Georgia Institute of Technology

One third of the workstation business is driven by industrial or scientific applications that heavily depend on the ability to interactively render 3D models whose complexity is increasing far more rapidly than the performance of the graphics subsystems. Simple scenes in 3D video-games may only need a few hundred textured polygons, but models used for mechanical CAD, scientific, geo-science, and medical applications involve scenes with millions of faces, sometimes grouped to form the boundaries of polyhedral solids of widely varying complexity. The successful exploitation of such large volumes of scientific and engineering three-dimensional data hinges on users' ability to access the data through phone lines or network connections and to manipulate significant portions of these 3D models interactively on the screen for scientific discovery, teaching, collaborative design, or engineering analysis. Currently available high-end graphics hardware is often insufficient to render the models at sufficient frame rates to support direct manipulation and interactive camera control.

The abundance and importance of complex 3D data bases in major industry segments, the affordability of interactive 3D rendering for office and consumer use, and the exploitation of the internet to distribute and share 3D data have exacerbated the need for an effective 3D geometric compression technique that would significantly reduce the time required to transmit 3D models over digital communication channels and the amount of memory or disk space required to store the models. Because the prevalent representation for 3Dshapes for graphics purposes is polyhedral and because polyhedra are in general triangulated for rendering, it is important to focus on the compression and decompression of complex triangulated models.

Although we anticipate further development of hardware acceleration graphics engines and new commercial explorations of hardware compression/decompression chips, we discuss software advances on both fronts, hoping that with maturity they will fuel hardware advances and impact data exchange standards.

Software techniques, which eliminate unnecessary or unessential rendering steps, may lead to dramatic performance improvements and hence reduce hardware costs for graphics. Many of these techniques require complex algorithmic pre-processing and a compromise on the quality and accuracy of the images. The relative impact of software techniques for accelerating the rendering of 3D scenes depends on the complexity and characteristics of the model, on the lighting model, on the image resolution, and on the hardware configuration. Acceleration techniques include hierarchical culling, memory management, visibility computation, reduced resolution or accuracy, model simplification, use of images, textures, or perturbation maps, and the optimization of the rendering library.

We first discuss a simple model of the rendering cost and review the impact of various acceleration techniques on the different cost factors and stress the need to combine the various techniques. We then focus 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. Such decreasing levels-of-detail (LOD) involve less faces and vertices, and hence require less memory and less geometry processing at rendering time. Using a lower level of detail when displaying small, distant, or background objects improves graphic performance without a significant loss of perceptual information, and thus enables real-time inspection of highly complex scenes. Original models are used for rendering objects close to the viewpoint and during navigation pauses for full precision static images.

Simplification is an automatic process that takes a polyhedral surface model S and produces a model S' that resembles S, but has significantly less vertices and faces. We discuss both geometric and visual measures of the discrepancy between S and S' and review the general principles for computing and evaluating such simplifications. We briefly review several simplification techniques based on curved-surface tessellation, space or surface sampling, and bounding hierarchies. We investigate in more details the variants of vertex clustering and face merging techniques, which include edge collapsing and triangle or vertex decimation. We illustrate these techniques through several approaches: Kalvin and Taylor aggregate nearly co-linear facets into connected regions, simplify their boundaries, and triangulate the regions; Ronfard and Rossignac expand on the edge-collapsing work of Hoppe et al. by maintaining an efficient point-plane distance criteria for estimating the errors associated with each candidate edge; Gueziec preserves the volume of the model and uses spheres as error bounds; Rossignac and Borrel use vertex quantization (integer rounding on vertex coordinates) to efficiently

compute the clusters. The latter simplification process is more efficient, more robust, and simpler to implement than alternative approaches, because it does not require building and maintaining a topological face-vertex incidence graph and thus does not impose any topological integrity restrictions on the original model. Although it is not adaptive and does not produce optimal simplifications for a given complexity reduction, it appears particularly well suited for mechanical and architectural CAD models, because it automatically groups and simplifies features that are geometrically close, but need not be topologically close nor even be part of a single connected component.

These techniques produce discrete simplifications, i.e. models that are uniformly simplified to different resolutions. They work well for scenes with many small objects uniformly distributed through space. Large objects may require an adaptive simplification model, where the resolution of the approximation is automatically adapted to the location of the view point (and decreases with the distance to the viewer). We illustrate this approach with Hoppe's Progressive Meshes and with the adaptive terrain models developed at the Georgia Tech's GVU center.

Simplification reduces the triangle counts, but does not address the problem of data compression. In fact, one needs to store the original model plus its simplified versions. 3D compression techniques may be applied to the original and the simplified models independently. They do not reduce the triangle count, but reduce the number of bits required to code the vertex coordinates (geometry), the triangles' references to their supporting vertices (incidence), and the normals and color attributes (photometry). We discuss Deering's extensions of the triangle strips encoding to more general triangular meshes, and Taubin and Rossignac's "topological surgery" approach, which captures the incidence data in less than 2 bits per triangle (instead of the naive 126) and uses vertex quantization, geometric prediction, and entropy coding to compress geometry and photometry with controllable loss. Finally, we expand on Hoppe's Progressive Meshes to lay the ground for further research towards the unification of simplification and compression and towards the use of such scaleable models for internet-based 3D data access and 3D collaborative applications.

CR Categories and Subject Descriptors: I.3.3 **Computer Graphics**: Picture/Image Generation -- display algorithms;
I.3.5 **Computer Graphics**: Computational Geometry and Object Modeling -- curve, surface, solid, and object representations;
I.3.7 **Computer Graphics**: Three Dimensional Graphics and Realism.
General Terms: 3D Polyhedron Simplification, Levels-of-detail, 3D compression, Algorithms, Graphics, Performance acceleration.

Introduction

Computer graphics is an effective means for communicating three-dimensional concepts, for inspecting CAD models, and for interacting with education and entertainment software. Although photo-realistic images and video sequences, typically computed off-line, have their own merits, interactive graphics is required when direct manipulation is important. Direct manipulation increases ease-of-use and enhances productivity by exploiting our natural ability to use visual cues when controlling our gestures. The closed loop involving the hand, the modeling or animation software, the graphics, and the human vision is only effective if graphic feedback is instantaneous. Delays between the gestures and the resulting image, lead to overshooting, reduce the feeling of control, and thus make the user less productive.

Interactive manipulation of 3D models and interactive inspection of 3D scenes cannot be effective when the graphics feedback requires more than a fraction of a second. Although graphics performance, has significantly increased in the recent years (benefiting from faster CPUs, leaner APIs, and dedicated graphics subsystems), it is lagging behind the galloping complexity of CAD models found in industrial and consumer applications and of scientific datasets [Rossignac94], 3D models of typical mechanical assemblies (appliances, engines, cars, aircrafts...) contain thousands of curved surfaces. Accurate polyhedral approximations of these exact models, typically constructed to interface with popular graphics APIs, involve millions of polygonal faces. Such complexity exceeds by one or two orders of magnitude the rendering performance of commercially available graphics adapters. The solution is not likely to come from hardware development alone. Algorithmic solutions must be employed to reduce the amount of redundant or unessential computation.

We define *redundant* computation as the processing steps that could be omitted without affecting the resulting images. *Unessential* computation is defined as the steps which, when omitted, affect the resulting image only to a moderate degree, so that the visual discrepancies between the correct image and the one produced do not hinder the user's perception and understanding of the scene. The elimination of all redundant and unessential computations remains an open research issue, because the computational cost associated with the identification of what is redundant and what is not may often offset the benefits of eliminating redundancy. Several techniques are reviewed below.

To better identify redundant and unessential steps for rendering polyhedral scenes, we use an over-simplified model of the rendering cost. The scene is described by its geometry (the location of the polygons and the location and characteristics of the view) and by the associated photometric properties (light sources, surface colors,

surface properties, textures). Rendering techniques compute, for each pixel, the amount of light reflected towards the eye by the objects in the scene. This computation may be arranged in various ways to cater to the desired degrees of photo-realism and to take advantage of the available graphics hardware or libraries. We focus here on rasterization techniques, which visit each polygon in the scene and combine the results in a frame buffer using a z-buffer for hidden surface elimination. This choice is dictated by the prevalence and performance of affordable 3D graphics hardware accelerators and rasterizers, and by the success of associated APIs.

Assume that the geometry of our scene is composed of T triangles. We use triangle counts in these arguments, instead of polygons, for simplicity and because polygons are typically converted into triangles, either during preprocessing [Rofard94] or during rendering, and are an accepted unit of graphics performance. Procedures for triangulating models bounded by curved parametric surfaces also exist.

The cost of rendering a scene comprising T triangles depends on a number of factors, such as the window size, the lighting model, the amount of memory paging involved, or how the triangles project onto the screen.

Rendering T triangles involves several costs:
- **F:** the cost of fetching the necessary triangles into cache memory
- **X**: the cost of transforming their vertices and lighting them using associated normals
- **C**: the cost of clipping the triangles and computing slopes
- **R**: the cost of rasterizing them

The overall rendering cost (i.e. performance) depends on the particular architecture of the 3D graphics subsystem. Three families of uniprocessor architectures are popular for graphics:
- all software rendering
- hardware rasterization with software geometric processing
- all hardware rendering

For purely software rendering architecture, the total cost is the sum of all the costs: **F+X+C+R**. For an architecture based on a hardware rasterizer, the total cost is the maximum of **max(F+X+C,hR**), where h provides the boost factor of the graphics rasterizer hardware. Because the hardware or the software may be bottlenecks, we use max instead of a sum when combining the individual costs. Similarly, because an all hardware rendering subsystem is pipelined, its cost is the maximum of the costs at each stage, i.e.: **max(F,kX+kC,hR**), where k provides the boost factor of the hardware-supported transformation, lighting, and clipping. Note that this simplified formula does not take into account the statistics effects of data buffers and load balancing in parallel architectures.

Review of graphics acceleration techniques

The performance enhancing techniques reviewed in this section help reduce the different aspects of the overall rendering cost. None of these techniques is usually sufficient to address the graphics performance problem and most systems exploit combinations of several techniques simultaneously.

Meshing or storing transformed vertices

In a polyhedral scene, the number **T** of triangles is roughly twice the number **V** of vertices. For example, for a triangulated manifold polyhedron with no holes or handles, the Euler-Poincarre formula becomes: **T-E+V=2,** where E is the number of edges. Notice that **2E=3T,** because there are three edges for each triangle, but each edge of a manifold is shared by two adjacent triangles. Combining these two equalities yields: **T-3T/2+V=2**, and finally: **T=2V-4.**

Therefore, independently processing the vertices of each triangle may unnecessarily increase **X**, and maybe even **F**, by a factor of 6, since on average a vertex is processed six times (3 times per triangle and there are twice more triangles than vertices).

To reduce this computational redundancy, 3D graphics adapters and associated APIs support triangle meshes, where each vertex is used in conjunction with two of the recently processed vertices to define the next triangle. If very long meshes were used, the majority of vertices, and associated normals would only be processed twice, reducing by three the geometric cost of transforming and lighting the vertices. Note however that half of this cost is still redundant. Furthermore, constructing long meshes is algorithmically expensive.

In graphics architectures where the geometric transformation and lighting is performed in software, all the vertices could be transformed only once and the resulting coordinates stored for clipping and rasterization, as needed for the triangles. Similarly, normals may be transformed and the associated colors computed and stored (depending on the lighting model, colors may be defined by normals only or by combinations of normals and

vertices). For compatibility reasons with hardware graphics adapters, this possibility is not systematically exploited by popular APIs.

Recent progress attempting to generalize the notion of a triangle strip to cover triangular meshes of arbitrary topology with minimal vertex duplication may be found in the context of geometric compression [Taubin96, Deering95, Hoppe96].

Frustum culling

Graphics performance may in general be significantly improved by avoiding to process the geometry of objects that do not project on the screen. Such objects may lie behind the viewpoint or outside of the viewing frustum. Simple bounds, such as min-max boxes or tight spheres around the objects may be easily pre-computed, updated during model editing or animation, and used for efficient culling. For scenes involving a large number of objects or for scenes where individual objects are relatively large and involve large numbers of triangles, pre-computed hierarchical spatial directories (see [Airey90, Clark76, Teller91, Naylor95]) are used to quickly prune portions of the scene outside the viewing frustum. These early culling techniques may have no effect when the entire models is examined and fits in the viewing frustum. In average however, they may significantly affect F, because one needs not fetch models whose bounds is outside the frustum, and of course also **X** and **C**. They do not affect R, because these triangles would have been rejected through clipping anyway. For example, culling may reduce **X** and **C** by a factor of if the viewpoint is in the center of the scene and if the viewing frustum spans a 90 degrees angle.

Smart caching and pre-fetching

Arranging the data so that contiguous memory locations are accessed by the graphics subsystem and using secondary processors for pre-fetching data may eliminate the delays caused by page faults and reduce **F**. This is particularly effective if the size of the model that is not culled out exceeds the available memory [Funkhouser93].

Pre-computed visibility

Culling objects or individual triangles that intersect the frustum but are invisible in the current view, because hidden by other objects, would directly impact the overall cost for all three types of architectures. Visibility information may be pre-computed for a given granularity of the space and of the models. For example, [Teller92] uses the model's geometry to subdivide space into cells, associating with each cell **c** the list of other cells visible from at least one point in **c**. Cells may correspond to a natural, semantic, partition of the scene (floors, rooms, corridors) or to more general partitions induced by planes that contain the faces of the model. The preprocessing is slow and requires storing vast amounts of visibility information, unless the number of cells and the number of objects in the scene is small. To improve performance, portals (i.e, holes, such as doors and windows, in the boundary of cells) may be approximated by an enclosing axis aligned 2D box in image space, which provides only a necessary condition for visibility, but reduces visibility tests to clipping against the intersection of such rectangles [Luebke95], A different approach was used in [Greene93], where a hierarchical quadtree representation of the z-buffer generated after displaying the front most objects is used to quickly cull hidden objects. Front most objects candidates are computed from the previous frame. Culling compares a hierarchical bound system of the 3D scene against the z-buffer quadtree. Both techniques are effective under the appropriate conditions, but are of little help in scenes, such as factories or exterior views, where very few objects are completely hidden.

Use of images

A distant or small group of objects may be replaced with a "3D sprite" or with a few textured polygons [Beigbeder91] showing the approximate image of the object(s) from the current viewpoint. A hierarchical clustering approach where groups of objects were displayed using such images was proposed in [Maciel95]. Previously rendered images with the associated z-buffer information provide an approximation of the model as seen from a specific location. They may be warped and reused to produce views from nearby locations on an inexpensive graphics system. Because the new view may reveal details hidden in the previous view, a more powerful graphic server may identify the discrepancies and send the missing information to the low-end client [Mann97].

Levels of detail

Using simplified models with a lower triangle count instead of the original models may in certain cases reduce paging **F**, and will in general reduce **X** and **C** significantly [Funkhouser93]. Lower levels of detail are usually appropriate for rendering distant features that appear small on the screen during camera motions or scene animations [Crow82]. During navigation pauses, the system will start computing the best quality image and, if not interrupted by the user, will automatically display it [Bergman86]. A review of the early techniques may be found in [Erikson96]. Two main approaches to the construction and use of such levels of detail (LOD), multi-resolution models have been investigated by a large number of researchers:

- Pre-compute a series of **static** view-independent simplifications for each subset of the scene (solid, group) which approximate the original model to decreasing levels of accuracy and select during navigation the appropriate level of details for each subset, depending on the viewing conditions and on the desired performance/accuracy trade-off.
- Pre-compute a unique **adaptive** model for the entire scene and during navigation adapt it to the current viewing conditions to produce a graphics model which approximates features close to the viewer with higher accuracy than distant features, which appear small on the screen.

A comparative discussion of some of the early techniques [Blake87] for computing static simplifications may be found in [Erikson96, Heckbert94]. We summarize here several trends to provide the context for the rest of the paper, which focuses on vertex clustering and edge collapsing techniques.

Polyhedral models are often constructed to approximate curved shapes through **surface tessellation**. Polyhedral approximations with fewer vertices and faces can thus in principle be produced by using coarser tessellation parameters. Emerging high-end graphic architectures support adaptive tessellation for trimmed NURBS surfaces [Rockwood89], However, the simplification process should be made independent of the design history and should be automatic [Crow82]. These surface tessellation techniques are of little help when trying to simplify complex shapes that involve thousands of surface matches, because they can at best simplify each surface patch to a few triangles, but cannot merge the triangulations of adjacent patches into much simpler models.

Surface fitting techniques to regularly spaced scanner data points [Schmitt86, Algorri96] may also be used for producing levels of detail polyhedral approximations [DeHaemer91]. These techniques have been recently extended to unorganized data points [Hoppe92, DeRose92] and to new sparse points automatically distributed over an existing triangulated surface (either evenly or according to curvature) [Turk92], These techniques suffer from expensive preprocessing. but yield highly optimized results.

Techniques for constructing approximating **inner and outer bounds** (or offset surfaces) for polyhedra have been extended to create simplified models that separate the inner and the outer offsets [Varshney94, Cohen96, Mitchell95]. The creation of valid, intersection-free offset surfaces still poses some challenges, although techniques based on extended octree representations of the interior and of the boundary of the polyhedron provide inner and outer bounds [Andujar96].

The rest of this paper focuses on **polygon count reduction** techniques that exploit an original triangular mesh and derive simplified models by eliminating vertices or triangles, by collapsing edges, or by merging adjacent faces.

Vertex clustering, the simplest to implement and most efficient approach, 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 [Rossignac93]. 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 Figure 3).

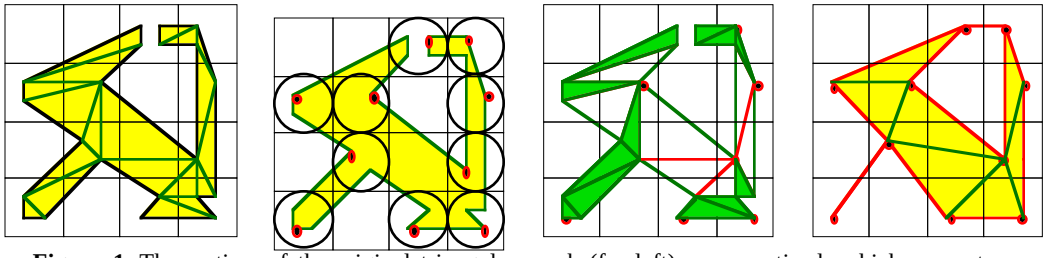


Figure 1: The vertices of the original triangular mesh (far left) are quantized, which amounts to associating each vertex with a single cell of a regular subdivision of a box then encloses the object. Cells which contain one or more vertices are marked with a circle (center left). All the vertices that 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.

A different technique, **edge collapsing,** merges the two end points of the edges of the polyhedron one edge at a time [Hoppe93, Ronfard96].. Each edge collapse operation removes two triangles (see Figure 2). The accuracy of the resulting model is estimated as the error cumulated by the sequence of edge-collapses. This incremental process permits to achieve the desired accuracy or the desired triangle count and is well suited for optimization (selection of the sequence of edge collapsing operations which results in the lowest error). Early version of edge collapsing techniques are more complex to implement than vertex clustering approaches (they require maintaining a complete incidence graph), are significantly slower, and impose strong topological constraints on the input polyhedra. Edge collapsing is in fact a restricted form of vertex clustering, since an edge collapse operation merges two clusters that are linked by an edge of the polyhedron. Both techniques may be combined [Hoppe97].

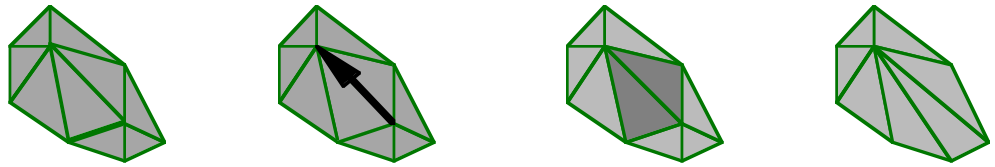


Figure 2: From left to right: the original triangle mesh, a selected edge showing the direction of collapse, the two triangles that will be eliminated during the collapse, and the resulting simpler mesh.

The **decimation** of a nearly-flat vertex [Schroeder92] is equivalent to an edge collapsing operation, possibly combined with edge-flips [Lawson72] (Fig. 3), and thus results in a hierarchical clustering of vertices. Its generalization [Kalvin91, Kalvin96] re-triangulates nearly flat regions constructed by incrementally merging triangles. Re-triangulating a region amounts to clustering all the internal vertices into a single "star" vertex, the apex of the new triangulation for the region.

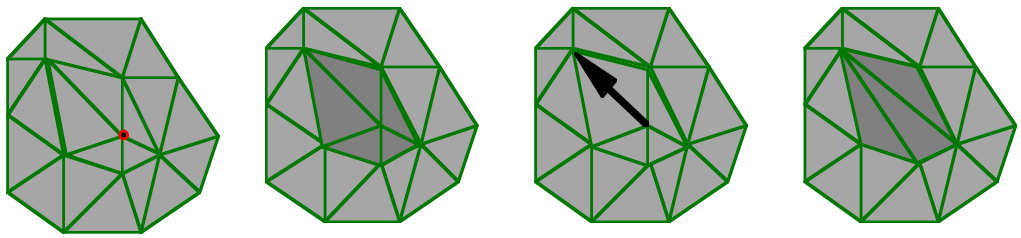


Figure 3: Decimating the highlighted vertex (far left) will affect the faces marked (center left). The affected region will be retriangulated (far right). The transformation corresponds to the edge collapse indicated by the arrow (center right).

Static simplifications are effective for **large and complex objects** only when these objects are viewed from far. Inspecting the details of a local feature on such a large complex object requires using the highest resolution for the entire object, which imposes that full resolution be used for the distant parts of the objects that could otherwise be displayed at much lower resolution. It is not uncommon that the complexity of a single object significantly exceeds what the graphics subsystem can render at interactive rates. An **adaptive multi-resolution** model is better suited for such situations [Xia96]. In fact, an adaptive model may be computed for the entire scene. An hierarchical version of the vertex clustering approach of [Rossignac93] is the basis of a new adaptive scheme [Luebke96], where octree nodes [Samet90] correspond to the hierarchy of vertex clusters. The main research challenges in the perfection of adaptive multiresolution models lie in: (1) the rapid generation of triangle strips for each view dependent simplification., (2) a fast decision process updating the levels of resolution at each new frame, and (3) the estimation of a tight error bound. Although it may be expensive to convert arbitrary triangular meshes into subdivision surfaces, wavelet-based hierarchical models offer an attractive and elegant solution to this problem [Eck95, Lounsbery94].

Adaptive level-of-detail for **terrain models** have been studied extensively [Garland95], because a terrain model is typically a single complex surface that must be rendered with non-uniform resolution and partly because error estimations for terrain models is simpler than for arbitrary 3D polyhedra. For example, hierarchical triangulated

irregular network (HTIN), were proposed [DeFloriani92] as an adaptive resolution model for topographic surfaces. Researchers at Georgia Tech's GVU center have developed an adaptive multi-resolution model for the realtime visualization of complex terrain data over a regular grid [Lindstrom96]. Their approach uses a recursive decision tree to indicate which edges may be collapsed and also to set preconditions: an edge may not be collapsed unless its children have been collapsed.

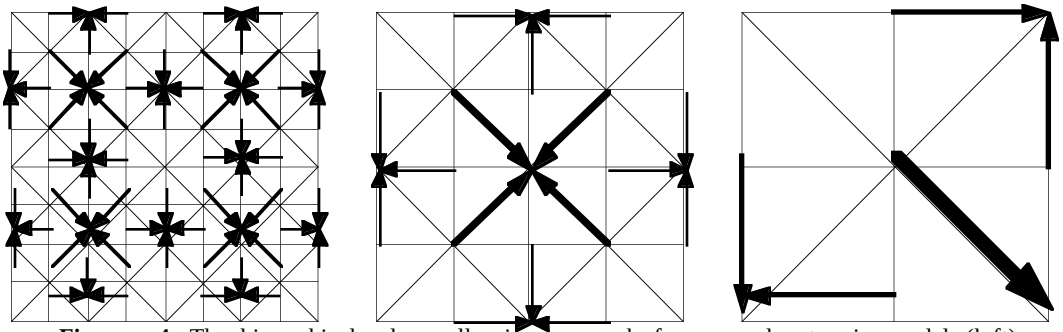


Figure 4: The hierarchical edge collapsing approach for a regular terrain model (left) produces a similar lower-resolution models by a sequence of vertical and then diagonal edge collapses (center). The process may be repeated iteratively (right). Not all edges at a given level need to collapse.

Review of 3D compression techniques

In comparison to image and video compression, little attention has been devoted to the compression of 3D shapes, both from the research community and from 3D data exchange standards committees. This situation is likely to change rapidly for three reasons: (1) the exploding complexity of industrial CAD models raises significantly the cost of the memory and auxiliary storage required by these models, (2) the distribution of 3D models over networks for collaborative design, gaming, rapid prototyping, or virtual interactions is seriously limited by the available bandwidth, and (3) the graphics performance of high level hardware adapters is limited by insufficient on-board memory to store the entire model or by a data transfer bottleneck.

We focus our study of 3D compression on triangular meshes which are shells of pairwise adjacent triangles. We chose triangle-based representations, because more general polygonal faces may be efficiently triangulated [Ronfard94], and because triangles provide a common denominator for most representation schemes. Furthermore, the number of triangles is a convenient measure of a model's complexity, which is important for comparing various compression techniques. We will mostly focus here on simply connected manifold meshes, where triangles are mutually disjoint (except at their edges and vertices), where each edge is adjacent to exactly two incident triangles, and where each vertex is adjacent to exactly one cone of incident triangles. Furthermore, we will assume for simplicity that the mesh is a connected surface with zero handles. These restrictions are made purely for the sake of simplicity, and most of the compression schemes reviewed here work or may be expanded to cope with more general meshes.

A triangular mesh is defined by the position of its vertices (geometry), by the association between each triangle and its sustaining vertices (incidence), and by color, normal, and texture information (photometry), which does not affect the 3D geometry, but influences the way the triangle is shaded.. What is the minimum number of bits required to encode a triangle mesh of **T** triangles and **V** vertices? Because for our simple meshes there are roughly twice more triangles than vertices, we will assume that **T=2V** and use **V** as a measure of the complexity of the mesh. For uniformity, we will assume that a unique normal (photometry) is associated with each vertex and that these normals are different for all vertices. To illustrate the storage requirement, we will pick **V** to be 2^{16} (i.e. 64K), which is a reasonable compromise between an average of about 500 vertices per solid in mechanical assemblies and significantly larger vertex counts for large architectural objects, terrain models, or medical datasets.

We compare below the storage requirements for several representation schemes. Of course other general purpose loss-less compression schemes [Pennebakerl93] may be applied to the bit stream resulting from these approaches, they will not be considered in our comparison.

An **array of triangles** is the simplest representation of a triangle mesh, It represents each triangle independently by the list of its vertex and normal coordinates, each represented by a 4 byte floating point number. Hence the number of bits per vertex for this simple representation of a triangle mesh of **V** vertices is **1152,** the product of the following terms:

- 2 triangles per vertex
- 3 vertex uses per triangle
- 2 vectors (vertex location and normal) per vertex use
- 3 coordinates per vector
- 4 bytes per coordinate
- 8 bits per byte

Note that a vertex is on average adjacent to 6 triangles and therefore, there are 6 vertex uses (i.e., descriptions of the same vertex (geometry and photometry) stored in the simple representation above, which does not need to encode explicitly the triangle-vertex incidence relation, since it is dictated by the place of the vertices in the data stream.

This redundancy can be eliminated by dissociating the representation of the vertices (location and normal) from the representation of the incidence relation, which requires 3 vertex references per triangle. Because a vertex reference often requires less bits than a vertex description, such schemes based on a **vertex and normal table** are more compact than the simple representations. In our example, if we store vertices in a table and reference them by an integer index, we need 16 bits for each vertex reference, since $\mathbf{V=2^{16}}$**.** Consequently, the vertex and normal table representation requires **288** bits per vertex. **192** bits for the geometry and photometry:

- 2 vectors (vertex location and normal) per vertex
- 3 coordinates per vector
- 4 bytes per coordinate
- 8 bits per byte

and **96** bits for the incidence information:

- 2 triangles per vertex
- 3 vertex references per triangle
- 16 bits per vertex reference

However, such a compression scheme requires random access to the vertices, which is acceptable in graphics systems with software geometry processing, but not suitable for fully hardware graphics systems, where there is limited register memory for a few vertices and associated normals.

A representation based on **triangle strips**, supported by popular graphics APIs, such as OpenGL [Neider93], is used to provide a good compromise for graphics. It reduces the repeated use of vertices from an average of 6 to an average of 2.4 (assuming an average length 10 triangles per strip, which is not easily achieved) and is compatible with the graphics hardware constraints. Basically, in a triangle strip, a triangle is formed by combining a new vertex description with the descriptions of the two previously sent vertices, which are temporarily stored in two buffers. The first two vertices are the overhead for each strip, so it is desirable to build long strips, but the automation of this task remains a challenging problem [Evans96].With our assumptions, triangle strips require **461** bits, the product of the following terms:

- 2 triangles per vertex
- 1.2 vertex uses per triangle
- 2 vectors (vertex location and normal) per vertex use
- 3 coordinates per vector
- 4 bytes per coordinate
- 8 bits per byte

The absence of the swap operation in the OpenGL further increases this vertex-use redundancy, since the same vertex may be sent several times in a triangle strip to overcome the left-right-left-right patterns of vertices imposed by OpenGL. This latter constraint does not affect the suitability of triangle strips for data compression.

Because on average a vertex is used twice, either as part of the same triangle strip or of two different ones, the use of triangle strips requires sending most vertices multiple times. Deering's **generalized strips** [Deering95] extend the two registers used for OpenGL triangle strips to a 16 registers stack-buffer where previous vertices may be stored for later use. Deering generalizes the triangle strip syntax by providing more general control over how the next vertex is used and by allowing the temporary inclusion of the current vertex in the stack-buffer and the reuse of any one of the 16 vertices of the stack-buffer. One bit per vertex is used to indicate whether the vertex should be pushed onto the stack-buffer. Two bits per triangle are used to indicate how to continue the current strip. One bit per triangle indicates whether the next vertex should be read from the input stream or retrieved from the stack. 4 bits of address are used for randomly selecting a vertex from the stack-buffer, each time an old vertex is reused. Assuming that each vertex is reused only once, the total cost for encoding the connectivity information is: 1+4 bits per vertex plus 2+1 bits per triangle. Assuming 2 triangles per vertex, this amounts to 11 bits per vertex. Algorithms for systematically creating good traversals of general meshes using Deering's generalized triangle mesh syntax are not available and naive traversal of arbitrary meshes may result in many isolated triangles or

small runs, implying that a significant portion (for example 20%) of the vertices will be sent more than once, and hence increasing the number of incidence bits per vertex to **13.** Deering also proposed to use coordinate and normal quantization and entropy encoding on the vertex coordinates expressed in a local coordinate system and on the normal coordinates to reduce the geometric and photometric storage costs. Basically, a minimax box around the mesh is used to define an optimal resolution coordinate system for the desired number of bits.
This quantization (which results from the integer rounding of the vertex coordinates to the units of the local coordinate system) is a lossy compression step. Deering then further compresses the coordinates by using an optimal variable length coding for the most frequent values of the discrete vertex coordinates and normal parameters. Because lossy compression schemes are difficult to compare, we will assume that a vertex-normal pair can be encoded using **48** bits per vertex for the geometry and photometry. Deering reports about 36 bits for the geometry alone. This yields a total of **61** bits per vertex.

Hoppe's **Progressive Meshes** [Hoppe96] define each vertex the identification of 2 incident edges in the previously constructed mesh, and a displacement vector for computing the location of the new vertex from the location of the common vertex to these two edges The construction is illustrated in Figure 5. Given that there are 3 times more edges than vertices, an index for identifying the first edge takes 18 bits. Given that there are on average 10 edges adjacent to an edge, we need 4 extra bits to identify the second edge The total connectivity storage cost per vertex would then be 22 bits. An entropy coding could further reduce this cost. Hoppe actually uses a vertex index to identify the shared vertex (which would require 16 bits in our example) and a combined 5 bit index identifying the two incident edges, resulting in **21** bits per vertex for the incidence.
Because Hoppe's corrective vectors are in general shorter than Deering's relative vectors, a Huffman coding should further improve the geometric and photometric compression. For simplicity, we will use the same cost for the geometry and photometry of **48** bits per vertex. The total reaches **69** bits per vertex, but the advantage of Hoppe's scheme lies not in its compression benefits, but in its suitability for scaleable model transmission, where a rough model is sent first and then a sequence of vertex insertion operations is sent to progressively refine the model. A progressive model is generated by storing in inverse order a sequence of simplifying edge collapses.

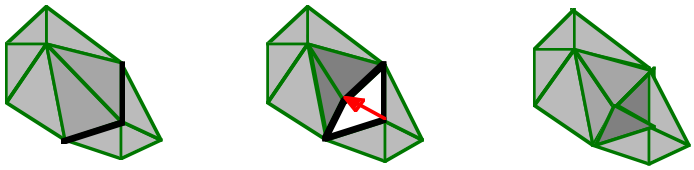


Figure 5: Two adjacent edges are identified (left). These edges are duplicated to produce a cut. The replicated instance of their common vertex is displaced the specified displacement vector (center). This creates a gap, which implicitly defines two new triangles.

Under the assumption that all vertex coordinates are available for random access during decompression, Taubin and Rossignac's **Topological Surgery** method [Taubin96] yields results comparable to Hoppe's Progressive Meshes for the geometric and photometric compression (**48** bits per vertex) and improves on Deering's approach for coding the incidence information by a factor of 4 (about between **2** and **4** bits, depending on the incidence graph). This yields a total of **52** bits. The approach is reviewed in more details in this report.

Estimating simplification errors

A simplification process takes a triangulated surface S and produces a simplified triangulated surface S'. The error that may be perceived when using S' instead of S in a scene depends on the shape of S and S' and also on the viewing conditions (size and orientation of the shape on the screen). It is vital to have a precise error evaluation or at least a tight and guaranteed upper bound on the error, otherwise, simplification schemes may produce simplified models that are not suitable substitutes for the original shape. The error bound will guide our selection of the appropriate level of detail that meets the desired graphics fidelity.

Although the image is defined only by the color components displayed at each pixel, it is useful to distinguish two types of errors: **geometric errors** (how far are the pixels from where they should be) and **color errors** (how do the pixels covered by the same feature of the shape differ in color), when evaluating the accuracy of a simplification technique. These two errors are discussed below.

Since, in general, we cannot predict the viewing conditions, a view-independent error estimation is usually computed during simplification and used during rendering to guide the choice of level of detail. To make this possible, one needs to compute an view-independent error representation that leads to a simple and efficient estimation of the corresponding view-dependent error. For example, if the a point on the surface was displaced

along the normal to that surface by a vector **D**, the perceived error will depend on the projected size of **D** on the screen and on the resulting side effects (for instance the changes in the orientation of the surrounding faces). Hence, the worst viewing conditions for the geometric error may occur when the vector **D** is orthogonal to the viewing direction, while the worst viewing condition for the color error depends on the relative orientation of the light sources, but will typically be proportional to the area occupied on the screen by the faces whose shape was altered by the displacement of the point. On average, this area is the largest when **D** is parallel to the viewing direction. From this example, it should be clear that it is rather difficult to provide a tight bound on the color error and that it is useful to consider separately the tasks of estimating the two errors.

Hausdorff estimate of the geometric error

Several simplification techniques reported in the literature fail to provide a proper error bound and use instead heuristic estimates to guide the simplification process or the selection of the level of detail during rendering. Such strategies often results in simpler algorithms and sometimes to faster processing, and lead to impressive demonstrations. They may be well suited for many entertainment applications, but may not be appropriate for professional applications, where unexpectedly high errors may mislead the users.

Several techniques have been proposed for computing a provable upper bound on the geometric error. They cover a wide spectrum of compromises between the simplicity of the calculations, the performance of the preprocessing steps, and the tightness of the bound.

The geometric error may be measured using the Hausdorff distance between S and S', defined as $\mathbf{H}(S,S')=\mathbf{max}(\mathbf{dev}(S,S'),\mathbf{dev}(S'S))$, where $\mathbf{dev}(A,B)=\mathbf{max}(\mathbf{dist}(a,B))$ for $a \in A$, and $\mathbf{dist}(a,B)=\mathbf{min}(||a=b||)$ for $b \in B$. It measures the worst case distance that a point on one surfaces would have to travel to reach the other surface. Note that **H** is a symmetric version (i.e, $\mathbf{H}(A,B)=\mathbf{H}(B,A)$) of the deviation, **dev**(A,B), of A from B, and is not to be confused with the minimum distance, **dist**(A,B), between the two surfaces, as illustrated in Figure 6.

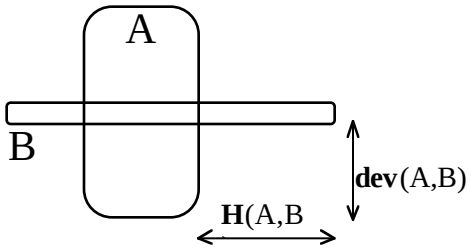


Figure 6: In this example, **dist**(A,B)=0 because the two sets intersect. The Hausdorff distance **H**(A,B) between the two sets is equal to the deviation, **dev**(B,A), but not to the deviation **dev**(A,B).

The Hausdorff distance provides a bound on the maximal geometric deviation between the two shapes and hence provides the maximum distance between a point on the profile (i.e. visible silhouette) of one object and the profile of another. We use the term profile to refer to visible edges that are adjacent to at least one front facing and one back facing face. Note that although the profile is a notion that depends on the view, the Hausdorff distance is not. The Hausdorff distance captures the worst case situation, where the line joining a point on one surface and its closest counterpart on the other surface is orthogonal to the viewing direction.

Although the Hausdorff distance of zero between two sets indicates that the sets are identical, in general, the Hausdorff distance is not a good measure of shape similarity, nor of proximity of surface orientation between two shapes (see Fig. 7 for a counterexample).

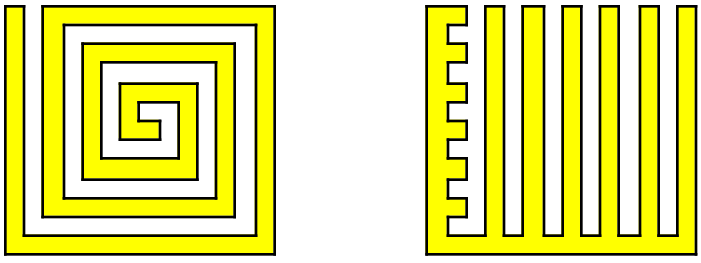


Figure 7: When translated for perfect alignment, these two shapes will have a relatively small Hausdorff distance, yet they differ significantly in their shape and structure.

The cost of computing the Hausdorff distance between two polyhedra is significant, because it does not suffice to check the distances between all the vertices of one set and the other set. One must also be able to detect configurations where the Hausdorff distance is realized at points that lie in the middle of faces. Figure 8 illustrates this concept in two dimensions. Consequently, simpler, although less precise bounds are often used.

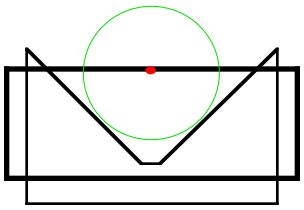


Figure 8: The Hausdorff distance between the two polygons is realized at a point on and edge, not at a vertex.

A geometric error bound based on vertex displacement

Because triangles are linear convex combinations of vertices, if no vertex of a triangle **T** has moved by more than a distance **D** then no interior point in **T** has either. Consequently, if we keep track of how much we move the original vertices, we will have an upper bound on the total error. Both vertex clustering and edge collapses may be viewed as a two step process:

1. Move selected vertices without changing the incidence graph,
2. Change the incidence graph to remove degenerate triangles without changing the geometry.

Voxels or octrees that contain the boundary may also be used to define tolerance zones for safe simplification steps within a prescribed tolerance [Airey90].

When the vertices of a cluster are collapsed into a single representative vertex, the maximum error is the **maximum distance** between the original vertices of the cluster and their representative vertex. This computation requires only one pass through the vertices of the cluster after the representative vertex was chosen. An even less expensive although more pessimistic error bound may be obtained by using half the length of the diagonal of a cell used for vertex quantization.

Although the same approach may be used for the first round of edge collapses, as clusters of vertices resulting from an previous edge collapses are merged into larger clusters, it becomes more expensive to keep track of all the vertices and to check their distance from the representative vertex. A pessimistic, yet simple estimate would be to use the maximum of the sum of all the length of the path from the representative vertex to the leaves of an edge collapse hierarchy.

A geometric error bound based on distances to supporting planes

Vertex displacement provides a rather pessimistic error bound. Imagine for example a simplification of a dense triangulation of a flat portion of a terrain. Although vertices may move during edge collapse or vertex clustering operations, the represented geometry remains flat, and therefore the error is zero. Ronfard and Rossignac have introduced an error estimator which measures the displacement of vertices in the direction orthogonal to their incident faces [Ronfard96]. 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 vertices, as shown in Figure 9. In the cases of sharp corners, the authors introduce an additional plane which suffices to guarantee a precise upper bound on the error.

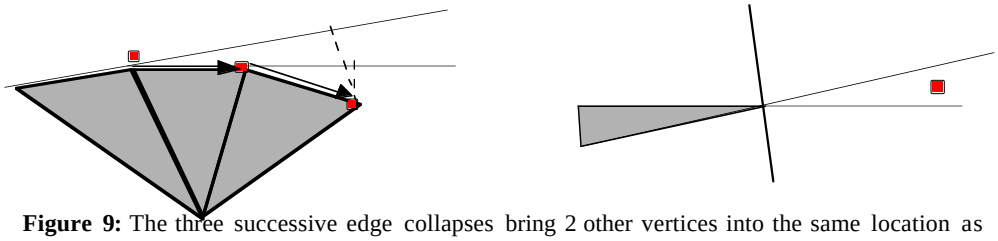


Figure 9: The three successive edge collapses bring 2 other vertices into the same location as a third vertex (marked left). The error is estimated by computing the maximum deviation of these three to the set of lines that support their incident edges. However, in sharp corners, the lines are almost parallel and the vertex could move far before the distance to the supporting lines for the incident edges in its initial position exceed the allowed threshold (right). Therefore, an additional line is introduced to limit this excursion.

Minimizing color errors

The color error is somewhat more subjective, because it requires computing distances in color space, and also harder to control, because slight variation in face orientations may result in considerable variation in specular reflections. Such color errors are best addressed by using photometric properties that are not affected by surface orientation, such as texture maps.

Note that using for the simplified model the normals of the original model may produce excellent results for faces that are roughly orthogonal to the viewing directions, but may produced undesirable black spots in situations where the face is still facing the viewer, but the normal inherited from the original model is not, end hence leads to a black color for the corresponding vertex.

Rossignac and Borrel's vertex quantization

The geometric simplification introduced in [Rossignac93] 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 used [Ronfard94]. 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 [Rossignac93] 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. The approach of [Rossignac93] has been re-engineered at IBM Research by Josh Mittleman and included in IBM's 3D Interaction Accelerator system [3DIX]. The implementation uses a few simple data-structures and sorting to achieve **O(V)** space and **O(V log V**) time complexities for a solid containing **V** vertices.

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.

This approach was recently improved by Low and Tan [Low97], who suggest:

- a better grading of vertices (use **cos(a/2)** rather than **1/a**, where **a** is maximum angle between all pairs of incident edges),
- a floating-cell clustering where the highest weight vertex in a cell attracts vertices in its vicinity from immediately adjacent cells,
- shading edges that approximate elongated objects.

Ronfard and Rossignac's edge collapsing

The principle of the edge-collapsing approaches is to iteratively collapse pairs of vertices that are connected by an edge of the polyhedron into a single new vertex that may be positioned at one of the original two vertices or in a new position, so as to minimize the error resulting from the transformation. The main differences between the various approaches lie in the techniques for estimating an error bound associated with each candidate edge and the optimization criteria for positioning the new vertex.

The technique developed by Ronfard and Rossignac at IBM Research [Ronfard96] associates with each vertex a compact description of the planes that support each of the incident triangles and possibly additional planes through sharp incident edges. These are used to define and evaluate approximation constraints. The user may wish for example to ensure that no vertex moves further away from each one of these planes than a prescribed distance. The maximum distance between a point and these planes is used as an error bound estimator. The advantage of this criteria for error estimation lies in the fact that it enables vertices to travel far away from their original location along nearly planar regions.

Initially, an error estimate is computed for each edge collapsing operation and the edges are sorted in a priority queue. At each step, the best edge is chosen, so as to minimize the total error estimate. When two vertex clusters are merged, their constraints (lists of planes) are merged and possibly pruned to keep the lists short.

Each collapsing operation only alters immediate neighbors and therefore, the a new error bound for each one of the neighboring edges may be quickly estimated and the priority queue updated.

Advantages and drawbacks

The method guarantees a tight error bound and permits to move vertices further away from their original location as long as they move along nearly parallel faces. Consequently, it produces much lower triangle counts than the vertex clustering method for the same error bound.

However, this method requires maintaining a face-edge-vertex incidence graph and special treatment to allow topological changes, such as the elimination of collapsed holes (see also [He96]).

Gueziec's volume preserving simplification

Gueziec's algorithm, developed at IBM Research, preserves the volume of the original polyhedron during simplification [Gueziec96], It favors the creation of near-equilateral triangles. Gueziec computes an upper bound to the approximation error and reports this bound using a novel tool, the error volume, which is constructed by taking the union of balls centered on the surface, whose radii, the error values vary linearly between surface vertices. Gueziec guarantees that the error will be less than user specified tolerances, which can vary across the surface, as opposed to a single tolerance. An originality of his method is that errors and tolerances are defined with respect to the simplified surface, as opposed to the original surface.

Following the approach of [Ronfard96], the algorithm uses a greedy strategy based upon the edge collapsing operation. Edges are weighted and sorted in a priority queue. Before collapsing the edge with the lowest weight into a simplified vertex, tests determine whether the simplification is appropriate. The simplified vertex is positioned such that the volume enclosed by a closed surface will stay the same. Gueziec shows that the simplified vertex must lie on a specific plane. On that plane, the vertex position minimizes the sum of squared distances to the planes of the edge star, which corresponds to minimizing a sum of distances to lines in the plane. Once the optimum position of the simplified vertex is found, Gueziec verifies that the triangle orientations are not perturbed by a rotation exceeding a user specification. Also, he verifies that the minimum value for the triangle aspect ratios, as measured with the triangle compactness or ratio between area and perimeter, is not degraded in excess of a pre-specified factor.

Gueziec also determines the effect of the edge collapse on the overall approximation error, and tests whether it is tolerable. The error volume is initialized with error values equal to zero, or with positive error values reported by a previous simplification process. As the simplification progresses, the error values at the remaining vertices are gradually updated. In a manner similar to Russian Dolls, a hierarchy of nested error volumes is built such that each new volume is guaranteed to enclose the previous error volume. A linear optimization is performed to compute error estimates that minimize their overall error volume. The simplification stops in a particular region when the width of the error volume reaches the tolerance. Assuming that a bound to the maximum valence at a vertex is respected, Gueziec shows that the overall computational complexity of his method is sub-quadratic in the number of edges of the surface.

The algorithm has the following limitations: Since it deals with the problem of finding a sub optimal approximation with a given error bound, it cannot guarantee that a given triangle reduction can be obtained, nor can it guarantee that a it reaches the minimum number of triangles. The current implementation is computationally intensive, mainly due to a linear programming step used for maintaining dynamically the error volume. Finally, although a pre-processing step could assign vertex tolerances that would prevent surface self-intersections, the current program allows them where tolerance volumes overlap. The algorithm assumes that the topology of the input surface is that of a manifold; this property is then also verified for the output surface.

Kalvin and Taylor's face-merging

Although it is domain independent, the Superfaces algorithm has been originally developed by Alan Kalvin and Russ Taylor at IBM Research for simplifying polyhedral meshes [Kalvin96] that result from building iso-surfaces of medical data-sets. The simplification is based on a bounded approximation criterion and produces a simplified mesh that approximates the original one to within a pre-specified tolerance. The vertices in the simplified mesh are a proper subset of the original vertices, so the algorithm is well-suited for creating hierarchical representations of polyhedra.

The algorithm simplifies a mesh in three phases:

- Superface creation: A "greedy", bottom-up face-merging procedure partitions the original faces into superface patches.
- Border Straightening: The borders of the superfaces are simplified, by merging boundary edges into superedges,
- Superface Triangulation: Triangulation points for the superfaces are defined. In this phase, a single superface may be decomposed into many superfaces, each with its own boundary and triangulation point.

Advantages and drawbacks

The Superfaces algorithm has the following advantages:

- It uses a bounded approximation approach which guarantees that a simplified mesh approximates the original mesh to within a pre-specified tolerance. That is, every vertex **v** in the original mesh is guaranteed to lie within a user-specified distance of the simplified mesh. Also, the vertices in the simplified mesh are a subset of the original vertices, so there is zero error distance between the simplified vertices and the original surface.
- It is fast. The face-merging procedure is efficient and greedy-- that is, it does not backtrack or undo any merging once completed. Thus the algorithm is practical for simplifying very large meshes.
- It is a general-purpose, domain-independent method.

Disadvantages of the Superfaces algorithm are:

- It can produce simplified surfaces that self-intersect.
- The current triangulation procedure (that uses no Steiner points) can produce skinny triangles, that are not desirable for rendering.

Taubin and Rossignac's 3D compression by topological surgery

The Topological surgery method for compressing the incidence relation is based on the following observations:

- Given the simply connected polygon that bounds a triangle strip and the selection of a starting seed edge, the internal triangulation of the strip may be encoded using **one** bit per triangle. This bit simply states whether one should advance the left or the right vertex of a progressing edge that sweeps the entire strip starting at the seed edge and moving its vertices one at a time along the boundary of the strip. (Note that if the strip was a regular alternations of left and right moves, we would need zero bits.)
- Given a set of non-overlapping triangle strips that cover the surface of a polyhedron and union of their boundaries identifies a set of "cut" edges. Each cut edge is used exactly twice in the boundary of the same or of different strips.
- A vertex spanning tree formed by selecting the minimum number of edges of the polyhedron that connect all the vertices but do not create loops cuts the boundary a single shell, genus zero, manifold polyhedron into a simply connected topological polygon that may be decomposed into the union of triangle strips. These strips and their adjacency may be encoded as a binary spanning tree of the triangles of the mesh.
- Cutting strategies that produce vertex and triangle spanning trees with few nodes that have more than one child lead to very efficient schemes for compressing the trees. Although encoding general trees and graphs is much more expensive [Jacobson89], the expected cost of our encoding is less than a bit per vertex. One basically needs to encode the length of a run of consecutive nodes that have a single child and the overall structure of the tree (which requires 2 bits for each node with more than one child.)

- The handling of non-manifold polyhedra and of meshes with higher genus is possible through simple extensions of this approach.
- The order imposed on the vertices by the vertex spanning tree may be exploited for computing good estimates for the location of the next vertex from the location of its 3 or 4 ancestors in the tree. Entropy coding the differences between the actual location and the estimate results in better compression for the geometric information than coding the absolute coordinates, because the coordinates of the difference vector are smaller on average.

The details of this approach may be found in [Taubin96].

Turan has shown that the incidence of a simple mesh can be encoded using **12** bits per vertex [Turan84]. However, Turan's study focused on the problem of triangulating a labeled graph. Taubin and Rossignac use the order of the vertices (i.e. a permutation on the vertex labels) to capture some of the incidence relation, and hence are able to reduce further improve the compression by a factor of 3.

Scaleable models

In order to support remote viewing of highly complex models, the user must be able to download a low resolution model and start using it while more detailed model description is still being transferred. Hoppe's progressive meshes [Hoppe96], discussed earlier, are suitable for such an adaptive scheme.

Furthermore, to accelerate graphics, it is important to transfer the different parts of the scene at different resolutions (models close to the viewer should receive more details early on, while distant models may never need to be displayed I full detail. Although the progressive mesh representation supports multi-resolution adaptive simplification, the constraints imposed on the order in which vertices may be inserted make it ill suited for adaptive LOD generation. Indeed, the system may often be forced to expand distant edges simply to create vertices whose descendants are needed to expand nearby edges. A more localized approach may be more effective.

Conclusion

Although the performance of state of the art hardware graphics and the internet bandwidth are growing rapidly, price constraints and the growing needs of industrial customers call for algorithmic solutions that improve the speed of visualizing and transmitting complex 3D scenes. We have presented several techniques, which automatically computes one or several simplified graphics representations of each object. These representations may be used selectively in lieu of the original model to accelerate the display process while preserving the overall perceptual information content of the scene. The described methods clusters vertices of the model and produces an approximate model where original faces are approximated with fewer faces defined in terms of selected vertices. Several simplified representations with different simplification factors may be stored in addition to the original model. Actual viewing conditions are used to establish automatically for each object which representation should be used for graphics. We have also reviewed several techniques for compressing the geometry, the incidence, and the photometry of a triangulated model. The best results yield a 20:1 compression ratio over naive representation schemes.

Bibliography

[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.

[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.

[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.

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

[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.

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

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

[DeFloriani92] L. De Floriani, 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.

[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.

[Funkhouser93] 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.

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

[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,
\URL{http://www.cs.cmu.edu/~garland/scape, Sept.1995.

[Greene93] N, Greene, M, Kass, and G, Miller, Hierarchical z-buffer visibility, ACM SIGGRAPH'93 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.

[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.

[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.

[Hoppe93] H, Hoppe, T, DeRose, T, Duchamp, J, McDonald, and W, Stuetzle, Mesh optimization, Proceedings SIGGRAPH'93, pp:19-26, August 1993.

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

[Hoppe97] H, Hoppe, Progressive Simplicial Complexes, 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.

[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] AD, Kalvin, RH, Taylor, Superfaces: Polyhedral Approximation with Bounded Error, IEEE Computer Graphics \& Applications, 16(3):64-77, May 1996.

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

[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.

[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.

[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.

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

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

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

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

[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] R. Ronfard and J, Rossignac, Full-range approximation of triangulated polyhedra, to appear in Proc. Eurographics'96 and in Computer Graphics Forum. IBM Research Report RC20432, 4/2/96.

[Rossignac93] 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] 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.

[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.

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

[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).

[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.

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

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

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

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

[3DIX] IBM 3D Interaction Accelerator, Product Description, http://www.research.ibm.com/3dix. 1996.

Mesh Optimization

Hugues Hoppe* Tony DeRose* Tom Duchamp†
John McDonald‡ Werner Stuetzle‡

University of Washington
Seattle, WA 98195

Abstract

We present a method for solving the following problem: Given a set of data points scattered in three dimensions and an initial triangular mesh M_0, produce a mesh M, of the same topological type as M_0, that fits the data well and has a small number of vertices. Our approach is to minimize an energy function that explicitly models the competing desires of conciseness of representation and fidelity to the data. We show that mesh optimization can be effectively used in at least two applications: surface reconstruction from unorganized points, and mesh simplification (the reduction of the number of vertices in an initially dense mesh of triangles).

CR Categories and Subject Descriptors: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling.

Additional Keywords: Geometric Modeling, Surface Fitting, Three-Dimensional Shape Recovery, Range Data Analysis, Model Simplification.

1 Introduction

The *mesh optimization* problem considered in this paper can be roughly stated as follows: Given a collection of data points X in $\mathbf{R}^3$ and an initial triangular mesh M_0 near the data, find a mesh M of the same topological type as M_0 that fits the data well and has a small number of vertices.

As an example, Figure 7b shows a set of 4102 data points sampled from the object shown in Figure 7a. The input to the mesh optimization algorithm consists of the points together with the initial mesh shown in Figure 7c. The optimized mesh is shown in Figure 7h. Notice that the sharp edges and corners indicated by the data have been faithfully recovered and that the number of vertices has been significantly reduced (from 1572 to 163).

*Department of Computer Science and Engineering, FR-35
†Department of Mathematics, GN-50
‡Department of Statistics, GN-22

This work was supported in part by Bellcore, the Xerox Corporation, IBM, Hewlett-Packard, AT&T Bell Labs, the Digital Equipment Corporation, the Department of Energy under grant DE-FG06-85-ER25006, the National Library of Medicine under grant NIH LM-04174, and the National Science Foundation under grants CCR-8957323 and DMS-9103002.

To solve the mesh optimization problem we minimize an *energy function* that captures the competing desires of tight geometric fit and compact representation. The tradeoff between geometric fit and compact representation is controlled via a user-selectable parameter c_{rep}. A large value of c_{rep} indicates that a sparse representation is to be strongly preferred over a dense one, usually at the expense of degrading the fit.

We use the input mesh M_0 as a starting point for a non-linear optimization process. During the optimization we vary the number of vertices, their positions, and their connectivity. Although we can give no guarantee of finding a global minimum, we have run the method on a wide variety of data sets; the method has produced good results in all cases (see Figure 1).

We see at least two applications of mesh optimization: surface reconstruction and mesh simplification.

The problem of surface reconstruction from sampled data occurs in many scientific and engineering applications. In [2], we outlined a two phase procedure for reconstructing a surface from a set of unorganized data points. The goal of phase one is to determine the topological type of the unknown surface and to obtain a crude estimate of its geometry. An algorithm for phase one was described in [5]. The goal of phase two is to improve the fit and reduce the number of faces. Mesh optimization can be used for this purpose.

Although we were originally led to consider the mesh optimization problem by our research on surface reconstruction, the algorithm we have developed can also be applied to the problem of mesh simplification. Mesh simplification, as considered by Turk [15] and Schroeder et al. [10], refers to the problem of reducing the number of faces in a dense mesh while minimally perturbing the shape. Mesh optimization can be used to solve this problem as follows: sample data points X from the initial mesh and use the initial mesh as the starting point M_0 of the optimization procedure. For instance, Figure 7q shows a triangular approximation of a minimal surface with 2032 vertices. Application of our mesh optimization algorithm to a sample of 6752 points (Figure 7r) from this mesh produces the meshes shown in Figures 7s (487 vertices) and 7t (239 vertices). The mesh of Figure 7s corresponds to a relatively small value of c_{rep}, and therefore has more vertices than the mesh of Figure 7t which corresponds to a somewhat larger value of c_{rep}.

The principal contributions of this paper are:

- It presents an algorithm for fitting a mesh of arbitrary topological type to a set of data points (as opposed to volume data, etc.). During the fitting process, the number and connectivity of the vertices, as well as their positions, are allowed to vary.
- It casts mesh simplification as an optimization problem with an energy function that directly measures deviation of the final mesh from the original. As a consequence, the final mesh

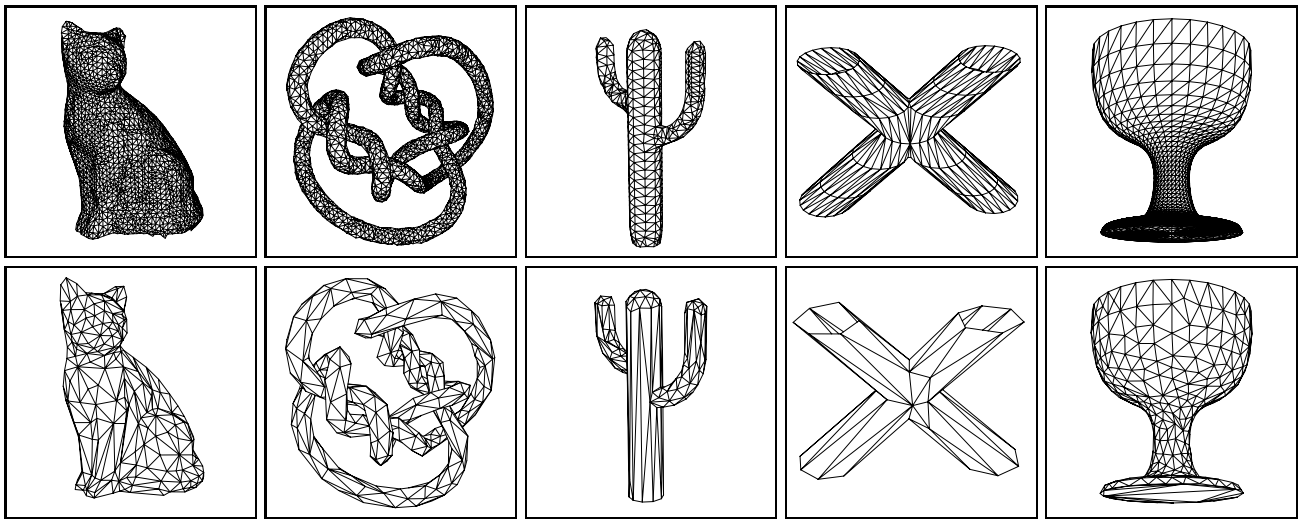


Figure 1: Examples of mesh optimization. The meshes in the top row are the initial meshes M_0; the meshes in the bottom row are the corresponding optimized meshes. The first 3 columns are reconstructions; the last 2 columns are simplifications.

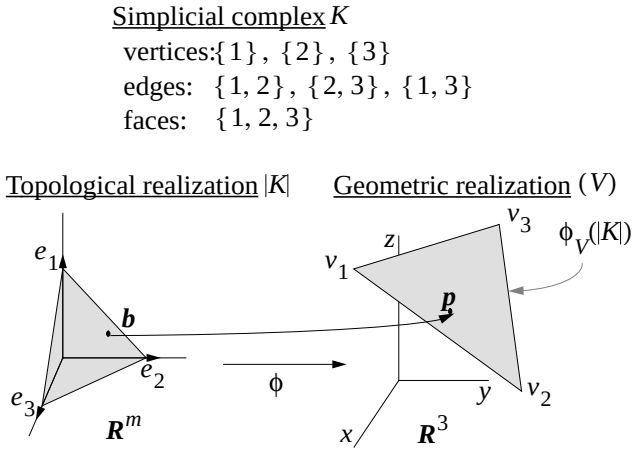


Figure 2: Example of mesh representation: a mesh consisting of a single face.

naturally adapts to curvature variations in the original mesh.

- It demonstrates how the algorithm's ability to recover sharp edges and corners can be exploited to automatically segment the final mesh into smooth connected components (see Figure 7i).

2 Mesh Representation

Intuitively, a *mesh* is a piecewise linear surface, consisting of triangular faces pasted together along their edges. For our purposes it is important to maintain the distinction between the connectivity of the vertices and their geometric positions. Formally, a mesh M is a pair (K, V), where: K is a *simplicial complex* representing the connectivity of the vertices, edges, and faces, thus determining the topological type of the mesh; $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$, $\mathbf{v}_i \in \mathbf{R}^3$ is a set of vertex positions defining the shape of the mesh in $\mathbf{R}^3$ (its geometric realization).

A simplicial complex K consists of a set of vertices $\{1, \ldots, m\}$, together with a set of non-empty subsets of the vertices, called the simplices of K, such that any set consisting of exactly one vertex is a simplex in K, and every non-empty subset of a simplex in K is again a simplex in K (cf. Spanier [14]). The 0-simplices $\{i\} \in K$ are called vertices, the 1-simplices $\{i, j\} \in K$ are called edges, and the 2-simplices $\{i, j, k\} \in K$ are called faces.

A geometric realization of a mesh as a surface in $\mathbf{R}^3$ can be obtained as follows. For a given simplicial complex K, form its *topological realization* $|K|$ in $\mathbf{R}^m$ by identifying the vertices $\{1, \ldots, m\}$ with the standard basis vectors $\{\mathbf{e}_1, \ldots, \mathbf{e}_m\}$ of $\mathbf{R}^m$. For each simplex $s \in K$ let $|s|$ denote the convex hull of its vertices in $\mathbf{R}^m$, and let $|K| = \cup_{s \in K} |s|$. Let $\phi : \mathbf{R}^m \to \mathbf{R}^3$ be the linear map that sends the i-th standard basis vector $\mathbf{e}_i \in \mathbf{R}^m$ to $\mathbf{v}_i \in \mathbf{R}^3$ (see Figure 2).

The *geometric realization* of M is the image $\phi_V(|K|)$, where we write the map as ϕ_V to emphasize that it is fully specified by the set of vertex positions $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$. The map ϕ_V is called an *embedding* if it is 1-1, that is if $\phi_V(|K|)$ is not self-intersecting. Only a restricted set of vertex positions V result in ϕ_V being an embedding.

If ϕ_V is an embedding, any point $\mathbf{p} \in \phi_V(|K|)$ can be parameterized by finding its unique pre-image on $|K|$. The vector $\mathbf{b} \in |K|$ with $\mathbf{p} = \phi_V(\mathbf{b})$ is called the *barycentric coordinate vector* of $\mathbf{p}$ (with respect to the simplicial complex K). Note that barycentric coordinate vectors are convex combinations of standard basis vectors $\mathbf{e}_i \in \mathbf{R}^m$ corresponding to the vertices of a face of K. Any barycentric coordinate vector has at most three non-zero entries; it has only two non-zero entries if it lies on an edge of $|K|$, and only one if it is a vertex.

3 Definition of the Energy Function

Recall that the goal of mesh optimization is to obtain a mesh that provides a good fit to the point set X and has a small number of vertices. We find a simplicial complex K and a set of vertex positions V defining a mesh $M = (K, V)$ that minimizes the energy function

$$E(K, V) = E_{dist}(K, V) + E_{rep}(K) + E_{spring}(K, V).$$

The first two terms correspond to the two stated goals; the third term is motivated below.

The distance energy E_{dist} is equal to the sum of squared distances

from the points $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ to the mesh,

$$E_{dist}(K, V) = \sum_{i=1}^{n} d^2(\mathbf{x}_i, \phi_V(|K|)).$$

The representation energy E_{rep} penalizes meshes with a large number of vertices. It is set to be proportional to the number of vertices m of K:

$$E_{rep}(K) = c_{rep} m.$$

The optimization allows vertices to be both added to and removed from the mesh. When a vertex is added, the distance energy E_{dist} is likely to be reduced; the term E_{rep} makes this operation incur a penalty so that vertices are not added indefinitely. Similarly, one wants to remove vertices from a dense mesh even if E_{dist} increases slightly; in this case E_{rep} acts to encourage the vertex removal. The user-specified parameter c_{rep} provides a controllable trade-off between fidelity of geometric fit and parsimony of representation.

We discovered, as others have before us [8], that minimizing $E_{dist} + E_{rep}$ does not produce the desired results. As an illustration of what can go wrong, Figure 7d shows the result of minimizing E_{dist} alone. The estimated surface has several spikes in regions where there is no data. These spikes are a manifestation of the fundamental problem that a minimum of $E_{dist} + E_{rep}$ may not exist.

To guarantee the existence of a minimum [6], we add the third term, the spring energy E_{spring}. It places on each edge of the mesh a spring of rest length zero and spring constant κ:

$$E_{spring}(K, V) = \sum_{\{j,k\} \in K} \kappa \|\mathbf{v}_j - \mathbf{v}_k\|^2$$

It is worthwhile emphasizing that the spring energy is not a smoothness penalty. Our intent is not to penalize sharp dihedral angles in the mesh, since such features may be present in the underlying surface and should be recovered. We view E_{spring} as a regularizing term that helps guide the optimization to a desirable local minimum. As the optimization converges to the solution, the magnitude of E_{spring} can be gradually reduced. We return to this issue in Section 4.4.

For some applications we want the procedure to be scale-invariant, which is equivalent to defining a unitless energy function E. To achieve invariance under Euclidean motion and uniform scaling, the points X and the initial mesh M_0 are pre-scaled uniformly to fit in a unit cube. After optimization, a post-processing step can undo this initial transformation.

4 Minimization of the Energy Function

Our goal is to minimize the energy function

$$E(K, V) = E_{dist}(K, V) + E_{rep}(K) + E_{spring}(K, V)$$

over the set $\mathcal{K}$ of simplicial complexes K homeomorphic to the initial simplicial complex K_0, and the vertex positions V defining the embedding. We now present an outline of our optimization algorithm, a pseudo-code version of which appears in Figure 3. The details are deferred to the next two subsections.

To minimize $E(K, V)$ over both K and V, we partition the problem into two nested subproblems: an inner minimization over V for fixed simplicial complex K, and an outer minimization over K.

In Section 4.1 we describe an algorithm that solves the inner minimization problem. It finds $E(K) = \min_V E(K, V)$, the energy of the best possible embedding of the fixed simplicial complex K, and the corresponding vertex positions V, given an initial guess for V. This corresponds to the procedure OptimizeVertexPositions in Figure 3.

```
OptimizeMesh(K0,V0) {
    K := K0
    V := OptimizeVertexPositions(K0,V0)
    – Solve the outer minimization problem.
    repeat {
        (K',V') := GenerateLegalMove(K,V)
        V' = OptimizeVertexPositions(K',V')
        if E(K', V') < E(K, V) then
            (K,V) := (K',V')
        endif
    } until convergence
    return (K,V)
}

– Solve the inner optimization problem
–   E(K) = min_V E(K, V)
– for fixed simplicial complex K.
OptimizeVertexPositions(K,V) {
    repeat {
        – Compute barycentric coordinates by projection.
        B := ProjectPoints(K,V)
        – Minimize E(K, V, B) over V using conjugate gradients.
        V := ImproveVertexPositions(K,B)
    } until convergence
    return V
}

GenerateLegalMove(K,V) {
    Select a legal move K ⇒ K'.
    Locally modify V to obtain V' appropriate for K'.
    return (K',V')
}
```

Figure 3: An idealized pseudo-code version of the minimization algorithm.

Whereas the inner minimization is a continuous optimization problem, the outer minimization of $E(K)$ over the simplicial complexes $K \in \mathcal{K}$ (procedure OptimizeMesh) is a discrete optimization problem. An algorithm for its solution is presented in Section 4.2.

The energy function $E(K, V)$ depends on two parameters c_{rep} and κ. The parameter c_{rep} controls the tradeoff between conciseness and fidelity to the data and should be set by the user. The parameter κ, on the other hand, is a regularizing parameter that, ideally, would be chosen automatically. Our method of setting κ is described in Section 4.4.

4.1 Optimization for Fixed Simplicial Complex
(Procedure OptimizeVertexPositions)

In this section, we consider the problem of finding a set of vertex positions V that minimizes the energy function $E(K, V)$ for a given simplicial complex K. As $E_{rep}(K)$ does not depend on V, this amounts to minimizing $E_{dist}(K, V) + E_{spring}(K, V)$.

To evaluate the distance energy $E_{dist}(K, V)$, it is necessary to compute the distance of each data point $\mathbf{x}_i$ to $M = \phi_V(|K|)$. Each of these distances is itself the solution to the minimization problem

$$d^2(\mathbf{x}_i, \phi_V(|K|)) = \min_{\mathbf{b}_i \in |K|} \|\mathbf{x}_i - \phi_V(\mathbf{b}_i)\|^2,$$

in which the unknown is the barycentric coordinate vector $\mathbf{b}_i \in |K| \subset \mathbf{R}^m$ of the projection of $\mathbf{x}_i$ onto M. Thus, minimizing

$E(K, V)$ for fixed K is equivalent to minimizing the new objective function

$$\begin{aligned} E(K, V, B) &= \sum_{i=1}^{n} \|\mathbf{x}_i - \phi_V(\mathbf{b}_i)\|^2 + E_{spring}(K, V) \\ &= \sum_{i=1}^{n} \|\mathbf{x}_i - \phi_V(\mathbf{b}_i)\|^2 + \sum_{\{j,k\} \in K} \kappa \|\mathbf{v}_j - \mathbf{v}_k\|^2 \end{aligned}$$

over the vertex positions $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$, $\mathbf{v}_i \in \mathbf{R}^3$ and the barycentric coordinates $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, $\mathbf{b}_i \in |K| \subset \mathbf{R}^m$.

To solve this optimization problem (procedure OptimizeVertexPositions), our method alternates between two subproblems:

1. For fixed vertex positions V, find optimal barycentric coordinate vectors B by *projection* (procedure ProjectPoints).
2. For fixed barycentric coordinate vectors B, find optimal vertex positions V by solving a *linear* least squares problem (procedure ImproveVertexPositions).

Because we find optimal solutions to both of these subproblems, $E(K, V, B)$ can never increase, and since it is bounded from below, it must converge. In principle, one could iterate until some formal convergence criterion is met. Instead, as is common, we perform a fixed number of iterations. As an example, Figure 7e shows the result of optimizing the mesh of Figure 7c over the vertex positions while holding the simplicial complex fixed.

It is conceivable that procedure OptimizeVertexPositions returns a set V of vertices for which the mesh is self-intersecting, i.e. ϕ_V is not an embedding. While it is possible to check *a posteriori* whether ϕ_V is an embedding, constraining the optimization to always produce an embedding appears to be difficult. This has not presented a problem in the examples we have run.

4.1.1 Projection Subproblem
(Procedure ProjectPoints)

The problem of optimizing $E(K, V, B)$ over the barycentric coordinate vectors $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, while holding the vertex positions $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ and the simplicial complex K constant, decomposes into n separate optimization problems:

$$\mathbf{b}_i = \operatorname*{argmin}_{\mathbf{b} \in |K|} \|\mathbf{x}_i - \phi_V(\mathbf{b})\|$$

In other words, $\mathbf{b}_i$ is the barycentric coordinate vector corresponding to the point $\mathbf{p} \in \phi_V(|K|)$ closest to $\mathbf{x}_i$.

A naive approach to computing $\mathbf{b}_i$ is to project $\mathbf{x}_i$ onto all of the faces of M, and then find the projection with minimal distance. To speed up the projection, we first enter the faces of the mesh into a spatial partitioning data structure (similar to the one used in [16]). Then for each point $\mathbf{x}_i$ only a nearby subset of the faces needs to be considered, and the projection step takes expected time $O(n)$. For additional speedup we exploit coherence between iterations. Instead of projecting each point globally onto the mesh, we assume that a point's projection lies in a neighborhood of its projection in the previous iteration. Specifically, we project the point onto all faces that share a vertex with the previous face. Although this is a heuristic that can fail, it has performed well in practice.

4.1.2 Linear Least Squares Subproblem
(Procedure ImproveVertexPositions)

Minimizing $E(K, V, B)$ over the vertex positions V while holding B and K fixed is a linear least squares problem. It decomposes into three independent subproblems, one for each of the three coordinates of the vertex positions. We will write down the problem for the first coordinate.

Let e be the number of edges (1-simplices) in K; note that e is $O(m)$. Let $\mathbf{v}^1$ be the m-vector whose i-th element is the first coordinate of $\mathbf{v}_i$. Let $\mathbf{d}^1$ be the $(n+e)$-vector whose first n elements are the first coordinates of the data points $\mathbf{x}_i$, and whose last e elements are zero. With these definitions we can express the least squares problem for the first coordinate as minimizing $\|A\mathbf{v}^1 - \mathbf{d}^1\|^2$ over $\mathbf{v}^1$. The design matrix A is an $(n+e) \times m$ matrix of scalars. The first n rows of A are the barycentric coordinate vectors $\mathbf{b}_i$. Each of the trailing e rows contains 2 non-zero entries with values $\sqrt{\kappa}$ and $-\sqrt{\kappa}$ in the columns corresponding to the indices of the edge's endpoints. The first n rows of the least squares problem correspond to $E_{dist}(K, V)$, while the last e rows correspond to $E_{spring}(K, V)$. An important feature of the matrix A is that it contains at most 3 non-zero entries in each row, for a total of $O(n + m)$ non-zero entries.

To solve the least squares problem, we use the conjugate gradient method (cf. [3]). This is an iterative method guaranteed to find the exact solution in as many iterations as there are distinct singular values of A, i.e. in at most m iterations. Usually far fewer iterations are required to get a result with acceptable precision. For example, we find that for m as large as 10^4, as few as 200 iterations are sufficient.

The two time-consuming operations in each iteration of the conjugate gradient algorithm are the multiplication of A by an $(n+e)$-vector and the multiplication of A^T by an m-vector. Because A is sparse, these two operations can be executed in $O(n + m)$ time. We store A in a sparse form that requires only $O(n + m)$ space. Thus, an acceptable solution to the least squares problem is obtained in $O(n+m)$ time. In contrast, a typical noniterative method for solving dense least squares problems, such as QR decomposition, would require $O((n + m)m^2)$ time to find an exact solution.

4.2 Optimization over Simplicial Complexes
(Procedure OptimizeMesh)

To solve the outer minimization problem, minimizing $E(K)$ over K, we define a set of three elementary transformations, *edge collapse*, *edge split*, and *edge swap*, taking a simplicial complex K to another simplicial complex K' (see Figure 4).

We define a *legal move* to be the application of one of these elementary transformations to an edge of K that leaves the topological type of K unchanged. The set of elementary transformations is complete in the sense that *any* simplicial complex in $\mathcal{K}$ can be obtained from K_0 through a sequence of legal moves[1].

Our goal then is to find such a sequence taking us from K_0 to a minimum of $E(K)$. We do this using a variant of random descent: we randomly select a legal move, $K \Rightarrow K'$. If $E(K') < E(K)$, we accept the move, otherwise we try again. If a large number of trials fails to produce an acceptable move, we terminate the search.

More elaborate selection strategies, such as steepest descent or simulated annealing, are possible. As we have obtained good results with the simple strategy of random descent, we have not yet implemented the other strategies.

Identifying Legal Moves An edge split transformation is always a legal move, as it can never change the topological type of K. The other two transformations, on the other hand, can cause a change of topological type, so tests must be performed to determine if they are legal moves.

[1] In fact, we prove in [6] that edge collapse and edge split are sufficient; we include edge swap to allow the optimization procedure to "tunnel" through small hills in the energy function.

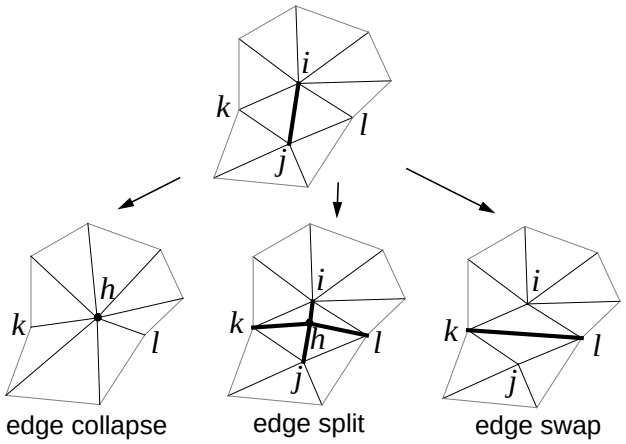


Figure 4: Local simplicial complex transformations

We define an edge $\{i,j\} \in K$ to be a *boundary edge* if it is a subset of only one face $\{i,j,k\} \in K$, and a vertex $\{i\}$ to be a *boundary vertex* if there exists a boundary edge $\{i,j\} \in K$.

An edge collapse transformation $K \Rightarrow K'$ that collapses the edge $\{i,j\} \in K$ is a legal move if and only if the following conditions are satisfied (proof in [6]):

- For all vertices $\{k\}$ adjacent to both $\{i\}$ and $\{j\}$ ($\{i,k\} \in K$ and $\{j,k\} \in K$), $\{i,j,k\}$ is a face of K.
- If $\{i\}$ and $\{j\}$ are both boundary vertices, $\{i,j\}$ is a boundary edge.
- K has more than 4 vertices if neither $\{i\}$ nor $\{j\}$ are boundary vertices, or K has more than 3 vertices if either $\{i\}$ or $\{j\}$ are boundary vertices.

An edge swap transformation $K \Rightarrow K'$ that replaces the edge $\{i,j\} \in K$ with $\{k,l\} \in K'$ is a legal move if and only if $\{k,l\} \notin K$.

4.3 Exploiting Locality

The idealized algorithm described so far is too inefficient to be of practical use. In this section, we describe some heuristics which dramatically reduce the running time. These heuristics capitalize on the fact that a local change in the structure of the mesh leaves the optimal positions of distant vertices essentially unchanged.

4.3.1 Heuristics for Evaluating the Effect of Legal Moves

Our strategy for selecting legal moves requires evaluation of $E(K') = \min_V E(K', V)$ for a simplicial complex K' obtained from K through a legal move. Ideally, we would use procedure OptimizeVertexPositions of Section 4.1 for this purpose, as indicated in Figure 3. In practice, however, this is too slow. Instead, we use fast local heuristics to estimate the effect of a legal move on the energy function.

Each of the heuristics is based on extracting a submesh in the neighborhood of the transformation, along with the subset of the data points projecting onto the submesh. The change in overall energy is estimated by only considering the contribution of the submesh and the corresponding point set. This estimate is always pessimistic, as full optimization would only further reduce the energy. Therefore, the heuristics never suggest changes that will increase the true energy of the mesh.

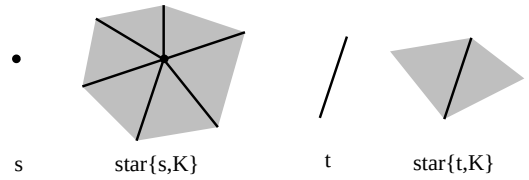


Figure 5: Neighborhood subsets of K.

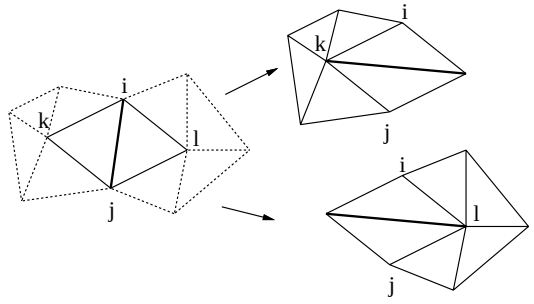


Figure 6: Two local optimizations to evaluate edge swap

Definition of neighborhoods in a simplicial complex To refer to neighborhoods in a simplicial complex, we need to introduce some further notation. We write $s' \leq s$ to denote that simplex s' is a non-empty subset of simplex s. For simplex $s \in K$, $\text{star}(s;K) = \{s' \in K : s \leq s'\}$ (Figure 5).

Evaluation of Edge Collapse To evaluate a transformation $K \Rightarrow K'$ collapsing an edge $\{i,j\}$ into a single vertex $\{h\}$ (Figure 4), we take the submesh to be $\text{star}(\{i\};K) \cup \text{star}(\{j\};K)$, and optimize over the single vertex position $\mathbf{v}_h$ while holding all other vertex positions constant.

Because we perform only a small number of iterations (for reasons of efficiency), the initial choice of $\mathbf{v}_h$ greatly influences the accuracy of the result. Therefore, we attempt three optimizations, with $\mathbf{v}_h$ starting at $\mathbf{v}_i$, $\mathbf{v}_j$, and $\frac{1}{2}(\mathbf{v}_i + \mathbf{v}_j)$, and accept the best one.

The edge collapse should be allowed only if the new mesh does not intersect itself. Checking for this would be costly; instead we settle for a less expensive heuristic check. If, after the local optimization, the maximum dihedral angle of the edges in $\text{star}(\{h\};K')$ is greater than some threshold, the edge collapse is rejected.

Evaluation of Edge Split The procedure is the same as for edge collapse, except that the submesh is defined to be $\text{star}(\{i,j\};K)$, and the initial value of the new vertex $\mathbf{v}_h$ is chosen to be $\frac{1}{2}(\mathbf{v}_i + \mathbf{v}_j)$.

Evaluation of Edge Swap To evaluate an edge swap transformation $K \Rightarrow K'$ that replaces an edge $\{i,j\} \in K$ with $\{k,l\} \in K'$, we consider two local optimizations, one with submesh $\text{star}(\{k\};K')$, varying vertex $\mathbf{v}_k$, and one with submesh $\text{star}(\{l\};K')$, varying vertex $\mathbf{v}_l$ (Figure 6). The change in energy is taken to best of these. As is the case in evaluating an edge collapse, we reject the transformation if the maximum dihedral angle after the local optimization exceeds a threshold.

4.3.2 Legal Move Selection Strategy
(Procedure GenerateLegalMove)

The simple strategy for selecting legal moves described in Section 4.2 can be improved by exploiting locality. Instead of selecting edges completely at random, edges are selected from a candidate set. This candidate set consists of all edges that may lead to beneficial moves, and initially contains all edges.

To generate a legal move, we randomly remove an edge from the candidate set. We first consider collapsing the edge, accepting the move if it is legal and reduces the total energy. If the edge collapse is not accepted, we then consider edge swap and edge split in that order. If one of the transformations is accepted, we update the candidate set by adding all neighboring edges. The candidate set becomes very useful toward the end of optimization, when the fraction of beneficial moves diminishes.

4.4 Setting of the Spring Constant

We view the spring energy E_{spring} as a regularizing term that helps guide the optimization process to a good minimum. The spring constant κ determines the contribution of this term to the total energy. We have obtained good results by making successive calls to procedure OptimizeMesh, each with a different value of κ, according to a schedule that gradually decreases κ.

As an example, to obtain the final mesh in Figure 7h starting from the mesh in Figure 7c, we successively set κ to $10^{-2}, 10^{-3}, 10^{-4}$, and 10^{-8} (see Figures 7f–7h). This same schedule was used in all the examples.

5 Results

5.1 Surface Reconstruction

From the set of points shown in Figure 7b, phase one of our reconstruction algorithm [5] produces the mesh shown in Figure 7c; this mesh has the correct topological type, but it is rather dense, is far away from the data, and lacks the sharp features of the original model (Figure 7a). Using this mesh as a starting point, mesh optimization produces the mesh in Figure 7h.

Figures 7i–7k,7m–7o show two examples of surface reconstruction from actual laser range data (courtesy of Technical Arts, Redmond, WA). Figures 7j and 7n show sets of points obtained by sampling two physical objects (a distributor cap and a golf club head) with a laser range finder. The outputs of phase one are shown in Figures 7k and 7o. The holes present in the surface of Figure 7k are artifacts of the data, as self-shadowing prevented some regions of the surface from being scanned. Adaptive selection of scanning paths preventing such shadowing is an interesting area of future research. In this case, we manually filled the holes, leaving a single boundary at the bottom. Figures 7l and 7p show the optimized meshes obtained with our algorithm.

5.2 Mesh Simplification

For mesh simplification, we first sample a set of points randomly from the original mesh using uniform random sampling over area. Next, we add the vertices of the mesh to this point set. Finally, to more faithfully preserve the boundaries of the mesh, we sample additional points from boundary edges.

As an example of mesh simplification, we start with the mesh containing 2032 vertices shown in Figure 7q. From it, we obtain a sample of 6752 points shown in Figure 7r (4000 random points, 2032 vertex points, and 720 boundary points). Mesh optimization, with $c_{rep} = 10^{-5}$, reduces the mesh down to 487 vertices (Figure 7s).

Fig.	#vert. m	#faces	#data n	Parameters c_{rep}	Parameters κ	Resulting energies E_{dist}	Resulting energies E	time (min.)
7c	1572	3152	4102	-	-	8.57×10^{-2}	-	-
7e	1572	3152	4102	10^{-5}	10^{-2}	8.04×10^{-4}	4.84×10^{-2}	1.5
7f	508	1024	4102	10^{-5}	10^{-2}	6.84×10^{-4}	3.62×10^{-2}	(+3.0)
7g	270	548	4102	10^{-5}	10^{-3}	6.08×10^{-4}	6.94×10^{-3}	(+2.2)
7h	163	334	4102	10^{-5}	*varied*	4.86×10^{-4}	2.12×10^{-3}	17.0
7k	9220	18272	12745	-	-	6.41×10^{-2}	-	-
7l	690	1348	12745	10^{-5}	*varied*	4.23×10^{-3}	1.18×10^{-2}	47.0
7o	4059	8073	16864	-	-	2.20×10^{-2}	-	-
7p	262	515	16864	10^{-5}	*varied*	2.19×10^{-3}	4.95×10^{-3}	44.5
7q	2032	3832	-	-	-	-	-	-
7s	487	916	6752	10^{-5}	*varied*	1.86×10^{-3}	8.05×10^{-3}	9.9
7t	239	432	6752	10^{-4}	*varied*	9.19×10^{-3}	4.39×10^{-2}	10.2

Table 1: Performance statistics for meshes shown in Figure 7.

By setting $c_{rep} = 10^{-4}$, we obtain a coarser mesh of 239 vertices (Figure 7t).

As these examples illustrate, basing mesh simplification on a measure of distance between the simplified mesh and the original has a number of benefits:

- Vertices are dense in regions of high Gaussian curvature, whereas a few large faces span the flat regions.
- Long edges are aligned in directions of low curvature, and the aspect ratios of the triangles adjust to local curvature.
- Edges and vertices of the simplified mesh are placed near sharp features of the original mesh.

5.3 Segmentation

Mesh optimization enables us to detect sharp features in the underlying surface. Using a simple thresholding method, the optimized mesh can be segmented into smooth components. To this end, we build a graph in which the nodes are the faces of mesh. Two nodes of this graph are connected if the two corresponding faces are adjacent and their dihedral angle is smaller than a given threshold. The connected components of this graph identify the desired smooth segments. As an example, Figure 7i shows the segmentation of the optimized mesh into 11 components. After segmentation, vertex normals can be estimated from neighboring faces within each component, and a smoothly shaded surface can be created (Figure 7m).

5.4 Parameter Settings and Performance Statistics

Table 1 lists the specific parameter values of c_{rep} and κ used to generate the meshes in the examples, along with other performance statistics. In all these examples, the table entry *"varied"* refers to a spring constant schedule of $\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-8}\}$. In fact, all meshes in Figure 1 are also created using the same parameters (except that c_{rep} was changed in two cases). Execution times were obtained on a DEC uniprocessor Alpha workstation.

6 Related Work

Surface Fitting There is a large body of literature on fitting embeddings of a rectangular domain; see Bolle and Vemuri [1] for a review. Schudy and Ballard [11, 12] fit embeddings of a sphere to point data. Goshtasby [4] works with embeddings of cylinders and tori. Sclaroff and Pentland [13] consider embeddings of a deformed superquadric. Miller et al. [9] approximate an isosurface of volume data by fitting a mesh homeomorphic to a sphere. While it appears that their method could be extended to finding isosurfaces of arbitrary topological type, it it less obvious how it could be modified to

handle point instead of volume data. Mallet [7] discusses interpolation of functions over simplicial complexes of arbitrary topological type.

Our method allows fitting of a parametric surface of arbitrary topological type to a set of three-dimensional points. In [2], we sketched an algorithm for fitting a mesh of *fixed* vertex connectivity to the data. The algorithm presented here is an extension of this idea in which we also allow the number of vertices and their connectivity to vary. To the best of our knowledge, this has not been done before.

Mesh Simplification Two notable papers discussing the mesh simplification problem are Schroeder et al. [10] and Turk [15].

The motivation of Schroeder et al. is to simplify meshes generated by "marching cubes" that may consist of more than a million triangles. In their iterative approach, the basic operation is removal of a vertex and re-triangulation of the hole thus created. The criterion for vertex removal in the simplest case (interior vertex not on edge or corner) is the distance from the vertex to the plane approximating its surrounding vertices. It is worthwhile noting that this criterion only considers deviation of the new mesh from the mesh created in the previous iteration; deviation from the original mesh does not figure in the strategy.

Turk's goal is to reduce the amount of detail in a mesh while remaining faithful to the original topology and geometry. His basic idea is to distribute points on the existing mesh that are to become the new vertices. He then creates a triangulation containing both old and new vertices, and finally removes the old vertices. The density of the new vertices is chosen to be higher in areas of high curvature.

The principal advantage of our mesh simplification method compared to the techniques mentioned above is that we cast mesh simplification as an optimization problem: we find a new mesh of lower complexity that is as close as possible to the original mesh. This is recognized as a desirable property by Turk (Section 8, p. 63): "Another topic is finding measures of how closely matched a given re-tiling is to the original model. Can such a quality measure be used to guide the re-tiling process?". Optimization automatically retains more vertices in areas of high curvature, and leads to faces that are elongated along directions of low curvature, another property recognized as desirable by Turk.

7 Summary and Future Work

We have described an energy minimization approach to solving the mesh optimization problem. The energy function we use consists of three terms: a distance energy that measures the closeness of fit, a representation energy that penalizes meshes with a large number of vertices, and a regularizing term that conceptually places springs of rest length zero on the edges of the mesh. Our minimization algorithm partitions the problem into two nested subproblems: an inner continuous minimization and an outer discrete minimization. The search space consists of all meshes homeomorphic to the starting mesh.

Mesh optimization has proven effective as the second phase of our method for surface reconstruction from unorganized points, as discussed in [5]. (Phase two is responsible for improving the geometric fit and reducing the number of vertices of the mesh produced in phase one.)

Our method has also performed well for mesh simplification, that is, the reduction of the number of vertices in a dense triangular mesh. It produces meshes whose edges align themselves along directions of low curvature, and whose vertices concentrate in areas of high Gaussian curvature. Because the energy does not penalize surfaces with sharp dihedral angles, the method can recover sharp edges and corners.

A number of areas of future research still remain, including:

- Investigate the use of more sophisticated optimization methods, such as simulated annealing for discrete optimization and quadratic methods for non-linear least squares optimization, in order to avoid undesirable local minima in the energy and to accelerate convergence.
- Gain more insight into the use of the spring energy as a regularizing term, especially in the presence of appreciable noise.
- Improve the speed of the algorithm and investigate implementations on parallel architectures.
- Develop methods for fitting higher order splines to more accurately and concisely model curved surfaces.
- Experiment with sparse, non-uniform, and noisy data.
- Extend the current algorithm to other distance measures such as maximum error (L^∞ norm) or average error (L^1 norm), instead of the current L^2 norm.

References

[1] Ruud M. Bolle and Baba C. Vemuri. On three-dimensional surface reconstruction methods. *IEEE PAMI*, 13(1):1–13, January 1991.

[2] T. DeRose, H. Hoppe, T. Duchamp, J. McDonald, and W. Stuetzle. Fitting of surfaces to scattered data. *SPIE*, 1830:212–220, 1992.

[3] Gene Golub and Charles Van Loan. *Matrix Computations*. John Hopkins University Press, 2nd edition, 1989.

[4] Ardeshir Goshtasby. Surface reconstruction from scattered measurements. *SPIE*, 1830:247–256, 1992.

[5] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):71–78, July 1992.

[6] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. TR 93-01-01, Dept. of Computer Science and Engineering, University of Washington, January 1993.

[7] J.L. Mallet. Discrete smooth interpolation in geometric modeling. *CAD*, 24(4):178–191, April 1992.

[8] Samuel Marin and Philip Smith. Parametric approximation of data using ODR splines. GMR 7057, General Motors Research Laboratories, May 1990.

[9] J.V. Miller, D.E. Breen, W.E. Lorensen, R.M. O'Bara, and M.J. Wozny. Geometrically deformed models: A method for extracting closed geometric models from volume data. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):217–226, July 1991.

[10] William Schroeder, Jonathan Zarge, and William Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):65–70, July 1992.

[11] R. B. Schudy and D. H. Ballard. Model detection of cardiac chambers in ultrasound images. Technical Report 12, Computer Science Department, University of Rochester, 1978.

[12] R. B. Schudy and D. H. Ballard. Towards an anatomical model of heart motion as seen in 4-d cardiac ultrasound data. In *Proceedings of the 6th Conference on Computer Applications in Radiology and Computer-Aided Analysis of Radiological Images*, 1979.

[13] Stan Sclaroff and Alex Pentland. Generalized implicit functions for computer graphics. *Computer Graphics (SIGGRAPH '91 Proceedings)*, 25(4):247–250, July 1991.

[14] E. H. Spanier. *Algebraic Topology*. McGraw-Hill, New York, 1966.

[15] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):55–64, July 1992.

[16] G. Wyvill, C. McPheeters, and B. Wyvill. Data structures for soft objects. *The Visual Computer*, 2(4):227–234, August 1986.

(a) Object to be sampled
(b) Sampled points X (n = 4102)
(c) Output of phase one (M_0)
(d) Optimization without E_{spring}
(e) Optimum for fixed K_0
(f) Optimum with $\kappa = 10^{-2}$
(g) Optimum with $\kappa = 10^{-3}$
(h) Final optimum with $\kappa = 10^{-8}$
(i) Segmented surface (11 comp.)
(j) Laser range data (n = 12, 745)
(k) Output of phase one
(l) Output of phase two
(m) Smooth shading from segments
(n) Laser range data (n = 16, 864)
(o) Output of phase one
(p) Output of phase two
(q) Original mesh M_0
(r) Sampled points X (n = 6752)
(s) Simplified mesh ($c_{rep} = 10^{-5}$)
(t) Simplified mesh ($c_{rep} = 10^{-4}$)

Figure 7: Examples of surface reconstruction and mesh simplification.

Progressive Meshes

Hugues Hoppe
Microsoft Research

ABSTRACT

Highly detailed geometric models are rapidly becoming commonplace in computer graphics. These models, often represented as complex triangle meshes, challenge rendering performance, transmission bandwidth, and storage capacities. This paper introduces the *progressive mesh* (PM) representation, a new scheme for storing and transmitting arbitrary triangle meshes. This efficient, lossless, continuous-resolution representation addresses several practical problems in graphics: smooth geomorphing of level-of-detail approximations, progressive transmission, mesh compression, and selective refinement.

In addition, we present a new mesh simplification procedure for constructing a PM representation from an arbitrary mesh. The goal of this optimization procedure is to preserve not just the geometry of the original mesh, but more importantly its overall appearance as defined by its discrete and scalar appearance attributes such as material identifiers, color values, normals, and texture coordinates. We demonstrate construction of the PM representation and its applications using several practical models.

CR Categories and Subject Descriptors: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - surfaces and object representations.

Additional Keywords: mesh simplification, level of detail, shape interpolation, progressive transmission, geometry compression.

1 INTRODUCTION

Highly detailed geometric models are necessary to satisfy a growing expectation for realism in computer graphics. Within traditional modeling systems, detailed models are created by applying versatile modeling operations (such as extrusion, constructive solid geometry, and freeform deformations) to a vast array of geometric primitives. For efficient display, these models must usually be tessellated into polygonal approximations—meshes. Detailed meshes are also obtained by scanning physical objects using range scanning systems [5]. In either case, the resulting complex meshes are expensive to store, transmit, and render, thus motivating a number of practical problems:

Email: hhoppe@microsoft.com
Web: http://www.research.microsoft.com/research/graphics/hoppe/

- *Mesh simplification*: The meshes created by modeling and scanning systems are seldom optimized for rendering efficiency, and can frequently be replaced by nearly indistinguishable approximations with far fewer faces. At present, this process often requires significant user intervention. Mesh simplification tools can hope to automate this painstaking task, and permit the porting of a single model to platforms of varying performance.
- *Level-of-detail (LOD) approximation*: To further improve rendering performance, it is common to define several versions of a model at various levels of detail [3, 8]. A detailed mesh is used when the object is close to the viewer, and coarser approximations are substituted as the object recedes. Since instantaneous switching between LOD meshes may lead to perceptible "popping", one would like to construct smooth visual transitions, *geomorphs*, between meshes at different resolutions.
- *Progressive transmission*: When a mesh is transmitted over a communication line, one would like to show progressively better approximations to the model as data is incrementally received. One approach is to transmit successive LOD approximations, but this requires additional transmission time.
- *Mesh compression*: The problem of minimizing the storage space for a model can be addressed in two orthogonal ways. One is to use mesh simplification to reduce the number of faces. The other is mesh compression: minimizing the space taken to store a particular mesh.
- *Selective refinement*: Each mesh in a LOD representation captures the model at a uniform (view-independent) level of detail. Sometimes it is desirable to adapt the level of refinement in selected regions. For instance, as a user flies over a terrain, the terrain mesh need be fully detailed only near the viewer, and only within the field of view.

In addressing these problems, this paper makes two major contributions. First, it introduces the *progressive mesh* (PM) representation. In PM form, an arbitrary mesh $\hat{M}$ is stored as a much coarser mesh M^0 together with a sequence of n detail records that indicate how to incrementally refine M^0 exactly back into the original mesh $\hat{M} = M^n$. Each of these records stores the information associated with a *vertex split*, an elementary mesh transformation that adds an additional vertex to the mesh. The PM representation of $\hat{M}$ thus defines a continuous sequence of meshes $M^0, M^1, \ldots, M^n$ of increasing accuracy, from which LOD approximations of any desired complexity can be efficiently retrieved. Moreover, geomorphs can be efficiently constructed between any two such meshes. In addition, we show that the PM representation naturally supports progressive transmission, offers a concise encoding of $\hat{M}$ itself, and permits selective refinement. In short, progressive meshes offer an efficient, lossless, continuous-resolution representation.

The other contribution of this paper is a new simplification procedure for constructing a PM representation from a given mesh $\hat{M}$. Unlike previous simplification methods, our procedure seeks to preserve not just the geometry of the mesh surface, but more importantly its overall appearance, as defined by the discrete and scalar attributes associated with its surface.

2 MESHES IN COMPUTER GRAPHICS

Models in computer graphics are often represented using triangle meshes.[1] Geometrically, a triangle mesh is a piecewise linear surface consisting of triangular faces pasted together along their edges. As described in [9], the mesh geometry can be denoted by a tuple (K, V), where K is a *simplicial complex* specifying the connectivity of the mesh simplices (the adjacency of the vertices, edges, and faces), and $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ is the set of vertex positions defining the shape of the mesh in $\mathbf{R}^3$. More precisely (cf. [9]), we construct a parametric domain $|K| \subset \mathbf{R}^m$ by identifying each vertex of K with a canonical basis vector of $\mathbf{R}^m$, and define the mesh as the image $\phi_V(|K|)$ where $\phi_V : \mathbf{R}^m \to \mathbf{R}^3$ is a linear map.

Often, surface appearance attributes other than geometry are also associated with the mesh. These attributes can be categorized into two types: *discrete* attributes and *scalar* attributes.

Discrete attributes are usually associated with faces of the mesh. A common discrete attribute, the *material identifier*, determines the shader function used in rendering a face of the mesh [18]. For instance, a trivial shader function may involve simple look-up within a specified texture map.

Many scalar attributes are often associated with a mesh, including diffuse color (r, g, b), normal (n_x, n_y, n_z), and texture coordinates (u, v). More generally, these attributes specify the local parameters of shader functions defined on the mesh faces. In simple cases, these scalar attributes are associated with vertices of the mesh. However, to represent discontinuities in the scalar fields, and because adjacent faces may have different shading functions, it is common to associate scalar attributes not with vertices, but with corners of the mesh [1]. A *corner* is defined as a (vertex,face) tuple. Scalar attributes at a corner (v, f) specify the shading parameters for face f at vertex v. For example, along a *crease* (a curve on the surface across which the normal field is not continuous), each vertex has two distinct normals, one associated with the corners on each side of the crease.

We express a mesh as a tuple $M = (K, V, D, S)$ where V specifies its geometry, D is the set of discrete attributes d_f associated with the faces $f = \{j, k, l\} \in K$, and S is the set of scalar attributes $s_{(v,f)}$ associated with the corners (v, f) of K.

The attributes D and S give rise to discontinuities in the visual appearance of the mesh. An edge $\{v_j, v_k\}$ of the mesh is said to be *sharp* if either (1) it is a boundary edge, or (2) its two adjacent faces f_l and f_r have different discrete attributes (i.e. $d_{fl} \neq d_{fr}$), or (3) its adjacent corners have different scalar attributes (i.e. $s_{(v_j,f_l)} \neq s_{(v_j,f_r)}$ or $s_{(v_k,f_l)} \neq s_{(v_k,f_r)}$). Together, the set of sharp edges define a set of *discontinuity curves* over the mesh (e.g. the yellow curves in Figure 12).

3 PROGRESSIVE MESH REPRESENTATION

3.1 Overview

Hoppe et al. [9] describe a method, *mesh optimization*, that can be used to approximate an initial mesh $\hat{M}$ by a much simpler one. Their optimization algorithm, reviewed in Section 4.1, traverses the space of possible meshes by successively applying a set of 3 mesh transformations: edge collapse, edge split, and edge swap.

We have discovered that in fact a single one of those transformations, *edge collapse*, is sufficient for effectively simplifying meshes. As shown in Figure 1, an edge collapse transformation $ecol(\{v_s, v_t\})$

[1] We assume in this paper that more general meshes, such as those containing n-sided faces and faces with holes, are first converted into triangle meshes by triangulation. The PM representation could be generalized to handle the more general meshes directly, at the expense of more complex data structures.

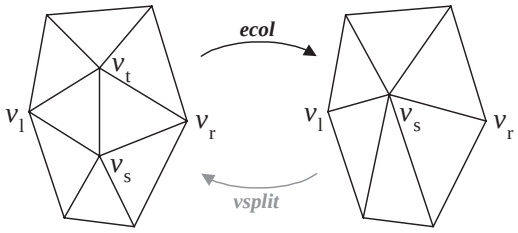


Figure 1: Illustration of the edge collapse transformation.

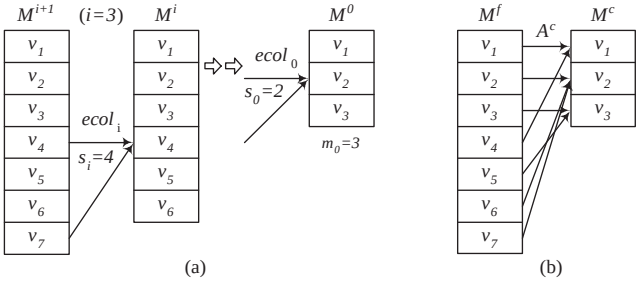


Figure 2: (a) Sequence of edge collapses; (b) Resulting vertex correspondence.

unifies 2 adjacent vertices v_s and v_t into a single vertex v_s. The vertex v_t and the two adjacent faces $\{v_s, v_t, v_l\}$ and $\{v_t, v_s, v_r\}$ vanish in the process. A position $\mathbf{v}_s$ is specified for the new unified vertex.

Thus, an initial mesh $\hat{M} = M^n$ can be simplified into a coarser mesh M^0 by applying a sequence of n successive edge collapse transformations:

$$(\hat{M} = M^n) \xrightarrow{ecol_{n-1}} \ldots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0 .$$

The particular sequence of edge collapse transformations must be chosen carefully, since it determines the quality of the approximating meshes $M^i, i < n$. A scheme for choosing these edge collapses is presented in Section 4.

Let m_0 be the number of vertices in M^0, and let us label the vertices of mesh M^i as $V^i = \{v_1, \ldots, v_{m_0+i}\}$, so that edge $\{v_{s_i}, v_{m_0+i+1}\}$ is collapsed by $ecol_i$ as shown in Figure 2a. As vertices may have different positions in the different meshes, we denote the position of v_j in M^i as $\mathbf{v}_j^i$.

A key observation is that an edge collapse transformation is invertible. Let us call that inverse transformation a *vertex split*, shown as *vsplit* in Figure 1. A vertex split transformation $vsplit(s, l, r, t, A)$ adds near vertex v_s a new vertex v_t and two new faces $\{v_s, v_t, v_l\}$ and $\{v_t, v_s, v_r\}$. (If the edge $\{v_s, v_t\}$ is a boundary edge, we let $v_r = 0$ and only one face is added.) The transformation also updates the attributes of the mesh in the neighborhood of the transformation. This attribute information, denoted by A, includes the positions $\mathbf{v}_s$ and $\mathbf{v}_t$ of the two affected vertices, the discrete attributes $d_{\{v_s,v_t,v_l\}}$ and $d_{\{v_t,v_s,v_r\}}$ of the two new faces, and the scalar attributes of the affected corners ($s_{(v_s,\cdot)}$, $s_{(v_t,\cdot)}$, $s_{(v_l,\{v_s,v_t,v_l\})}$, and $s_{(v_r,\{v_t,v_s,v_r\})}$).

Because edge collapse transformations are invertible, we can therefore represent an arbitrary triangle mesh $\hat{M}$ as a simple mesh M^0 together with a sequence of n *vsplit* records:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \ldots \xrightarrow{vsplit_{n-1}} (M^n = \hat{M})$$

where each record is parametrized as $vsplit_i(s_i, l_i, r_i, A_i)$. We call $(M^0, \{vsplit_0, \ldots, vsplit_{n-1}\})$ a *progressive mesh* (PM) representation of $\hat{M}$.

As an example, the mesh $\hat{M}$ of Figure 5d (13,546 faces) was simplified down to the coarse mesh M^0 of Figure 5a (150 faces) using

6,698 edge collapse transformations. Thus its PM representation consists of M^0 together with a sequence of $n = 6698$ *vsplit* records. From this PM representation, one can extract approximating meshes with any desired number of faces (actually, within ± 1) by applying to M^0 a prefix of the *vsplit* sequence. For example, Figure 5 shows approximating meshes with 150, 500, and 1000 faces.

3.2 Geomorphs

A nice property of the vertex split transformation (and its inverse, edge collapse) is that a smooth visual transition (a *geomorph*) can be created between the two meshes M^i and M^{i+1} in $M^i \xrightarrow{vsplit_i} M^{i+1}$. For the moment let us assume that the meshes contain no attributes other than vertex positions. With this assumption the vertex split record is encoded as $vsplit_i(s_i, l_i, r_i, A_i = (\mathbf{v}_{s_i}^{i+1}, \mathbf{v}_{m_0+i+1}^{i+1}))$. We construct a geomorph $M^G(\alpha)$ with blend parameter $0 \le \alpha \le 1$ such that $M^G(0)$ looks like M^i and $M^G(1)$ looks like M^{i+1}—in fact $M^G(1) = M^{i+1}$—by defining a mesh

$$M^G(\alpha) = (K^{i+1}, V^G(\alpha))$$

whose connectivity is that of M^{i+1} and whose vertex positions linearly interpolate from $v_{s_i} \in M^i$ to the split vertices $v_{s_i}, v_{m_0+i+1} \in M^{i+1}$:

$$\mathbf{v}_j^G(\alpha) = \begin{cases} (\alpha)\mathbf{v}_j^{i+1} + (1-\alpha)\mathbf{v}_{s_i}^i & , \ j \in \{s_i, m_0+i+1\} \\ \mathbf{v}_j^{i+1} = \mathbf{v}_j^i & , \ j \notin \{s_i, m_0+i+1\} \end{cases}$$

Using such geomorphs, an application can smoothly transition from a mesh M^i to meshes M^{i+1} or M^{i-1} without any visible "snapping" of the meshes.

Moreover, since individual *ecol* transformations can be transitioned smoothly, so can the composition of any sequence of them. Geomorphs can therefore be constructed between *any* two meshes of a PM representation. Indeed, given a finer mesh M^f and a coarser mesh M^c, $0 \le c < f \le n$, there exists a natural correspondence between their vertices: each vertex of M^f is related to a unique ancestor vertex of M^c by a surjective map A^c obtained by composing a sequence of *ecol* transformations (Figure 2b). More precisely, each vertex v_j of M^f corresponds with the vertex $v_{A^c(j)}$ in M^c where

$$A^c(j) = \begin{cases} j & , \ j \le m_0 + c \\ A^c(s_{j-m_0-1}) & , \ j > m_0 + c \end{cases} .$$

(In practice, this ancestor information A^c is gathered in a forward fashion as the mesh is refined.) This correspondence allows us to define a geomorph $M^G(\alpha)$ such that $M^G(0)$ looks like M^c and $M^G(1)$ equals M^f. We simply define $M^G(\alpha) = (K^f, V^G(\alpha))$ to have the connectivity of M^f and the vertex positions

$$\mathbf{v}_j^G(\alpha) = (\alpha)\mathbf{v}_j^f + (1-\alpha)\mathbf{v}_{A^c(j)}^c .$$

So far we have outlined the construction of geomorphs between PM meshes containing only position attributes. We can in fact construct geomorphs for meshes containing both discrete and scalar attributes.

Discrete attributes by their nature cannot be smoothly interpolated. Fortunately, these discrete attributes are associated with faces of the mesh, and the "geometric" geomorphs described above smoothly introduce faces. In particular, observe that the faces of M^c are a proper subset of the faces of M^f, and that those faces of M^f missing from M^c are invisible in $M^G(0)$ because they have been collapsed to degenerate (zero area) triangles. Other geomorphing schemes [10, 11, 17] define well-behaved (invertible) parametrizations between meshes at different levels of detail, but these do not permit the construction of geomorphs between meshes with different discrete attributes.

Scalar attributes defined on corners can be smoothly interpolated much like the vertex positions. There is a slight complication in that a corner (v, f) in a mesh M is not naturally associated with any "ancestor corner" in a coarser mesh M^c if f is not a face of M^c. We can still attempt to infer what attribute value (v, f) would have in M^c as follows. We examine the mesh M^{i+1} in which f is first introduced, locate a neighboring corner (v, f') in M^{i+1} whose attribute value is the same, and recursively backtrack from it to a corner in M^c. If there is no neighboring corner in M^{i+1} with an identical attribute value, then the corner (v, f) has no equivalent in M^c and we therefore keep its attribute value constant through the geomorph.

The interpolating function on the scalar attributes need not be linear; for instance, normals are best interpolated over the unit sphere, and colors may be interpolated in a color space other than RGB.

Figure 6 demonstrates a geomorph between two meshes M^{175} (500 faces) and M^{425} (1000 faces) retrieved from the PM representation of the mesh in Figure 5d.

3.3 Progressive transmission

Progressive meshes are a natural representation for progressive transmission. The compact mesh M^0 is transmitted first (using a conventional uni-resolution format), followed by the stream of $vsplit_i$ records. The receiving process incrementally rebuilds $\hat{M}$ as the records arrive, and animates the changing mesh. The changes to the mesh can be geomorphed to avoid visual discontinuities. The original mesh $\hat{M}$ is recovered exactly after all n records are received, since PM is a lossless representation.

The computation of the receiving process should be balanced between the reconstruction of $\hat{M}$ and interactive display. With a slow communication line, a simple strategy is to display the current mesh whenever the input buffer is found to be empty. With a fast communication line, we find that a good strategy is to display meshes whose complexities increase exponentially. (Similar issues arise in the display of images transmitted using progressive JPEG.)

3.4 Mesh compression

Even though the PM representation encodes both $\hat{M}$ and a continuous family of approximations, it is surprisingly space-efficient, for two reasons. First, the locations of the vertex split transformations can be encoded concisely. Instead of storing all three vertex indices (s_i, l_i, r_i) of $vsplit_i$, one need only store s_i and approximately 5 bits to select the remaining two vertices among those adjacent to v_{s_i}.[2] Second, because a vertex split has local effect, one can expect significant coherence in mesh attributes through each transformation. For instance, when vertex $v_{s_i}^i$ is split into $v_{s_i}^{i+1}$ and $v_{m_0+i+1}^{i+1}$, we can predict the positions $\mathbf{v}_{s_i}^{i+1}$ and $\mathbf{v}_{m_0+i+1}^{i+1}$ from $\mathbf{v}_{s_i}^i$, and use delta-encoding to reduce storage. Scalar attributes of corners in M^{i+1} can similarly be predicted from those in M^i. Finally, the material identifiers $d_{\{v_s, v_t, v_l\}}$ and $d_{\{v_t, v_s, v_r\}}$ of the new faces in mesh M^{i+1} can often be predicted from those of adjacent faces in M^i using only a few control bits.

As a result, the size of a carefully designed PM representation should be competitive with that obtained from methods for compressing uni-resolution meshes. Our current prototype implementation was not designed with this goal in mind. However, we analyze the compression of the connectivity K, and report results on the compression of the geometry V. In the following analysis, we assume for simplicity that $m_0 = 0$ since typically $m_0 \ll n$.

A common representation for the mesh connectivity K is to list the three vertex indices for each face. Since the number of vertices is n and the number of faces approximately $2n$, such a list requires $6\lceil \log_2(n) \rceil n$ bits of storage. Using a buffer of 2 vertices, *generalized triangle strip* representations reduce this number to about

[2] On average, v_{s_i} has 6 neighbors, and the number of permutations $P_2^6 = 30$ can be encoded in $\lceil \log_2(P_2^6) \rceil = 5$ bits.

$(\lceil \log_2(n) \rceil + 2k)n$ bits, where vertices are back-referenced once on average and $k \simeq 2$ bits capture the vertex replacement codes [6]. By increasing the vertex buffer size to 16, Deering's *generalized triangle mesh* representation [6] further reduces storage to about $(\frac{1}{8}\lceil \log_2(n) \rceil + 8)n$ bits. Turan [16] shows that planar graphs (and hence the connectivity of closed genus 0 meshes) can be encoded in $12n$ bits. Recent work by Taubin and Rossignac [15] addresses more general meshes. With the PM representation, each $vsplit_i$ requires specification of s_i and its two neighbors, for a total storage of about $(\lceil \log_2(n) \rceil + 5)n$ bits. Although not as concise as [6, 15], this is comparable to generalized triangle strips.

A traditional representation of the mesh geometry V requires storage of $3n$ coordinates, or $96n$ bits with IEEE single-precision floating point. Like Deering [6], we assume that these coordinates can be quantized to 16-bit fixed precision values without significant loss of visual quality, thus reducing storage to $48n$ bits. Deering is able to further compress this storage by delta-encoding the quantized coordinates and Huffman compressing the variable-length deltas. For 16-bit quantization, he reports storage of $35.8n$ bits, which includes both the deltas and the Huffman codes. Using a similar approach with the PM representation, we encode V in $31n$ to $50n$ bits as shown in Table 1. To obtain these results, we exploit a property of our optimization algorithm (Section 4.3): when considering the collapse of an edge $\{v_s, v_t\}$, it considers three starting points for the resulting vertex position $\mathbf{v}_n$: $\{\mathbf{v}_s, \mathbf{v}_t, \frac{\mathbf{v}_s+\mathbf{v}_t}{2}\}$. Depending on the starting point chosen, we delta-encode either $\{\mathbf{v}_s - \mathbf{v}_n, \mathbf{v}_t - \mathbf{v}_n\}$ or $\{\frac{\mathbf{v}_s+\mathbf{v}_t}{2} - \mathbf{v}_n, \frac{\mathbf{v}_t-\mathbf{v}_s}{2}\}$, and use separate Huffman tables for all four quantities.

To further improve compression, we could alter the construction algorithm to forego optimization and let $\mathbf{v}_n \in \{\mathbf{v}_s, \mathbf{v}_t, \frac{\mathbf{v}_s+\mathbf{v}_t}{2}\}$. This would degrade the accuracy of the approximating meshes somewhat, but allows encoding of V in $30n$ to $37n$ bits in our examples. Arithmetic coding [19] of delta lengths does not improve results significantly, reflecting the fact that the Huffman trees are well balanced. Further compression improvements may be achievable by adapting both the quantization level and the delta length models as functions of the *vsplit* record index i, since the magnitude of successive changes tends to decrease.

3.5 Selective refinement

The PM representation also supports selective refinement, whereby detail is added to the model only in desired areas. Let the application supply a callback function REFINE(v) that returns a Boolean value indicating whether the neighborhood of the mesh about v should be further refined. An initial mesh M^c is selectively refined by iterating through the list $\{vsplit_c, \ldots, vsplit_{n-1}\}$ as before, but only performing $vsplit_i(s_i, l_i, r_i, A_i)$ if

(1) all three vertices $\{v_{s_i}, v_{l_i}, v_{r_i}\}$ are present in the mesh, and

(2) REFINE(v_{s_i}) evaluates to TRUE.

(A vertex v_j is absent from the mesh if the prior vertex split that would have introduced it, $vsplit_{j-m_0-1}$, was not performed due to the above conditions.)

As an example, to obtain selective refinement of the model within a view frustum, REFINE(v) is defined to be TRUE if either v or any of its neighbors lies within the frustum. As seen in Figure 7a, condition (1) described above is suboptimal. The problem is that a vertex v_{s_i} within the frustum may fail to be split because its expected neighbor v_{l_i} lies just outside the frustum and was not previously created. The problem is remedied by using a less stringent version of condition (1). Let us define the *closest living ancestor* of a vertex v_j to be the vertex with index

$$A'(j) = \begin{cases} j & , \text{ if } v_j \text{ exists in the mesh} \\ A'(s_{j-m_0-1}) & , \text{ otherwise} \end{cases}$$

The new condition becomes:

(1') v_{s_i} is present in the mesh (i.e. $A'(s_i) = s_i$) and the vertices $v_{A'(l_i)}$ and $v_{A'(r_i)}$ are both adjacent to v_{s_i}.

As when constructing the geomorphs, the ancestor information A' is carried efficiently as the *vsplit* records are parsed. If conditions (1') and (2) are satisfied, $vsplit(s_i, A'(l_i), A'(r_i), A_i)$ is applied to the mesh. A mesh selectively refined with this new strategy is shown in Figure 7b. This same strategy was also used for Figure 10. Note that it is still possible to create geomorphs between M^c and selectively refined meshes thus created.

An interesting application of selective refinement is the transmission of view-dependent models over low-bandwidth communication lines. As the receiver's view changes over time, the sending process need only transmit those *vsplit* records for which REFINE evaluates to TRUE, and of those only the ones not previously transmitted.

4 PROGRESSIVE MESH CONSTRUCTION

The PM representation of an arbitrary mesh $\hat{M}$ requires a sequence of edge collapses transforming $\hat{M} = M^n$ into a base mesh M^0. The quality of the intermediate approximations $M^i, i < n$ depends largely on the algorithm for selecting which edges to collapse and what attributes to assign to the affected neighborhoods, for instance the positions $\mathbf{v}_{s_i}^i$.

There are many possible PM construction algorithms with varying trade-offs of speed and accuracy. At one extreme, a crude and fast scheme for selecting edge collapses is to choose them completely at random. (Some local conditions must be satisfied for an edge collapse to be legal, i.e. manifold preserving [9].) More sophisticated schemes can use heuristics to improve the edge selection strategy, for example the "distance to plane" metric of Schroeder et al. [14]. At the other extreme, one can attempt to find approximating meshes that are optimal with respect to some appearance metric, for instance the E_{dist} geometric metric of Hoppe et al. [9].

Since PM construction is a preprocess that can be performed offline, we chose to design a simplification procedure that invests some time in the selection of edge collapses. Our procedure is similar to the mesh optimization method introduced by Hoppe et al. [9], which is outlined briefly in Section 4.1. Section 4.2 presents an overview of our procedure, and Sections 4.3–4.6 present the details of our optimization scheme for preserving both the shape of the mesh and the scalar and discrete attributes which define its appearance.

4.1 Background: mesh optimization

The goal of mesh optimization [9] is to find a mesh $M = (K, V)$ that both accurately fits a set X of points $\mathbf{x}_i \in \mathbf{R}^3$ and has a small number of vertices. This problem is cast as minimization of an energy function

$$E(M) = E_{dist}(M) + E_{rep}(M) + E_{spring}(M) .$$

The first two terms correspond to the two goals of accuracy and conciseness: the *distance energy* term

$$E_{dist}(M) = \sum_i d^2(\mathbf{x}_i, \phi_V(|K|))$$

measures the total squared distance of the points from the mesh, and the *representation energy* term $E_{rep}(M) = c_{rep}m$ penalizes the number m of vertices in M. The third term, the *spring energy* $E_{spring}(M)$ is introduced to regularize the optimization problem. It corresponds to placing on each edge of the mesh a spring of rest length zero and tension κ:

$$E_{spring}(M) = \sum_{\{j,k\} \in K} \kappa \|\mathbf{v}_j - \mathbf{v}_k\|^2 .$$

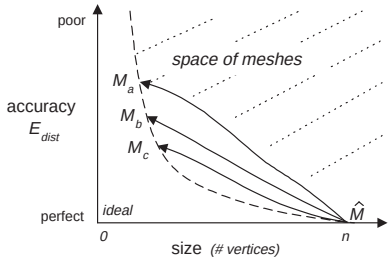


Figure 3: Illustration of the paths taken by mesh optimization using three different settings of c_{rep}.

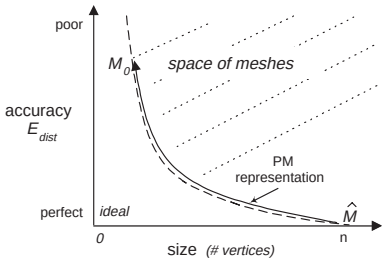


Figure 4: Illustration of the path taken by the new mesh simplification procedure in a graph plotting accuracy vs. mesh size.

The energy function $E(M)$ is minimized using a nested optimization method:

- *Outer loop*: The algorithm optimizes over K, the connectivity of the mesh, by randomly attempting a set of three possible mesh transformations: edge collapse, edge split, and edge swap. This set of transformations is complete, in the sense that any simplicial complex K of the same topological type as $\hat{K}$ can be reached through a sequence of these transformations. For each candidate mesh transformation, $K \rightarrow K'$, the continuous optimization described below computes $E_{K'}$, the minimum of E subject to the new connectivity K'. If $\Delta E = E_{K'} - E_K$ is found to be negative, the mesh transformation is applied (akin to a zero-temperature simulated annealing method).
- *Inner loop*: For each candidate mesh transformation, the algorithm computes $E_{K'} = \min_V E_{dist}(V) + E_{spring}(V)$ by optimizing over the vertex positions V. For the sake of efficiency, the algorithm in fact optimizes only one vertex position $\mathbf{v}_s$, and considers only the subset of points X that project onto the neighborhood affected by $K \rightarrow K'$. To avoid surface self-intersections, the edge collapse is disallowed if the maximum dihedral angle of edges in the resulting neighborhood exceeds some threshold.

Hoppe et al. [9] find that the regularizing spring energy term $E_{spring}(M)$ is most important in the early stages of the optimization, and achieve best results by repeatedly invoking the nested optimization method described above with a schedule of decreasing spring constants κ.

Mesh optimization is demonstrated to be an effective tool for mesh simplification. Given an initial mesh $\hat{M}$ to approximate, a dense set of points X is sampled both at the vertices of $\hat{M}$ and randomly over its faces. The optimization algorithm is then invoked with $\hat{M}$ as the starting mesh. Varying the setting of the representation constant c_{rep} results in optimized meshes with different trade-offs of accuracy and size. The paths taken by these optimizations are shown illustratively in Figure 3.

4.2 Overview of the simplification algorithm

As in mesh optimization [9], we also define an explicit energy metric $E(M)$ to measure the accuracy of simplified meshes $M = (K, V, D, S)$ with respect to the original $\hat{M}$, and we also modify the mesh M starting from $\hat{M}$ while minimizing $E(M)$.

Our energy metric has the following form:

$$E(M) = E_{dist}(M) + E_{spring}(M) + E_{scalar}(M) + E_{disc}(M) \ .$$

The first two terms, $E_{dist}(M)$ and $E_{spring}(M)$ are identical to those in [9]. The next two terms of $E(M)$ are added to preserve attributes associated with M: $E_{scalar}(M)$ measures the accuracy of its scalar attributes (Section 4.4), and $E_{disc}(M)$ measures the geometric accuracy of its discontinuity curves (Section 4.5). (To achieve scale invariance of the terms, the mesh is uniformly scaled to fit in a unit cube.)

Our scheme for optimizing over the connectivity K of the mesh is rather different from [9]. We have discovered that a mesh can be effectively simplified using edge collapse transformations alone. The edge swap and edge split transformations, useful in the context of surface reconstruction (which motivated [9]), are not essential for simplification. Although in principle our simplification algorithm can no longer traverse the entire space of meshes considered by mesh optimization, we find that the meshes generated by our algorithm are just as good. In fact, because of the priority queue approach described below, our meshes are usually better. Moreover, considering only edge collapses simplifies the implementation, improves performance, and most importantly, gives rise to the PM representation (Section 3).

Rather than randomly attempting mesh transformations as in [9], we place all (legal) candidate edge collapse transformations into a priority queue, where the priority of each transformation is its estimated energy cost ΔE. In each iteration, we perform the transformation at the front of the priority queue (with lowest ΔE), and recompute the priorities of edges in the neighborhood of this transformation. As a consequence, we eliminate the need for the awkward parameter c_{rep} as well as the energy term $E_{rep}(M)$. Instead, we can explicitly specify the number of faces desired in an optimized mesh. Also, a single run of the optimization can generate several such meshes. Indeed, it generates a continuous-resolution family of meshes, namely the PM representation of $\hat{M}$ (Figure 4).

For each edge collapse $K \rightarrow K'$, we compute its cost $\Delta E = E_{K'} - E_K$ by solving a continuous optimization

$$E_{K'} = \min_{V,S} E_{dist}(V) + E_{spring}(V) + E_{scalar}(V, S) + E_{disc}(V)$$

over both the vertex positions V and the scalar attributes S of the mesh with connectivity K'. This minimization is discussed in the next three sections.

4.3 Preserving surface geometry ($E_{dist} + E_{spring}$)

As in [9], we "record" the geometry of the original mesh $\hat{M}$ by sampling from it a set of points X. At a minimum, we sample a point at each vertex of $\hat{M}$. If requested by the user, additional points are sampled randomly over the surface of $\hat{M}$. The energy terms $E_{dist}(M)$ and $E_{spring}(M)$ are defined as in Section 4.1.

For a mesh of fixed connectivity, our method for optimizing the vertex positions to minimize $E_{dist}(V) + E_{spring}(V)$ closely follows that of [9]. Evaluating $E_{dist}(V)$ involves computing the distance of each point $\mathbf{x}_i$ to the mesh. Each of these distances is itself a minimization problem

$$d^2(\mathbf{x}_i, \phi_V(|K|)) = \min_{\mathbf{b}_i \in |K|} \|\mathbf{x}_i - \phi_V(\mathbf{b}_i)\|^2 \qquad (1)$$

where the unknown $\mathbf{b}_i$ is the parametrization of the projection of $\mathbf{x}_i$ on the mesh. The nonlinear minimization of $E_{dist}(V) + E_{spring}(V)$ is performed using an iterative procedure alternating between two steps:

1. For fixed vertex positions V, compute the optimal parametrizations $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_{|X|}\}$ by projecting the points X onto the mesh.
2. For fixed parametrizations B, compute the optimal vertex positions V by solving a sparse linear least-squares problem.

As in [9], when considering $ecol(\{v_s, v_t\})$, we optimize only one vertex position, $\mathbf{v}_s^i$. We perform three different optimizations with different starting points, $\mathbf{v}_s^i = (1-\alpha)\mathbf{v}_s^{i+1} + (\alpha)\mathbf{v}_t^{i+1}$ for $\alpha = \{0, \frac{1}{2}, 1\}$, and accept the best one.

Instead of defining a global spring constant κ for E_{spring} as in [9], we adapt κ each time an edge collapse transformation is considered. Intuitively, the spring energy is most important when few points project onto a neighborhood of faces, since in this case finding the vertex positions minimizing $E_{dist}(V)$ may be an under-constrained problem. Thus, for each edge collapse transformation considered, we set κ as a function of the ratio of the number of points to the number of faces in the neighborhood.[3] With this adaptive scheme, the influence of $E_{spring}(M)$ decreases gradually and adaptively as the mesh is simplified, and we no longer require the expensive schedule of decreasing spring constants.

4.4 Preserving scalar attributes (E_{scalar})

As described in Section 2, we represent piecewise continuous scalar fields by defining scalar attributes S at the mesh corners. We now present our scheme for preserving these scalar fields through the simplification process. For exposition, we find it easier to first present the case of continuous scalar fields, in which the corner attributes at a vertex are identical. The generalization to piecewise continuous fields is discussed shortly.

Optimizing scalar attributes at vertices Let the original mesh $\hat{M}$ have at each vertex v_j not only a position $\mathbf{v}_j \in \mathbf{R}^3$ but also a scalar attribute $\underline{\mathbf{v}}_j \in \mathbf{R}^d$. To capture scalar attributes, we sample at each point $\mathbf{x}_i \in X$ the attribute value $\underline{\mathbf{x}}_i \in \mathbf{R}^d$. We would then like to generalize the distance metric E_{dist} to also measure the deviation of the sampled attribute values $\underline{X}$ from those of M.

One natural way to achieve this is to redefine the distance metric to measure distance in $\mathbf{R}^{3+d}$:

$$d^2((\mathbf{x}_i \;\; \underline{\mathbf{x}}_i), M(K, V, \underline{V})) = \min_{\mathbf{b}_i \in |K|} \left\| (\mathbf{x}_i \;\; \underline{\mathbf{x}}_i) - (\phi_V(\mathbf{b}_i) \;\; \phi_{\underline{V}}(\mathbf{b}_i)) \right\|^2 .$$

This new distance functional could be minimized using the iterative approach of Section 4.3. However, it would be expensive since finding the optimal parametrization $\mathbf{b}_i$ of each point $\mathbf{x}_i$ would require projection in $\mathbf{R}^{3+d}$, and would be non-intuitive since these parametrizations would not be geometrically based.

Instead we opted to determine the parametrizations $\mathbf{b}_i$ using only geometry with equation (1), and to introduce a separate energy term E_{scalar} to measure attribute deviation based on these parametrizations:

$$E_{scalar}(\underline{V}) = (c_{scalar})^2 \sum_i \left\| \underline{\mathbf{x}}_i - \phi_{\underline{V}}(\mathbf{b}_i) \right\|^2$$

where the constant c_{scalar} assigns a relative weight between the scalar attribute errors (E_{scalar}) and the geometric errors (E_{dist}).

Thus, to minimize $E(V, \underline{V}) = E_{dist}(V) + E_{spring}(V) + E_{scalar}(\underline{V})$, our algorithm first finds the vertex position $\mathbf{v}_s$ minimizing $E_{dist}(V) + E_{spring}(V)$ by alternately projecting the points onto the mesh (obtaining the parametrizations $\mathbf{b}_i$) and solving a linear least-squares problem (Section 4.1). Then, using those same parametrizations $\mathbf{b}_i$, it finds the vertex attribute $\underline{\mathbf{v}}_s$ minimizing E_{scalar} by solving a single linear least-squares problem. Hence introducing E_{scalar} into the optimization causes negligible performance overhead.

Since ΔE_{scalar} contributes to the estimated cost ΔE of an edge collapse, we obtain simplified meshes whose faces naturally adapt to the attribute fields, as shown in Figures 8 and 11.

Optimizing scalar attributes at corners Our scheme for optimizing the scalar corner attributes S is a straightforward generalization of the scheme just described. Instead of solving for a single unknown attribute value $\underline{\mathbf{v}}_s$, the algorithm partitions the corners around v_s into continuous sets (based on equivalence of corner attributes) and for each continuous set solves independently for its optimal attribute value.

Range constraints Some scalar attributes have constrained ranges. For instance, the components (r, g, b) of color are typically constrained to lie between 0 and 1. Least-squares optimization may yield color values outside this range. In these cases we clip the optimized values to the given range. For least-squares minimization of a Euclidean norm at a single vertex, this is in fact optimal.

Normals Surface normals (n_x, n_y, n_z) are typically constrained to have unit length, and thus their domain is non-Cartesian. Optimizing over normals would therefore require minimization of a nonlinear functional with nonlinear constraints. We decided to instead simply carry the normals through the simplification process. Specifically, we compute the new normals at vertex $v_{s_i}^i$ by interpolating between the normals at vertices $v_{s_i}^{i+1}$ and $v_{m_0+i+1}^{i+1}$ using the α value that resulted in the best vertex position $\mathbf{v}_{s_i}^i$ in Section 4.3. Fortunately, the absolute directions of normals are less visually important than their discontinuities, and we have a scheme for preserving such discontinuities, as described in the next section.

4.5 Preserving discontinuity curves (E_{disc})

Appearance attributes give rise to a set of discontinuity curves on the mesh, both from differences between discrete face attributes (e.g. material boundaries), and from differences between scalar corner attributes (e.g. creases and shadow boundaries). As these discontinuity curves form noticeable features, we have found it useful to preserve them both topologically and geometrically.

We can detect when a candidate edge collapse would modify the topology of the discontinuity curves using some simple tests on the presence of sharp edges in its neighborhood. Let $sharp(v_j, v_k)$ denote that an edge $\{v_j, v_k\}$ is sharp, and let $\#sharp(v_j)$ be the number of sharp edges adjacent to a vertex v_j. Then, referring to Figure 1, $ecol(\{v_s, v_t\})$ modifies the topology of discontinuity curves if either:

- $sharp(v_s, v_l)$ and $sharp(v_t, v_l)$, or
- $sharp(v_s, v_r)$ and $sharp(v_t, v_r)$, or
- $\#sharp(v_s) \geq 1$ and $\#sharp(v_t) \geq 1$ and not $sharp(v_s, v_t)$, or
- $\#sharp(v_s) \geq 3$ and $\#sharp(v_t) \geq 3$ and $sharp(v_s, v_t)$, or
- $sharp(v_s, v_t)$ and $\#sharp(v_s) = 1$ and $\#sharp(v_t) \neq 2$, or
- $sharp(v_s, v_t)$ and $\#sharp(v_t) = 1$ and $\#sharp(v_s) \neq 2$.

If an edge collapse would modify the topology of discontinuity curves, we either disallow it, or penalize it as discussed in Section 4.6.

To preserve the geometry of the discontinuity curves, we sample an additional set of points X_{disc} from the sharp edges of $\hat{M}$, and define an additional energy term E_{disc} equal to the total squared distances of each of these points to the discontinuity curve from which it was sampled. Thus E_{disc} is defined just like E_{dist}, except that the points X_{disc} are constrained to project onto a set of sharp edges in the mesh. In effect, we are solving a curve fitting problem embedded within the surface fitting problem. Since all boundaries of the surface are defined to be discontinuity curves, our procedure preserves bound-

[3] The neighborhood of an edge collapse transformation is the set of faces shown in Figure 1. Using C notation, we set $\kappa = r < 4 \;?\; 10^{-2} : r < 8 \;?\; 10^{-4} : 10^{-8}$ where r is the ratio of the number of points to faces in the neighborhood.

ary geometry more accurately than [9]. Figure 9 demonstrates the importance of using the E_{disc} energy term in preserving the material boundaries of a mesh with discrete face attributes.

4.6 Permitting changes to topology of discontinuity curves

Some meshes contain numerous discontinuity curves, and these curves may delimit features that are too small to be visible when viewed from a distance. In such cases we have found that strictly preserving the topology of the discontinuity curves unnecessarily curtails simplification. We have therefore adopted a hybrid strategy, which is to permit changes to the topology of the discontinuity curves, but to penalize such changes. When a candidate edge collapse $ecol(\{v_s, v_t\})$ changes the topology of the discontinuity curves, we add to its cost ΔE the value $|X_{disc,\{v_s,v_t\}}| \cdot \|\mathbf{v}_s - \mathbf{v}_t\|^2$ where $|X_{disc,\{v_s,v_t\}}|$ is the number of points of X_{disc} projecting onto $\{v_s, v_t\}$. That simple strategy, although ad hoc, has proven very effective. For example, it allows the dark gray window frames of the "cessna" (visible in Figure 9) to vanish in the simplified meshes (Figures 5a–c).

Table 1: Parameter settings and quantitative results.

Object	Original $\hat{M}$		Base M^0		User param.		$\lvert X_{disc}\rvert$	V	Time
	m_0+n	#faces	m_0	#faces	$\lvert X\rvert-(m_0+n)$	c_{color}		$\frac{\text{bits}}{n}$	mins
cessna	6,795	13,546	97	150	100,000	-	46,811	46	23
terrain	33,847	66,960	3	1	0	-	3,796	46	16
mandrill	40,000	79,202	3	1	0	0.1	4,776	31	19
radiosity	78,923	150,983	1,192	1,191	200,000	0.01	74,316	37	106
fandisk	6,475	12,946	27	50	10,000	-	5,924	50	19

5 RESULTS

Table 1 shows, for the meshes in Figures 5–12, the number of vertices and faces in both $\hat{M}$ and M^0. In general, we let the simplification proceed until no more legal edge collapse transformations are possible. For the "cessna", we stopped at 150 faces to obtain a visually aesthetic base mesh. As indicated, the only user-specified parameters are the number of additional points (besides the $m_0 + n$ vertices of $\hat{M}$) sampled to increase fidelity, and the c_{scalar} constants relating the scalar attribute accuracies to the geometric accuracy. The only scalar attribute we optimized is color, and its c_{scalar} constant is denoted as c_{color}. The number $|X_{disc}|$ of points sampled from sharp edges is set automatically so that the densities of X and X_{disc} are proportional.[4] Execution times were obtained on a 150MHz Indigo2 with 128MB of memory.

Construction of the PM representation proceeds in three steps. First, as the simplification algorithm applies a sequence $ecol_{n-1} \ldots ecol_0$ of transformations to the original mesh, it writes to a file the sequence $vsplit_{n-1} \ldots vsplit_0$ of corresponding inverse transformations. When finished, the algorithm also writes the resulting base mesh M^0. Next, we reverse the order of the *vsplit* records. Finally, we renumber the vertices and faces of $(M^0, vsplit_0 \ldots vsplit_{n-1})$ to match the indexing scheme of Section 3.1 in order to obtain a concise format.

Figure 6 shows a single geomorph between two meshes M^{175} and M^{425} of a PM representation. For interactive LOD, it is useful to select a sequence of meshes from the PM representation, and to construct successive geomorphs between them. We have obtained good results by selecting meshes whose complexities grow exponentially, as in Figure 5. During execution, an application can adjust the granularity of these geomorphs by sampling additional meshes from the PM representation, or freeing some up.

In Figure 10, we selectively refined a terrain (grid of 181×187 vertices) using a new REFINE(v) function that keeps more detail near silhouette edges and near the viewer. More precisely, for the faces F_v adjacent to v, we compute the signed projected screen areas $\{a_f : f \in F_v\}$. We let REFINE(v) return TRUE if

(1) any face $f \in F_v$ lies within the view frustum, and either

(2a) the signs of a_f are not all equal (i.e. v lies near a silhouette edge) or

(2b) $\sum_{f \in F_v} a_f > thresh$ for a screen area threshold $thresh = 0.16^2$ (where total screen area is 1).

6 RELATED WORK

Mesh simplification methods A number of schemes construct a discrete sequence of approximating meshes by repeated application of a simplification procedure. Turk [17] sprinkles a set of points on a mesh, with density weighted by estimates of local curvature, and then retriangulates based on those points. Both Schroeder et al. [14] and Cohen et al. [4] iteratively remove vertices from the mesh and retriangulate the resulting holes. Cohen et al. are able to bound the maximum error of the approximation by restricting it to lie between two offset surfaces. Hoppe et al. [9] find accurate approximations through a general mesh optimization process (Section 4.1). Rossignac and Borrel [12] merge vertices of a model using spatial binning. A unique aspect of their approach is that the topological type of the model may change in the process. Their method is extremely fast, but since it ignores geometric qualities like curvature, the resulting approximations can be far from optimal. Some of the above methods [12, 17] permit the construction of geomorphs between successive simplified meshes.

Multiresolution analysis (MRA) Lounsbery et al. [10, 11] generalize the concept of multiresolution analysis to surfaces of arbitrary topological type. Eck et al. [7] describe how MRA can be applied to the approximation of an arbitrary mesh. Certain et al. [2] extend MRA to capture color, and present a multiresolution Web viewer supporting progressive transmission. MRA has many similarities with the PM scheme, since both store a simple base mesh together with a stream of detail records, and both produce a continuous-resolution representation. It is therefore worthwhile to highlight their differences:

Advantages of PM over MRA:

- MRA requires that the detail terms (wavelets) lie on a domain with subdivision connectivity, and as a result an arbitrary initial mesh $\hat{M}$ can only be recovered to within an ϵ tolerance. In contrast, the PM representation is lossless since $M^n = \hat{M}$.
- Because the approximating meshes $M^i, i < n$ in a PM may have arbitrary connectivity, they can be much better approximations than their MRA counterparts (Figure 12).
- The MRA representation cannot deal effectively with surface creases, unless those creases lie parametrically along edges of the base mesh (Figure 12). PM's can introduce surface creases anywhere and at any level of detail.
- PM's capture continuous, piecewise-continuous, and discrete appearance attributes. MRA schemes can represent discontinuous functions using a piecewise-constant basis (such as the Haar basis as used in [2, 13]), but the resulting approximations have too many discontinuities since none of the basis functions meet continuously. Also, it is not clear how MRA could be extended to capture discrete attributes.

[4] We set $|X_{disc}|$ such that $|X_{disc}|/perim = c(|X|/area)^{\frac{1}{2}}$ where $perim$ is the total length of all sharp edges in $\hat{M}$, $area$ is total area of all faces, and the constant $c = 4.0$ is chosen empirically.

Advantages of MRA over PM:

- The MRA framework provides a parametrization between meshes at various levels of detail, thus making possible multiresolution surface editing. PM's also offer such a parametrization, but it is not smooth, and therefore multiresolution editing may be non-intuitive.
- Eck et al. [7] construct MRA approximations with guaranteed maximum error bounds to $\hat{M}$. Our PM construction algorithm does not provide such bounds, although one could envision using simplification envelopes [4] to achieve this.
- MRA allows geometry and color to be compressed independently [2].

Other related work There has been relatively little work in simplifying arbitrary surfaces with functions defined over them. One special instance is image compression, since an image can be thought of as a set of scalar color functions defined on a quadrilateral surface. Another instance is the framework of Schröder and Sweldens [13] for simplifying functions defined over the sphere. The PM representation, like the MRA representation, is a generalization in that it supports surfaces of arbitrary topological type.

7 SUMMARY AND FUTURE WORK

We have introduced the progressive mesh representation and shown that it naturally supports geomorphs, progressive transmission, compression, and selective refinement. In addition, as a PM construction method, we have presented a new mesh simplification procedure designed to preserve not just the geometry of the original mesh, but also its overall appearance.

There are a number of avenues for future work, including:

- Development of an explicit metric and optimization scheme for preserving surface normals.
- Experimentation with PM editing.
- Representation of articulated or animated models.
- Application of the work to progressive subdivision surfaces.
- Progressive representation of more general simplicial complexes (not just 2-d manifolds).
- Addition of spatial data structures to permit efficient selective refinement.

We envision many practical applications for the PM representation, including streaming of 3D geometry over the Web, efficient storage formats, and continuous LOD in computer graphics applications. The representation may also have applications in finite element methods, as it can be used to generate coarse meshes for multigrid analysis.

ACKNOWLEDGMENTS

I wish to thank Viewpoint Datalabs for providing the "cessna" mesh, Pratt & Whitney for the gas turbine engine component ("fandisk"), Softimage for the "terrain" mesh, and especially Steve Drucker for creating several radiosity models using Lightscape. Thanks also to Michael Cohen, Steven "Shlomo" Gortler, and Jim Kajiya for their enthusiastic support, and to Rick Szeliski for helpful comments on the paper. Mark Kenworthy first coined the term "geomorph" in '92 to distinguish them from image morphs.

REFERENCES

[1] APPLE COMPUTER, INC. *3D graphics programming with QuickDraw 3D.* Addison Wesley, 1995.

[2] CERTAIN, A., POPOVIC, J., DUCHAMP, T., SALESIN, D., STUETZLE, W., AND DEROSE, T. Interactive multiresolution surface viewing. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 91–98.

[3] CLARK, J. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19,* 10 (October 1976), 547–554.

[4] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., BROOKS, F., AND WRIGHT, W. Simplification envelopes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 119–128.

[5] CURLESS, B., AND LEVOY, M. A volumetric method for building complex models from range images. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 303–312.

[6] DEERING, M. Geometry compression. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 13–20.

[7] ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. Multiresolution analysis of arbitrary meshes. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 173–182.

[8] FUNKHOUSER, T., AND SÉQUIN, C. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 247–254.

[9] HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. Mesh optimization. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 19–26.

[10] LOUNSBERY, J. M. *Multiresolution analysis for surfaces of arbitrary topological type.* PhD thesis, Department of Computer Science and Engineering, University of Washington, 1994.

[11] LOUNSBERY, M., DEROSE, T., AND WARREN, J. Multiresolution analysis for surfaces of arbitrary topological type. Submitted for publication. (TR 93-10-05b, Dept. of Computer Science and Engineering, U. of Washington, January 1994.).

[12] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics,* B. Falcidieno and T. L. Kunii, Eds. Springer-Verlag, 1993, pp. 455–465.

[13] SCHRÖDER, P., AND SWELDENS, W. Spherical wavelets: efficiently representing functions on the sphere. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 161–172.

[14] SCHROEDER, W., ZARGE, J., AND LORENSEN, W. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings) 26,* 2 (1992), 65–70.

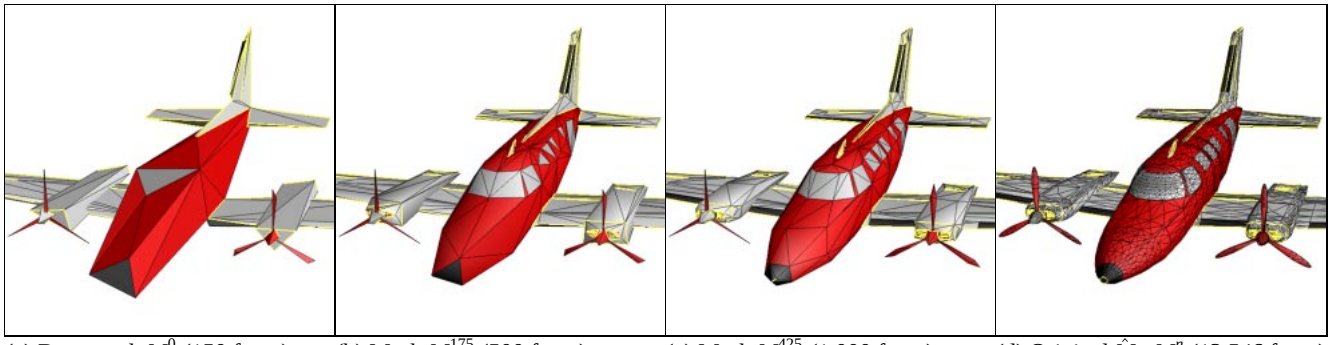
[15] TAUBIN, G., AND ROSSIGNAC, J. Geometry compression through topological surgery. Research Report RC-20340, IBM, January 1996.

[16] TURAN, G. Succinct representations of graphs. *Discrete Applied Mathematics 8* (1984), 289–294.

[17] TURK, G. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proceedings) 26,* 2 (1992), 55–64.

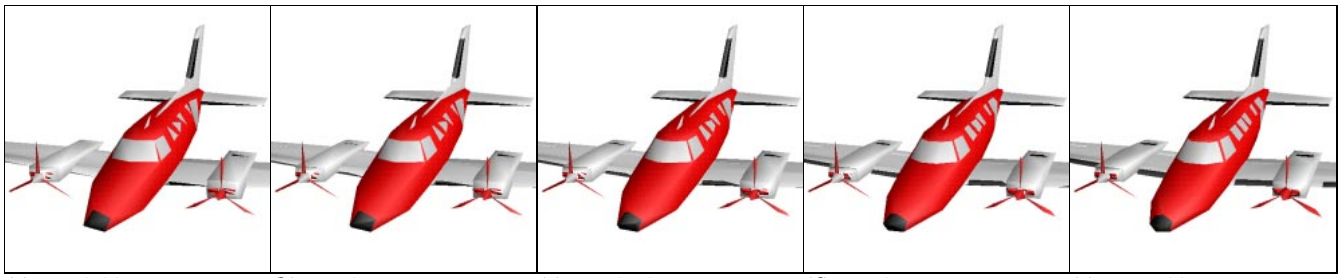
[18] UPSTILL, S. *The RenderMan Companion.* Addison-Wesley, 1990.

[19] WITTEN, I., NEAL, R., AND CLEARY, J. Arithmetic coding for data compression. *Communications of the ACM 30,* 6 (June 1987), 520–540.



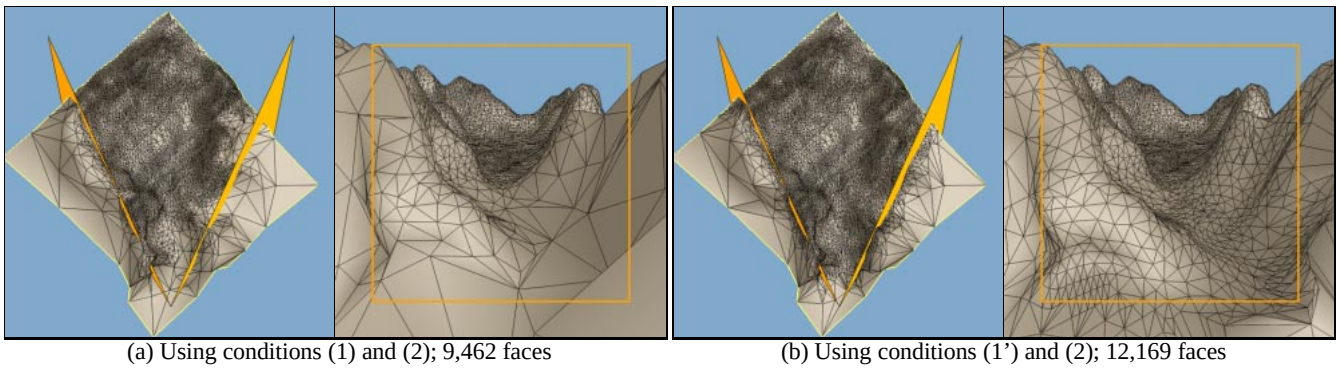
(a) Base mesh M^0 (150 faces) (b) Mesh M^{175} (500 faces) (c) Mesh M^{425} (1,000 faces) (d) Original $\hat{M} = M^n$ (13,546 faces)

Figure 5: The PM representation of an arbitrary mesh $\hat{M}$ captures a continuous-resolution family of approximating meshes $M^0 \ldots M^n = \hat{M}$.



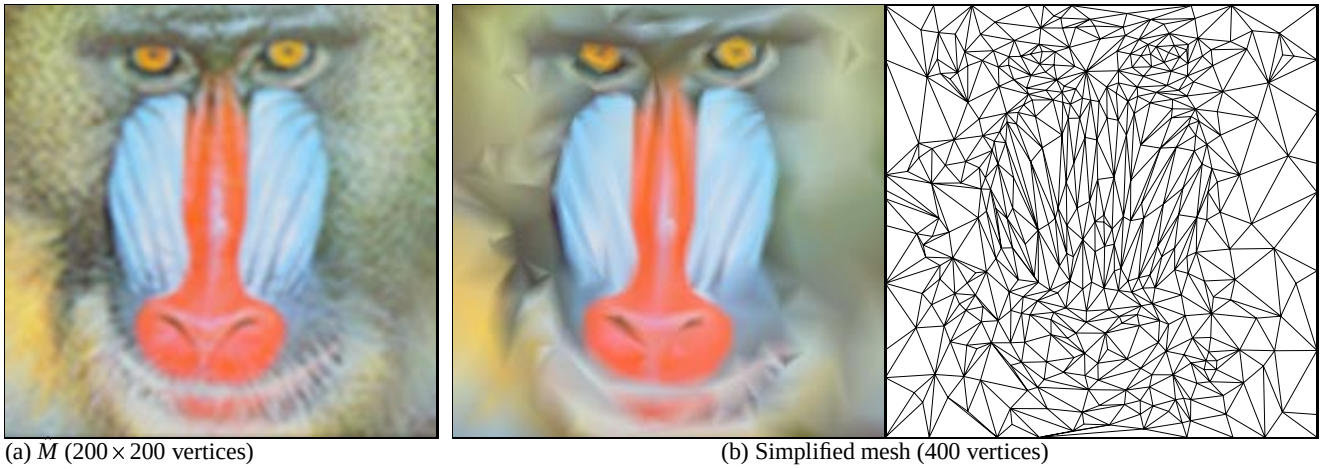
(a) α = 0.00 (b) α = 0.25 (c) α = 0.50 (d) α = 0.75 (e) α = 1.00

Figure 6: Example of a geomorph $M^G(\alpha)$ defined between $M^G(0) \doteq M^{175}$ (with 500 faces) and $M^G(1) = M^{425}$ (with 1,000 faces).



(a) Using conditions (1) and (2); 9,462 faces (b) Using conditions (1') and (2); 12,169 faces

Figure 7: Example of selective refinement within the view frustum (indicated in orange).



(a) $\hat{M}$ (200×200 vertices) (b) Simplified mesh (400 vertices)

Figure 8: Demonstration of minimizing E_{scalar}: simplification of a mesh with trivial geometry (a square) but complex scalar attribute field. ($\hat{M}$ is a mesh with regular connectivity whose vertex colors correspond to the pixels of an image.)

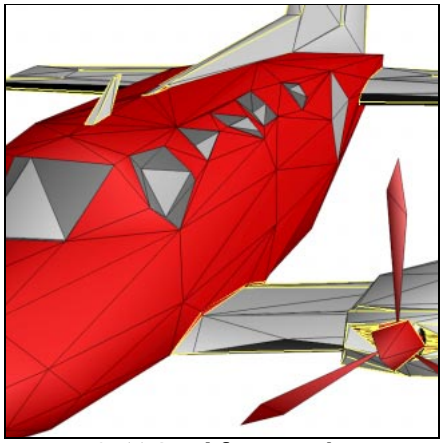


Figure 9: (a) Simplification without E_{disc}

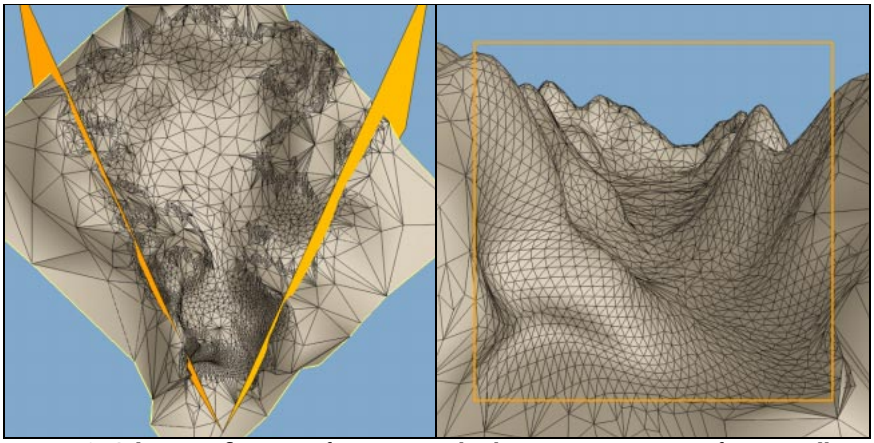
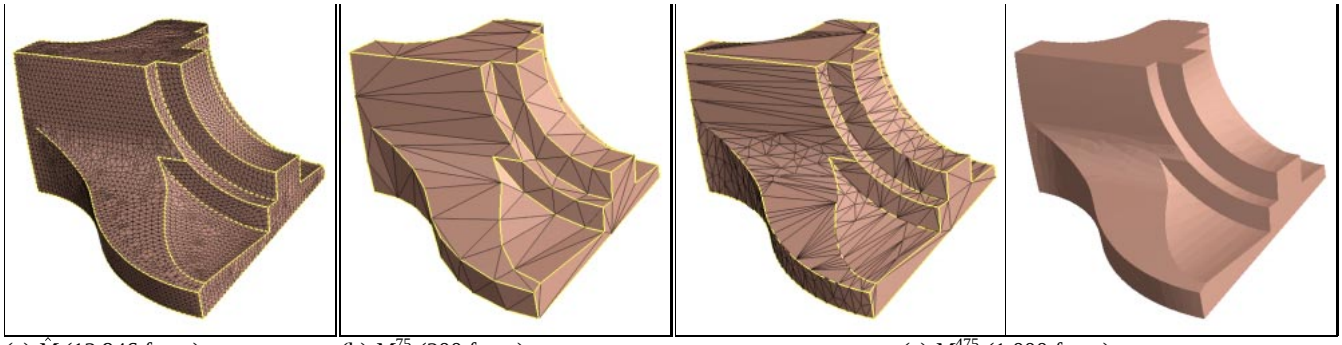
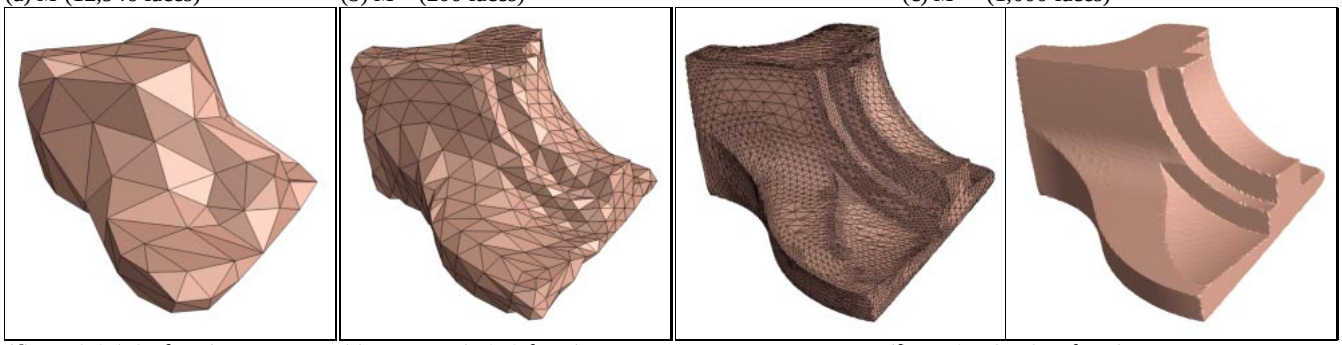


Figure 10: Selective refinement of a terrain mesh taking into account view frustum, silhouette regions, and projected screen size of faces (7,438 faces).

Figure 11: Simplification of a radiosity solution; left: original mesh (150,983 faces); right: simplified mesh (10,000 faces).



(a) $\hat{M}$ (12,946 faces) (b) M^{75} (200 faces) (c) M^{475} (1,000 faces)



(d) ϵ = 9.0 (192 faces) (e) ϵ = 2.75 (1,070 faces) (f) ϵ = 0.1 (15,842 faces)

Figure 12: Approximations of a mesh $\hat{M}$ using (b–c) the PM representation, and (d–f) the MRA scheme of Eck et al. [7]. As demonstrated, MRA cannot recover $\hat{M}$ exactly, cannot deal effectively with surface creases, and produces approximating meshes of inferior quality.

View-Dependent Refinement of Progressive Meshes

Hugues Hoppe
Microsoft Research

ABSTRACT

Level-of-detail (LOD) representations are an important tool for real-time rendering of complex geometric environments. The previously introduced *progressive mesh* representation defines for an arbitrary triangle mesh a sequence of approximating meshes optimized for view-independent LOD. In this paper, we introduce a framework for selectively refining an arbitrary progressive mesh according to changing view parameters. We define efficient refinement criteria based on the view frustum, surface orientation, and screen-space geometric error, and develop a real-time algorithm for incrementally refining and coarsening the mesh according to these criteria. The algorithm exploits view coherence, supports frame rate regulation, and is found to require less than 15% of total frame time on a graphics workstation. Moreover, for continuous motions this work can be amortized over consecutive frames. In addition, smooth visual transitions (geomorphs) can be constructed between any two selectively refined meshes.

A number of previous schemes create view-dependent LOD meshes for height fields (e.g. terrains) and parametric surfaces (e.g. NURBS). Our framework also performs well for these special cases. Notably, the absence of a rigid subdivision structure allows more accurate approximations than with existing schemes. We include results for these cases as well as for general meshes.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation - Display algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - surfaces and object representations.

Additional Keywords: mesh simplification, level-of-detail, multiresolution representations, dynamic tessellation, shape interpolation.

1 INTRODUCTION

Rendering complex geometric models at interactive rates is a challenging problem in computer graphics. While rendering performance is continually improving, significant gains are obtained by adapting the complexity of a model to its contribution to the rendered image. The ideal solution would be to efficiently determine the coarsest model that satisfies some perceptual image qualities. One common heuristic technique is to author several versions of a model at various *levels of detail* (LOD); a detailed triangle mesh is used when the object is close to the viewer, and coarser approximations are substituted as the object recedes [4, 8]. Such LOD meshes can be computed automatically using mesh simplification techniques (e.g. [5, 10, 19, 21]). The recently introduced *progressive mesh* (PM) representation [10] captures a continuous sequence of meshes optimized for view-independent LOD control, and allows fast traversal of the sequence at runtime.

Sets or sequences of view-independent LOD meshes are appropriate for many applications, but difficulties arise when rendering large-scale models, such as environments, that may surround the viewer:

- Many faces of the model may lie outside the view frustum and thus do not contribute to the image (Figure 12a). While these faces are typically culled early in the rendering pipeline, this processing incurs a cost.
- Similarly, it is often unnecessary to render faces oriented away from the viewer, and such faces are usually culled using a "back-facing" test, but again at a cost.
- Within the view frustum, some regions of the model may lie much closer to the viewer than others. View-independent LOD meshes fail to provide the appropriate level of detail over the entire model (e.g. as does the mesh in Figure 12b).

Some of these problems can be addressed by representing a graphics scene as a hierarchy of meshes. Parts of the scene outside the view frustum can then be removed efficiently using hierarchical culling, and LOD can be adjusted independently for each mesh in the hierarchy [4, 8]. However, establishing such hierarchies on continuous surfaces is a challenging problem. For instance, if a terrain mesh (Figure 11d) is partitioned into blocks, and these blocks are rendered at different levels of detail, one has to address the problem of cracks between the blocks [14]. In addition, the block boundaries are unlikely to correspond to natural features in the surface, resulting in suboptimal approximations. Similar problems also arise in the adaptive tessellation of smooth parametric surfaces [1, 13, 18].

Specialized schemes have been presented to adaptively refine meshes for the cases of height fields and parametric surfaces, as summarized in Section 2.1. In this paper, we offer a general runtime LOD framework for selectively refining arbitrary meshes according to changing view parameters. A similar approach was developed independently by Xia and Varshney [24]; their scheme is summarized and compared in Section 2.3.

The principal contributions of this paper are:

- It presents a framework for real-time selective refinement of arbitrary progressive meshes (Section 3).
- It defines fast view-dependent refinement criteria involving the view frustum, surface orientation, and screen-space projected error (Section 4).
- It presents an efficient algorithm for incrementally adapting the mesh refinement based on these criteria (Section 5). The algorithm exploits view coherence, supports frame rate regulation, and may be amortized over consecutive frames. To reduce popping, geomorphs can be constructed between any two selectively refined meshes.

Email: hhoppe@microsoft.com
Web: http://research.microsoft.com/∼hoppe/

- It shows that triangle strips can be generated for efficient rendering even though the mesh connectivity is irregular and dynamic (Section 6).
- Finally, it demonstrates the framework's effectiveness on the important special cases of height fields and tessellated parametric surfaces, as well as on general meshes (Section 8).

Notation We denote a triangle mesh M as a tuple (V, F), where V is a set of vertices v_j with positions $\mathbf{v}_j \in \mathbf{R}^3$, and F is a set of ordered vertex triples $\{v_j, v_k, v_l\}$ specifying vertices of triangle faces in counter-clockwise order. The *neighborhood* of a vertex v, denoted N_v, refers to the set of faces adjacent to v.

2 RELATED WORK

2.1 View-dependent LOD for domains in $\mathbf{R}^2$

Previous view-dependent refinement methods for domains in $\mathbf{R}^2$ fall into two categories: height fields and parametric surfaces.

Although there exist numerous methods for simplifying height fields, only a subset support efficient view-dependent LOD. These are based on hierarchical representations such as grid quadtrees [14, 23], quaternary triangular subdivisions [15], and more general triangulation hierarchies [3, 6, 20]. (The subdivision approach of [15] generalizes to 2-dimensional domains of arbitrary topological type.) Because quadtrees and quaternary subdivisions are based on a regular subdivision structure, the view-dependent meshes created by these schemes have constrained connectivities, and therefore require more polygons for a given accuracy than so-called *triangulated irregular networks* (TIN's). It was previously thought that dynamically adapting a TIN at interactive rates would be prohibitively expensive [14]. In this paper we demonstrate real-time modification of highly adaptable TIN's. Moreover, our framework extends to arbitrary meshes.

View-dependent tessellation of parametric surfaces such as NURBS requires fairly involved algorithms to deal with parameter step sizes, trimming curves, and stitching of adjacent patches [1, 13, 18]. Most real-time schemes sample a regular grid in the parametric domain of each patch to exploit fast forward differencing and to simplify the patch stitching process. Our framework allows real-time adaptive tessellations that adapt to surface curvature and view parameters.

2.2 Review of progressive meshes

In the PM representation [10], an arbitrary mesh $\hat{M}$ is simplified through a sequence of *n edge collapse* transformations (*ecol* in Figure 1) to yield a much simpler base mesh M^0 (see Figure 11):

$$(\hat{M} = M^n) \xrightarrow{ecol_{n-1}} \ldots \xrightarrow{ecol_1} M^1 \xrightarrow{ecol_0} M^0 .$$

Because each *ecol* has an inverse, called a *vertex split* transformation, the process can be reversed:

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \ldots \xrightarrow{vsplit_{n-1}} (M^n = \hat{M}) .$$

The tuple $(M^0, \{vsplit_0, \ldots, vsplit_{n-1}\})$ forms a PM representation of $\hat{M}$. Each vertex split, parametrized as $vsplit(v_s, v_l, v_r, v_t, f_l, f_r)$, modifies the mesh by introducing one new vertex v_t and two new faces $f_l = \{v_s, v_t, v_l\}$ and $f_r = \{v_s, v_r, v_t\}$ as shown in Figure 1. The resulting sequence of meshes $M^0, \ldots, M^n = \hat{M}$ is effective for view-independent LOD control (Figure 11). In addition, smooth visual transitions (*geomorphs*) can be constructed between any two meshes in this sequence.

To create view-dependent approximations, our earlier work [10] describes a scheme for selectively refining the mesh based on a user-specified query function $\mathsf{qrefine}(v_s)$. The basic idea is to traverse the

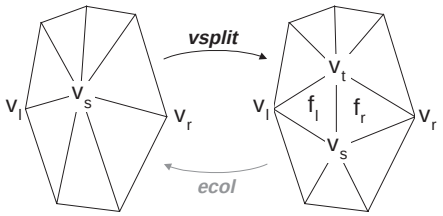


Figure 1: Original definitions of the refinement (*vsplit*) and coarsening (*ecol*) transformations.

$vsplit_i$ records in order, but to only perform $vsplit_i(v_{s_i}, v_{l_i}, v_{r_i}, \ldots)$ if

(1) $vsplit_i$ is a *legal* transformation, that is, if the vertices $\{v_{s_i}, v_{l_i}, v_{r_i}\}$ satisfy some conditions in the mesh refined so far, and

(2) $\mathsf{qrefine}(v_{s_i})$ evaluates to *true*.

The scheme is demonstrated with a view-dependent qrefine function whose criteria include the view frustum, proximity to silhouettes, and screen-projected face areas.

However, some major issues are left unaddressed. The qrefine function is not designed for real-time performance, and fails to measure screen-space geometric error. More importantly, no facility is provided for efficiently adapting the selectively refined mesh as the view parameters change.

2.3 Vertex hierarchies

Xia and Varshney [24] use *ecol*/*vsplit* transformations to create a simplification hierarchy that allows real-time selective refinement. Their approach is to precompute for a given mesh $\hat{M}$ a *merge tree* bottom-up as follows. First, all vertices $\hat{V}$ are entered as leaves at level 0 of the tree. Then, for each level $l \geq 0$, a set of *ecol* transformations is selected to merge pairs of vertices, and the resulting proper subset of vertices is promoted to level $l+1$. The *ecol* transformations in each level are chosen based on edge lengths, but with the constraint that their neighborhoods do not overlap. The topmost level of the tree (or more precisely, forest) corresponds to the vertices of a coarse mesh M^0. (In some respects, this structure is similar to the subdivision hierarchy of [11].)

At runtime, selective refinement is achieved by moving a vertex front up and down through the hierarchy. For consistency of the refinement, an *ecol* or *vsplit* transformation at level l is only permitted if its neighborhood in the selectively refined mesh is identical to that in the precomputed mesh at level l; these additional dependencies are stored in the merge tree. As a consequence, the representation shares characteristics of quadtree-type hierarchies, in that only gradual change is permitted from regions of high refinement to regions of low refinement [24].

Whereas Xia and Varshney construct the hierarchy based on edge lengths and constrain the hierarchy to a set of levels with non-overlapping transformations, our approach is to let the hierarchy be formed by an unconstrained, geometrically optimized sequence of *vsplit* transformations (from an arbitrary PM), and to introduce as few dependencies as possible between these transformations, in order to minimize the complexity of approximating meshes.

Several types of view-dependent criteria are outlined in [24], including local illumination and screen-space projected edge length. In this paper we detail three view-dependent criteria. One of these measures screen-space surface approximation error, and therefore yields mesh refinement that naturally adapts to both surface curvature and viewing direction.

Another related scheme is that of Luebke [16], which constructs a vertex hierarchy using a clustering octree, and locally adapts the complexity of the scene by selectively coalescing the cluster nodes.

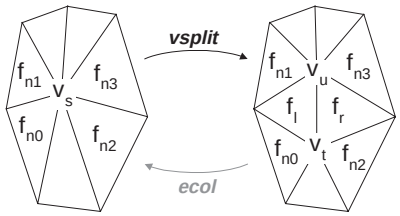


Figure 2: New definitions of *vsplit* and *ecol*.

3 SELECTIVE REFINEMENT FRAMEWORK

In this section, we show that a real-time selective refinement framework can be built upon an arbitrary PM.

Let a selectively refined mesh M^S be defined as the mesh obtained by applying to the base mesh M^0 a subsequence $S \subseteq \{0, \ldots, n-1\}$ of the PM *vsplit* sequence. As noted in Section 2.2, an arbitrary subsequence S may not correspond to a well-defined mesh, since a *vsplit* transformation is *legal* only if the current mesh satisfies some preconditions. These preconditions are analogous to the vertex or face dependencies found in most hierarchical representations [6, 14, 24]. Several definitions of *vsplit* legality have been presented (two in [10] and one in [24]); ours is yet another, which we will introduce shortly. Let $\mathcal{M}$ be the set of all meshes M^S produced from M^0 by a subsequence S of legal *vsplit* transformations.

To support incremental refinement, it is necessary to consider not just *vsplit*'s, but also *ecol*'s, and to perform these transformations in an order possibly different from that in the PM sequence. A major concern is that a selectively refined mesh should be unique, regardless of the sequence of (legal) transformations that leads to it, and in particular, it should still be a mesh in $\mathcal{M}$.

We first sought to extend the selective refinement scheme of [10] with a set of legality preconditions for *ecol* transformations, but were unable to form a consistent framework without overly restricting it. Instead, we began anew with modified definitions of *vsplit* and *ecol*, and found a set of legality preconditions sufficient for consistency, yet flexible enough to permit highly adaptable refinement. The remainder of this section presents these new definitions and preconditions.

New transformation definitions The new definitions of *vsplit* and *ecol* are illustrated in Figure 2. Note that their effects on the mesh are still the same; they are simply parametrized differently. The transformation $vsplit(v_s, v_t, v_u, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$, replaces the *parent* vertex v_s by two *children* v_t and v_u. Two new faces f_l and f_r are created between the two pairs of neighboring faces (f_{n0}, f_{n1}) and (f_{n2}, f_{n3}) adjacent to v_s. The edge collapse transformation $ecol(v_s, v_t, v_u, \ldots)$ has the same parameters as *vsplit* and performs the inverse operation. To support meshes with boundaries, face neighbors $f_{n0}, f_{n1}, f_{n2}, f_{n3}$ may have a special *nil* value, and vertex splits with $f_{n2} = f_{n3} = nil$ create only the single face f_l.

Let $\mathcal{V}$ denote the set of vertices in all meshes of the PM sequence. Note that $|\mathcal{V}|$ is approximately twice the number $|\hat{V}|$ of original vertices because of the vertex renaming in each *vsplit*. In contrast, the faces of a selectively refined mesh M^S are always a subset of the original faces $\hat{F}$. We number the vertices and faces in the order that they are created, so that $vsplit_i$ introduces the vertices $t_i = |V^0| + 2i + 1$ and $u_i = |V^0| + 2i + 2$. We say that a vertex or face is *active* if it exists in the selectively refined mesh M^S.

Vertex hierarchy As in [24], the parent-child relation on the vertices establishes a vertex hierarchy (Figure 3), and a selectively refined mesh corresponds to a "vertex front" through this hierarchy (e.g. M^0 and $\hat{M}$ in Figure 3). Our vertex hierarchy differs in two respects. First, vertices are renamed as they are split, and this renaming contributes to the refinement dependencies. Second, the hierarchy is constructed top-down after loading a PM using a simple traversal of the *vsplit* records. Although our hierarchies may be unbalanced, they typically have fewer levels than in [24] (e.g. 24 instead of 65 for the bunny) because they are unconstrained.

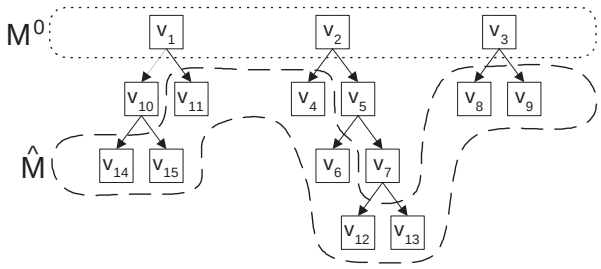


Figure 3: The vertex hierarchy on $\mathcal{V}$ forms a "forest", in which the root nodes are the vertices of the coarsest mesh (base mesh M^0) and the leaf nodes are the vertices of the most refined mesh (original mesh $\hat{M}$).

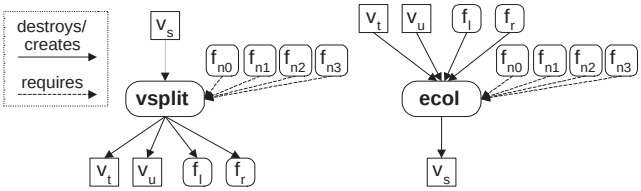


Figure 4: Preconditions and effects of *vsplit* and *ecol* transformations.

Preconditions We define a set of preconditions for *vsplit* and *ecol* to be legal (refer to Figure 4).

A $vsplit(v_s, v_t, v_u, \ldots)$ transformation is legal if

(1) v_s is an active vertex, and

(2) the faces $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$ are all active faces.

An $ecol(v_s, v_t, v_u, \ldots)$ transformation is legal if

(1) v_t and v_u are both active vertices, and

(2) the faces adjacent to f_l and f_r are $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$, in the configuration of Figure 2.

Properties Let $\mathcal{M}^\star$ be the set of meshes obtained by transitive closure of legal *vsplit* and *ecol* transformations from M^0 (or equivalently from $\hat{M}$ since the PM sequence $M^0 \longleftrightarrow \hat{M}$ is legal). For any mesh $M = (V, F) \in \mathcal{M}^\star$, we observe the following properties:[1]

- If $vsplit(v_s, v_t, v_u, \ldots)$ is legal, then $\{f_{n0}, f_{n1}\}$ and $\{f_{n2}, f_{n3}\}$ must be pairwise adjacent and adjacent to v_s as in Figure 2.
- If the active vertex front lies below $ecol(v_s, v_t, v_u, \ldots)$ (i.e. $f_l, f_r \in F$), then $\{f_{n0}, f_{n1}, f_{n2}, f_{n3}\}$ must all be active.
- $M \in \mathcal{M}$, i.e. $M = M^S$ for some subsequence S, i.e. $\mathcal{M}^\star = \mathcal{M}$.
- $M = M^S$ is identical to the mesh obtained by applying to $\hat{M}$ the complement subsequence $\{n-1, \ldots, 0\} \setminus S$ of *ecol* transformations, which are legal.

Implementation To make these ideas more concrete, Figure 5 lists the C++ data structures used in our implementation. A selectively refinable mesh consists of an array of vertices and an array of faces. Of these vertices and faces, only a subset are active, as specified by two doubly-linked lists that thread through a subset of

[1] Although these properties have held for the numerous experiments we have performed, we unfortunately do not have formal proofs for them as yet.

```
struct ListNode {              // Node possibly on a linked list
    ListNode* next;            // 0 if this node is not on the list
    ListNode* prev;
};
struct Vertex {
    ListNode active;           // list stringing active vertices V
    Point point;
    Vector normal;
    Vertex* parent;            // 0 if this vertex is in M^0
    Vertex* vt;                // 0 if this vertex is in M̂; (vu=vt+1)
 // Remaining fields encode vsplit information, defined if vt ≠ 0.
    Face* fl;                  // (fr=fl+1)
    Face* fn[4];               // required neighbors fn0, fn1, fn2, fn3
    RefineInfo refine_info;    // defined in Section 4
};
struct Face {
    ListNode active;           // list stringing active faces F
    int matid;                 // material identifier
 // Remaining fields are used if the face is active.
    Vertex* vertices[3];       // ordered counter-clockwise
    Face* neighbors[3];        // neighbors[i] across from vertices[i]
};
struct SRMesh {                // Selectively refinable mesh
    Array<Vertex> vertices;    // set V of all vertices
    Array<Face> faces;         // set F̂ of all faces
    ListNode active_vertices;  // head of list V ⊆ V
    ListNode active_faces;     // head of list F ⊆ F̂
};
```

Figure 5: Principal C++ data structures.

the records. In the Vertex records, the fields *parent* and *vt* encode the vertex hierarchy of Figure 3. If a vertex can be split, its *fl* and *fn*[0..3] fields encode the remaining parameters of the *vsplit* (and hence the dependencies of Figure 4). Each Face record contains links to its current vertices, links to its current face neighbors, and a material identifier used for rendering.

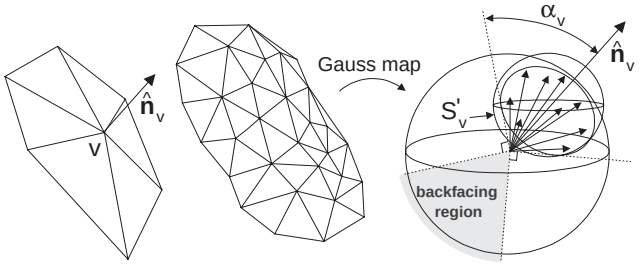
4 REFINEMENT CRITERIA

In this section, we describe a query function qrefine(v_s) that determines whether a vertex v_s should be split based on the current view parameters. As outlined below, the function uses three criteria: the view frustum, surface orientation, and screen-space geometric error. Because qrefine is often evaluated thousands of times per frame, it has been designed to be fast, at the expense of a few simplifying approximations where noted.

```
function qrefine(v_s)
    // Refine only if it affects the surface within the view frustum.
    if outside_view_frustum(v_s) return false
    // Refine only if part of the affected surface faces the viewer.
    if oriented_away(v_s) return false
    // Refine only if screen-projected error exceeds tolerance τ.
    if screen_space_error(v_s) ≤ τ return false
    return true
```

View frustum This first criterion seeks to coarsen the mesh outside the view frustum in order to reduce graphics load. Our approach is to compute for each vertex $v \in \mathcal{V}$ the radius r_v of a sphere centered at $\mathbf{v}$ that bounds the region of $\hat{M}$ supported by v and all its descendants. We let qrefine(v) return *false* if this bounding sphere lies completely outside the view frustum.

The radii r_v are computed after a PM representation is loaded into memory using a bounding sphere hierarchy as follows. First, we compute for each $v \in \hat{V}$ (the leaf nodes of the vertex hierarchy) a sphere S_v that bounds its adjacent vertices in $\hat{M}$. Next, we perform a postorder traversal of the vertex hierarchy (by scanning the *vsplit* sequence backwards) to assign each parent vertex v_s the smallest sphere S_{v_s} that bounds the spheres S_{v_t}, S_{v_u} of its two children. Finally, since the resulting spheres S_v are not centered on the vertices, we compute at each vertex v the radius r_v of a larger sphere centered at $\mathbf{v}$ that bounds S_v.



(a) N_v (b) region of $\hat{M}$ (c) S^2

Figure 6: Illustration of (a) the neighborhood of v, (b) the region in $\hat{M}$ affected by v, and (c) the space of normals over that region and the cone of normals that bounds it.

Since the view frustum is a 4-sided semi-infinite pyramid, a sphere of radius r_v centered at $\mathbf{v} = (v_x, v_y, v_z)$ lies outside the frustum if

$$a_i v_x + b_i v_y + c_i v_z + d_i < -r_v \quad \text{for any } i = 1 \ldots 4$$

where each linear functional $a_i x + b_i y + c_i z + d_i$ measures the signed Euclidean distance to a side of the frustum. Selective refinement based solely on the view frustum is demonstrated in Figure 12a.

Surface orientation The purpose of the second criterion is to coarsen regions of the mesh oriented away from the viewer, again to reduce graphics load. Our approach is analogous to the view frustum criterion, except that we now consider the space of normals over the surface (the Gauss map) instead of the surface itself. The space of normals is a subset of the unit sphere $S^2 = \{\mathbf{p} \in \mathbf{R}^3 : \|\mathbf{p}\| = 1\}$; for a triangle mesh $\hat{M}$, it consists of a discrete set of points, each corresponding to the normal of a triangle face of $\hat{M}$.

For each vertex v, we bound the space of normals associated with the region of $\hat{M}$ supported by v and its descendants, using a *cone of normals* [22] defined by a semiangle α_v about the vector $\hat{\mathbf{n}}_v$ = *v.normal* (Figure 6). The semiangles α_v are computed after a PM representation is loaded into memory using a *normal space hierarchy* [12]. As before, we first hierarchically compute at each vertex v a sphere S'_v that bounds the associated space of normals. Next, we compute at each vertex v the semiangle α_v of a cone about $\hat{\mathbf{n}}_v$ that bounds the intersection of S'_v and S^2. We let $\alpha_v = \frac{\pi}{2}$ if no bounding cone (with $\alpha_v < \frac{\pi}{2}$) exists.

Given a viewpoint $\mathbf{e}$, it is unnecessary to split v if $\mathbf{e}$ lies in the *backfacing region* of v, that is, if

$$\frac{\mathbf{a}_v - \mathbf{e}}{\|\mathbf{a}_v - \mathbf{e}\|} \cdot \hat{\mathbf{n}}_v > \sin \alpha_v \ ,$$

where $\mathbf{a}_v$ is a cone *anchor point* that takes into account the geometric bounding volume S_v (see [22] for details). However, to improve both space and time efficiency, we approximate $\mathbf{a}_v$ by $\mathbf{v}$ (it amounts to a parallel projection approximation [13]), and instead use the test

$$(\mathbf{v} - \mathbf{e}) \cdot \hat{\mathbf{n}}_v > 0 \quad \textbf{and} \quad ((\mathbf{v} - \mathbf{e}) \cdot \hat{\mathbf{n}}_v)^2 > \|\mathbf{v} - \mathbf{e}\|^2 \sin^2 \alpha_v \ .$$

The effect of this test is seen in Figures 13c, 14, and 16c, where the backfacing regions of the meshes are kept coarse.

Screen-space geometric error The goal of the third criterion is to adapt the mesh refinement such that the distance between the approximate surface M and the original $\hat{M}$, when projected on the screen, is everywhere less than a screen-space tolerance τ.

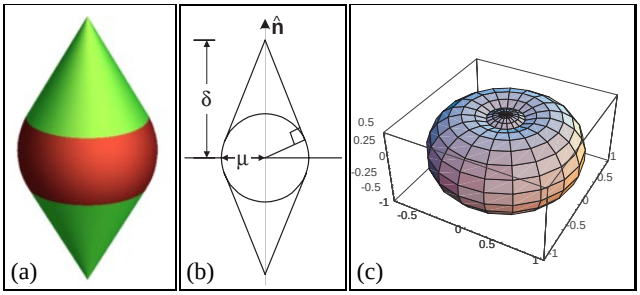


Figure 7: Illustration of (a) the deviation space $D_{\hat{\mathbf{n}}}(\mu, \delta)$, (b) its cross-section, and (c) the extent of its screen-space projection as a function of viewing angle (with $\mu = 0.5$ and $\delta = 1$).

To determine whether a vertex $v \in \mathcal{V}$ should be split, we seek a measure of the deviation between its current neighborhood N_v (the set of faces adjacent to v) and the corresponding region $\hat{N}_v$ in $\hat{M}$. One quantitative measure is the *Hausdorff distance* $\mathcal{H}(N_v, \hat{N}_v)$, defined as the smallest scalar r such that any point on N_v is within distance r of a point on $\hat{N}_v$, and vice versa. Mathematically, $\mathcal{H}(N_v, \hat{N}_v)$ is the smallest r for which $N_v \subset \hat{N}_v \oplus B(r)$ and $\hat{N}_v \subset N_v \oplus B(r)$ where $B(r)$ is the closed ball of radius r and $\oplus$ denotes the Minkowski sum[2]. If $\mathcal{H}(N_v, \hat{N}_v) = r$, the screen-space approximation error is bounded by the screen-space projection of the ball $B(r)$.

If N_v and $\hat{N}_v$ are similar and approximately planar, a tighter distance bound can be obtained by replacing the ball $B(r)$ in the above definition by a more general *deviation space* D. For instance, Lindstrom et al. [14] record deviation of height fields (graphs of functions over the $\mathbf{xy}$ plane) by associating to each vertex a scalar value δ representing a vertical deviation space $D_{\hat{\mathbf{z}}}(\delta) = \{h\,\hat{\mathbf{z}} : -\delta \le h \le \delta\}$. The main advantage of using $D_{\hat{\mathbf{z}}}(\delta)$ is that its screen-space projection vanishes as its principal axis $\hat{\mathbf{z}}$ becomes parallel to the viewing direction, unlike the corresponding $B(\delta)$.

To generalize these ideas to arbitrary surfaces, we define a deviation space $D_{\hat{\mathbf{n}}}(\mu, \delta)$ shown in Figure 7a–b. The motivation is that most of the deviation is orthogonal to the surface and is captured by a directional component $\delta\hat{\mathbf{n}}$, but a uniform component μ may be required when $\hat{N}_v$ is curved. The uniform component also allows accurate approximation of discontinuity curves (such as surface boundaries and material boundaries) whose deviations are often tangent to the surface. The particular definition of $D_{\hat{\mathbf{n}}}(\mu, \delta)$ corresponds to the shape whose projected radius along a direction $\vec{\mathbf{v}}$ has the simple formula $\max(\mu, \delta\|\hat{\mathbf{n}} \times \vec{\mathbf{v}}\|)$. As shown in Figure 7c, the graph of this radius as a function of view direction has the shape of a sphere of radius μ unioned with a "bialy" [14] of radius δ.

During the construction of a PM representation, we precompute μ_v, δ_v for deviation space $D_{\hat{\mathbf{n}}_v}(\mu_v, \delta_v)$ at each vertex $v \in \mathcal{V}$ as follows. After each $ecol(v_s, v_t, v_u, \ldots)$ transformation is applied, we estimate the deviation between N_{v_s} and $\hat{N}_{v_s}$ by examining the residual error vectors $E = \{\mathbf{e}_i\}$ from a dense set of points X sampled on $\hat{M}$ that locally project onto N_{v_s}, as explained in more detail in [10]. We use $\max_{\mathbf{e}_i \in E}(\mathbf{e}_i \cdot \hat{\mathbf{n}}_v) \,/\, \max_{\mathbf{e}_i \in E} \|\mathbf{e}_i \times \hat{\mathbf{n}}_v\|$ to fix the ratio δ_v/μ_v, and find the smallest $D_{\hat{\mathbf{n}}_v}(\mu_v, \delta_v)$ with that ratio that bounds E. Alternatively, other simplification schemes such as [2, 5, 9] could be adapted to obtain deviation spaces with guaranteed bounds.

Note that the computation of μ_v, δ_v does not measure parametric distortion. This is appropriate for texture-mapped surfaces if the texture is geometrically projected or "wrapped". If instead, vertices were to contain explicit texture coordinates, the residual computation could be altered to measure deviation parametrically.

Given viewpoint $\mathbf{e}$, screen-space tolerance τ (as a fraction of viewport size), and field-of-view angle φ, qrefine(v) returns *true* if the screen-space projection of $D_{\hat{\mathbf{n}}_v}(\mu_v, \delta_v)$ exceeds τ, that is, if

$$\max\left(\mu_v,\ \delta_v \left\| \hat{\mathbf{n}}_v \times \frac{\mathbf{v}-\mathbf{e}}{\|\mathbf{v}-\mathbf{e}\|} \right\|\right) / \|\mathbf{v}-\mathbf{e}\| \ \ge\ \left(2\cot\frac{\varphi}{2}\right)\tau\ .$$

For efficiency, we use the equivalent test

$$\begin{aligned} \mu_v^2 &\ge \kappa^2 \|\mathbf{v}-\mathbf{e}\|^2 \quad \textbf{or} \\ \delta_v^2\left(\|\mathbf{v}-\mathbf{e}\|^2 - ((\mathbf{v}-\mathbf{e})\cdot\hat{\mathbf{n}}_v)^2\right) &\ge \kappa^2\|\mathbf{v}-\mathbf{e}\|^4\ , \end{aligned}$$

where $\kappa^2 = (2\cot\frac{\varphi}{2})^2\tau^2$ is computed once per frame. Note that the test reduces to that of [14] when $\mu_v = 0$ and $\hat{\mathbf{n}}_v = \hat{\mathbf{z}}$, and requires only a few more floating point operations in the general case. As seen in Figures 13b and 16b, our test naturally results in more refinement near the model silhouette where surface deviation is orthogonal to the view direction.

Our test provides only an approximate bound on the screen-space projected error, for a number of reasons. First, the test slightly underestimates error away from the viewport center, as pointed out in [14]. Second, a parallel projection assumption is made when projecting $D_{\hat{\mathbf{n}}}$ on the screen, as in [14]. Third, the neighborhood about v when evaluating qrefine(v) may be different from that in the PM sequence since M is selectively refined; thus the deviation spaces $D_{\hat{\mathbf{n}}}$ provide strict bounds only at the vertices themselves. Nonetheless, the criterion works well in practice, as demonstrated in Figures 12–16.

Implementation We store in each Vertex.RefineInfo record the four scalar values $\{-r_v, \sin^2\alpha_v, \mu_v^2, \delta_v^2\}$. Because the three refinement tests share several common subexpressions, evaluation of the complete qrefine function requires remarkably few CPU cycles on average (230 cycles per call as shown in Table 2).

5 INCREMENTAL SELECTIVE REFINEMENT ALGORITHM

We now present an algorithm for incrementally adapting a mesh within the selective refinement framework of Section 3, using the qrefine function of Section 4. The basic idea is to traverse the list of active vertices V before rendering each frame, and for each vertex $v \in V$, either leave it as is, split it, or collapse it. The core of the traversal algorithm is summarized below.

```
procedure adapt_refinement()
    for each v ∈ V
        if v.vt and qrefine(v)
            force_vsplit(v)
        else if v.parent and ecol_legal(v.parent) and
                not qrefine(v.parent)
            ecol(v.parent)      // (and reconsider some vertices)
procedure force_vsplit(v') {
    stack ← v'
    while v ← stack.top()
        if v.vt and v.fl ∈ F
            stack.pop()         // v was split earlier in the loop
        else if v ∉ V
            stack.push(v.parent)
        else if vsplit_legal(v)
            stack.pop()
            vsplit(v)           // (placing v.vt and v.vu next in list V)
        else for i ∈ {0...3}
            if v.fn[i] ∉ F
                // force vsplit that creates face v.fn[i]
                stack.push(v.fn[i].vertices[0].parent) [3]
```

[2]The Minkowski sum is simply $A \oplus B = \{\mathbf{a} + \mathbf{b} : \mathbf{a} \in A, \mathbf{b} \in B\}$.

[3]Implementation detail: the vertex that should be split to create an inactive face f is found in $f.vertices[0].parent$ because we always set both $f_l.vertices[0] = v_t$ and $f_r.vertices[0] = v_t$ when creating faces, thereby obviating the need for a Face.*parent* field.

We iterate through the doubly linked list of active vertices V. For any active vertex $v \notin \hat{M}$, if qrefine(v) evaluates to *true*, the vertex should be split. If *vsplit*(v) is not legal (i.e. if any of the faces $v.fn[0..3]$ are not active), a chain of other vertex splits are performed in order for *vsplit*(v) to become legal (procedure force_vsplit), namely those that introduce the faces $v.fn[0..3]$, and recursively, any others required to make those vertex splits legal.

For any active vertex $v \notin M^0$, if qrefine($v.parent$) returns *false*, the vertex v should be collapsed. However, this edge collapse is only performed if it is legal (i.e. if the sibling of v is also active and the neighboring faces of $v.parent.fl$ and $v.parent.fr$ match those of $v.parent.fn[0..3]$).

In short, the strategy is to force refinement when desired, but to coarsen only when possible. After a *vsplit* or *ecol* is performed, some vertices in the resulting neighborhood should be considered for further transformations. Since these vertices may have been previously visited in the traversal of V, we relocate them in the list to lie immediately after the list iterator. Specifically, following *vsplit*(v), we add $v.vt, v.vu$ after the iterator; and, following *ecol*($v.parent$), we add $v.parent$ and relocate v_l, v_r after the iterator (where v_l and v_r are the current neighbors of v as in Figure 1).

Time complexity The time complexity for adapt_refinement, transforming M^A into M^B, is $O(|V^A| + |V^B|)$ in the worst case since $M^A \rightarrow M^0 \rightarrow M^B$ could possibly require $O(|V^A|)$ *ecol*'s and $O(|V^B|)$ *vsplit*'s, each taking constant time. For continuous view changes, V^B is usually similar to V^A, and the simple traversal of the active vertex list is the bottleneck of the incremental refinement algorithm, as shown in Table 2. Note that the number $|V|$ of active vertices is typically much smaller than the number $|\hat{V}|$ of original vertices. The rendering process, which has the same time complexity ($|F| \simeq 2|V|$), in fact has a larger time constant. Indeed, adapt_refinement requires only about 14% of total frame time, as discussed in Section 8.

Regulation For a given PM and a constant screen-space tolerance τ, the number $|F|$ of active faces can vary dramatically depending on the view. Since both refinement times and rendering times are closely correlated to $|F|$, this leads to high variability in frame rates (Figure 9). We have implemented a simple scheme for regulating τ so as to maintain $|F|$ at a nearly constant level. Let m be the desired number of faces. Prior to calling adapt_refinement at time frame t, we set $\tau_t = \tau_{t-1}(|F_{t-1}|/m)$ where $|F_{t-1}|$ is the number of active faces in the previously drawn frame. As shown in Figure 10, this simple feedback control system exhibits good stability for our terrain flythrough. More sophisticated control strategies may be necessary for heterogeneous, irregular models. Direct regulation of frame rate could be attempted, but since frame rate is more sensitive to operating system "hiccups", it may be best achieved indirectly using a secondary, slower controller adjusting m.

Amortization Since the main loop of adapt_refinement is a simple traversal of the list V, we can distribute its work over consecutive frames by traversing only a fraction of V each frame. For slowly changing view parameters, this reduces the already low overhead of selective refinement while introducing few visual artifacts.

With amortization, however, regulation of $|F|$ through adjustment of τ becomes more difficult, since the response in $|F|$ may lag several frames. Our current strategy is to wait several frames until the entire list V has been traversed before making changes to τ. To reduce overshooting, we disallow *vsplit* refinement if the number of active faces reaches an upper limit (e.g. $|F| \geq 1.2m$). but do count the number of faces that would be introduced towards the next adjustment to τ. In the flythrough example of Figure 10, where the average frame rate is 7.2 frames/sec, amortization increases frame rate to 8 frames/sec.

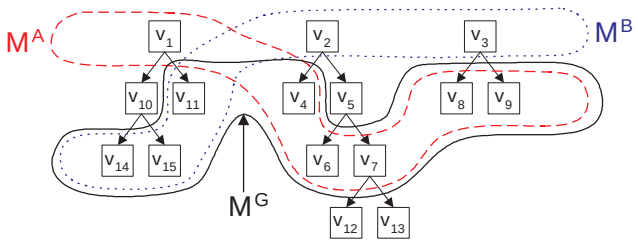


Figure 8: Illustration of two selectively refined meshes M^A and M^B, and of the mesh M^G used to geomorph between them.

Geomorphs The selective refinement framework also supports geomorphs between any two selectively refined meshes M^A and M^B. That is, one can construct a mesh $M^G(\alpha)$ whose vertices vary as a function of a parameter $0 \leq \alpha \leq 1$, such that $M^G(0)$ looks identical to M^A and $M^G(1)$ looks identical to M^B. The key is to first find a mesh M^G whose active vertex front is everywhere lower than or equal to that of M^A and M^B, as illustrated in Figure 8. Mesh $\hat{M}$ trivially satisfies this property, but a simpler mesh M^G is generally obtained by starting from either M^A or M^B and successively calling force_vsplit to advance the vertex front towards that of the other mesh. The mesh M^G has the property that its faces F^G are a superset of both F^A and F^B, and that any vertex $v_j \in V^G$ has a unique ancestor $v_{\rho^{G\rightarrow A}(j)} \in V^A$ and a unique ancestor $v_{\rho^{G\rightarrow B}(j)} \in V^B$. The geomorph $M^G(\alpha)$ is the mesh $(F^G, V^G(\alpha))$ with

$$\mathbf{v}_j^G(\alpha) = (1-\alpha)\mathbf{v}_{\rho^{G\rightarrow A}(j)} + (\alpha)\mathbf{v}_{\rho^{G\rightarrow B}(j)} .$$

In the case that M^B is the result of calling adapt_refinement on M^A, the mesh M^G can be obtained more directly. Instead of a single pass through V in adapt_refinement, we make two passes: a refinement pass $M^A \rightarrow M^G$ where only *vsplit* are considered, and a coarsening pass $M^G \rightarrow M^B$ where only *ecol* are considered. In each pass, we record the sequence of transformations performed, allowing us to backtrack through the inverse of the *ecol* sequence to recover the intermediate mesh M^G, and to construct the desired ancestry functions $\rho^{G\rightarrow A}$ and $\rho^{G\rightarrow B}$. Such a geomorph is demonstrated on the accompanying video. Because of view coherence, the number of vertices that require interpolation is generally smaller than the number of active vertices. More research is needed to determine the feasibility and usefulness of generating geomorphs at runtime.

6 RENDERING

Many graphics systems require triangle strip representations for optimal rendering performance [7]. Because the mesh connectivity in our incremental refinement scheme is dynamic, it is not possible to precompute triangle strips. We use a greedy algorithm to generate triangle strips at every frame, as shown in Figure 12e. Surprisingly, the algorithm produces strips of adequate length (on average, 10–15 faces per "generalized" triangle strip under IRIS GL, and about 4.2 faces per "sequential" triangle strip under OpenGL), and does so efficiently (Table 2).

The algorithm traverses the list of active faces F, and at any face not yet rendered, begins a new triangle strip. Then, iteratively, it renders the face, checks if any of its neighbor(s) has not yet been rendered, and if so continues the strip there. Only neighbors with the same material are considered, so as to reduce graphics state changes. To reduce fragmentation, we always favor continuing generalized triangle strips in a clockwise spiral (Figure 12e). When the strip reaches a dead end, traversal of the list F resumes. One bit of the Face.*matid* field is used as a boolean flag to record rendered faces; these bits are cleared using a quick second pass through F.

Recently, graphics libraries have begun to support interfaces for immediate-mode rendering of (V, F) mesh representations (e.g. Direct3D DrawIndexedPrimitive and OpenGL glArrayElementArrayEXT). Although not used in our current prototype, such interfaces may be ideal for rendering selectively refined meshes.

7 OPTIMIZING PM CONSTRUCTION FOR SELECTIVE REFINEMENT

The PM construction algorithm of [10] finds a sequence of *vsplit* refinement transformations optimized for accuracy, without regard to the shape of the resulting vertex hierarchy. We have experimented with introducing a small penalty function to the cost metric of [10] to favor balanced hierarchies in order to minimize unnecessary dependencies. The penalty for $ecol(v_t, v_u)$ is $c\,(n_{v_t}+n_{v_u})$ where n_v is the number of descendants of v (including itself) and c is a user-specified parameter. We find that a small value of c improves results slightly for some examples (i.e. reduces the number of faces for a given error tolerance τ), but that as c increases, the hierarchies become quadtree-like and the results worsen markedly (Figure 17). Our conclusion is that it is beneficial to introduce a small bias to favor balanced hierarchies in the absence of geometric preferences.

8 RESULTS

Timing results We constructed a PM representation of a Grand Canyon terrain mesh of 600^2 vertices (717,602 faces), and truncated this PM representation to 400,000 faces. This preprocessing requires several hours but is done off-line (Table 1). Loading this PM from disk and constructing the SRMesh requires less than a minute (most of it spent computing r_v and α_v). Figures 9 and 10 show measurements from a 3-minute real-time flythrough of the terrain without and with regulation, on an SGI Indigo2 Extreme (150MHz R4400 with 128MB of memory). The measurements show that the time spent in adapt_refinement is approximately 14% of total frame time. In the accompanying video, amortization is used to reduce this overhead to 8% of total frame time. For the flythrough of Figure 10, code profiling and system monitoring reveal the timing breakdown shown in Table 2. Note that triangle strip generation is efficient enough to keep CPU utilization below 100%; the graphics system is in fact the bottleneck. On another computer with the same CPU but with an Impact graphics system, the average frame rate increases from 7.2 to 14.0 frames/sec.

Space requirements Table 1 shows the disk space required to store the PM representations and associated deviation parameters; both are compressed using GNU gzip. Positions, normals, and deviation parameters are currently stored as floating point, and should be quantized to improve compression.

Since $|\mathcal{V}| \simeq 2|\hat{V}|$ and $|\hat{F}| \simeq 2|\hat{V}|$, memory requirement for SRMesh is $O(|\hat{V}|)$. The current implementation is not optimized for space, and requires about $224\,|\hat{V}|$ bytes. The memory footprint could be reduced as follows. Since only about half of all vertices $\mathcal{V}$ can be split, it would be best to store the split information (fl, $fn[0..3]$, *refine_info*) in a separate array of "Vsplit" records indexed by *vt*. If space is always allocated for 2 faces per *vsplit*, the Vertex.*fl* field can be deleted and instead computed from *vt*. Scalar values in the RefineInfo record can be quantized to 8 bits with an exponential map as in [14]. Coordinates of points and normals can be quantized to 16 bits. Material identifiers are unnecessary if the mesh has only one material. Overall, these changes would reduce memory requirements down to about $140\,|\hat{V}|$ bytes.

For the case of height fields, the memory requirement per vertex far exceeds that of regular grid schemes [14]. However, the fully detailed mesh $\hat{M}$ may have arbitrary connectivity, and may therefore be obtained by pre-simplifying a given grid representation, possibly

Table 1: Statistics for the various data sets.

Model	Fully detailed $\hat{M}$		Disk (MB)		Mem.	V hier.	Constr.
	$\lvert\hat{V}\rvert$	$\lvert\hat{F}\rvert$	PM	$\{\mu,\delta\}$	(MB)	height	(mins)
canyon_{200}	40,000	79,202	1.3	0.3	8.9	29	47
canyon_{400}	160,000	318,402	5.0	1.1	35.8	32	244
canyon_{600}	360,000	717,602	11.0	2.6	80.6	36	627
" trunc.	200,600	400,000	6.6	1.5	44.9	35	627
sphere	9,902	19,800	0.3	0.1	2.2	19	11
teapot trunc.	5,090	10,000	0.2	0.0	1.1	20	12
gameguy	21,412	42,712	0.8	0.2	4.8	26	30
bunny	34,835	69,473	1.2	0.2	7.8	24	51

Table 2: CPU utilization (on a 150MHz MIPS R4400).

	procedure	% of frame time	cycles/call
User	adapt_refinement	14 %	-
	(vsplit)	(0 %)	2200
	(ecol)	(1 %)	4000
	(qrefine)	(4 %)	230
	render (tstrip/face)	26 %	600
	GL library	19 %	-
System	OS + graphics	21 %	-
	CPU idle	20 %	-

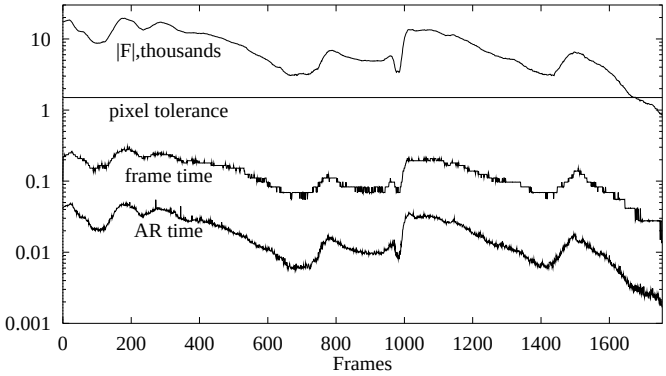


Figure 9: Measurements in flythrough for constant $\tau = 0.25\%$ (1.5 pixels in 600^2 window). From top: number of faces in thousands, τ in pixels, frame times and adapt_refinement times in seconds.

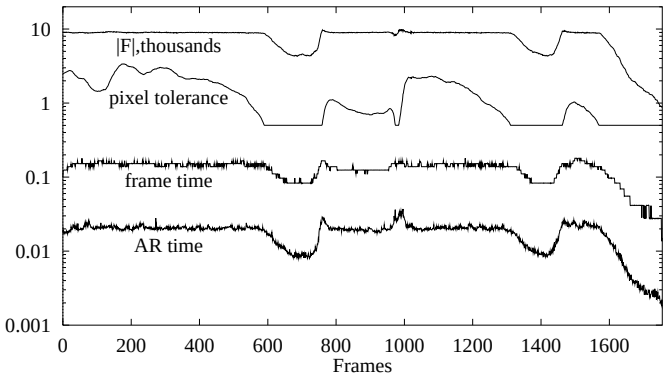


Figure 10: Same but with regulation to maintain $|F| \simeq 9000$. (τ is never allowed below 0.5 pixels.)

by an order of magnitude or more, without significant loss of accuracy. This pre-simplification may be achieved by simply truncating the PM representation, either at creation time or at load time.

Applications that use height fields often require efficient geometric queries, such as point search. Because the vertex hierarchies in our framework have $O(\log n)$ height in the average case (this can be enforced using the approach in Section 7), such queries can be performed in $O(\log n)$ time by iteratively calling force_vsplit on vertices in the neighborhood of the query point.

Parametric surfaces Our framework offers a novel approach to real-time adaptive tessellation of parametric surfaces. As a precomputation, we first obtain a dense tessellation of the surface, then construct from this dense mesh a PM representation, and finally truncate the PM sequence to a desired level of maximum accuracy. At runtime, we selectively refine this truncated PM representation according to the viewpoint (Figure 14). The main drawback of this approach is that the resolution of the most detailed tessellation is fixed a priori. However, the benefits include simplicity of runtime implementation (no trimming or stitching), efficiency (incremental, amortized work), and most importantly, high adaptability of the tessellations (accurate TIN's whose connectivities adapt both to surface curvature and to the viewpoint).

General meshes Figures 15 and 16 demonstrate selective refinement applied to general meshes. We expect this to be of practical use for rendering complex models and environments that do not conveniently admit scene hierarchies.

9 SUMMARY AND FUTURE WORK

We have introduced an efficient framework for selectively refining arbitrary progressive meshes, developed fast view-dependent refinement criteria, and presented an algorithm for incrementally adapting the approximating meshes according to these criteria. We have demonstrated real-time selective refinement on a number of meshes, including terrains, parametric surface tessellations, and general meshes. As the adaptive refinement algorithm exploits frame-to-frame coherence and is easily amortized, it consumes only a small fraction of total frame time. Because the selectively refined meshes stem from a geometrically optimized set of vertex split transformations with few dependencies, they quickly adapt to the underlying model, requiring fewer polygons for a given level of approximation than previous schemes.

There are a number of areas for future work, including:

- Memory management for large models, particularly terrains.
- Experimentation with runtime generation of geomorphs.
- Extension of refinement criteria to account for surface shading [24], or for surface velocity and proximity to gaze center [17].
- Adaptive refinement for animated models.
- Applications of selective refinement to collision detection.

ACKNOWLEDGMENTS

The Grand Canyon data is from the United States Geological Survey, with in-house processing by Chad McCabe of the Microsoft Geography Product Unit; the "gameguy" mesh is courtesy of Viewpoint DataLabs; the "bunny" is from the Stanford University Computer Graphics Laboratory. I also wish to thank Jed Lengyel, John Snyder, and Rick Szeliski for helpful comments, and Bobby Bodenheimer for useful discussions on control theory.

REFERENCES

[1] ABI-EZZI, S. S., AND SUBRAMANIAM, S. Fast dynamic tessellation of trimmed NURBS surfaces. *Computer Graphics Forum (Proceedings of Eurographics '94) 13*, 3 (1994), 107–126.

[2] BAJAJ, C., AND SCHIKORE, D. Error-bounded reduction of triangle meshes with multivariate data. *SPIE 2656* (1996), 34–45.

[3] CIGNONI, P., PUPPO, E., AND SCOPIGNO, R. Representation and visualization of terrain surfaces at variable resolution. In *Scientific Visualization '95* (1995), R. Scateni, Ed., World Scientific, pp. 50–68.

[4] CLARK, J. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10 (October 1976), 547–554.

[5] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., BROOKS, F., AND WRIGHT, W. Simplification envelopes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 119–128.

[6] DE FLORIANI, L., MARZANO, P., AND PUPPO, E. Multiresolution models for topographic surface description. *The Visual Computer 12*, 7 (1996), 317–345.

[7] EVANS, F., SKIENA, S., AND VARSHNEY, A. Optimizing triangle strips for fast rendering. In *Visualization '96 Proceedings* (1996), IEEE, pp. 319–326.

[8] FUNKHOUSER, T., AND SÉQUIN, C. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 247–254.

[9] GUÉZIEC, A. Surface simplification with variable tolerance. In *Proceedings of the Second International Symposium on Medical Robotics and Computer Assisted Surgery* (November 1995), pp. 132–139.

[10] HOPPE, H. Progressive meshes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 99–108.

[11] KIRKPATRICK, D. Optimal search in planar subdivisions. *SIAM Journal on Computing 12*, 1 (February 1983), 28–35.

[12] KUMAR, S., AND MANOCHA, D. Hierarchical visibility culling for spline models. In *Proceedings of Graphics Interface '96* (1996), pp. 142–150.

[13] KUMAR, S., MANOCHA, D., AND LASTRA, A. Interactive display of large-scale NURBS models. In *1995 Symposium on Interactive 3D Graphics* (1995), ACM SIGGRAPH, pp. 51–58.

[14] LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L., FAUST, N., AND TURNER, G. Real-time, continuous level of detail rendering of height fields. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 109–118.

[15] LOUNSBERY, M., DEROSE, T., AND WARREN, J. Multiresolution surfaces of arbitrary topological type. *ACM Transactions on Graphics 16*, 1 (January 1997), 34–73.

[16] LUEBKE, D. Hierarchical structures for dynamic polygonal simplification. TR 96-006, Department of Computer Science, University of North Carolina at Chapel Hill, 1996.

[17] OHSHIMA, T., YAMAMOTO, H., AND TAMURA, H. Gaze-directed adaptive rendering for interacting with virtual space. In *Proc. of IEEE 1996 Virtual Reality Annual Intnl. Symp.* (1996), pp. 103–110.

[18] ROCKWOOD, A., HEATON, K., AND DAVIS, T. Real-time rendering of trimmed surfaces. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (1989), vol. 23, pp. 107–116.

[19] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunii, Eds. Springer-Verlag, 1993, pp. 455–465.

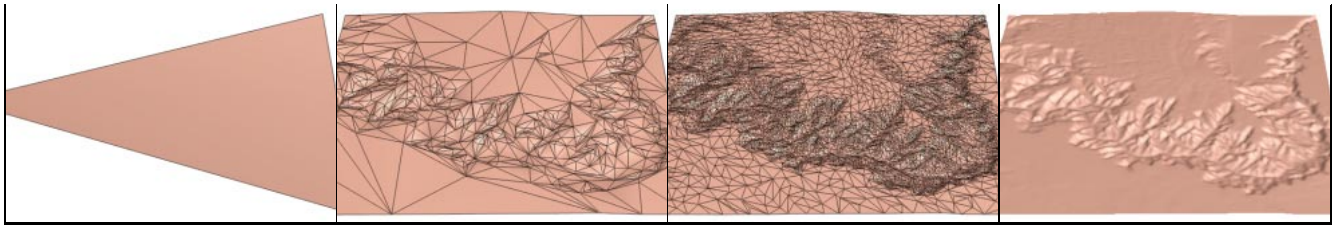
[20] SCARLATOS, L. L. A refined triangulation hierarchy for multiple levels of terrain detail. In *Proceedings, IMAGE V Conference* (June 1990), pp. 115–122.

[21] SCHROEDER, W., ZARGE, J., AND LORENSEN, W. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings) 26*, 2 (1992), 65–70.

[22] SHIRMAN, L., AND ABI-EZZI, S. The cone of normals technique for fast processing of curved patches. *Computer Graphics Forum (Proceedings of Eurographics '93) 12*, 3 (1993), 261–272.

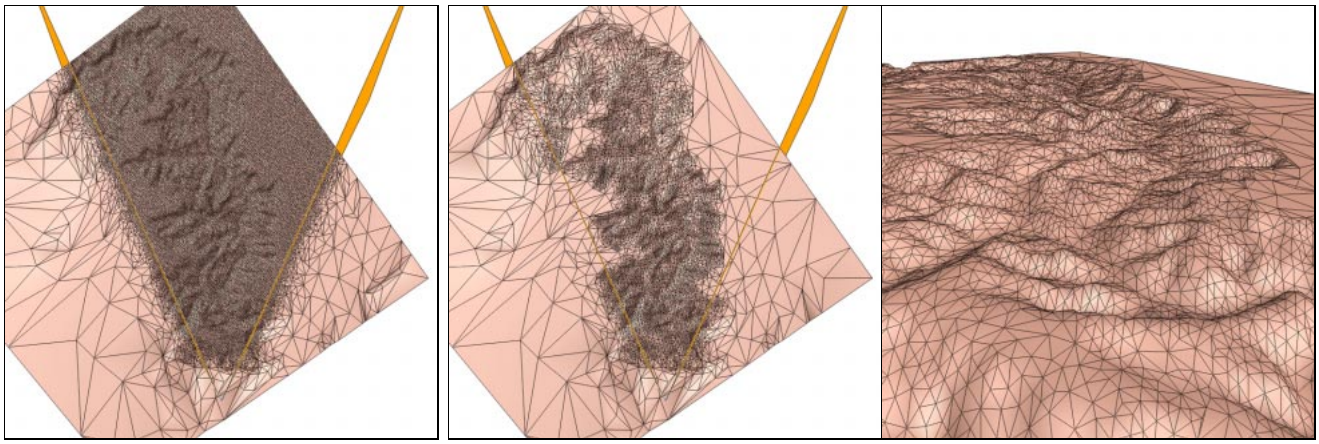
[23] TAYLOR, D. C., AND BARRETT, W. A. An algorithm for continuous resolution polygonalizations of a discrete surface. In *Proceedings of Graphics Interface '94* (1994), pp. 33–42.

[24] XIA, J., AND VARSHNEY, A. Dynamic view-dependent simplification for polygonal models. In *Visualization '96 Proceedings* (1996), IEEE, pp. 327–334.

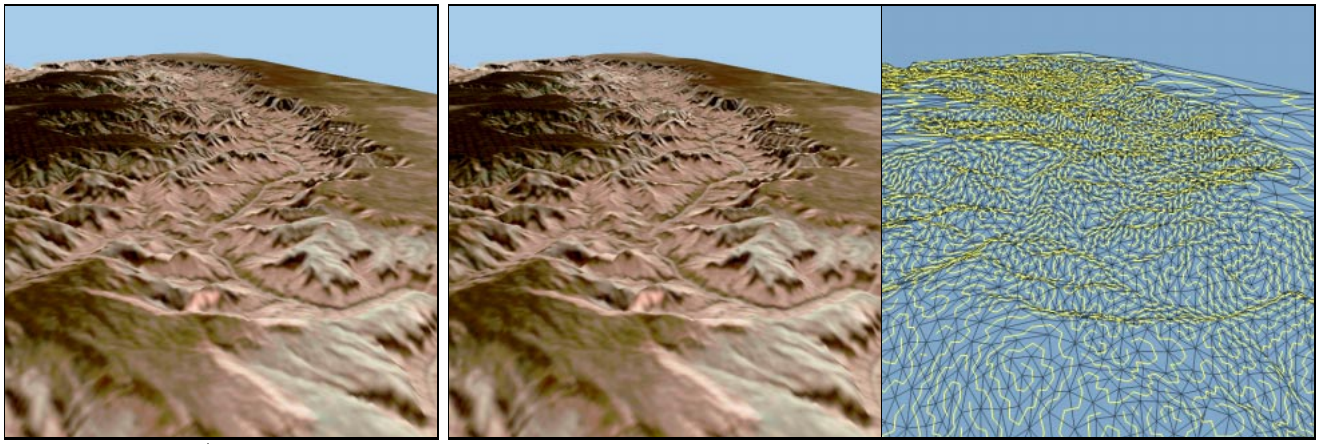


(a) Base mesh M^0 (1 face) (b) M^{514} (1,000 faces) (c) M^{5066} (10,000 faces) (d) $\hat{M} = M^n$ (79,202 faces)

Figure 11: The PM representation of a mesh $\hat{M}$ captures a continuous sequence of view-independent LOD meshes $M^0 \ldots M^n = \hat{M}$.

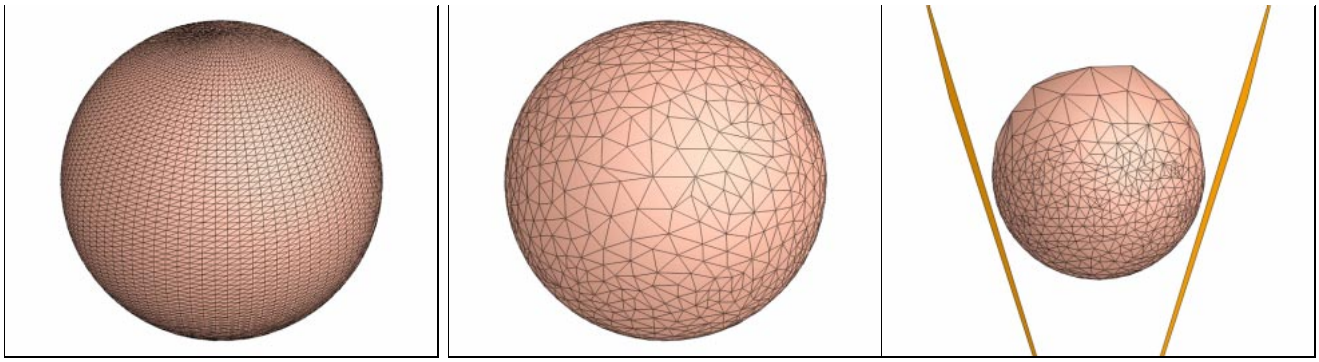


(a) Top view ($\tau = 0.0\%$; 33,119 faces) (b) Top and regular views ($\tau = 0.33\%$; 10,013 faces)



(c) Texture mapped $\hat{M}$ (79,202 faces) (d) Texture mapped (10,013 faces) (e) 764 generalized triangle strips

Figure 12: View-dependent refinement of the same PM, using the view frustum (highlighted in orange) and a screen-space geometric error tolerance of (a) 0% and (b,d,e) 0.33% of window size (i.e. 2 pixels for a 600×600 image).



(a) Original $\hat{M}$ (19,800 faces) (b) Front view and (c) Top view ($\tau = 0.075\%$; 1,422 faces)

Figure 13: View-dependent refinement of a tessellated sphere, demonstrating (b) the directionality of the deviation space $D_{\hat{\mathbf{n}}}$ (more refinement near silhouettes) and (c) the surface orientation criterion (coarsening of backfacing regions).

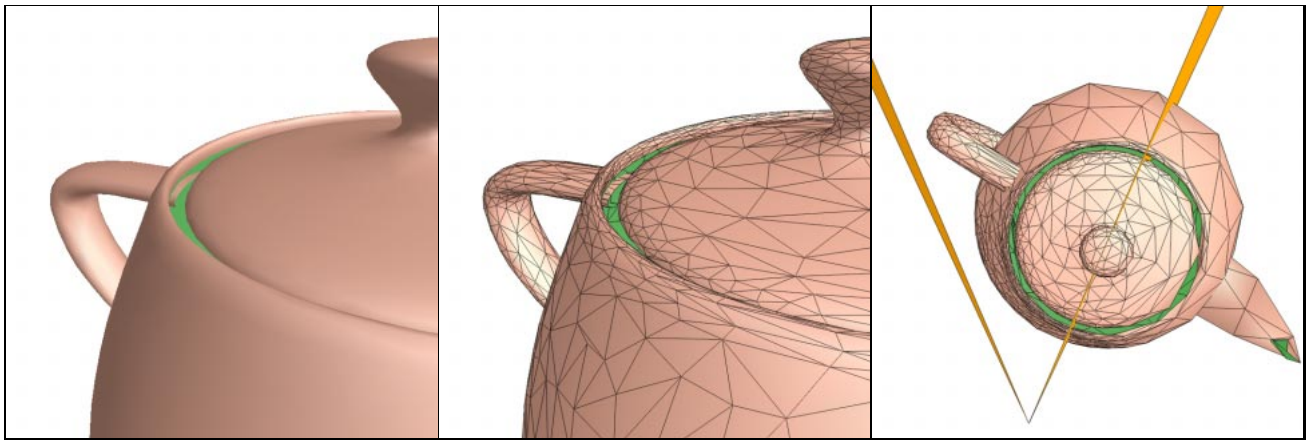
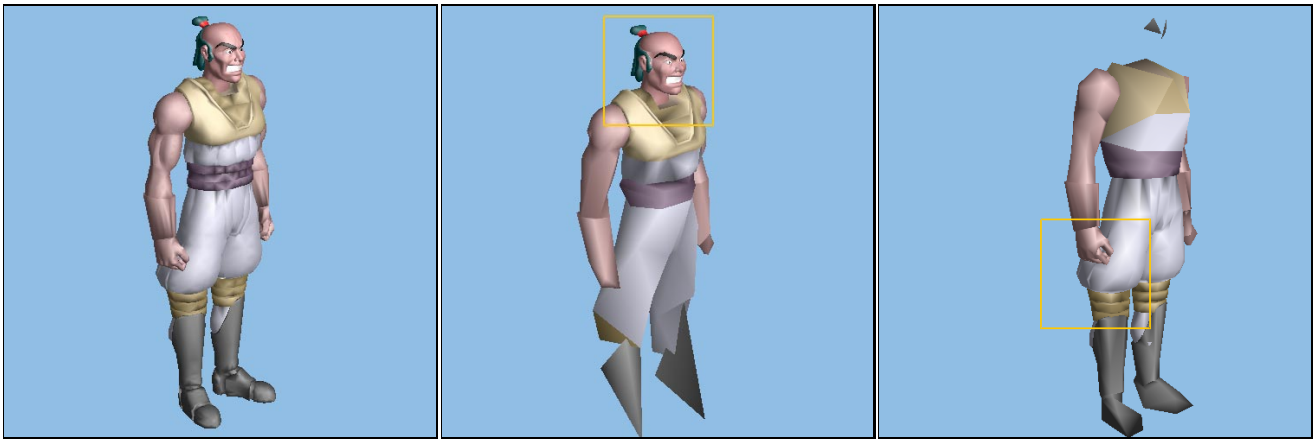
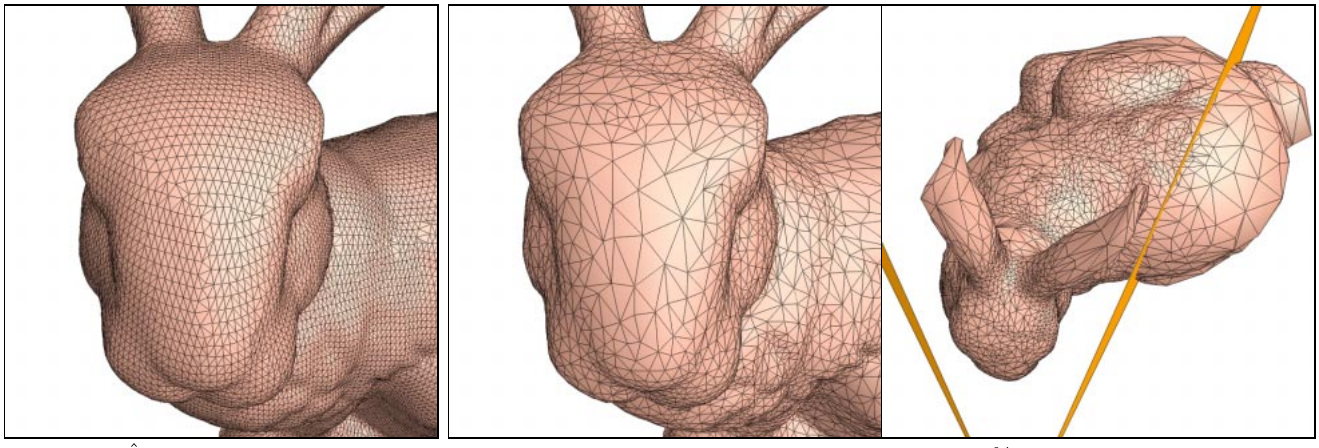


Figure 14: View-dependent refinement (τ = 0.15%; 1,782 faces) of a truncated PM representation (10,000 faces in $\hat{M}$) created from a tessellated parametric surface (25,440 faces). Interactive frame rate near this viewpoint is 14.7 frames/sec, versus 6.8 frames/sec using $\hat{M}$.



(a) Original $\hat{M}$ (42,712 faces) (b) View 1 (3,157 faces) (c) View 2 (2,559 faces)

Figure 15: Two view-dependent refinements of a general mesh $\hat{M}$ using view frustums highlighted in orange and with τ set to 0.6%.



(a) Original $\hat{M}$ (69,473 faces) (b) Front view and (c) Top view (τ = 0.1%; 10,528 faces)

Figure 16: View-dependent refinement. Interactive frame rate near this viewpoint is 6.7 frames/sec, versus 1.9 frames/sec using $\hat{M}$.

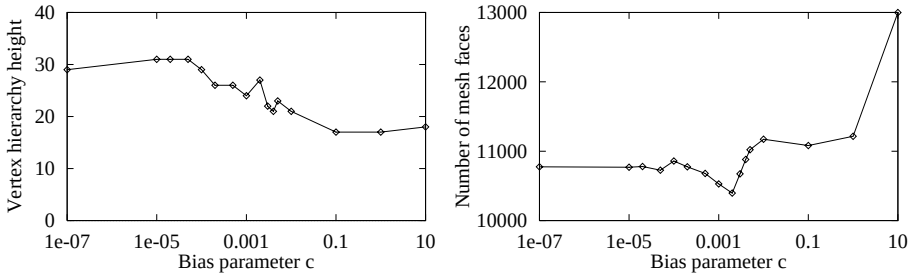


Figure 17: Height of vertex hierarchy, and number of faces in mesh of Figure 16b, as functions of the bias parameter c used in PM construction of bunny.

Progressive Simplicial Complexes

Jovan Popović* Carnegie Mellon University
Hugues Hoppe Microsoft Research

ABSTRACT

In this paper, we introduce the progressive simplicial complex (PSC) representation, a new format for storing and transmitting triangulated geometric models. Like the earlier progressive mesh (PM) representation, it captures a given model as a coarse base model together with a sequence of refinement transformations that progressively recover detail. The PSC representation makes use of a more general refinement transformation, allowing the given model to be an arbitrary triangulation (e.g. any dimension, non-orientable, non-manifold, non-regular), and the base model to always consist of a single vertex. Indeed, the sequence of refinement transformations encodes both the geometry and the topology of the model in a unified multiresolution framework. The PSC representation retains the advantages of PM's. It defines a continuous sequence of approximating models for runtime level-of-detail control, allows smooth transitions between any pair of models in the sequence, supports progressive transmission, and offers a space-efficient representation. Moreover, by allowing changes to topology, the PSC sequence of approximations achieves better fidelity than the corresponding PM sequence.

We develop an optimization algorithm for constructing PSC representations for graphics surface models, and demonstrate the framework on models that are both geometrically and topologically complex.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - surfaces and object representations.

Additional Keywords: model simplification, level-of-detail representations, multiresolution, progressive transmission, geometry compression.

1 INTRODUCTION

Modeling and 3D scanning systems commonly give rise to triangle meshes of high complexity. Such meshes are notoriously difficult to render, store, and transmit. One approach to speed up rendering is to replace a complex mesh by a set of level-of-detail (LOD) approximations; a detailed mesh is used when the object is close to the viewer, and coarser approximations are substituted as the object recedes [6, 8]. These LOD approximations can be precomputed automatically using mesh simplification methods (e.g. [2, 10, 14, 20, 21, 22, 24, 27]). For efficient storage and transmission, mesh compression schemes [7, 26] have also been developed.

The recently introduced *progressive mesh* (PM) representation [13] provides a unified solution to these problems. In PM form, an arbitrary mesh $\hat{M}$ is stored as a coarse base mesh M^0 together with a sequence of n detail records that indicate how to incrementally refine M^0 into $M^n = \hat{M}$ (see Figure 7). Each detail record encodes the information associated with a *vertex split*, an elementary transformation that adds one vertex to the mesh. In addition to defining a continuous sequence of approximations $M^0 \ldots M^n$, the PM representation supports smooth visual transitions (geomorphs), allows progressive transmission, and makes an effective mesh compression scheme.

The PM representation has two restrictions, however. First, it can only represent *meshes*: triangulations that correspond to orientable[1] 2-dimensional manifolds. Triangulated[2] models that cannot be represented include 1-d manifolds (open and closed curves), higher dimensional polyhedra (e.g. triangulated volumes), non-orientable surfaces (e.g. Möbius strips), non-manifolds (e.g. two cubes joined along an edge), and non-regular models (i.e. models of mixed dimensionality). Second, the expressiveness of the PM vertex split transformations constrains all meshes $M^0 \ldots M^n$ to have the same topological type. Therefore, when $\hat{M}$ is topologically complex, the simplified base mesh M^0 may still have numerous triangles (Figure 7).

In contrast, a number of existing simplification methods allow topological changes as the model is simplified (Section 6). Our work is inspired by vertex unification schemes [21, 22], which merge vertices of the model based on geometric proximity, thereby allowing genus modification and component merging.

In this paper, we introduce the *progressive simplicial complex* (PSC) representation, a generalization of the PM representation that permits topological changes. The key element of our approach is the introduction of a more general refinement transformation, the *generalized vertex split*, that encodes changes to both the geometry and topology of the model. The PSC representation expresses an arbitrary triangulated model M (e.g. any dimension, non-orientable, non-manifold, non-regular) as the result of successive refinements applied to a base model M^1 that always consists of a single vertex (Figure 8). Thus both geometric and topological complexity are recovered progressively. Moreover, the PSC representation retains the advantages of PM's, including continuous LOD, geomorphs, progressive transmission, and model compression.

In addition, we develop an optimization algorithm for constructing a PSC representation from a given model, as described in Section 4.

* Work performed while at Microsoft Research.
Email: jovan@cs.cmu.edu, hhoppe@microsoft.com
Web: http://www.cs.cmu.edu/∼jovan/
Web: http://research.microsoft.com/∼hoppe/

[1] The particular parametrization of vertex splits in [13] assumes that mesh triangles are consistently oriented.

[2] Throughout this paper, we use the words "triangulated" and "triangulation" in the general dimension-independent sense.

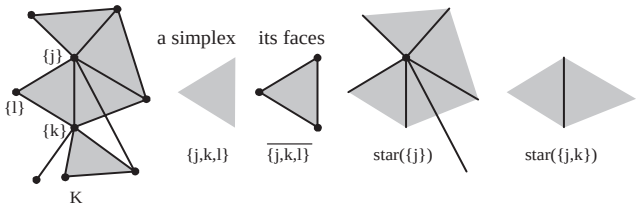


Figure 1: Illustration of a simplicial complex K and some of its subsets.

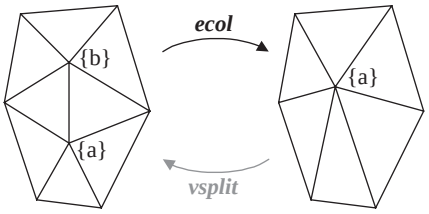


Figure 2: Illustration of the edge collapse transformation and its inverse, the vertex split.

2 BACKGROUND

2.1 Concepts from algebraic topology

To precisely define both triangulated models and their PSC representations, we find it useful to introduce some elegant abstractions from algebraic topology (e.g. [15, 25]).

The geometry of a triangulated model is denoted as a tuple (K, V) where the *abstract simplicial complex* K is a combinatorial structure specifying the adjacency of vertices, edges, triangles, etc., and V is a set of vertex positions specifying the shape of the model in $\mathbf{R}^3$.

More precisely, an abstract simplicial complex K consists of a set of vertices $\{1, \ldots, m\}$ together with a set of non-empty subsets of the vertices, called the *simplices* of K, such that any set consisting of exactly one vertex is a simplex in K, and every non-empty subset of a simplex in K is also a simplex in K.

A simplex containing exactly d+1 vertices has *dimension d* and is called a d-simplex. As illustrated pictorially in Figure 1, the *faces* of a simplex s, denoted $\overline{s}$, is the set of non-empty subsets of s. The *star* of s, denoted star(s), is the set of simplices of which s is a face. The *children* of a d-simplex s are the $(d \Leftrightarrow 1)$-simplices of $\overline{s}$, and its *parents* are the $(d+1)$-simplices of star(s). A simplex with exactly one parent is said to be a *boundary simplex*, and one with no parents a *principal simplex*. The dimension of K is the maximum dimension of its simplices; K is said to be *regular* if all its principal simplices have the same dimension.

To form a triangulation from K, identify its vertices $\{1, \ldots, m\}$ with the standard basis vectors $\{\mathbf{e}_1, \ldots, \mathbf{e}_m\}$ of $\mathbf{R}^m$. For each simplex s, let the *open simplex* $\langle s \rangle \subset \mathbf{R}^m$ denote the interior of the convex hull of its vertices:

$$\langle s \rangle = \{\mathbf{b} \in \mathbf{R}^m \ : \ \mathbf{b}_j \geq 0 \ , \ \sum_{j=1}^{m} \mathbf{b}_j = 1 \ , \ \mathbf{b}_j > 0 \Leftrightarrow \{j\} \subseteq s\}.$$

The *topological realization* $|K|$ is defined as $|K| = \langle K \rangle = \cup_{s \in K} \langle s \rangle$. The *geometric realization* of K is the image $\phi_V(|K|)$ where $\phi_V : \mathbf{R}^m \rightarrow \mathbf{R}^3$ is the linear map that sends the j-th standard basis vector $\mathbf{e}_j \in \mathbf{R}^m$ to $\mathbf{v}_j \in \mathbf{R}^3$. Only a restricted set of vertex positions $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ lead to an embedding of $\phi_V(|K|) \subset \mathbf{R}^3$, that is, prevent self-intersections. The geometric realization $\phi_V(|K|)$ is often called a *simplicial complex* or *polyhedron*; it is formed by an arbitrary union of points, segments, triangles, tetrahedra, etc. Note that there generally exist many triangulations (K, V) for a given polyhedron. (Some of the vertices V may lie in the polyhedron's interior.)

Two sets are said to be *homeomorphic* (denoted $\cong$) if there exists a continuous one-to-one mapping between them. Equivalently, they are said to have the same *topological type*. The topological realization $|K|$ is a *d-dimensional manifold without boundary* if for each vertex $\{j\}$, $\langle \text{star}(\{j\}) \rangle \cong \mathbf{R}^d$. It is a *d-dimensional manifold* if each $\langle \text{star}(\{v\}) \rangle$ is homeomorphic to either $\mathbf{R}^d$ or $\mathbf{R}^d_+$, where $\mathbf{R}^d_+ = \{\mathbf{x} \in \mathbf{R}^d : \mathbf{x}_1 \geq 0\}$. Two simplices s_1 and s_2 are *d-adjacent* if they have a common d-dimensional face. Two d-adjacent $(d+1)$-simplices s_1 and s_2 are *manifold-adjacent* if $\langle \text{star}(s_1 \cap s_2) \rangle \cong \mathbf{R}^{d+1}$.

Transitive closure of 0-adjacency partitions K into *connected components*. Similarly, transitive closure of manifold-adjacency partitions K into *manifold components*.

2.2 Review of progressive meshes

In the PM representation [13], a mesh with appearance attributes is represented as a tuple $M = (K, V, D, S)$, where the abstract simplicial complex K is restricted to define an orientable 2-dimensional manifold, the vertex positions $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ determine its geometric realization $\phi_V(|K|)$ in $\mathbf{R}^3$, D is the set of discrete material attributes d_f associated with 2-simplices $f \in K$, and S is the set of scalar attributes $s_{(v,f)}$ (e.g. normals, texture coordinates) associated with corners (vertex-face tuples) of K.

An initial mesh $\hat{M} = M^n$ is simplified into a coarser base mesh M^0 by applying a sequence of n successive edge collapse transformations:

$$(\hat{M} = M^n) \overset{ecol_{n-1}}{\Leftrightarrow} \ \ldots \ \overset{ecol_1}{\Leftrightarrow} \ M^1 \ \overset{ecol_0}{\Leftrightarrow} \ M^0 \ .$$

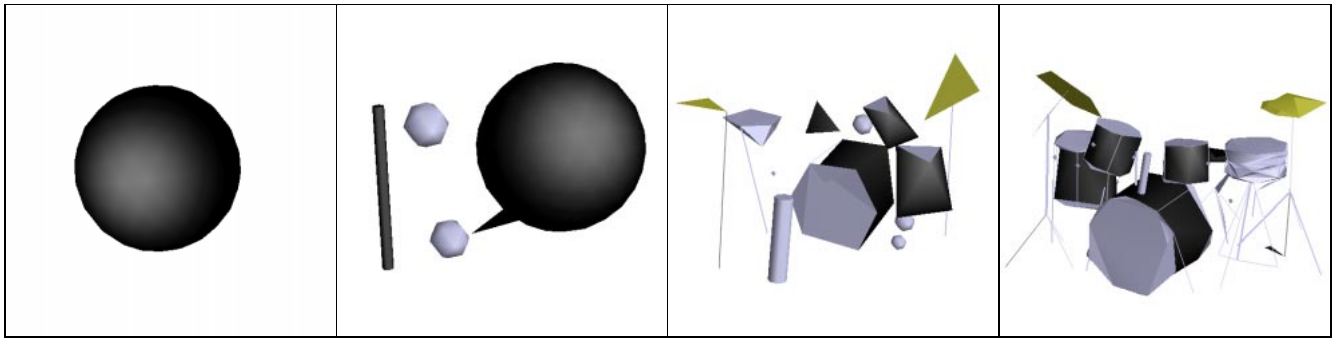
As shown in Figure 2, each *ecol* unifies the two vertices of an edge $\{a, b\}$, thereby removing one or two triangles. The position of the resulting unified vertex can be arbitrary. Because the edge collapse transformation has an inverse, called the *vertex split* transformation (Figure 2), the process can be reversed, so that an arbitrary mesh $\hat{M}$ may be represented as a simple mesh M^0 together with a sequence of n *vsplit* records:

$$M^0 \overset{vsplit_0}{\Leftrightarrow} M^1 \overset{vsplit_1}{\Leftrightarrow} \ \ldots \ \overset{vsplit_{n-1}}{\Leftrightarrow} \ (M^n = \hat{M})$$

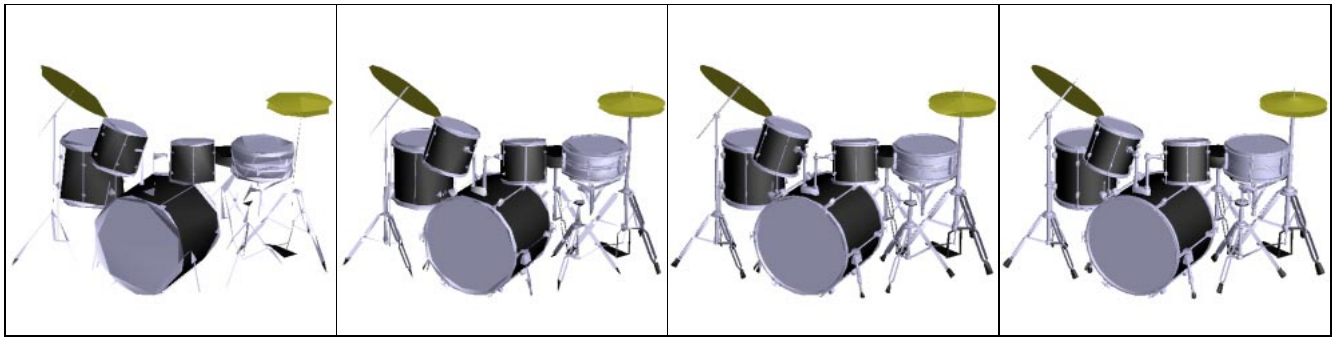
The tuple $(M^0, \{vsplit_0, \ldots, vsplit_{n-1}\})$ forms a *progressive mesh* (PM) representation of $\hat{M}$.

The PM representation thus captures a continuous sequence of approximations $M^0 \ldots M^n$ that can be quickly traversed for interactive level-of-detail control. Moreover, there exists a correspondence between the vertices of any two meshes M^c and M^f $(0 \leq c < f \leq n)$ within this sequence, allowing for the construction of smooth visual transitions (geomorphs) between them. A sequence of such geomorphs can be precomputed for smooth runtime LOD. In addition, PM's support progressive transmission, since the base mesh M^0 can be quickly transmitted first, followed the *vsplit* sequence. Finally, the *vsplit* records can be encoded concisely, making the PM representation an effective scheme for mesh compression.

Topological constraints Because the definitions of *ecol* and *vsplit* are such that they preserve the topological type of the mesh (i.e. all $|K^i|$ are homeomorphic), there is a constraint on the minimum complexity that K^0 may achieve. For instance, it is known that the minimal number of vertices for a closed genus g mesh (orientable 2-manifold) is $\lceil (7+(48g+1)^{\frac{1}{2}})/2 \rceil$ if $g \neq 2$ (10 if $g = 2$) [16]. Also, the presence of boundary components may further constrain the complexity of K^0. Most importantly, $\hat{K}$ may consist of a number of components, and each is required to appear in the base mesh. For example, the meshes in Figure 7 each have 117 components. As evident from the figure, the geometry of PM meshes may deteriorate severely as they approach topological lower bound.



M^1; $\{1,0,0\}$; (1) M^{10}; $\{5,1,1\}$; (7) M^{50}; $\{4,6,56\}$; (12) M^{200}; $\{15,52,277\}$; (28)



M^{500}; $\{39,68,690\}$; (58) M^{2000}; $\{14,25,3219\}$; (108) M^{5000}; $\{0,2,9010\}$; (176) $M^{n=34794}$; $\{0,0,68776\}$; (207)

Figure 3: Example of a PSC representation. The image captions indicate the number of principal $\{0,1,2\}$-simplices respectively and the number of connected components (in parenthesis).

3 PSC REPRESENTATION

3.1 Triangulated models

The first step towards generalizing PM's is to let the PSC representation encode more general triangulated models, instead of just meshes.

We denote a triangulated model as a tuple $M = (K, V, D, A)$. The abstract simplicial complex K is not restricted to 2-manifolds, but may in fact be arbitrary. To represent K in memory, we encode the *incidence graph* of the simplices using the following linked structures (in C++ notation):

```
struct Simplex {
    int dim;      // 0=vertex, 1=edge, 2=triangle, ...
    int id;
    Simplex* children[MAXDIM+1];      // [0..dim]
    List<Simplex*> parents;
};
```

To render the model, we draw only the principal simplices of K, denoted $\mathcal{P}(K)$ (i.e. vertices not adjacent to edges, edges not adjacent to triangles, etc.). The discrete attributes D associate a material identifier d_s with each simplex $s \in \mathcal{P}(K)$. For the sake of simplicity, we avoid explicitly storing surface normals at "corners" (using a set S) as done in [13]. Instead we let the material identifier d_s contain a *smoothing group* field [28], and let a normal discontinuity (*crease*) form between any pair of adjacent triangles with different smoothing groups.

Previous vertex unification schemes [21, 22] render principal simplices of dimension 0 and 1 (denoted $\mathcal{P}_{01}(K)$) as points and lines respectively with fixed, device-dependent screen widths. To better approximate the model, we instead define a set A that associates an area $a_s \in A$ with each simplex $s \in \mathcal{P}_{01}(K)$. We think of a 0-simplex $s_0 \in \mathcal{P}_0(K)$ as approximating a sphere with area a_{s_0}, and a 1-simplex $s_1 = \{j,k\} \in \mathcal{P}_1(K)$ as approximating a cylinder (with axis $(\mathbf{v}_j, \mathbf{v}_k)$) of area a_{s_1}. To render a simplex $s \in \mathcal{P}_{01}(K)$, we determine the radius r_{model} of the corresponding sphere or cylinder in modeling space, and project the length r_{model} to obtain the radius r_{screen} in screen pixels. Depending on r_{screen}, we render the simplex as a polygonal sphere or cylinder with radius r_{model}, a 2D point or line with thickness $2r_{screen}$, or do not render it at all. This choice based on r_{screen} can be adjusted to mitigate the overhead of introducing polygonal representations of spheres and cylinders.

As an example, Figure 3 shows an initial model $\hat{M}$ of 68,776 triangles. One of its approximations M^{500} is a triangulated model with $\{39,68,690\}$ principal $\{0,1,2\}$-simplices respectively.

3.2 Level-of-detail sequence

As in progressive meshes, from a given triangulated model $\hat{M} = M^n$, we define a sequence of approximations M^i:

$$M^1 \overset{op_1}{\longleftrightarrow} M^2 \overset{op_2}{\longleftrightarrow} \ \ldots\ M^{n-1} \overset{op_{n-1}}{\longleftrightarrow} M^n\ .$$

Here each model M^i has exactly i vertices. The simplification operator $M^i \overset{vunify_i}{\longleftarrow} M^{i+1}$ is the *vertex unification* transformation, which merges two vertices (Section 3.3), and its inverse $M^i \overset{gvspl_i}{\longrightarrow} M^{i+1}$ is the *generalized vertex split* transformation (Section 3.4). The tuple $(M^1, \{gvspl_1, \ldots, gvspl_{n-1}\})$ forms a *progressive simplicial complex* (PSC) representation of $\hat{M}$.

To construct a PSC representation, we first determine a sequence of *vunify* transformations simplifying $\hat{M}$ down to a single vertex, as described in Section 4. After reversing these transformations, we renumber the simplices in the order that they are created, so that each $gvspl_i(\{a_i\}, \ldots)$ splits the vertex $\{a_i\} \in K^i$ into two vertices $\{a_i\}, \{i{+}1\} \in K^{i+1}$. As vertices may have different positions in the different models, we denote the position of $\{j\}$ in M^i as $\mathbf{v}_j^i$.

To better approximate a surface model $\hat{M}$ at lower complexity levels, we initially associate with each (principal) 2-simplex s an area a_s equal to its triangle area in $\hat{M}$. Then, as the model is simplified, we

keep constant the sum of areas a_s associated with principal simplices within each manifold component. When 2-simplices are eventually reduced to principal 1-simplices and 0-simplices, their associated areas will provide good estimates of the original component areas.

3.3 Vertex unification transformation

The transformation $vunify(\{a_i\}, \{b_i\}, midp_i) : M^i \leftarrow M^{i+1}$ takes an arbitrary pair of vertices $\{a_i\}, \{b_i\} \in K^{i+1}$ (simplex $\{a_i, b_i\}$ need not be present in K^{i+1}) and merges them into a single vertex $\{a_i\} \in K^i$.

Model M^i is created from M^{i+1} by updating each member of the tuple (K, V, D, A) as follows:

K: References to $\{b_i\}$ in all simplices of K are replaced by references to $\{a_i\}$. More precisely, each simplex s in star($\{b_i\}$) $\subset K^{i+1}$ is replaced by simplex $(s \setminus \{b_i\}) \cup \{a_i\}$, which we call the *ancestor simplex* of s. If this ancestor simplex already exists, s is deleted.

V: Vertex $\mathbf{v}_b$ is deleted. For simplicity, the position of the remaining (unified) vertex is set to either the midpoint or is left unchanged. That is, $\mathbf{v}_a^i = (\mathbf{v}_a^{i+1} + \mathbf{v}_b^{i+1})/2$ if the boolean parameter $midp_i$ is *true*, or $\mathbf{v}_a^i = \mathbf{v}_a^{i+1}$ otherwise.

D: Materials are carried through as expected. So, if after the vertex unification an ancestor simplex $(s \setminus \{b_i\}) \cup \{a_i\} \in K^i$ is a new principal simplex, it receives its material from $s \in K^{i+1}$ if s is a principal simplex, or else from the single parent $s \cup \{a_i\} \in K^{i+1}$ of s.

A: To maintain the initial areas of manifold components, the areas a_s of deleted principal simplices are redistributed to manifold-adjacent neighbors. More concretely, the area of each principal d-simplex s deleted during the K update is distributed to a manifold-adjacent d-simplex not in star($\{a_i, b_i\}$). If no such neighbor exists and the ancestor of s is a principal simplex, the area a_s is distributed to that ancestor simplex. Otherwise, the manifold component (star($\{a_i, b_i\}$)) of s is being squashed between two other manifold components, and a_s is discarded.

3.4 Generalized vertex split transformation

Constructing the PSC representation involves recording the information necessary to perform the inverse of each $vunify_i$. This inverse is the generalized vertex split $gvspl_i$, which splits a 0-simplex $\{a_i\}$ to introduce an additional 0-simplex $\{b_i\}$. (As mentioned previously, renumbering of simplices implies $b_i \equiv i{+}1$, so index b_i need not be stored explicitly.) Each $gvspl_i$ record has the form

$$gvspl_i(\{a_i\}, C_i^{\Delta K}, midp_i, (\Delta\mathbf{v})_i, C_i^{\Delta D}, C_i^{\Delta A}) ,$$

and constructs model M^{i+1} from M^i by updating the tuple (K, V, D, A) as follows:

K: As illustrated in Figure 4, any simplex adjacent to $\{a_i\}$ in K^i can be the *vunify* result of one of four configurations in K^{i+1}. To construct K^{i+1}, we therefore replace each ancestor simplex $s \in$ star($\{a_i\}$) in K^i by either **(1)** s, **(2)** $(s \setminus \{a_i\}) \cup \{i{+}1\}$, **(3)** s and $(s \setminus \{a_i\}) \cup \{i{+}1\}$, or **(4)** s, $(s \setminus \{a_i\}) \cup \{i{+}1\}$ and $s \cup \{i{+}1\}$. The choice is determined by a *split code* associated with s. These split codes are stored as a code string $C_i^{\Delta K}$, in which the simplices star($\{a_i\}$) are sorted first in order of increasing dimension, and then in order of increasing simplex id, as shown in Figure 5.

V: The new vertex is assigned position $\mathbf{v}_{i+1}^{i+1} = \mathbf{v}_{a_i}^i + (\Delta\mathbf{v})_i$. The other vertex is given position $\mathbf{v}_{a_i}^{i+1} = \mathbf{v}_{a_i}^i \Leftrightarrow (\Delta\mathbf{v})_i$ if the boolean parameter $midp_i$ is *true*; otherwise its position remains unchanged.

D: The string $C_i^{\Delta D}$ is used to assign materials d_s for each new principal simplex. Simplices in $C_i^{\Delta D}$, as well as in $C_i^{\Delta A}$ below, are sorted by simplex dimension and simplex id as in $C_i^{\Delta K}$.

A: During reconstruction, we are only interested in the areas a_s for $s \in \mathcal{P}_{01}(K)$. The string $C_i^{\Delta A}$ tracks changes in these areas.

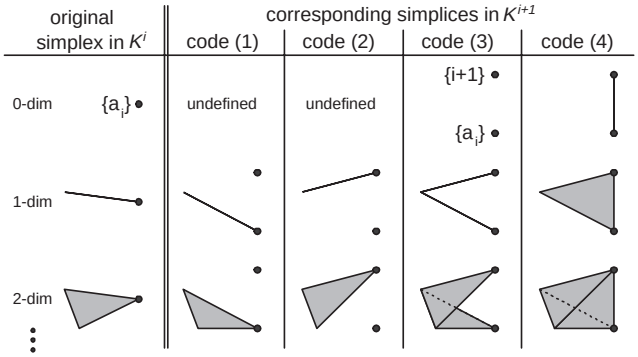


Figure 4: Effects of split codes on simplices of various dimensions.

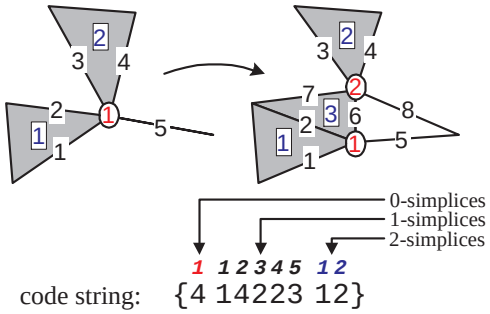


Figure 5: Example of split code encoding.

3.5 Properties

Levels of detail A graphics application can efficiently transition between models $M^1 \ldots M^n$ at runtime by performing a sequence of *vunify* or *gvspl* transformations. Our current research prototype was not designed for efficiency; it attains simplification rates of about 6000 *vunify*/sec and refinement rates of about 5000 *gvspl*/sec. We expect that a careful redesign using more efficient data structures would significantly improve these rates.

Geomorphs As in the PM representation, there exists a correspondence between the vertices of the models $M^1 \ldots M^n$. Given a coarser model M^c and a finer model M^f, $1 \le c < f \le n$, each vertex $\{j\} \in K^f$ corresponds to a unique ancestor vertex $\{\rho^{f\to c}(j)\} \in K^c$ found by recursively traversing the ancestor simplex relations:

$$\rho^{f\to c}(j) = \begin{cases} j & , j \le c \\ \rho^{f\to c}(a_{j-1}) & , j > c \end{cases} .$$

This correspondence allows the creation of a smooth visual transition (geomorph) $M^G(\alpha)$ such that $M^G(1)$ equals M^f and $M^G(0)$ looks identical to M^c. The geomorph is defined as the model

$$M^G(\alpha) = (K^f, V^G(\alpha), D^f, A^G(\alpha))$$

in which each vertex position is interpolated between its original position in V^f and the position of its ancestor in V^c:

$$\mathbf{v}_j^G(\alpha) = (\alpha)\mathbf{v}_j^f + (1 \Leftrightarrow \alpha)\mathbf{v}_{\rho^{f\to c}(j)}^c .$$

However, we must account for the special rendering of principal simplices of dimension 0 and 1 (Section 3.1). For each simplex $s \in \mathcal{P}_{01}(K^f)$, we interpolate its area using

$$a_s^G(\alpha) = (\alpha)a_s^f + (1 \Leftrightarrow \alpha)a_s^c ,$$

where $a_s^c = 0$ if $s \notin \mathcal{P}_{01}(K^c)$. In addition, we render each simplex $s \in \mathcal{P}_{01}(K^c) \setminus \mathcal{P}_{01}(K^f)$ using area $a_s^G(\alpha) = (1 \Leftrightarrow \alpha)a_s^c$. The resulting

geomorph is visually smooth even as principal simplices are introduced, removed, or change dimension. The accompanying video demonstrates a sequence of such geomorphs.

Progressive transmission As with PM's, the PSC representation can be progressively transmitted by first sending M^1, followed by the *gvspl* records. Unlike the base mesh of the PM, M^1 always consists of a single vertex, and can therefore be sent in a fixed-size record. The rendering of lower-dimensional simplices as spheres and cylinders helps to quickly convey the overall shape of the model in the early stages of transmission.

Model compression Although PSC *gvspl* are more general than PM *vsplit* transformations, they offer a surprisingly concise representation of $\hat{M}$. Table 1 lists the average number of bits required to encode each field of the *gvspl* records.

Using arithmetic coding [30], the vertex id field $\{a_i\}$ requires $\log_2 i$ bits, and the boolean parameter $midp_i$ requires 0.6–0.9 bits for our models. The $(\Delta\mathbf{v})_i$ delta vector is quantized to 16 bits per coordinate (48 bits per $\Delta\mathbf{v}$), and stored as a variable-length field [7, 13], requiring about 31 bits on average.

At first glance, each split code in the code string $C_i^{\Delta K}$ seems to have 4 possible outcomes (except for the split code for 0-simplex $\{a_i\}$ which has only 2 possible outcomes). However, there exist constraints between these split codes. For example, in Figure 5, the code 1 for 1-simplex id 1 implies that 2-simplex id 1 also has code 1. This in turn implies that 1-simplex id 2 cannot have code 2. Similarly, code 2 for 1-simplex id 3 implies a code 2 for 2-simplex id 2, which in turn implies that 1-simplex id 4 cannot have code 1. These constraints, illustrated in the "scoreboard" of Figure 6, can be summarized using the following two rules:

(1) If a simplex has split code $c \in \{1, 2\}$, all of its parents have split code c.

(2) If a simplex has split code 3, none of its parents have split code 4.

As we encode split codes in $C_i^{\Delta K}$ left to right, we apply these two rules (and their contrapositives) transitively to constrain the possible outcomes for split codes yet to be encoded. Using arithmetic coding with uniform outcome probabilities, these constraints reduce the code string length in Figure 6 from 15 bits to 10.2 bits. In our models, the constraints reduce the code string from 30 bits to 14 bits on average.

The code string is further reduced using a non-uniform probability model. We create an array $T[0..dim][0..15]$ of encoding tables, indexed by simplex dimension (0..dim) and by the set of possible (constrained) split codes (a 4-bit mask). For each simplex s, we encode its split code c using the probability distribution found in $T[s.dim][s.codes\_mask]$. For 2-dimensional models, only 10 of the 48 tables are non-trivial, and each table contains at most 4 probabilities, so the total size of the probability model is small. These encoding tables reduce the code strings to approximately 8 bits as shown in Table 1. By comparison, the PM representation requires approximately 5 bits for the same information, but of course it disallows topological changes.

To provide more intuition for the efficiency of the PSC representation, we note that capturing the connectivity of an average 2-manifold simplicial complex (n vertices, $3n$ edges, and $2n$ triangles) requires $\sum_{i=1}^{n}(\log_2 i+8) \simeq n(\log_2 n+7)$ bits with PSC encoding, versus $n(12\log_2 n + 9.5)$ bits with a traditional one-way incidence graph representation.

For improved compression, it would be best to use a hybrid PM + PSC representation, in which the more concise PM vertex split encoding is used when the local neighborhood is an orientable

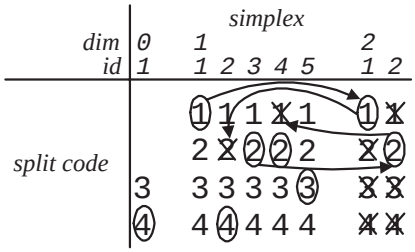


Figure 6: Constraints on the split codes for the simplices in the example of Figure 5.

Table 1: Compression results and construction times.

Object	#verts n	Space required (bits/n)							Trad. repr. bits/n	Con. time hrs.
		K		V		D	A	Σ		
		$\{a_i\}$	$C_i^{\Delta K}$	$midp_i$	$(\Delta\mathbf{v})_i$	$C_i^{\Delta D}$	$C_i^{\Delta A}$			
drumset	34,794	12.2	8.2	0.9	28.1	4.1	0.4	53.9	146.1	4.3
destroyer	83,799	13.3	8.3	0.7	23.1	2.1	0.3	47.8	154.1	14.1
chandelier	36,627	12.4	7.6	0.8	28.6	3.4	0.8	53.6	143.6	3.6
schooner	119,734	13.4	8.6	0.7	27.2	2.5	1.3	53.7	148.7	22.2
sandal	4,628	9.2	8.0	0.7	33.4	1.5	0.0	52.8	123.2	0.4
castle	15,082	11.0	1.2	0.6	30.7	0.0	-	43.5	-	0.5
cessna	6,795	9.6	7.6	0.6	32.2	2.5	0.1	52.6	132.1	0.5
harley	28,847	11.9	7.9	0.9	30.5	1.4	0.4	53.0	135.7	3.5

2-dimensional manifold (this occurs on average 93% of the time in our examples).

To compress $C_i^{\Delta D}$, we predict the material for each new principal simplex $s \in \text{star}(\{a_i\}) \cup \text{star}(\{b_i\}) \subset K^{i+1}$ by constructing an ordered set D_s of materials found in $\text{star}(\{a_i\}) \subset K^i$. To improve the coding model, the first materials in D_s are those of principal simplices in $\text{star}(s') \subset K^i$ where s' is the ancestor of s; the remaining materials in $\text{star}(\{a_i\}) \subset K^i$ are appended to D_s. The entry in $C_i^{\Delta D}$ associated with s is the index of its material in D_s, encoded arithmetically. If the material of s is not present in D_s, it is specified explicitly as a global index in D.

We encode $C_i^{\Delta A}$ by specifying the area a_s for each new principal simplex $s \in \mathcal{P}_{01}(\text{star}(\{a_i\}) \cup \text{star}(\{b_i\})) \subset K^{i+1}$. To account for this redistribution of area, we identify the principal simplex from which s receives its area by specifying its index in $\mathcal{P}_{01}(\text{star}(\{a_i\})) \subset K^i$.

The column labeled Σ in Table 1 sums the bits of each field of the *gvspl* records. Multiplying Σ by the number n of vertices in $\hat{M}$ gives the total number of bits for the PSC representation of the model (e.g. 500 KB for the destroyer). By way of comparison, the next column shows the number of bits per vertex required in a traditional "IndexedFaceSet" representation, with quantization of 16 bits per coordinate and arithmetic coding of face materials ($\simeq 3n\cdot 16 + 2n\cdot 3\cdot\log_2 n$ + materials).

4 PSC CONSTRUCTION

In this section, we describe a scheme for iteratively choosing pairs of vertices to unify, in order to construct a PSC representation. Our algorithm, a generalization of [13], is time-intensive, seeking high quality approximations. It should be emphasized that many quality metrics are possible. For instance, the *quadric error* metric recently introduced by Garland and Heckbert [9] provides a different trade-off of execution speed and visual quality.

As in [13, 20], we first compute a cost ΔE for each candidate *vunify* transformation, and enter the candidates into a priority queue ordered by ascending cost. Then, in each iteration $i = n \Leftrightarrow 1 \ldots 1$, we perform the *vunify* at the front of the queue and update the costs of affected candidates.

4.1 Forming set $\mathcal{C}$ of candidate vertex pairs

In principle, we could enter all possible pairs of vertices from $\hat{M}$ into the priority queue, but this would be prohibitively expensive since simplification would then require at least $O(n^2 \log n)$ time. Instead, we would like to consider only a smaller set $\mathcal{C}$ of candidate vertex pairs. Naturally, $\mathcal{C}$ should include the 1-simplices of K. Additional pairs should also be included in $\mathcal{C}$ to allow distinct connected components of M to merge and to facilitate topological changes. We considered several schemes for forming these additional pairs, including binning, octrees, and k-closest neighbor graphs, but opted for the Delaunay triangulation because of its adaptability on models containing components at different scales.

We compute the Delaunay triangulation of the vertices of $\hat{M}$, represented as a 3-dimensional simplicial complex $\hat{K}_{DT}$. We define the initial set $\mathcal{C}$ to contain both the 1-simplices of $\hat{K}$ and the subset of 1-simplices of $\hat{K}_{DT}$ that connect vertices in different connected components of $\hat{K}$. During the simplification process, we apply each vertex unification performed on M to $\mathcal{C}$ as well in order to keep consistent the set of candidate pairs.

For models in $\mathbf{R}^3$, $\mathcal{C} \cap \text{star}(\{a_i\})$ has constant size in the average case, and the overall simplification algorithm requires $O(n \log n)$ time. (In the worst case, it could require $O(n^2 \log n)$ time.)

4.2 Selecting vertex unifications from $\mathcal{C}$

For each candidate vertex pair $(a, b) \in \mathcal{C}$, the associated $vunify(\{a\}, \{b\}) : M^i \leftarrow M^{i+1}$ is assigned the cost

$$\Delta E = \Delta E_{dist} + \Delta E_{disc} + E_{\Delta area} + E_{fold} \ .$$

As in [13], the first term is $\Delta E_{dist} = E_{dist}(M^i) \Leftrightarrow E_{dist}(M^{i+1})$, where $E_{dist}(M)$ measures the geometric accuracy of the approximate model M. Conceptually, $E_{dist}(M)$ approximates the continuous integral

$$\int_{\mathbf{p} \in \hat{M}} d^2(\mathbf{p}, M) \ ,$$

where $d(\mathbf{p}, M)$ is the Euclidean distance of the point $\mathbf{p}$ to the closest point on M. We discretize this integral by defining $E_{dist}(M)$ as the sum of squared distances to M from a dense set of points X sampled from the original model $\hat{M}$. We sample X from the set of principal simplices in K — a strategy that generalizes to arbitrary triangulated models.

In [13], $E_{disc}(M)$ measures the geometric accuracy of discontinuity curves formed by a set of sharp edges in the mesh. For the PSC representation, we generalize the concept of sharp edges to that of *sharp simplices* in K — a simplex is sharp either if it is a boundary simplex or if two of its parents are principal simplices with different material identifiers. The energy E_{disc} is defined as the sum of squared distances from a set X_{disc} of points sampled from sharp simplices to the discontinuity components from which they were sampled. Minimization of E_{disc} therefore preserves the geometry of material boundaries, normal discontinuities (creases), and triangulation boundaries (including boundary curves of a surface and endpoints of a curve).

We have found it useful to introduce a term $E_{\Delta area}$ that penalizes surface stretching (a more sophisticated version of the regularizing E_{spring} term of [13]). Let A_N^{i+1} be the sum of triangle areas in the neighborhood $\text{star}(\{a_i\}) \cup \text{star}(\{b_i\}) \subset K^{i+1}$, and A_N^i the sum of triangle areas in $\text{star}(\{a_i\}) \subset K^i$. The mean squared displacement over the neighborhood N due to the change in area can be approximated as $\overline{disp^2} = \frac{1}{2}(\sqrt{A_N^{i+1}} \Leftrightarrow \sqrt{A_N^i})^2$. We let $E_{\Delta area} = |X_N| \, \overline{disp^2}$, where $|X_N|$ is the number of points X projecting in the neighborhood.

To prevent model self-intersections, the last term E_{fold} penalizes surface folding. We compute the rotation of each oriented triangle in the neighborhood due to the vertex unification (as in [10, 20]). If any rotation exceeds a threshold angle value, we set E_{fold} to a large constant.

Unlike [13], we do not optimize over the vertex position $\mathbf{v}_a^i$, but simply evaluate ΔE for $\mathbf{v}_a^i \in \{\mathbf{v}_a^{i+1}, \mathbf{v}_b^{i+1}, (\mathbf{v}_a^{i+1} + \mathbf{v}_b^{i+1})/2\}$ and choose the best one. This speeds up the optimization, improves model compression, and allows us to introduce non-quadratic energy terms like $E_{\Delta area}$.

5 RESULTS

Table 1 gives quantitative results for the examples in the figures and in the video. Simplification times for our prototype are measured on an SGI Indigo2 Extreme (150MHz R4400). Although these times may appear prohibitive, PSC construction is an off-line task that only needs to be performed once per model.

Figure 9 highlights some of the benefits of the PSC representation. The pearls in the chandelier model are initially disconnected tetrahedra; these tetrahedra merge and collapse into 1-d curves in lower-complexity approximations. Similarly, the numerous polygonal ropes in the schooner model are simplified into curves which can be rendered as line segments. The straps of the sandal model initially have some thickness; the top and bottom sides of these straps merge in the simplification. Also note the disappearance of the holes on the sandal straps. The castle example demonstrates that the original model need not be a mesh; here $\hat{M}$ is a 1-dimensional non-manifold obtained by extracting edges from an image.

6 RELATED WORK

There are numerous schemes for representing and simplifying triangulations in computer graphics. A common special case is that of subdivided 2-manifolds (meshes). Garland and Heckbert [12] provide a recent survey of mesh simplification techniques. Several methods simplify a given model through a sequence of edge collapse transformations [10, 13, 14, 20]. With the exception of [20], these methods constrain edge collapses to preserve the topological type of the model (e.g. disallow the collapse of a tetrahedron into a triangle).

Our work is closely related to several schemes that generalize the notion of edge collapse to that of vertex unification, whereby separate connected components of the model are allowed to merge and triangles may be collapsed into lower dimensional simplices. Rossignac and Borrel [21] overlay a uniform cubical lattice on the object, and merge together vertices that lie in the same cubes. Schaufler and Stürzlinger [22] develop a similar scheme in which vertices are merged using a hierarchical clustering algorithm. Luebke [18] introduces a scheme for locally adapting the complexity of a scene at runtime using a clustering octree. In these schemes, the approximating models correspond to simplicial complexes that would result from a set of *vunify* transformations (Section 3.3). Our approach differs in that we order the *vunify* in a carefully optimized sequence. More importantly, we define not only a simplification process, but also a new representation for the model using an encoding of $gvspl = vunify^{-1}$ transformations.

Recent, independent work by Schmalstieg and Schaufler [23] develops a similar strategy of encoding a model using a sequence of vertex split transformations. Their scheme differs in that it tracks only triangles, and therefore requires regular, 2-dimensional triangulations. Hence, it does not allow lower-dimensional simplices in the model approximations, and does not generalize to higher dimensions.

Some simplification schemes make use of an intermediate volumetric representation to allow topological changes to the model. He et al. [11] convert a mesh into a binary inside/outside function discretized on a three-dimensional grid, low-pass filter this function,

and convert it back to a simpler surface using an adaptive "marching cubes" algorithm. They demonstrate that aliasing is reduced by rendering the filtered volume as a set of nested translucent surfaces. Similarly, Andújar et al. [1] make use of an inside/outside octree representation.

Triangulations of subdivided manifolds (and non-manifolds) of higher dimension are used extensively in solid modeling. Paoluzzi et al. [19] provide an overview of related work and analyze the benefits of representing such triangulations using (regular) simplicial complexes. Bertolotto et al. [3, 4] present hierarchical simplicial representations for subdivided manifolds, but these do not support changes of topological type.

Polyhedra can also be represented using more general representations. The simplicial set representation of Lang and Lienhardt [17] generalizes simplicial complexes to allow incomplete and degenerate simplices. Cell complexes, formed by subdividing manifolds into non-simplicial cells, can be represented using the radial edge structure of Weiler [29] or the cell tuple structure of Brisson [5].

7 SUMMARY AND FUTURE WORK

We have introduced the progressive simplicial complex representation, a new format for arbitrary triangulated models that captures both geometry and topology in a unified multiresolution framework. It defines a continuous-resolution sequence of approximating models, from the original model down to a single vertex. In addition, it allows geomorphs between any pair of models in this sequence, supports progressive transmission, and offers a concise storage format. We presented an optimization algorithm for constructing PSC representations for computer graphics surface models.

Although we restricted our examples in this paper to models of dimension at most 2, the PSC representation is defined for arbitrary dimensions, and we expect that it will find useful applications in the representation of higher dimensional models such as volumes, light fields, and bidirectional reflection distribution functions. In particular, it offers an avenue for level-of-detail control in volume rendering applications.

ACKNOWLEDGMENTS

We are extremely grateful to Viewpoint Datalabs for providing us with numerous meshes with which to experiment. We also wish to thank Tom Duchamp for helpful discussions on algebraic topology.

REFERENCES

[1] ANDÚJAR, C., AYALA, D., BRUNET, P., JOAN-ARINYO, R., AND SOLÉ, J. Automatic generation of multiresolution boundary representations. *Computer Graphics Forum (Proceedings of Eurographics '96) 15*, 3 (1996), 87–96.

[2] BAJAJ, C., AND SCHIKORE, D. Error-bounded reduction of triangle meshes with multivariate data. *SPIE 2656* (1996), 34–45.

[3] BERTOLOTTO, M., DE FLORIANI, L., BRUZZONE, E., AND PUPPO, E. Multiresolution representation of volume data through hierarchical simplicial complexes. In *Aspects of visual form processing* (1994), C. Arcelli, L. Cordella, and G. Sanniti di Baja, Eds., World Scientific, pp. 73–82.

[4] BERTOLOTTO, M., DE FLORIANI, L., AND MARZANO, P. Pyramidal simplicial complexes. In *Solid Modeling '95* (May 1995), pp. 153–162.

[5] BRISSON, E. *Representation of d-dimensional geometric objects*. PhD thesis, Dept. of Computer Science and Engineering, U. of Washington, 1990.

[6] CLARK, J. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10 (October 1976), 547–554.

[7] DEERING, M. Geometry compression. *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), 13–20.

[8] FUNKHOUSER, T., AND SÉQUIN, C. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 247–254.

[9] GARLAND, M., AND HECKBERT, P. Surface simplification using quadric error metrics. *Computer Graphics (SIGGRAPH '97 Proceedings)* (1997).

[10] GUÉZIEC, A. Surface simplification inside a tolerance volume. Research Report RC-20440, IBM, March 1996.

[11] HE, T., HONG, L., VARSHNEY, A., AND WANG, S. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics 2*, 2 (June 1996), 171–184.

[12] HECKBERT, P., AND GARLAND, M. Survey of polygonal surface simplification algorithms. Tech. Rep. CMU-CS-95-194, Carnegie Mellon University, 1995.

[13] HOPPE, H. Progressive meshes. *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), 99–108.

[14] HOPPE, H., DEROSE, T., DUCHAMP, T., MCDONALD, J., AND STUETZLE, W. Mesh optimization. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 19–26.

[15] HUDSON, J. *Piecewise Linear Topology*. W.A. Benjamin, Inc, 1969.

[16] JUNGERMAN, M., AND RINGEL, G. Minimal triangulations on orientable surfaces. *Acta Mathematica 145*, 1-2 (1980), 121–154.

[17] LANG, V., AND LIENHARDT, P. Geometric modeling with simplicial sets. In *Pacific Graphics '95* (August 1995), pp. 475–493.

[18] LUEBKE, D. Hierarchical structures for dynamic polygonal simplification. TR 96-006, Department of Computer Science, University of North Carolina at Chapel Hill, 1996.

[19] PAOLUZZI, A., BERNARDINI, F., CATTANI, C., AND FERRUCCI, V. Dimension-independent modeling with simplicial complexes. *ACM Transactions on Graphics 12*, 1 (January 1993), 56–102.

[20] RONFARD, R., AND ROSSIGNAC, J. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum (Proceedings of Eurographics '96) 15*, 3 (1996), 67–76.

[21] ROSSIGNAC, J., AND BORREL, P. Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunii, Eds. Springer-Verlag, 1993, pp. 455–465.

[22] SCHAUFLER, G., AND STÜRZLINGER, W. Generating multiple levels of detail from polygonal geometry models. In *Virtual Environments '95 (Eurographics Workshop)* (January 1995), M. Göbel, Ed., Springer Verlag, pp. 33–41.

[23] SCHMALSTIEG, D., AND SCHAUFLER, G. Smooth levels of detail. In *Proc. of IEEE 1997 Virtual Reality Annual Intnl. Symp.* (1997), pp. 12–19.

[24] SCHROEDER, W., ZARGE, J., AND LORENSEN, W. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proceedings) 26*, 2 (1992), 65–70.

[25] SPANIER, E. H. *Algebraic Topology*. McGraw-Hill, New York, 1966.

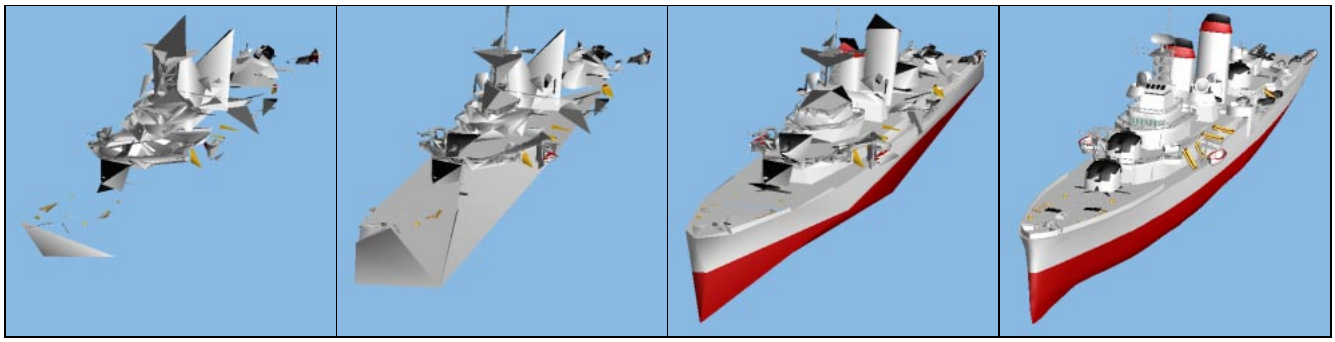
[26] TAUBIN, G., AND ROSSIGNAC, J. Geometry compression through topological surgery. Research Report RC-20340, IBM, January 1996.

[27] TURK, G. Re-tiling polygonal surfaces. *Computer Graphics (SIGGRAPH '92 Proceedings) 26*, 2 (1992), 55–64.

[28] WAVEFRONT TECHNOLOGIES, INC. *Wavefront File Formats, Version 4.0 RG-10-004*, first ed. Santa Barbara, CA, 1993.

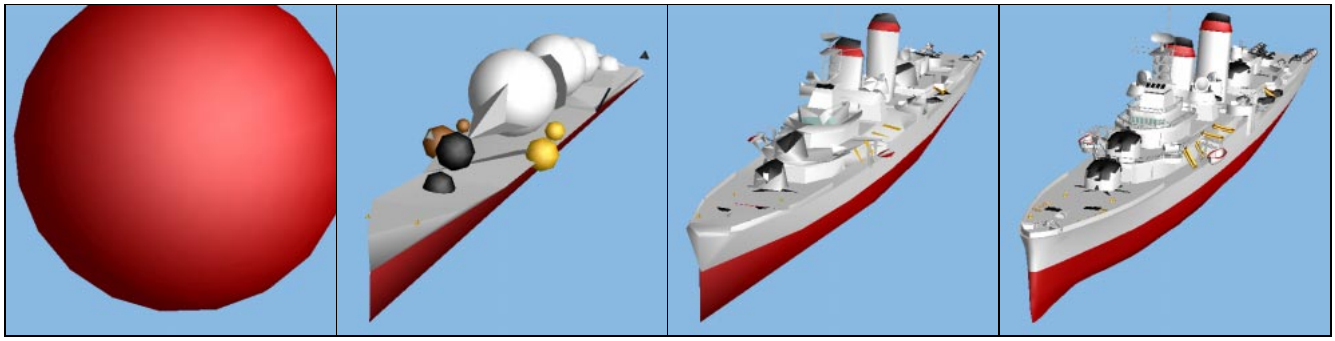
[29] WEILER, K. The radial edge structure: a topological representation for non-manifold geometric boundary modeling. In *Geometric modeling for CAD applications*. Elsevier Science Publish., 1988.

[30] WITTEN, I., NEAL, R., AND CLEARY, J. Arithmetic coding for data compression. *Communications of the ACM 30*, 6 (June 1987), 520–540.



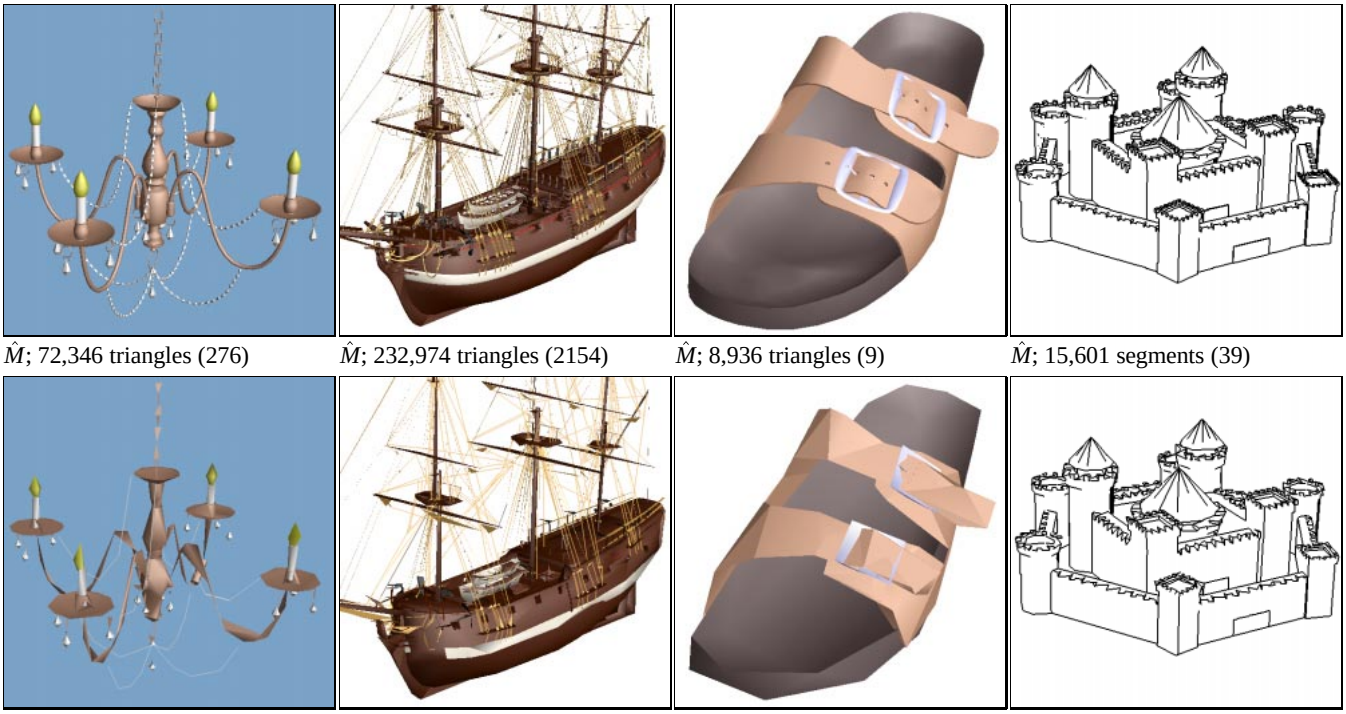
M^0; 1,154 verts; 2,522 tris M^{1739}; 2,893 verts; 6,000 tris M^{2739}; 3,893 verts; 8,000 tris $M^{n=82645}$; 83,799 verts; 167,744 tris

Figure 7: From a given mesh $\hat{M}$, the PM representation [13] captures a sequence of meshes $M^0 \ldots M^n = \hat{M}$. Because all approximations M^i must have the same topological type, the base mesh M^0 may still be complex.



M^1; $\{1, 0, 0\}$; (1) M^{50}; $\{14, 3, 66\}$; (18) M^{1000}; $\{5, 89, 1517\}$; (56) $M^{n=83799}$; $\{0, 0, 167744\}$; (117)

Figure 8: In contrast, the PSC representation captures a sequence of models $M^1 \ldots M^n = \hat{M}$ in which the base model M^1 always consists of a single vertex. All geometric and topological information is encoded progressively by a sequence of generalized vertex split transformations. The image captions indicate the number of principal $\{0, 1, 2\}$-simplices respectively and the number of connected components (in parenthesis). Note that even M^{1000} looks markedly better than the 8000-triangle PM approximation.



$\hat{M}$; 72,346 triangles (276) $\hat{M}$; 232,974 triangles (2154) $\hat{M}$; 8,936 triangles (9) $\hat{M}$; 15,601 segments (39)

M^{500}; $\{3, 52, 674\}$; (50) M^{3000};$\{239, 495, 3189\}$ (587) M^{100}; $\{0, 0, 170\}$; (2) M^{1000}; $\{20, 1265, 0\}$; (33)

Figure 9: For each column, the top row shows the original model and the bottom row shows one approximation in the PSC sequence. The image captions indicate the number of principal $\{0, 1, 2\}$-simplices respectively and the number of connected components (in parenthesis).

Progressive Meshes and Recent Extensions

Hugues Hoppe
Microsoft Research

SIGGRAPH 97 Course
Multiresolution Surface Modeling

Meshes in computer graphics

mesh *M*

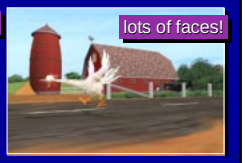
V *F*

Vertex 1 x_1 y_1 z_1
Vertex 2 x_2 y_2 z_2
…

Face 1 2 3
Face 3 2 4
Face 4 2 7
…

(appearance attributes: normals, colors, textures, ...)

Complex meshes



Challenges:
- rendering
- storage
- transmission

Talk outline

SIGGRAPH
'96 • Progressive mesh (PM) representation
- continuous-resolution
- efficient
- progressive transmission

'97 • View-dependent refinement of PM's

'97 • Progressive simplicial complex (PSC) repr.

Mesh simplification techniques

[Schroeder-etal92]
[Hoppe-etal93]
[Rossignac-Borrel93]
[Cohen-etal96]
[Garland-Heckbert97]
...

?

Traditional level-of-detail (LOD)

[Clark76]
[Funkhouser93]

close

distance from viewer?

far

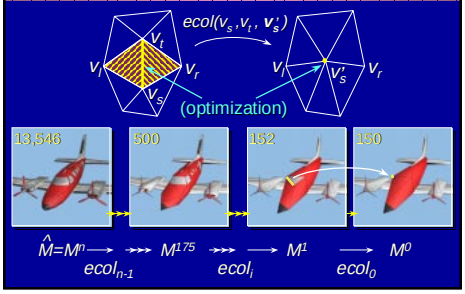
Concern: transitions may "pop"
→ would like smooth LOD

New PM representation

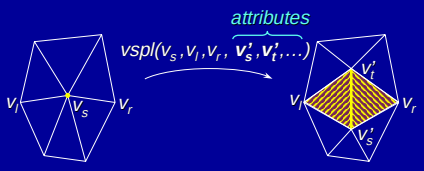
Basic idea:

- Simplify arbitrary mesh through sequence of edge collapse transformations.
- Record sequence of inverse transformations, called vertex splits.

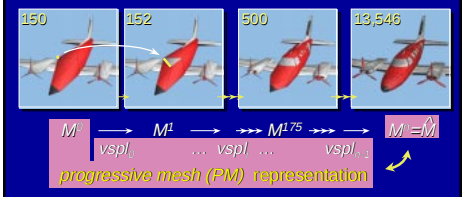
Edge collapse → Simplification



Invertible! Vertex split transformation

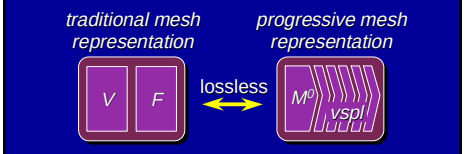


Reconstruction process



VIDEO: PM construction and LOD

Conversion

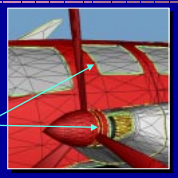


Optimization process

- Various metrics (speed vs. accuracy)
- Typically performed off-line

How to select edge collapses

- Preserve *appearance*:
  - geometric shape
  - scalar fields (e.g. color, normals)
  - discontinuity curves



$$E = \sum_{\text{points}} \int\int (e_{shape} + e_{scalars}) dA + \sum_{\text{points}} \int (e_{disc}) dL$$

Error metric: point sampling

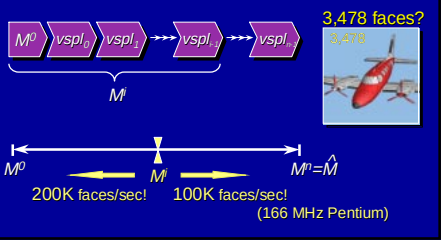
Selecting edge collapses

- Greedy algorithm: always collapse edge resulting in smallest ΔE.
- Optimize position and attributes of resulting vertex.

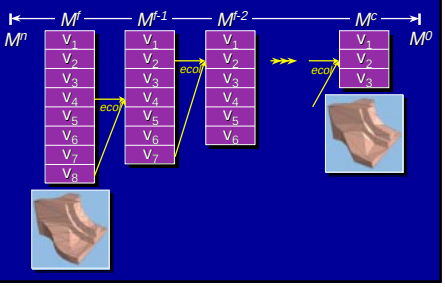
Simplification rates: ~ 30 faces/sec
- off-line process
- could use faster, simpler metrics

Application: Continuous-resolution LOD

From PM, extract M^i of any desired complexity.

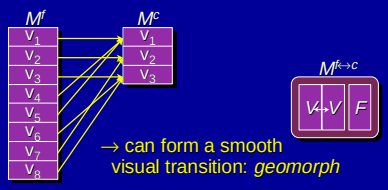


Property: Vertex correspondence



Application: Smooth transitions

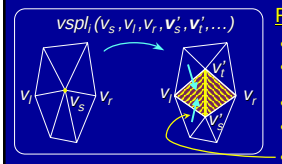
Correspondence is a surjection:



→ can form a smooth visual transition: *geomorph*

VIDEO: PM geomorphs

Application: Mesh compression



Record:

- v_s ($\log_2 i$ bits)
- v_l & v_r (~5 bits)
- v_t' - v_s (delta)
- v_s' - v_s (delta)
- predict materials
- …

Analysis:

- connectivity: $(4+\log_2 n)n$ bits vs. $(6\log_2 n)n$ bits
- geometry: $\sim 30n$ bits vs. $96n$ bits

[Deering95]

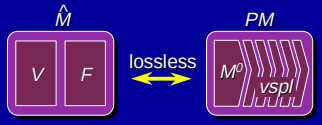
Application: Progressive transmission

Transmit records progressively:

Receiver displays: M^0, M^i, $\hat{M}$

(~ progressive GIF & JPEG)

PM Summary



- single resolution
- continuous-resolution
- smooth LOD
- space-efficient
- progressive

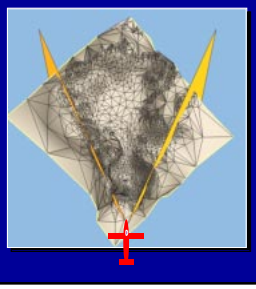
[Microsoft DirectX 5.0]

VIDEO/DEMO: PM results

View-Dependent Refinement of Progressive Meshes

[SIGGRAPH 97]

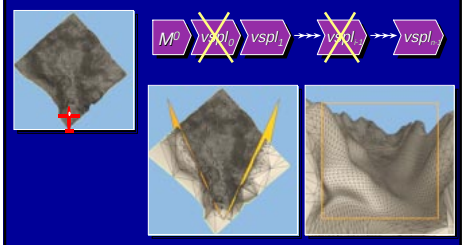
Adaptive refinement: motivation



Related work

- Height fields (e.g. terrains)
  - [Cignoni etal95]
  - [De Floriani etal96]
  - [Lindstrom etal96] …
- Parametric surfaces (e.g. NURBS)
  - [Rockwood etal89]
  - [Abi-Ezzi & Subramaniam93]
  - [Kumar etal95] …
- Arbitrary meshes
  - [Xia-Varshney96]

Using progressive meshes

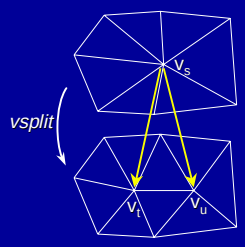


(e.g. view frustum)

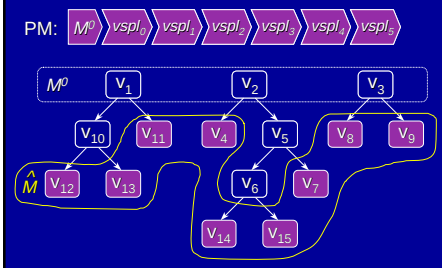
Contributions

- PM → vertex hierarchy
  → selective refinement
- Dependencies → consistent framework
- View-dependent refinement criteria

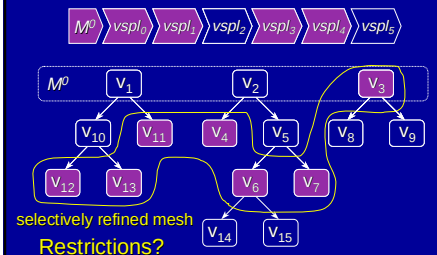
Parent-child vertex relations



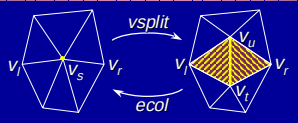
Vertex hierarchy



Selective refinement



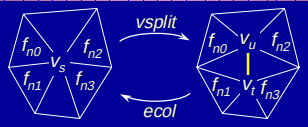
Legality conditions?



- *vsplit* legal if:
  - v_s is active
  - v_l and v_r are active
  } [SIGGRAPH96]
- *ecol* legal if:
  - ? problem: consistent?

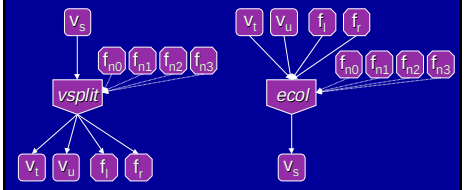
[Xia&Varshney96]: identical neighborhood

New vspl/ecol parametrizations

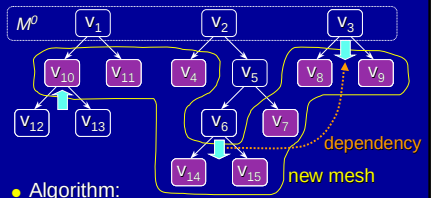


- *vsplit* legal if:
  - v_s is active
  - $f_{n0}, f_{n1}, f_{n2}, f_{n3}$ are active
- *ecol* legal if:
  - v_t, v_u are active
  - $f_{n0}, f_{n1}, f_{n2}, f_{n3}$ are adjacent

Dependencies



Runtime algorithm



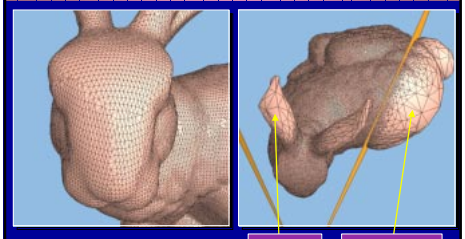
- Algorithm:
  - incremental (frame coherence)
  - efficient (~15% of frame time)
  - amortizable

View-dependent refinement criteria

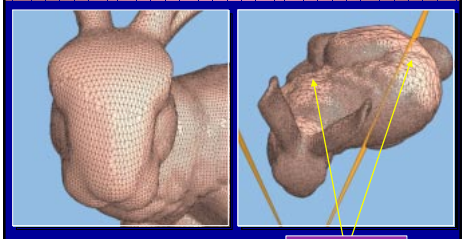
3 criteria:

- view frustum
- surface orientation
- screen-space geometric error

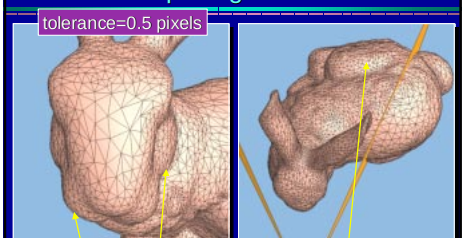
View frustum



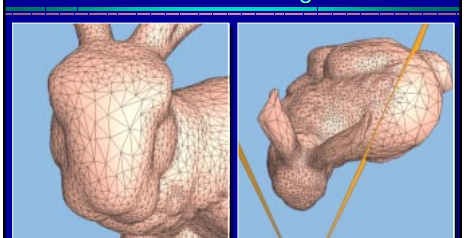
Surface orientation



Screen-space geometric error

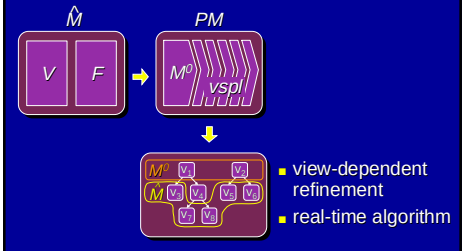


All three criteria together



VIDEO: Selective Refinement

Selective Refinement Summary



- view-dependent refinement
- real-time algorithm

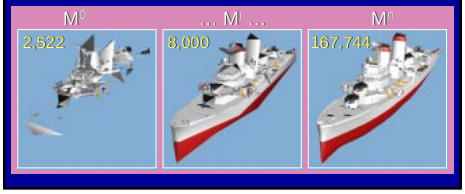
Progressive Simplicial Complexes

[SIGGRAPH 97]

(Joint work with Jovan Popovic)

PM restrictions:

- Supports only “meshes”
  (orientable, 2-dimensional manifolds)
- Preserves topological type

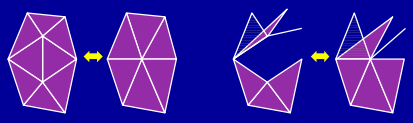


Progressive Simplicial Complexes

- Represent arbitrary triangulations:
  - any dimension,
  - non-orientable,
  - non-manifold,
  - non-regular, …
- Progressively encode both geometry **and topology**.

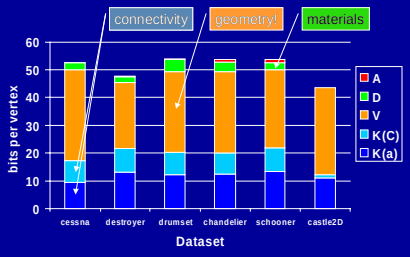
Generalization

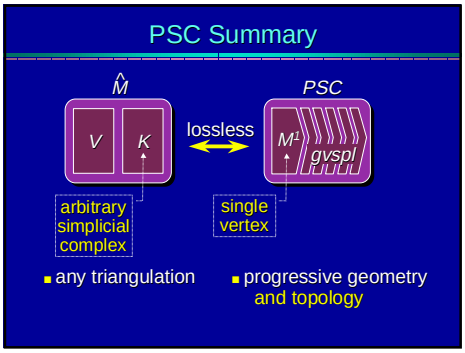
PM		*PSC*
edge collapse (*ecol*)	⟷	vertex unification (*vunify*)
vertex split (*vspl*)	⟷	generalized vertex split (*gvspl*)

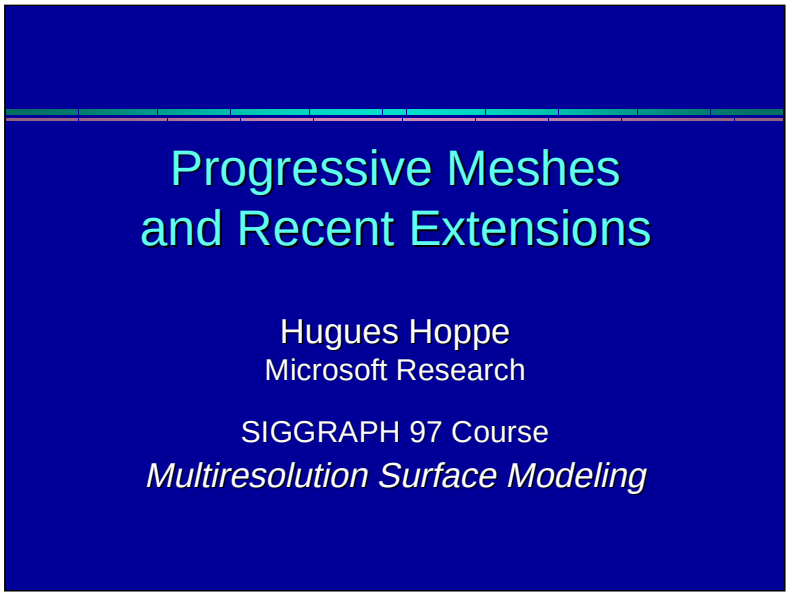


LOD sequence

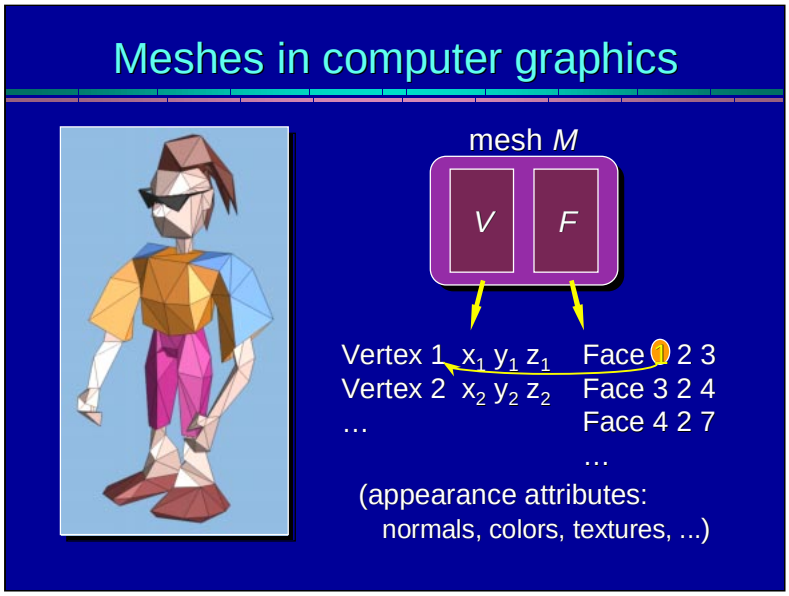
Space analysis




PSC Summary
$\hat{M}$
PSC
V
K
lossless
M^1
gvspl
arbitrary simplicial complex
single vertex
any triangulation
progressive geometry and topology



Hugues Hoppe - SIGGRAPH 97 course - Multiresolution Surface Modeling

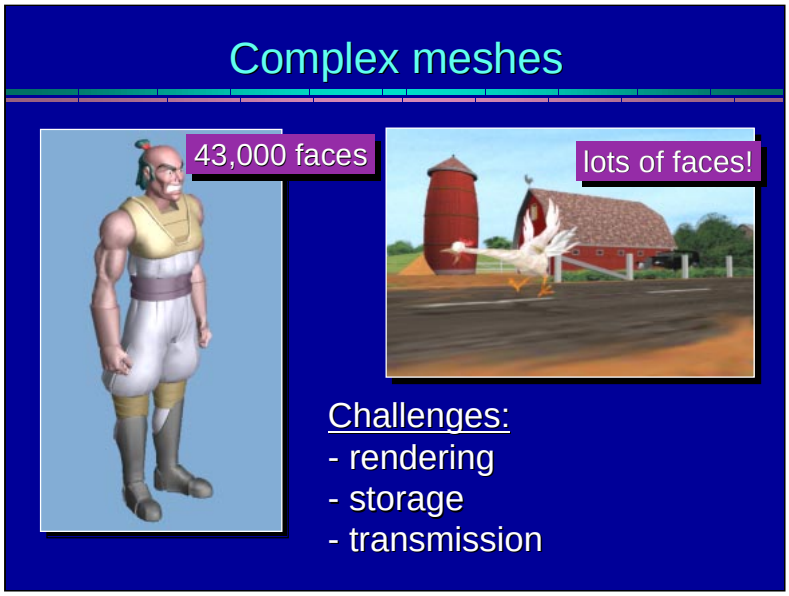


In computer graphics, geometric models are commonly represented as triangle meshes.

As shown here, a mesh consists of a set of triangular faces pasted together along their edges and meeting at a common set of vertices

In a traditional representation, a mesh is represented as a set of vertices containing xyz coordinates, and a set of faces containing indices referring back to the vertices.

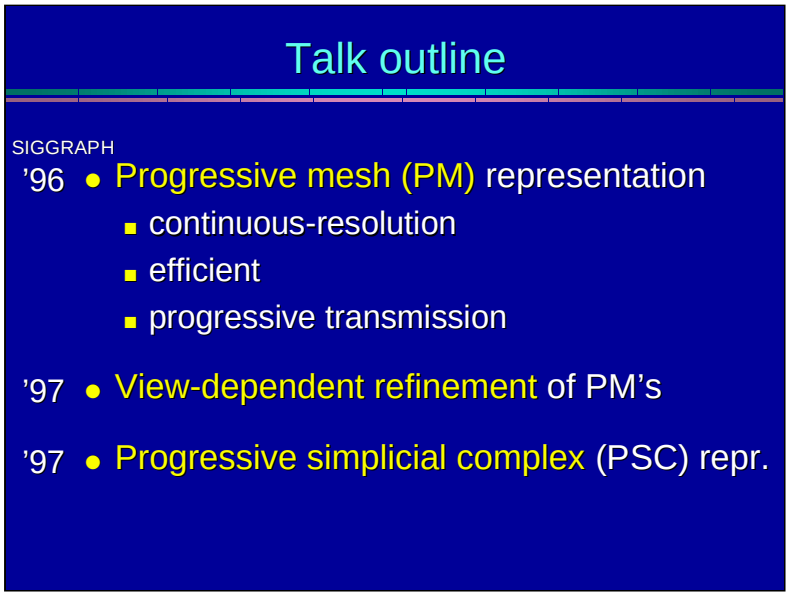
Besides geometry, other appearance attributes are often present in a mesh, such as vertex normals, vertex colors, material attributes, and texture maps. These appearance attributes are also associated with the vertices and faces of the mesh.



There is a growing expectation for realism in computer graphics, and as a result geometric models are becoming very complex.

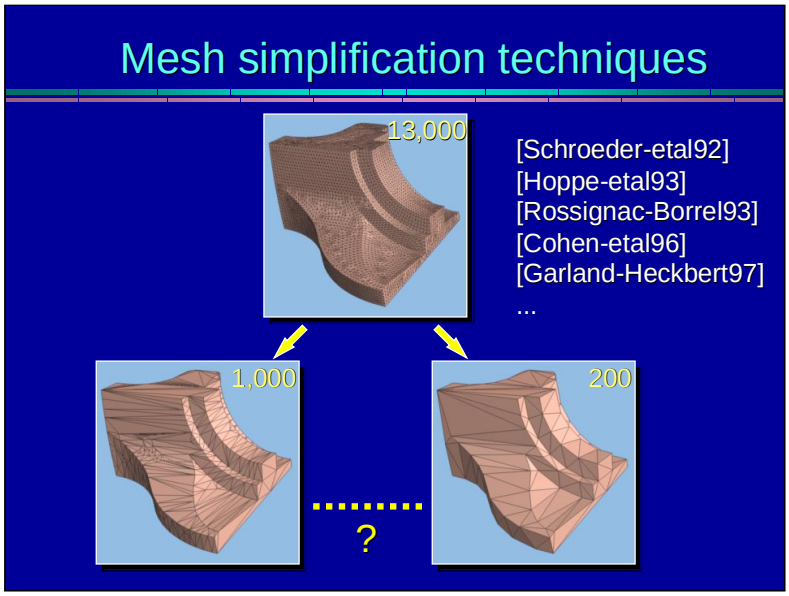
And, computer graphics scenes are typically composed of many meshes, so the overall complexity can be huge.

This presents several challenges, because of limitations in rendering performance, storage capacities, and transmission bandwidths.



In this talk, I'll first describe the progressive mesh representation introduced last year. The progressive mesh representation is a new format for storing and transmitting arbitrary triangle meshes. This new multiresolution representation has a number of nice properties. As I'll demonstrate in this talk, it captures a continuous-resolution family of approximations; it is a space-efficient representation; and, it can be transmitted progressively.

Next, I'll present an overview of two extensions to progressive meshes, which appear in SIGGRAPH 97.

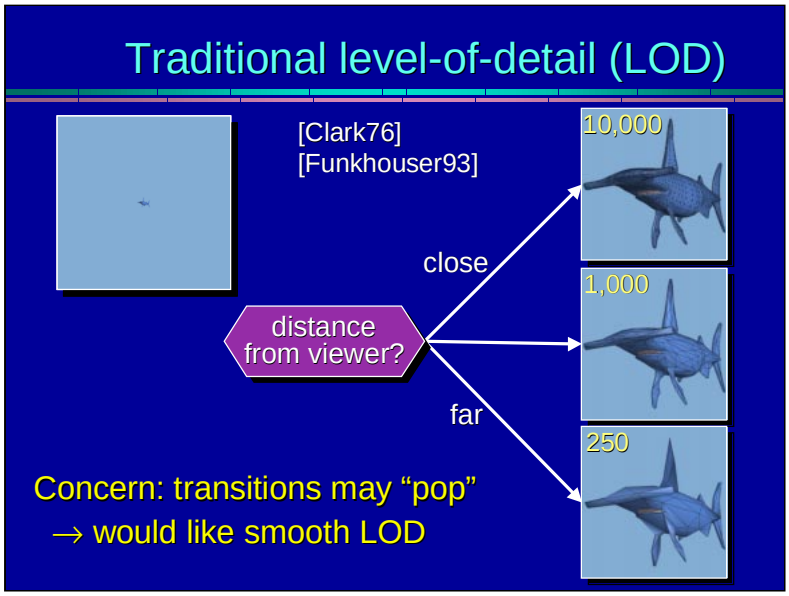


Several methods have been developed to simplify meshes. I've listed a number of these on this slide.

The goal of these methods is to reduce the number of faces in the mesh while attempting to preserve its original shape.

These methods typically contain some complexity parameters (knobs) that can be varied to obtain several meshes with different trade-off's of size and accuracy.

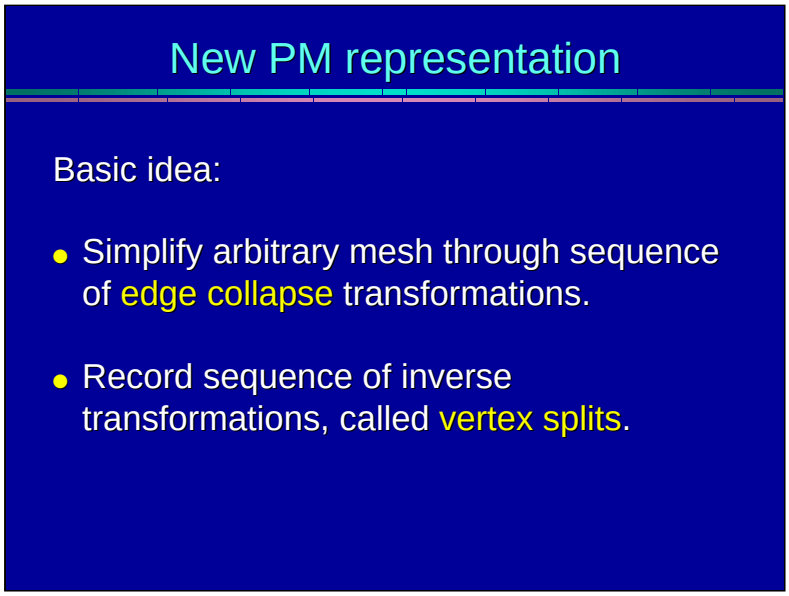
However this only yields a discrete set of meshes. Ideally we would want to capture a continuum that spans all these meshes.



These simplified meshes are often used in the context of level-of-detail.

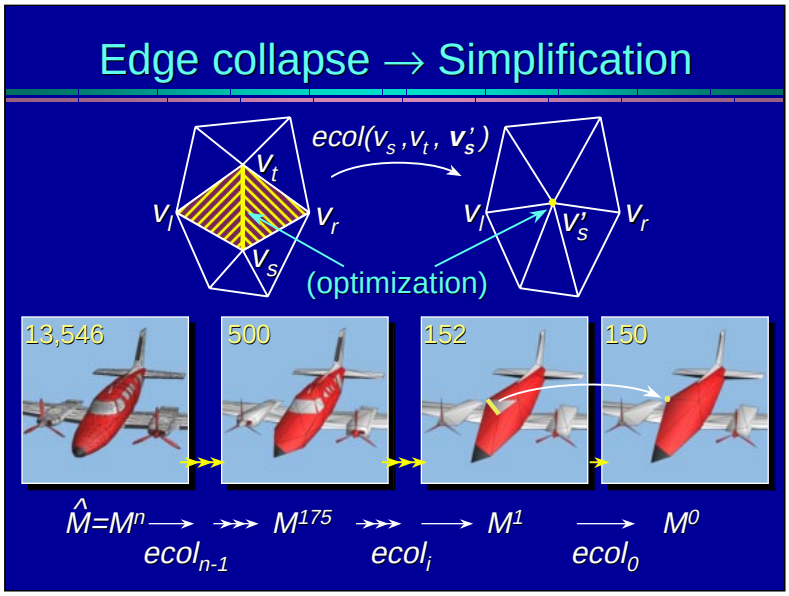
The idea is to use a fully-detailed mesh when the model is close to the viewer, and to substitute coarser and coarser approximations as the object recedes away from the viewer.

One concern with this approach is that instantaneous switching between the various meshes may result in a “popping” perceptible to the user. Therefore we would like to have a level-of-detail representation that supports smooth transitions.



I'll now introduce the progressive mesh representation.

The basic idea is to first simplify an arbitrary mesh through a sequence of edge collapse transformations, and while doing so, to record the sequence of inverse transformations, called vertex splits, that are sufficient to reconstruct the original mesh from the simplified one.

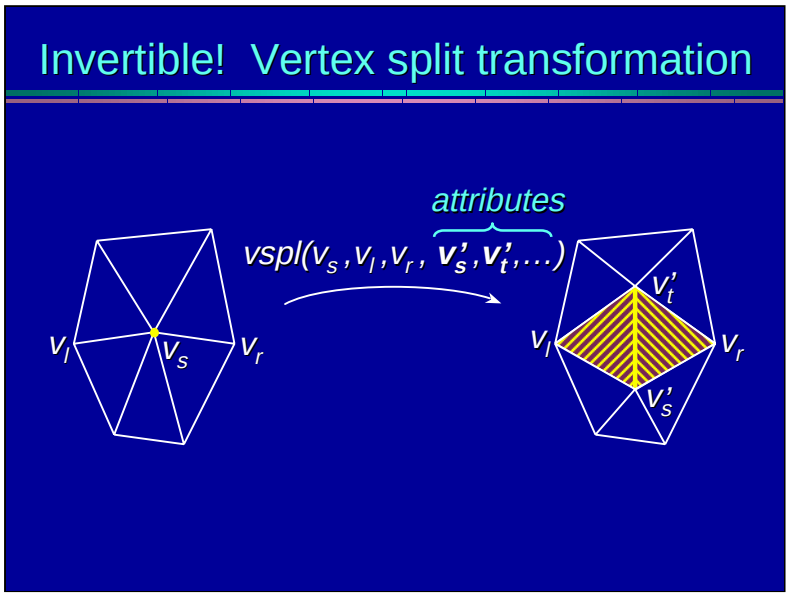


An edge collapse transformation takes two vertices adjacent to an edge of the mesh, and unifies them into one. In the process, one or two faces of the mesh are deleted.

Of course, there is an optimization procedure for selecting what edges to collapse (and in what order), and what position and attributes to assign to the resulting vertex. I'll return to that topic in a few slides.

Thus, from the original mesh, denoted M^n, we apply a sequence of edge collapse transformations to obtain a very coarse base mesh, M^0.

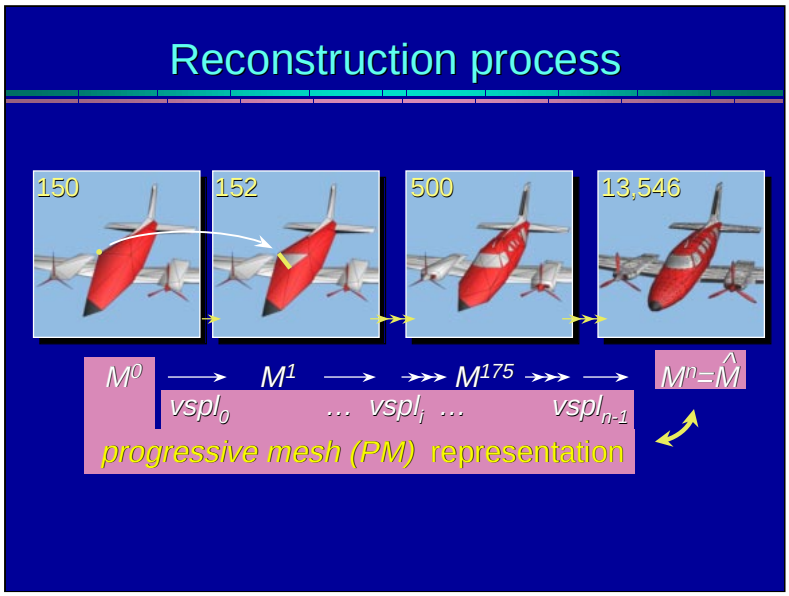
You can see the effect of the very last edge collapse transforming M^1 into M^0; it removes the last window from the airplane.



A key observation is that the edge collapse transformation is invertible. Its inverse is called a “vertex split”.

This operation takes as parameters a vertex v_s along with two of its neighbors v_l and v_r. It splits v_s into two vertices, creating two new faces in the process.

The vertex split record also stores the positions of the two split vertices as well as other appearance attributes associated with the new neighborhood.

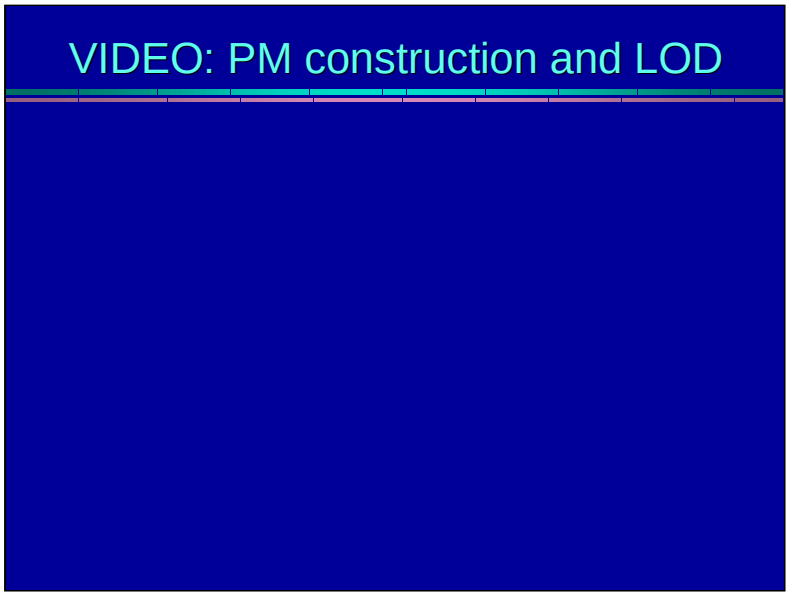


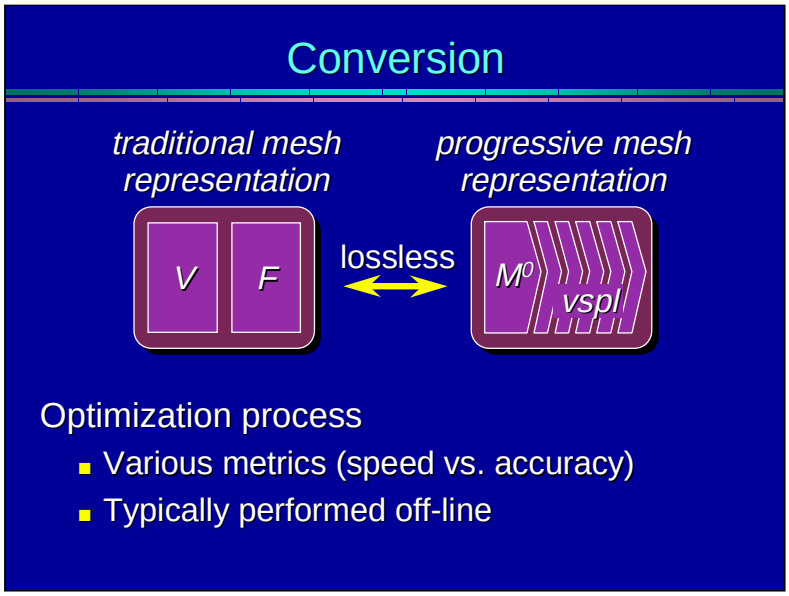
We can therefore flip the previous diagram, starting with the base mesh M^0, and apply to it a sequence of vertex split transformations.

The first vertex split introduces one vertex and two faces; subsequent vertex splits recover more and more detail; and, after all vertex splits have been applied, the original mesh M^n is recovered **exactly**.

Therefore, the base mesh M^0 together with the sequence of vertex split records forms an alternate representation for the mesh. This is what I refer to as the progressive mesh representation.

It is equivalent to knowing the original mesh. It just expresses it in a different format.


VIDEO: PM construction and LOD



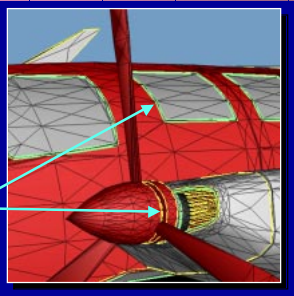
The construction of the progressive mesh representation typically involves an optimization algorithm so that the simplified meshes are good approximations to the original.

This optimization process may involve various error metrics and various searching techniques. Thus there is an obvious trade-off between the time spent constructing a PM representation and the accuracy of its approximations.

The optimization process can typically be performed off-line, as it only has to be done once per model. Thus in our work we have sought to invest some time in the optimization in order to produce approximations of good quality.

How to select edge collapses?

- Preserve *appearance*:
  - geometric shape
  - scalar fields
    (e.g. color, normals)
  - discontinuity curves



$$E = \sum_{points} \int\!\!\int (e_{shape} + e_{scalars}) dA + \sum_{points} \int (e_{disc}) dL$$

How are edge collapses to be selected?

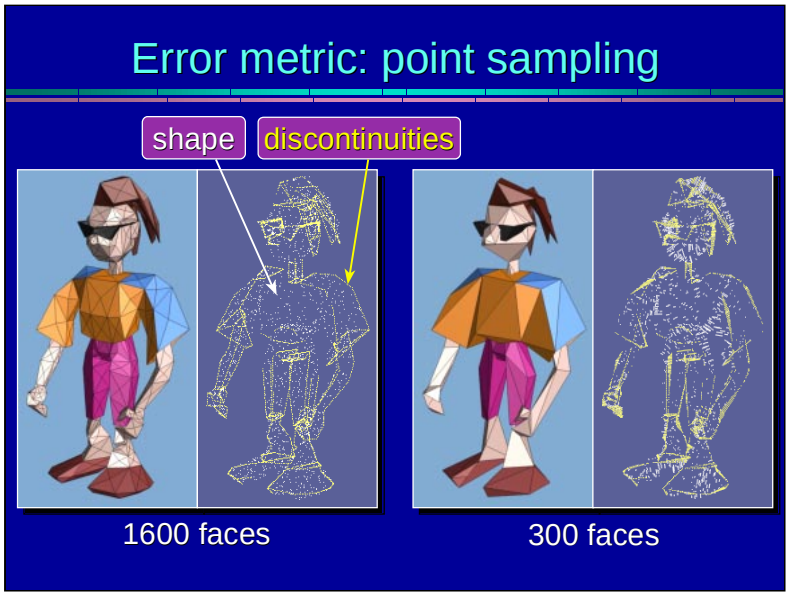
The goal of the simplification procedure is to preserve the appearance of the model.

This involves preserving its geometric shape, fields such as color and normals defined over its surface, and most importantly, discontinuity curves such as material boundaries and normal discontinuities.

These 3 goals are captured in an energy metric with 3 corresponding terms. The first two terms integrate deviation of geometric shape and scalar attributes over the surface of the model. The last term integrates the deviation of discontinuity curves over their length.

Ideally we would compute these integrals directly.

But, we approximate them by discrete sampling, using a set of points sampled from the original mesh.



The left pair of images show the original mesh and the dense set of points sampled from it.

The white points correspond to the discrete samples used for the first two energy terms (shape and scalars), and the yellow points correspond to the samples used for the last energy term (discontinuity curves).

The right pair of images show a simplified (approximating) mesh, and the projections of the sample points onto this approximation. The first and last energy terms correspond directly to the sum of squared lengths of the white and yellow segments respectively.

Selecting edge collapses

- Greedy algorithm: always collapse edge resulting in smallest ΔE.
- Optimize position and attributes of resulting vertex.

Simplification rates: ~ 30 faces/sec

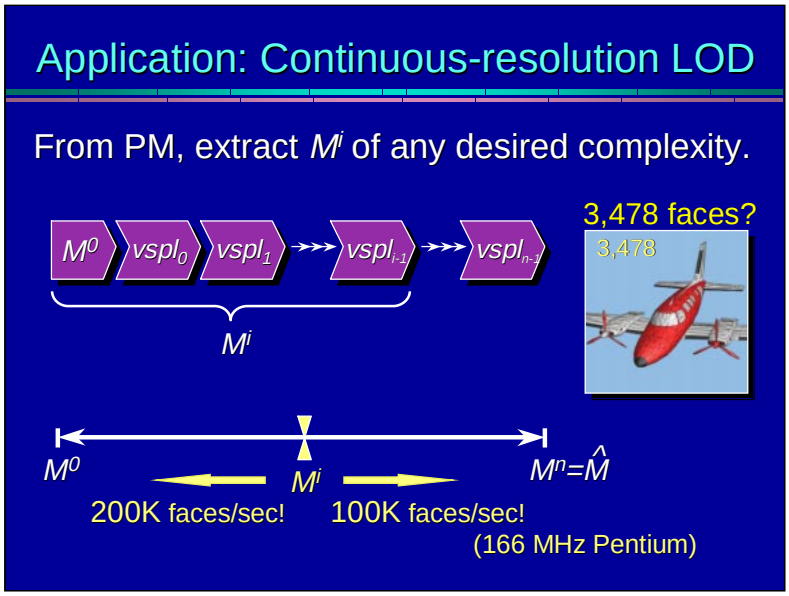
- off-line process
- could use faster, simpler metrics

We use a simple greedy algorithm to select edges to collapse. Specifically, we always collapse the edge that results in the smallest increase in overall energy.

To determine this change in energy, we optimize the position and attributes of the vertex resulting from edge collapse in order to minimize energy.

We obtain simplification rates of about 30 faces per second. This is quite slow, but the process can be performed off-line.

We could instead use simpler heuristics for simplification as done in other techniques. It's a trade-off of accuracy vs. time.



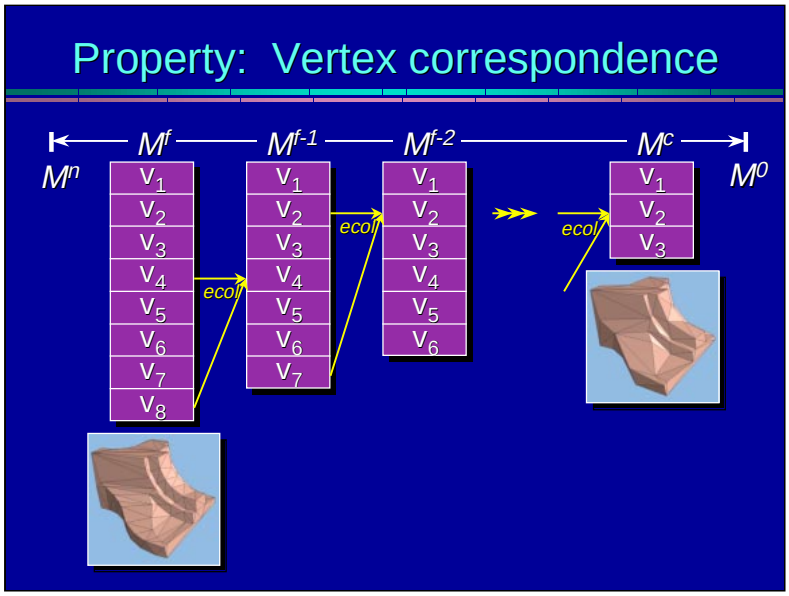
I'll now discuss several advantages of this progressive mesh representation.

First, it is a continuous-resolution representation. What I mean by this is that from it one can recover approximating meshes of any desired complexity. This is achieved by applying to the base mesh a prefix of the vertex split sequence.

So if one wants to recover a mesh with 3,478 faces, it can be retrieved very efficiently.

In effect, this defines a continuous family of approximating meshes (from the base mesh M^0 to the original mesh M^n), and we can iterate through these meshes by applying edge collapses and vertex splits.

In fact, this can be done very efficiently. On a Pentium processor, we obtain a reconstruction rate of about 100,000 faces/sec, and a simplification rate of about 200,000 faces/sec.

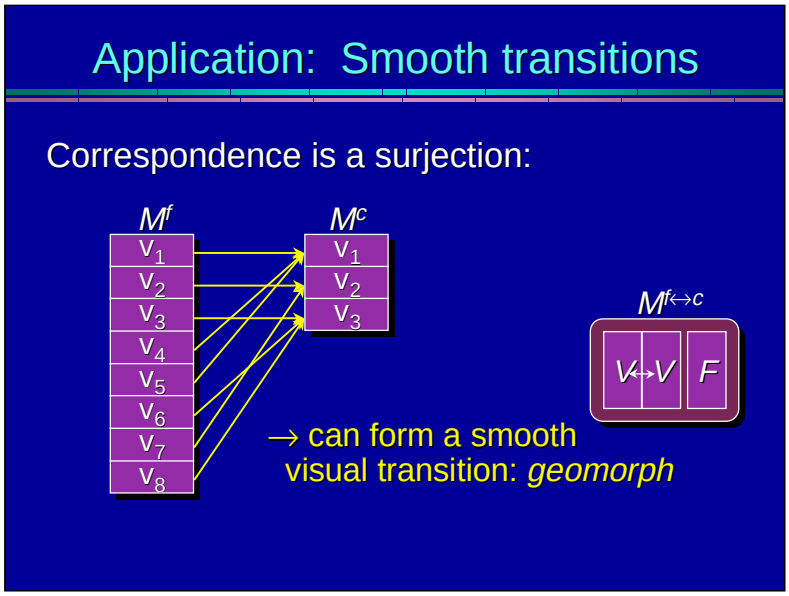


Another nice property is that there exists a natural correspondence between vertices of **any** two meshes in the family of approximations.

So, here are two meshes, a finer mesh M^f and a coarser mesh M^c.

From the finer mesh, each edge collapse unifies two vertices of the mesh into one, and this occurs repeatedly until the coarser mesh is obtained.

Now we see a correspondence between vertices of the various meshes.



When we fold together all of these correspondences, we find that they form a surjection, or “onto map” from the vertices of the finer mesh onto the vertices of the coarser mesh.

This allows us to define a smooth visual transition, or geomorph, between these two meshes.

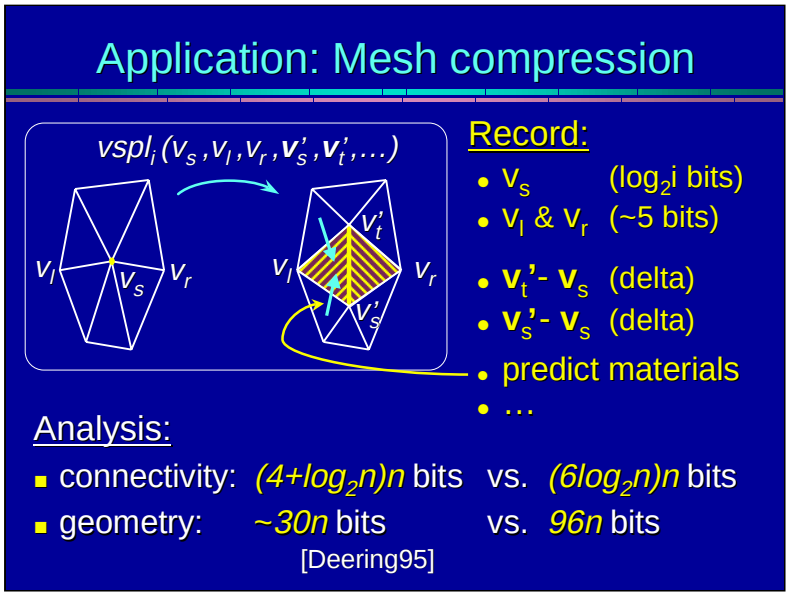
We simply take the finer mesh M^f, and for each of its vertices, interpolate between its current position and the position of its corresponding vertex in the coarser mesh.

I'll show a few examples to demonstrate this.

First, here is single geomorph between 2 meshes: the first has 400 faces and the second has 600 faces. As you can see, there is a smooth visual transition between the two.

This next example shows that we can define a sequence of geomorphs to represent an object. There are 12 geomorphs; the complexities of the meshes were chosen to be an exponential sequence.

Finally, this example shows how this can be applied to level-of-detail control. There are 8 geomorphs, from 1600 faces down to 200 faces. Even though the mesh connectivities are changing discretely, the visual transitions are all smooth.



In addition, PM is an effective scheme for compressing meshes. The key is that each vertex split applies a local perturbation to the mesh that can be encoded concisely. Let us consider the various parameters of a vertex split record.

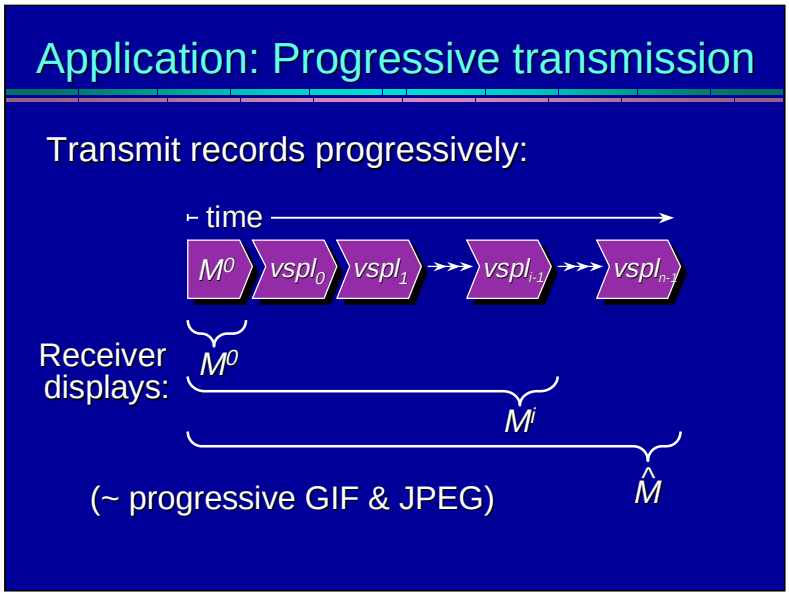
First, the vertex v_s must be specified among the vertices of the mesh reconstructed so far. Since there are I vertices in iteration I, this requires log_2 I bits. The vertices v_l and v_r can be specified among the neighbors of v_s and on average this requires only about 5 bits.

The positions of the two new split vertices can be specified as deltas from the position of the old vertex, and since the deltas are small in magnitude, this can be encoded more concisely than absolute positions.

Finally, other attributes such as material properties for the two new faces can be predicted from the neighboring faces.

The bottom line is that the mesh connectivity requires about (4+log_2 n)n bits, which is much smaller than a traditional IndexedFaceSet representation. It's about as concise as triangle strips, but not as optimal as work by [Deering95] or [Taubin&Rossignac96].

The geometry offers excellent delta-encoding (when quantized), particularly if we restrict the placement of simplified vertices v_s.



As you may already have guessed, progressive meshes are a natural representation for progressive transmission.

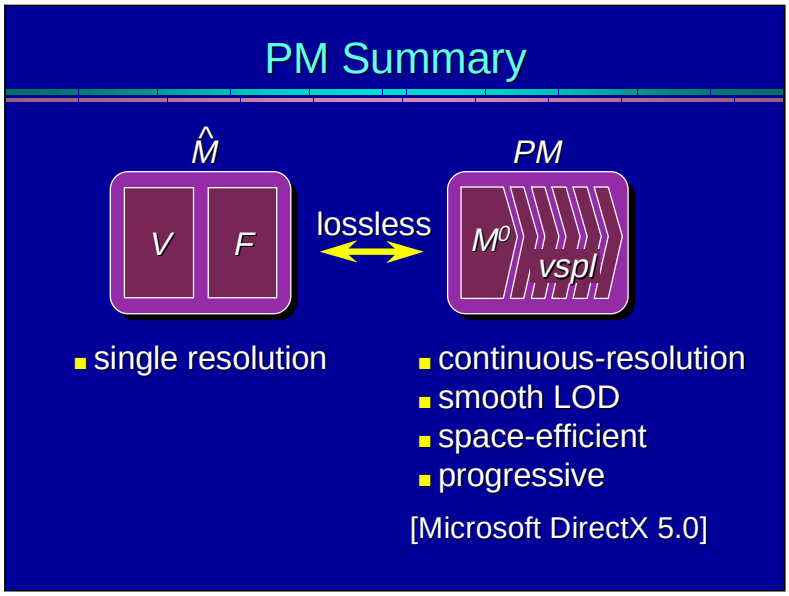
The base mesh is transmitted first, in a traditional format, followed by the stream of vertex split records.

The receiving process can quickly display this coarse base mesh.

Then, as more and more vertex split records arrive, it can at any point in time display the mesh constructed so far. Reconstruction can proceed at over 100,000 faces per second.

At the end of transmission, the original mesh has been recovered exactly.

Note that this is analogous to the way that images are transmitted over the Web using progressive GIF and JPEG.



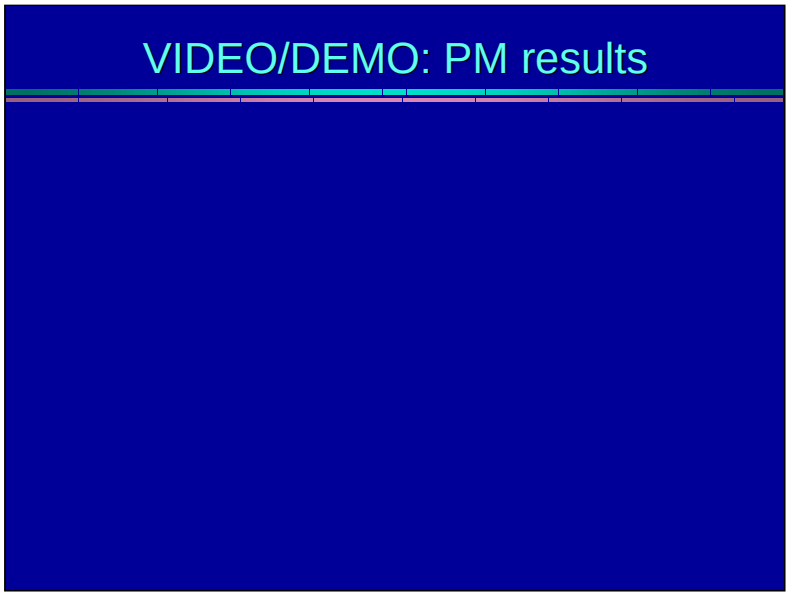
To summarize, a mesh is traditionally represented as a set of vertices and a set of faces.

Such a mesh can be converted losslessly into a progressive mesh representation, composed of a base mesh and a sequence of vertex split records.

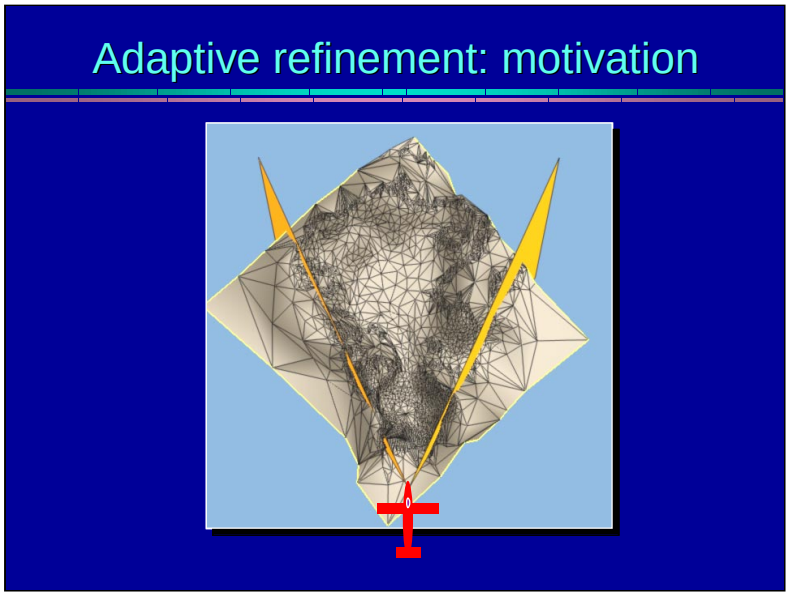
This new representation has a number of nice properties.

It captures a continuous-resolution family of approximations; it supports smooth level-of-detail; it is space-efficient; and, it can be transmitted progressively.

An initial implementation of progressive meshes will hopefully be available in Microsoft's DirectX 5.0 graphics interface.


VIDEO/DEMO: PM results

The progressive mesh can also be selectively refined based on changing view parameters.



With a large-scale model, it is often desirable to adapt the resolution of the mesh over the surface of the model.

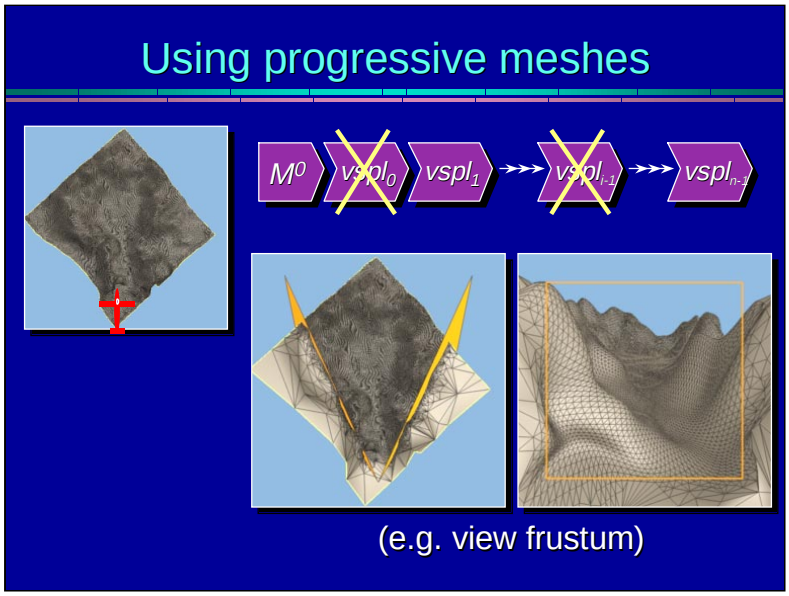
For instance, if the user is flying over a dense terrain model, the mesh need be refined only within the view frustum (the portion of the terrain visible in the viewport). In addition, it's desirable to refine the mesh more densely near the viewer.

Related work

- Height fields (e.g. terrains)
  - [Cignoni etal95]
  - [De Floriani etal96]
  - [Lindstrom etal96] …
- Parametric surfaces (e.g. NURBS)
  - [Rockwood etal89]
  - [Abi-Ezzi & Subramaniam93]
  - [Kumar etal95] …
- Arbitrary meshes
  - [Xia-Varshney96]

Several schemes adapt the refinement of meshes for the cases of height fields and parametric surfaces.

Like Xia & Varshney, we develop a framework for selectively refining an arbitrary mesh.

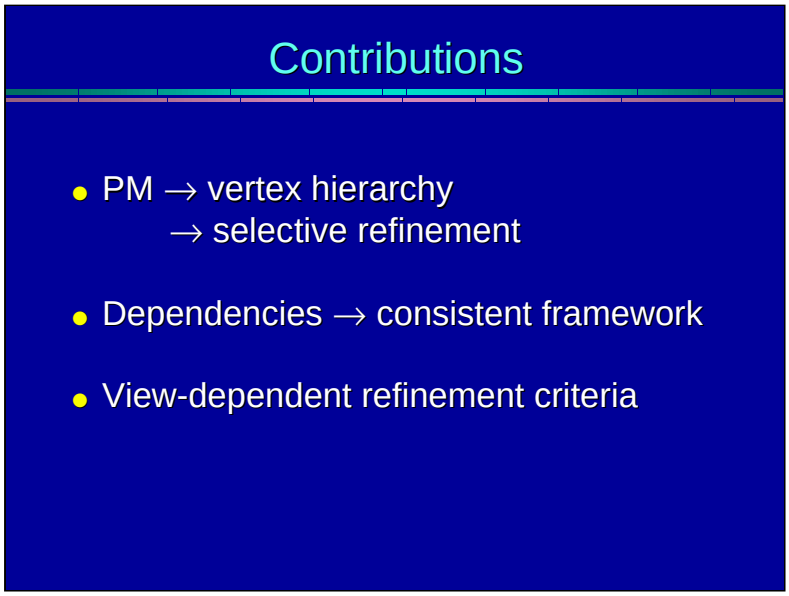


As showed earlier, the progressive mesh naturally defines a **linear** sequence of **view-independent** LOD approximations.

However, it's also possible to selectively refine the mesh, in order to adapt the resolution of the mesh in a non-uniform way.

The basic idea is apply to the base mesh only a selected subset of the vertex split sequence.

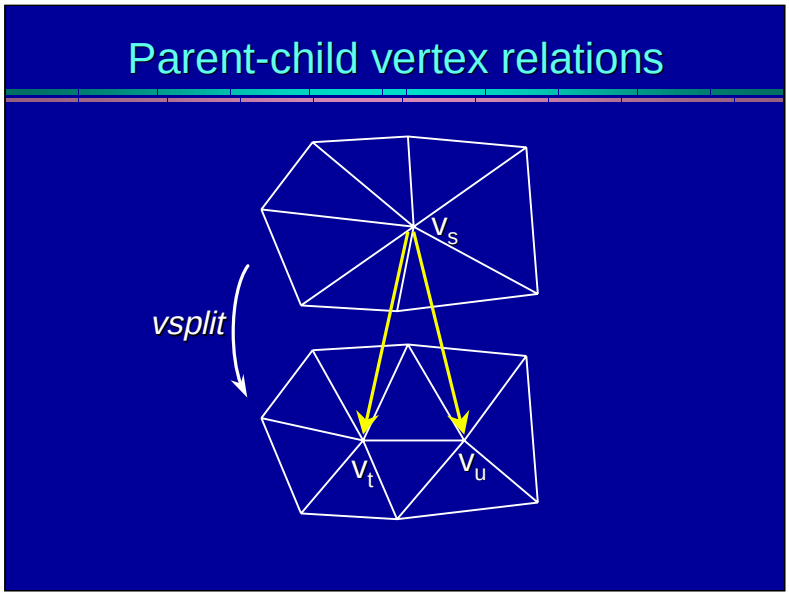
In this simple example, vertex splits were only applied if they modified the mesh within a given view frustum, show in orange.



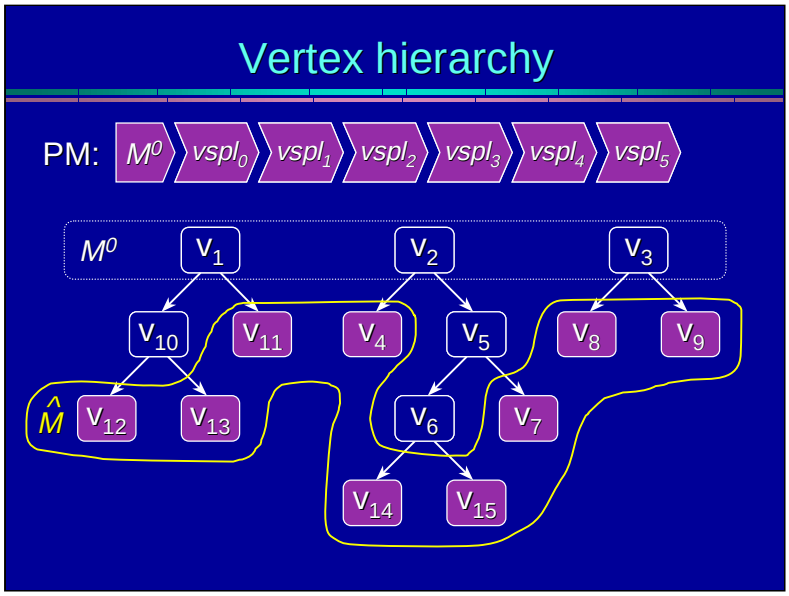
I'll first show that the progressive mesh representation naturally defines a vertex hierarchy, and that this hierarchy can be used to establish a selective refinement framework.

Next, I'll introduce a set of dependencies that make this framework consistent.

Finally, I'll present some criteria for determining where to refine and coarsen the mesh based on changing view parameters.



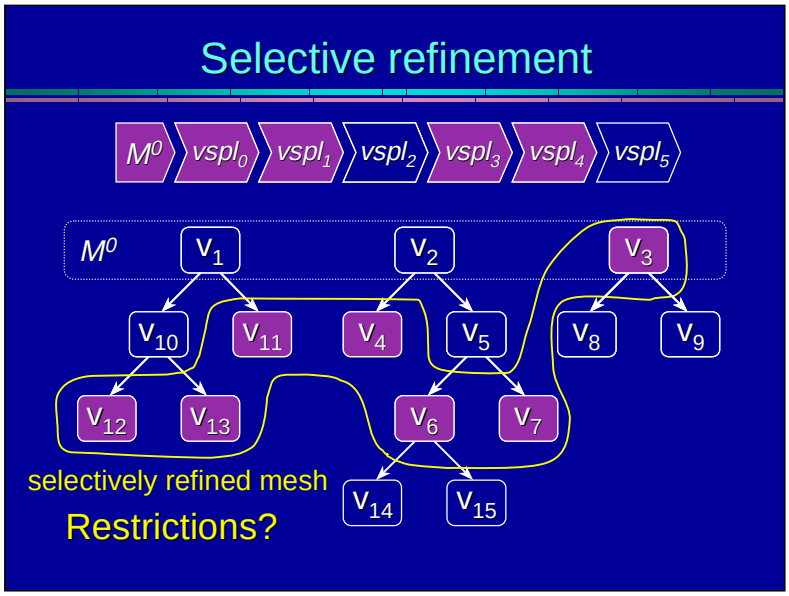
In this rotated diagram, we see that a vertex split refinement transformation defines a relation between the “parent” vertex v_s and the two “child” vertices that result from the split.



If we begin with the vertices of the base mesh M^0 and apply the vertex splits in order, we see that the parent-child vertex relations form a “forest” in which the vertices of M^0 are the root nodes and the vertices of the original, fully-refined mesh M^n are the leaf nodes.

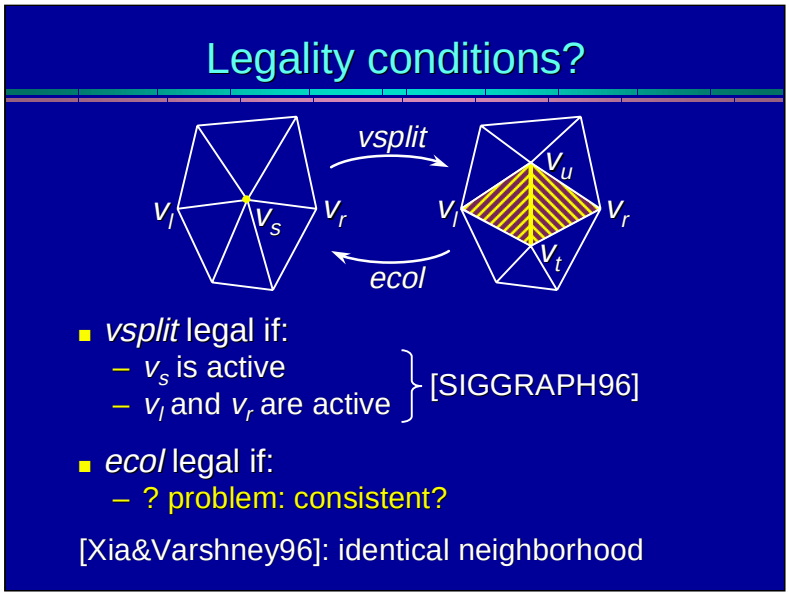
The intermediate meshes M^i, 0<i<n, from the progressive mesh sequence correspond to “vertex fronts” through this vertex hierarchy.

A similar vertex hierarchy also appears in the work of [Xia & Varshney 96] and [Luebke 96].



Given this vertex hierarchy, we can begin to think about applying both vertex splits and their inverses (edge collapses) selectively and out-of-sequence. This corresponds to incrementally moving the vertex front up and down in the hierarchy.

Are there any restrictions as to what **vspl**/**ecol** transformations can be applied? Yes, because these transformations expect that the mesh neighborhood which they modify has a particular configuration.

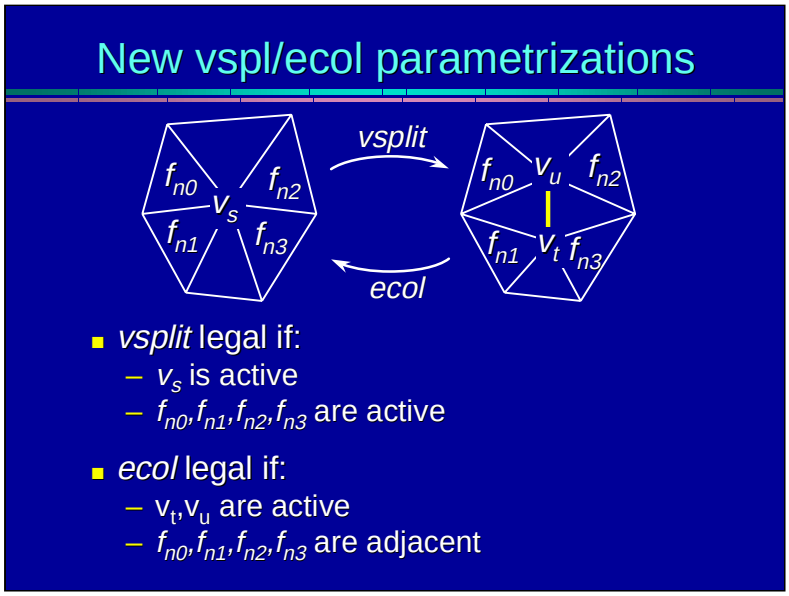


My SIGGRAPH 96 paper described a particular set of legality conditions for vertex split transformations.

However, no legality conditions were given for edge collapse transformations. Therefore, after a change in view parameters, it was necessary to start again from the base mesh to produce a new selectively refined mesh.

I attempted to find legality conditions for edge collapses, but was unable to form a consistent framework without introducing many dependencies between the transformations.

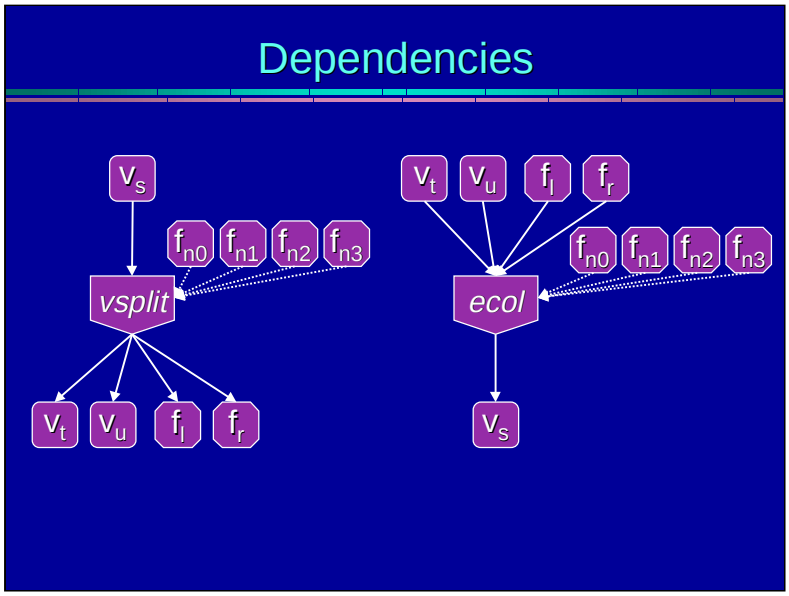
In the scheme of Xia and Varshney, a transformation is legal if the entire ring of vertices around v_s (or v_t and v_u) is present in the current mesh.



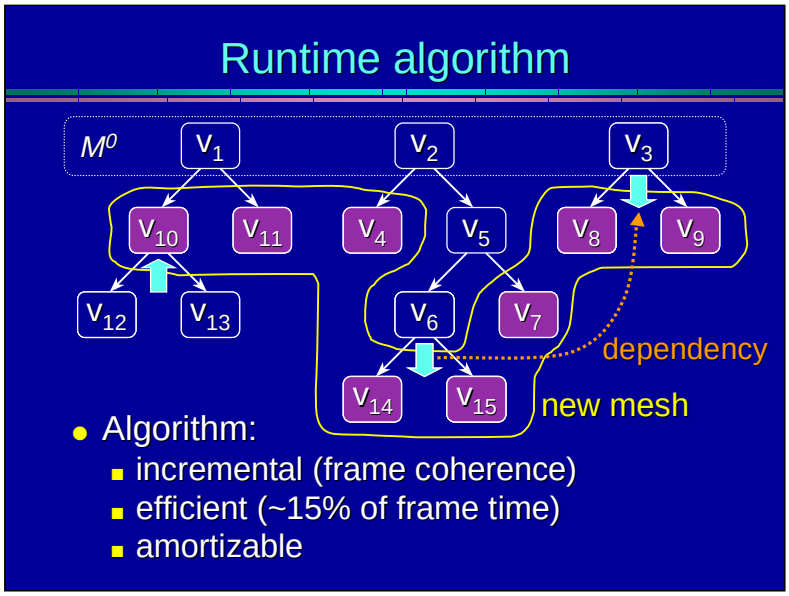
I instead found a new parametrization for the vertex split and edge collapse transformations, as shown at the top, that permits a consistent framework with few dependencies.

A vertex split is legal if the center vertex v_s is active (that is, the vertex front passes through that vertex), and if the four faces f_n0, f_n1, f_n2, f_n3 expected by the vertex split are also all active.

An edge collapse is legal if the two child vertices v_t and v_u are active (that is, the vertex front passes through two siblings), and if the four faces adjacent to the two faces f_l,f_r to remove are f_n0, f_n1, f_n2, f_n3.



These dependencies are illustrated in the above diagram.



At runtime, we traverse the active vertex front, and for each vertex, decided whether it should be refined or coarsened.

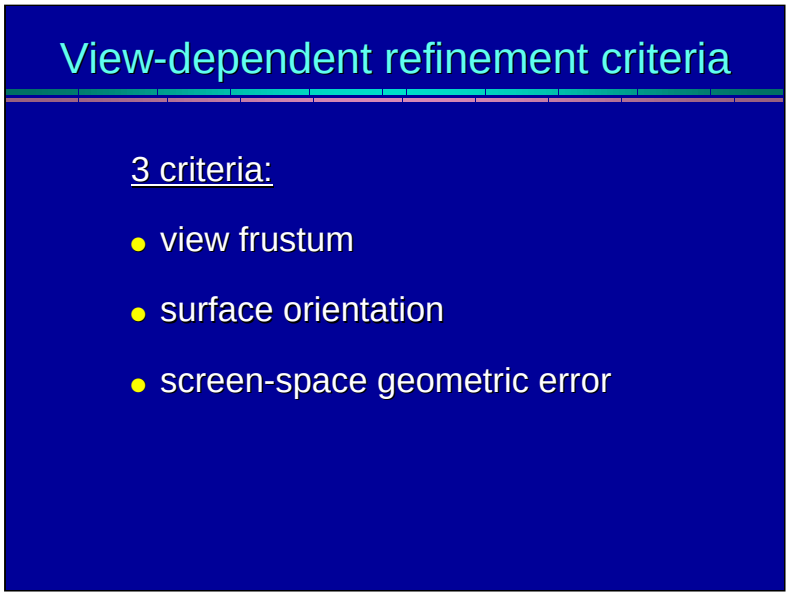
If we find that a vertex should be refined but that the vertex split is not legal, other vertex splits are performed (using a recursive procedure) in order to make the first vertex split legal.

A vertex is collapsed only if it is legal, in order to conservatively satisfy the refinement criteria.

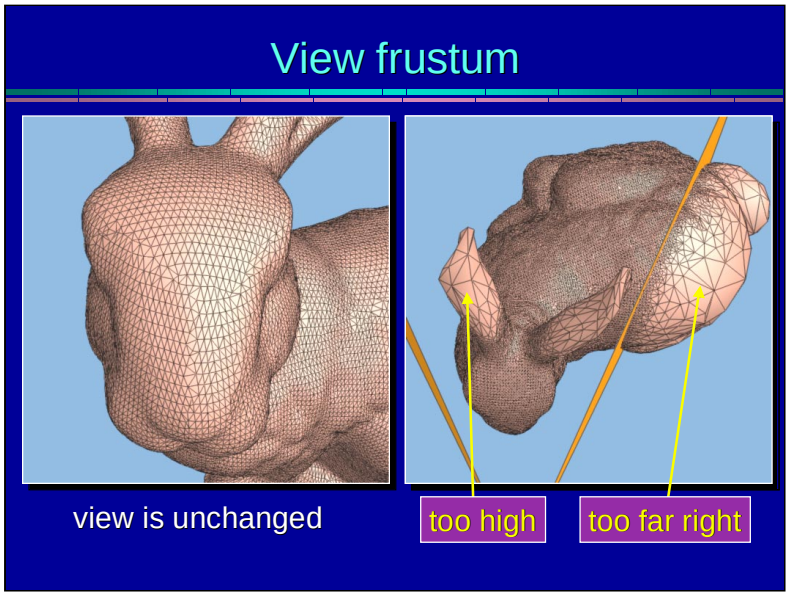
The algorithm exploits frame coherence since only incremental work is performed to adapt the mesh used in the previous frame to the new view parameters.

We find that this adaptive refinement algorithm requires only a small fraction of total frame time on a graphics workstation, because rendering has the same time complexity (linear on the size of the active mesh) and has a larger time constant.

Finally, the algorithm can be amortized over consecutive frames by considering only a fraction of the active vertex list each frame.



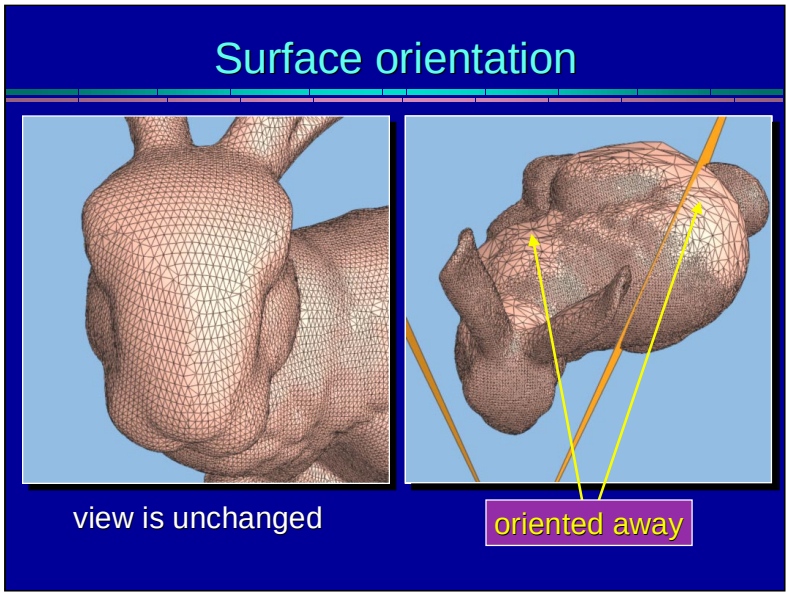
We use three criteria to adapt the refinement of the mesh based on the view parameters.



We demonstrate the 3 criteria using the view shown on the left. In order to see the global effect of selective refinement, the right window shows a top view with the real view frustum highlighted in orange.

The view frustum criterion acts to coarsen regions of the mesh outside the view frustum.

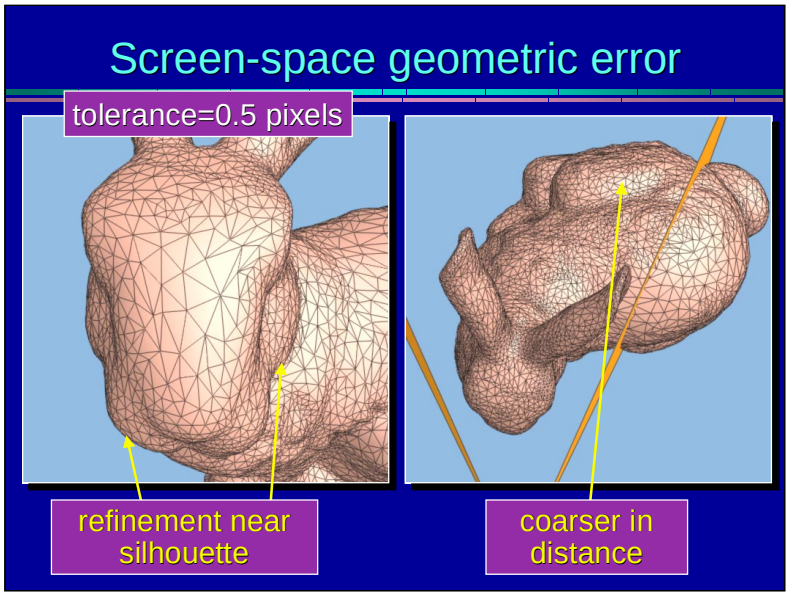
The criterion is conservative, so the view within the frustum is left unchanged.



The surface orientation criterion acts to coarsen regions of the mesh which are oriented away from the viewer.

For closed surfaces, it is unnecessary to render backfacing regions since they are obscured by nearer front-facing regions of the mesh.

Again, this criterion is conservative so that the visible portion of the model is unchanged.

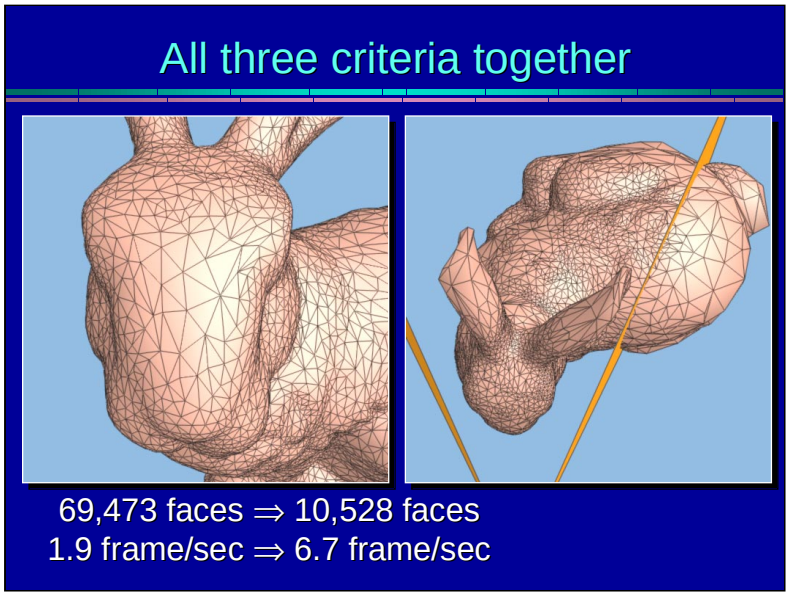


The third refinement criterion adapts the mesh refinement so that the deviation of the approximating mesh (from the fully refined mesh), when projected on the screen, is everywhere less than a screen-space error tolerance.

In this example, the tolerance was set to 0.5 pixels (the images are 500x500 pixels).

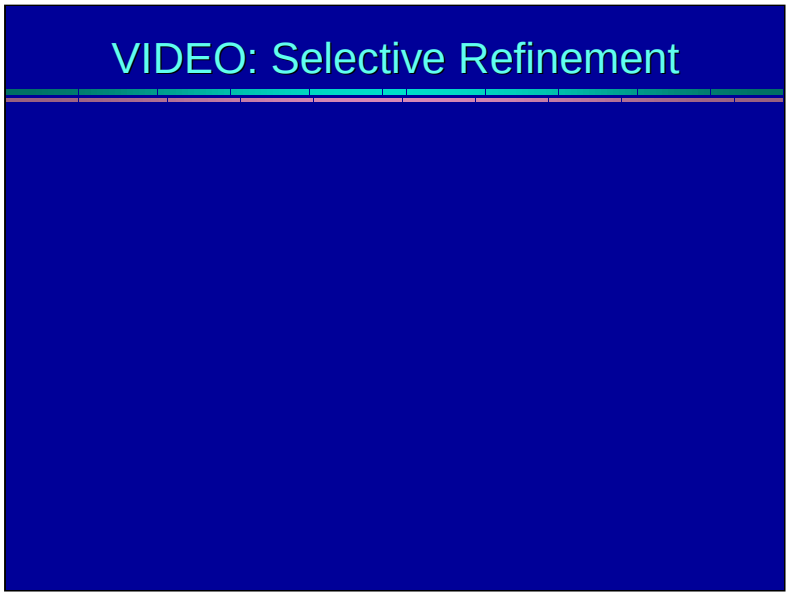
One natural byproduct of this criterion is that the refinement is denser near the object's silhouettes where the deviation of the surface is perpendicular to the viewing direction.

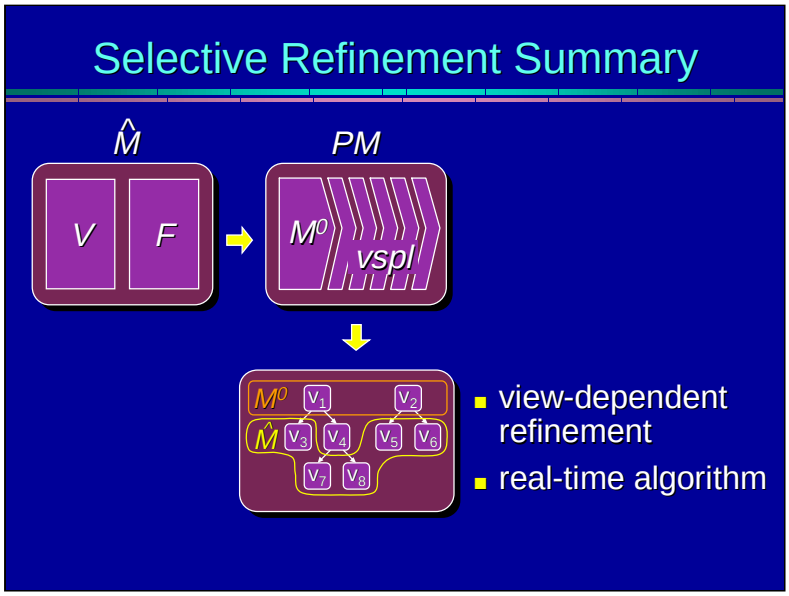
Also, the mesh is made coarser the farther it is from the viewpoint.



These images show the effect of using all 3 criteria together.

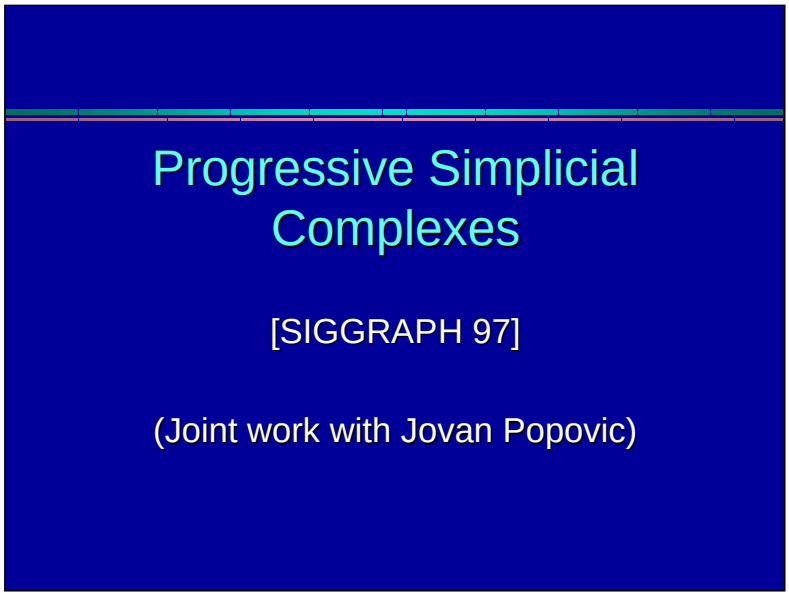
The fully refined mesh of 70,000 faces is simplified to about 10,000 faces. Interactive frame rate is thereby increased from about 2 to 7 frames/sec (including the time taken for adaptive refinement, which is only a small fraction of rendering time).


VIDEO: Selective Refinement

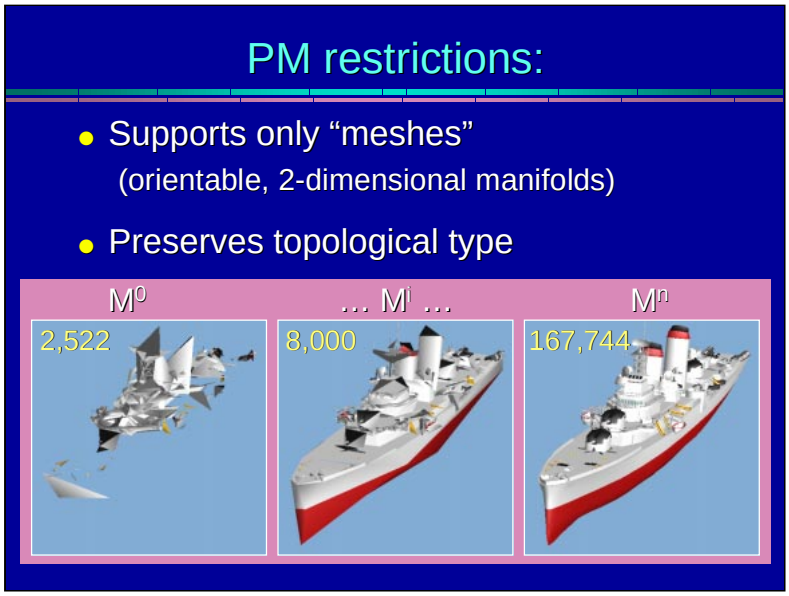


To summarize, the progressive mesh representation also allows real-time, view-dependent refinement of arbitrary triangle meshes.

Surprisingly, the incremental work of updating the mesh from frame to frame requires only a small fraction of total frame time.



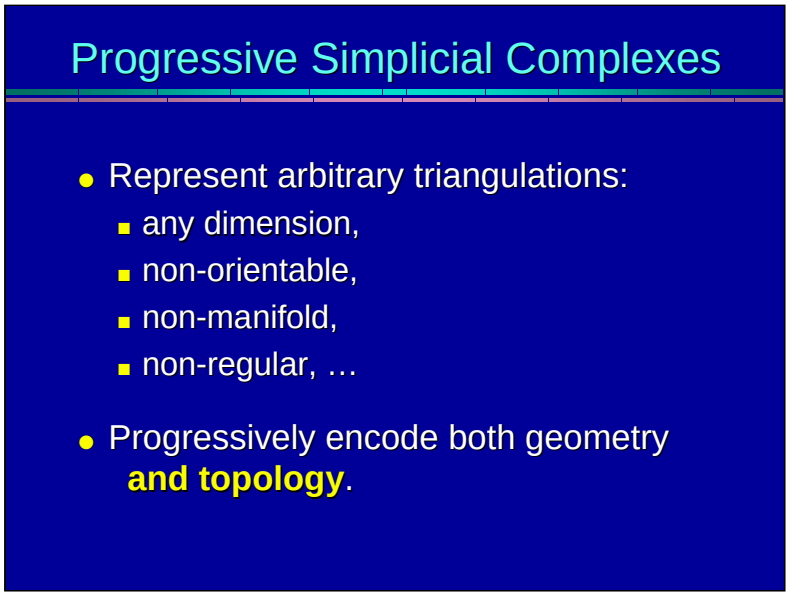
The final part of my talk describes another extension to the progressive mesh work: the progressive simplicial complex representation. It is a generalization that allows the representation of a larger class of models, and allows the progressive representation of both geometry and topology.



The progressive mesh representation has two restrictions.

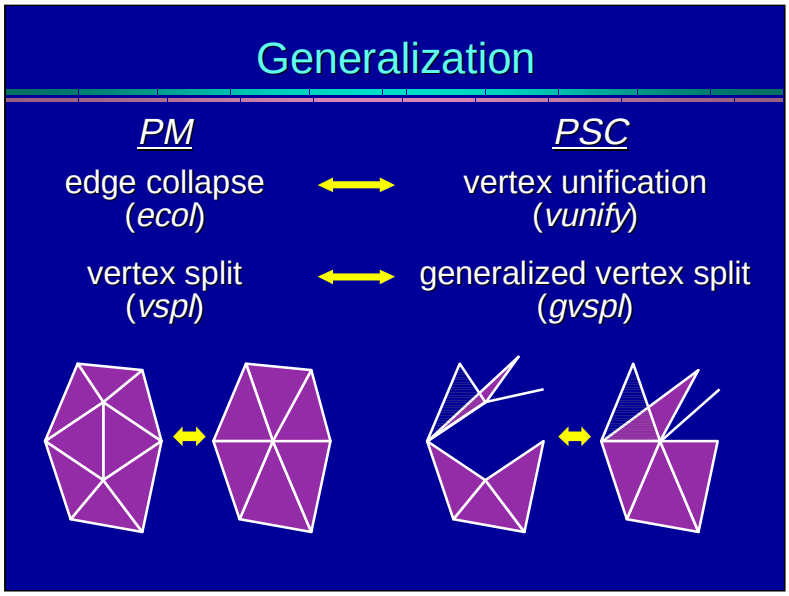
First, it supports only meshes. That is, 2-d triangulations that form orientable manifolds.

Second, and more importantly, all meshes in the progressive mesh sequence are required to have the same topological type. As a consequence, if the original mesh has a number of connected components, each component is required to appear (in simplified form) in the base mesh M^0. This introduces a lower bound on the complexity of M^0 as shown on the left.



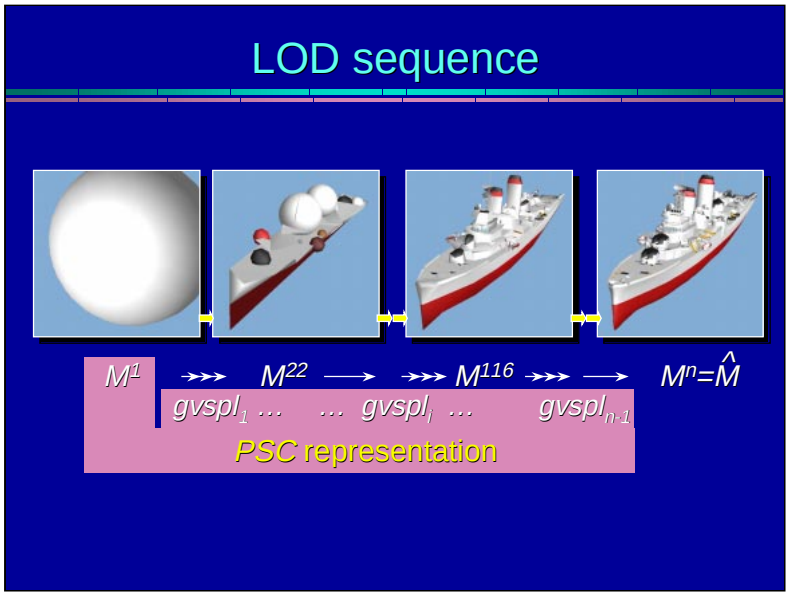
The progressive simplicial complex representation is able to represent arbitrary triangulations. These may have any dimension, may be non-orientable, non-manifold, and non-regular.

Second, it is able to capture such triangulations by progressively encoding both their geometry and topology.



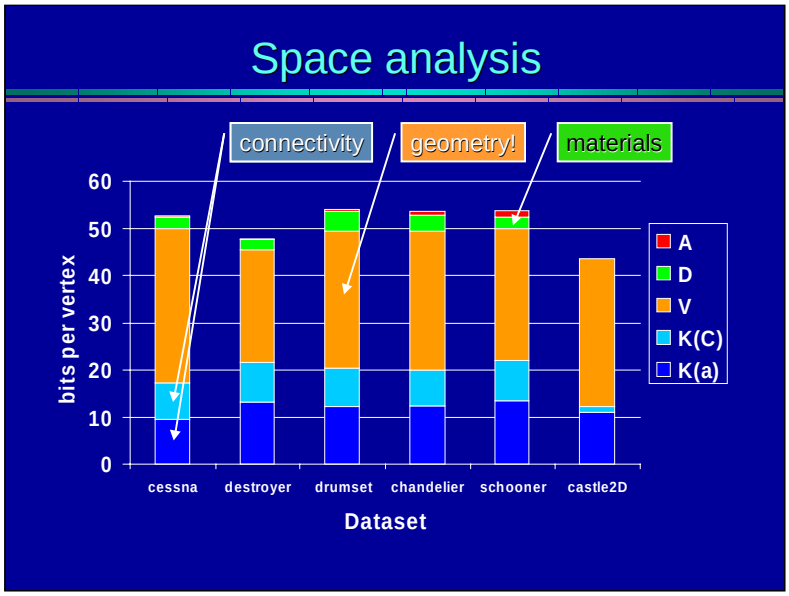
The key is to replace the **edge collapse** / **vertex split** transformations by the more general **vertex unification** / **generalized vertex split** transformations.

The vertex unification transformation takes an arbitrary pair of vertices (not necessarily connected by an edge) and unifies them into one vertex. The generalized vertex split is its inverse.



Using these two transformations, we again form a sequence of models by simplifying a given model, but now the simplest model always consists of a single vertex.

Solitary vertices are drawn as either spheres or circles in order to convey the overall shape of the model at coarse levels. Similarly, solitary edges are drawn as either cylinders or lines.

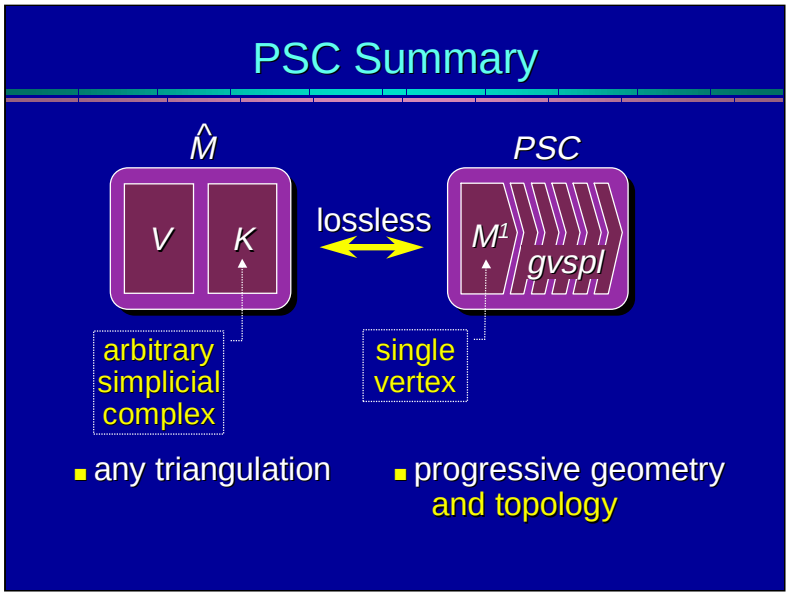


This diagram shows the average number of bits required to store each field of a generalized vertex split record in a PSC representation.

The two blue fields encode the connectivity of the model; these require about (7+log_2 n) bits per vertex. That is about 3 bits more per vertex than the PM representation. However, the PSC representation is able to capture topological changes.

Vertex coordinates (encoding the geometry) are quantized to 16 bits and delta-encoded as in the PM representation. As seen in the diagram, geometry still forms the bulk of the space required to store the model.

The green field shows the encoding of material attributes of the faces. These material attributes also include smoothing groups used to infer surface normals (i.e. surface creases).



Here is a summary of the progressive simplicial complex representation.

It is able to encode an arbitrary simplicial complex as a single vertex together with a sequence of refinement transformations that progressively encode both geometry and topology.

The PSC representations retains the advantages of progressive meshes. It defines a continuous sequence of approximating models for runtime level-of-detail control, allows smooth transitions between any pair of models in the sequence, supports progressive transmission , and still offers a space-efficient representation.

Moreover, by allowing changes to topology, the PSC sequence of approximations achieves better fidelity than the corresponding PM sequence.

Decimation of Triangle Meshes

William J. Schroeder
Jonathan A. Zarge
William E. Lorensen

General Electric Company
Schenectady, NY

1.0 INTRODUCTION

The polygon remains a popular graphics primitive for computer graphics application. Besides having a simple representation, computer rendering of polygons is widely supported by commercial graphics hardware and software. However, because the polygon is linear, often thousands or millions of primitives are required to capture the details of complex geometry. Models of this size are generally not practical since rendering speeds and memory requirements are proportional to the number of polygons. Consequently applications that generate large polygonal meshes often use domain-specific knowledge to reduce model size. There remain algorithms, however, where domain-specific reduction techniques are not generally available or appropriate.

One algorithm that generates many polygons is *marching cubes*. *Marching cubes* is a brute force surface construction algorithm that extracts isodensity surfaces from volume data, producing from one to five triangles within voxels that contain the surface. Although originally developed for medical applications, *marching cubes* has found more frequent use in scientific visualization where the size of the volume data sets are much smaller than those found in medical applications. A large computational fluid dynamics volume could have a finite difference grid size of order 100 by 100 by 100, while a typical medical computed tomography or magnetic resonance scanner produces over 100 slices at a resolution of 256 by 256 or 512 by 512 pixels each. Industrial computed tomography, used for inspection and analysis, has even greater resolution, varying from 512 by 512 to 1024 by 1024 pixels. For these sampled data sets, isosurface extraction using *marching cubes* can produce from 500k to 2,000k triangles. Even today's graphics workstations have trouble storing and rendering models of this size.

Other sampling devices can produce large polygonal models: range cameras, digital elevation data, and satellite data. The sampling resolution of these devices is also improving, resulting in model sizes that rival those obtained from medical scanners.

This paper describes an application independent algorithm that uses local operations on geometry and topology to reduce the number of triangles in a triangle mesh. Although our implementation is for the triangle mesh, it can be directly applied to the more general polygon mesh. After describing other work related to model creation from sampled data, we describe the triangle decimation process and its implementation. Results from two different geometric modeling applications illustrate the strengths of the algorithm.

2.0 THE DECIMATION ALGORITHM

The goal of the decimation algorithm is to reduce the total number of triangles in a triangle mesh, preserving the original topology and a good approximation to the original geometry.

2.1 OVERVIEW

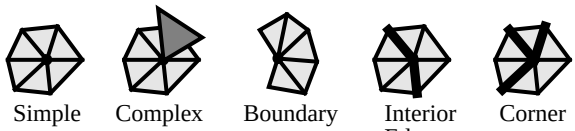
The decimation algorithm is simple. Multiple passes are made over all vertices in the mesh. During a pass, each vertex is a candidate for removal and, if it meets the specified decimation criteria, the vertex and all triangles that use the vertex are deleted. The resulting hole in the mesh is patched by forming a local triangulation. The vertex removal process repeats, with possible adjustment of the decimation criteria, until some termination condition is met. Usually the termination criterion is specified as a percent reduction of the original mesh (or equivalent), or as some maximum decimation value. The three steps of the algorithm are:

1. characterize the local vertex geometry and topology,
2. evaluate the decimation criteria, and
3. triangulate the resulting hole.

2.2 CHARACTERIZING LOCAL GEOMETRY / TOPOLOGY

The first step of the decimation algorithm characterizes the local geometry and topology for a given vertex. The outcome of this process determines whether the vertex is a potential candidate for deletion, and if it is, which criteria to use.

Each vertex may be assigned one of five possible classifications: simple, complex, boundary, interior edge, or corner vertex. Examples of each type are shown in the figure below.



A simple vertex is surrounded by a complete cycle of

triangles, and each edge that uses the vertex is used by exactly two triangles. If the edge is not used by two triangles, or if the vertex is used by a triangle not in the cycle of triangles, then the vertex is complex. These are non-manifold cases.

A vertex that is on the boundary of a mesh, i.e., within a semi-cycle of triangles, is a boundary vertex.

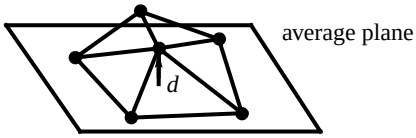
A simple vertex can be further classified as an interior edge or corner vertex. These classifications are based on the local mesh geometry. If the dihedral angle between two adjacent triangles is greater than a specified *feature angle*, then a *feature edge* exists. When a vertex is used by two feature edges, the vertex is an interior edge vertex. If one or three or more feature edges use the vertex, the vertex is classified a corner vertex.

Complex vertices are not deleted from the mesh. All other vertices become candidates for deletion.

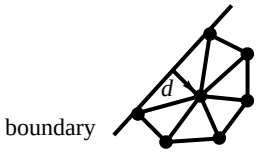
2.3 EVALUATING THE DECIMATION CRITERIA

The characterization step produces an ordered loop of vertices and triangles that use the candidate vertex. The evaluation step determines whether the triangles forming the loop can be deleted and replaced by another triangulation exclusive of the original vertex. Although the fundamental decimation criterion we use is based on vertex distance to plane or vertex distance to edge, others can be applied.

Simple vertices use the distance to plane criterion (see figure below). If the vertex is within the specified distance to the average plane it may be deleted. Otherwise it is retained.



Boundary and interior edge vertices use the distance to edge criterion (figure below). In this case, the algorithm determines the distance to the line defined by the two vertices creating the boundary or feature edge. If the distance to the line is less than *d*, the vertex can be deleted.



It is not always desirable to retain feature edges. For example, meshes may contain areas of relatively small triangles with large feature angles, contributing relatively little to the geometric approximation. Or, the small triangles may be the result of "noise" in the original mesh. In these situations, corner vertices, which are usually not deleted, and interior edge vertices, which are evaluated using the distance to edge criterion, may be evaluated using the distance to plane criterion. We call this edge preservation, a user specifiable parameter.

If a vertex can be eliminated, the loop created by removing the triangles using the vertex must be triangulated. For interior edge vertices, the original loop must be split into two halves, with the split line connecting the vertices forming the feature edge. If the loop can be split in this way, i.e., so that resulting two loops do not overlap, then the loop is split and each piece is triangulated separately.

2.4 TRIANGULATION

Deleting a vertex and its associated triangles creates one (simple or boundary vertex) or two loops (interior edge vertex). Within each loop a triangulation must be created whose triangles are non-intersecting and non-degenerate. In addition, it is desirable to create triangles with good aspect ratio and that approximate the original loop as closely as possible.

In general it is not possible to use a two-dimensional algorithm to construct the triangulation, since the loop is usually non-planar. In addition, there are two important characteristics of the loop that can be used to advantage. First, if a loop cannot be triangulated, the vertex generating the loop need not be removed. Second, since every loop is star-shaped, triangulation schemes based on recursive loop splitting are effective. The next section describes one such scheme.

Once the triangulation is complete, the original vertex and its cycle of triangles are deleted. From the Euler relation it follows that removal of a simple, corner, or interior edge vertex reduces the mesh by precisely two triangles. If a boundary vertex is deleted then the mesh is reduced by precisely one triangle.

3.0 IMPLEMENTATION

3.1 DATA STRUCTURES

The data structure must contain at least two pieces of information: the geometry, or coordinates, of each vertex, and the definition of each triangle in terms of its three vertices. In addition, because ordered lists of triangles surrounding a vertex are frequently required, it is desirable to maintain a list of the triangles that use each vertex.

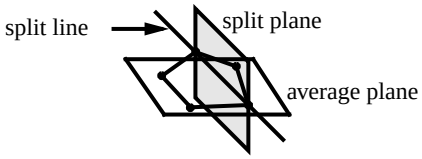
Although data structures such as Weiler's radial edge or Baumgart's winged-edge data structure can represent this information, our implementation uses a space-efficient vertex-triangle hierarchical ring structure. This data structure contains hierarchical pointers from the triangles down to the vertices, and pointers from the vertices back up to the triangles using the vertex. Taken together these pointers form a ring relationship. Our implementation uses three lists: a list of vertex coordinates, a list of triangle definitions, and another list of lists of triangles using each vertex. Edge definitions are not explicit, instead edges are implicitly defined as ordered vertex pairs in the triangle definition.

3.2 TRIANGULATION

Although other triangulation schemes can be used, we chose a recursive loop splitting procedure. Each loop to be triangulated is divided into two halves. The division is along a line (i.e., the split line) defined from two non-neighboring vertices in the loop. Each new loop is divided again, until only three vertices remain in each loop. A loop of three ver-

tices forms a triangle, that may be added to the mesh, and terminates the recursion process.

Because the loop is non-planar and star-shaped, the loop split is evaluated using a split plane. The split plane, as shown in the figure below, is the plane orthogonal to the average plane that contains the split line. In order to determine whether the split forms two non-overlapping loops, the split plane is used for a half-space comparison. That is, if every point in a candidate loop is on one side of the split plane, then the two loops do not overlap and the split plane is acceptable. Of course, it is easy to create examples where this algorithm will fail to produce a successful split. In such cases we simply indicate a failure of the triangulation process, and do not remove the vertex or surrounding triangle from the mesh.



Typically, however, each loop may be split in more than one way. In this case, the best splitting plane must be selected. Although many possible measures are available, we have been successful using a criterion based on aspect ratio. The aspect ratio is defined as the minimum distance of the loop vertices to the split plane, divided by the length of the split line. The best splitting plane is the one that yields the maximum aspect ratio. Constraining this ratio to be greater than a specified value,.e.g., 0.1, produces acceptable meshes.

Certain special cases may occur during the triangulation process. Repeated decimation may produce a simple closed surface such as a tetrahedron. Eliminating a vertex in this case would modify the topology of the mesh. Another special case occurs when “tunnels” or topological holes are present in the mesh. The tunnel may eventually be reduced to a triangle in cross section. Eliminating a vertex from the tunnel boundary then eliminates the tunnel and creates a non-manifold situation.

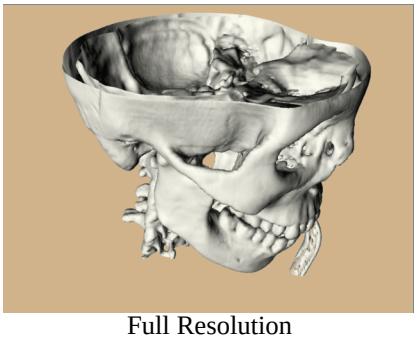
These cases are treated during the triangulation process. As new triangles are created, checks are made to insure that duplicate triangles and triangle edges are not created. This preserves the topology of the original mesh, since new connections to other parts of the mesh cannot occur.

4.0 RESULTS

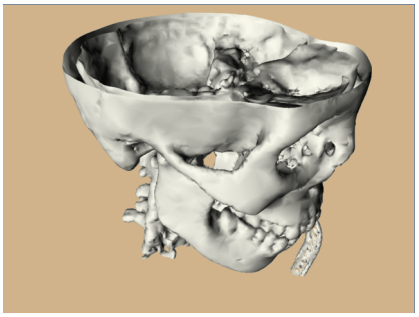
Two different applications illustrate the triangle decimation algorithm. Although each application uses a different scheme to create an initial mesh, all results were produced with the same decimation algorithm.

4.1 VOLUME MODELING

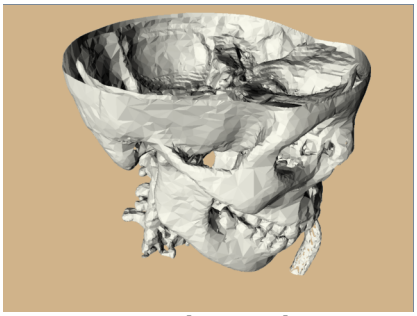
The first application applies the decimation algorithm to isosurfaces created from medical and industrial computed tomography scanners. *Marching cubes* was run on a 256 by 256 pixel by 93 slice study. Over 560,000 triangles were required to model the bone surface. Earlier work reported a triangle reduction strategy that used averaging to reduce the number of triangles on this same data set. Unfortunately, averaging applies uniformly to the entire data set, blurring



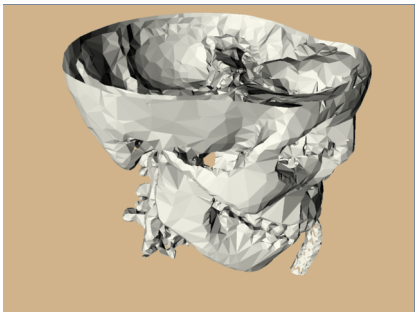
Full Resolution
(569K Gouraud shaded triangles)



75% decimated
(142K Gouraud shaded triangles)



75% decimated
(142K flat shaded triangles)



90% decimated
(57K flat shaded triangles)

high frequency features. The first set of figures shows the resulting bone isosurfaces for 0%, 75%, and 90% decimation, using a decimation threshold of 1/5 the voxel dimension. The next pair of figures shows decimation results for an industrial CT data set comprising 300 slices, 512 by 512, the largest we have processed to date. The isosurface created from the original blade data contains 1.7 million triangles. In fact, we could not render the original model because we exceeded the swap space on our graphics hardware. Even after decimating 90% of the triangles, the serial number on the blade dovetail is still evident.

4.2 TERRAIN MODELING

We applied the decimation algorithm to two digital elevation data sets: Honolulu, Hawaii and the Mariner Valley on Mars. In both examples we generated an initial mesh by creating two triangles for each uniform quadrilateral element in the sampled data. The Honolulu example illustrates the polygon savings for models that have large flat areas. First we applied a decimation threshold of zero, eliminating over 30% of the co-planar triangles. Increasing the threshold removed 90% of the triangles. The next set of four figures shows the resulting 30% and 90% triangulations. Notice the transitions from large flat areas to fine detail around the shore line.

The Mars example is an appropriate test because we had access to sub-sampled resolution data that could be compared with the decimated models. The data represents the western end of the Mariner Valley and is about 1000km by 500km on a side. The last set of figures compares the shaded and wireframe models obtained via sub-sampling and decimation. The original model was 480 by 288 samples. The sub-sampled data was 240 by 144. After a 77% reduction, the decimated model contains fewer triangles, yet shows more fine detail around the ridges.

5.0 REFERENCES

[1] Baumgart, B. G., "Geometric Modeling for Computer Vision," Ph.D. Dissertation, Stanford University, August 1974.

[2] Bloomenthal, J., "Polygonalization of Implicit Surfaces," *Computer Aided Geometric Design*, Vol. 5, pp. 341-355, 1988.

[3] Cline, H. E., Lorensen, W. E., Ludke, S., Crawford, C. R., and Teeter, B. C., "Two Algorithms for the Three Dimensional Construction of Tomograms," *Medical Physics*, Vol. 15, No. 3, pp. 320-327, June 1988.

[4] DeHaemer, M. J., Jr. and Zyda, M. J., "Simplification of Objects Rendered by Polygonal Approximations," *Computers & Graphics*, Vol. 15, No. 2, pp 175-184, 1992.

[5] Dunham, J. G., "Optimum Uniform Piecewise Linear Approximation of Planar Curves," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, No. 1, pp. 67-75, January 1986.

[6] Finnigan, P., Hathaway, A., and Lorensen, W., "Merging CAT and FEM," *Mechanical Engineering*, Vol. 112, No. 7, pp. 32-38, July 1990.

[7] Fowler, R. J. and Little, J. J., "Automatic Extraction of Irregular Network Digital Terrain Models," *Computer Graphics*, Vol. 13, No. 2, pp. 199-207, August 1979.

[8] Ihm, I. and Naylor, B., "Piecewise Linear Approximations of Digitized Space Curves with Applications," in *Scientific Visualization of Physical Phenomena*, pp. 545-569, Springer-Verlag, June 1991.

[9] Kalvin, A. D., Cutting, C. B., Haddad, B., and Noz, M. E., "Constructing Topologically Connected Surfaces for the Comprehensive Analysis of 3D Medical Structures," *SPIE Image Processing*, Vol. 1445, pp. 247-258, 1991.

[10] Lorensen, W. E. and Cline, H. E., "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, Vol. 21, No. 3, pp. 163-169, July 1987.

[11] Miller, J. V., Breen, D. E., Lorensen, W. E., O'Bara, R. M., and Wozny, M. J., "Geometrically Deformed Models: A Method for Extracting Closed Geometric Models from Volume Data," *Computer Graphics*, Vol. 25, No. 3, July 1991.

[12] Preparata, F. P. and Shamos, M. I., *Computational Geometry*, Springer-Verlag, 1985.

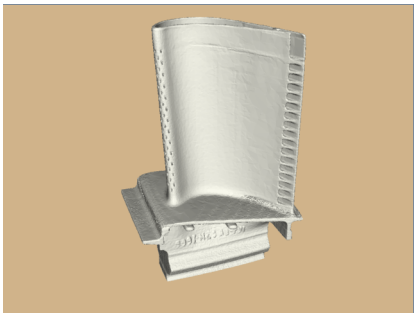
[13] Schmitt, F. J., Barsky, B. A., and Du, W., "An Adaptive Subdivision Method for Surface-Fitting from Sampled Data," *Computer Graphics*, Vol. 20, No. 4, pp. 179-188, August 1986.

[14] Schroeder, W. J., "Geometric Triangulations: With Application to Fully Automatic 3D Mesh Generation," PhD Dissertation, Rensselaer Polytechnic Institute, May 1991.

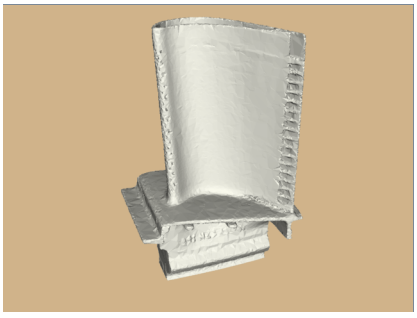
[15] Terzopoulos, D. and Fleischer, K., "Deformable Models," *The Visual Computer*, Vol. 4, pp. 306-311, 1988.

[16] Turk, G., "Re-Tiling of Polygonal Surfaces," *Computer Graphics*, Vol. 26, No. 3, July 1992.

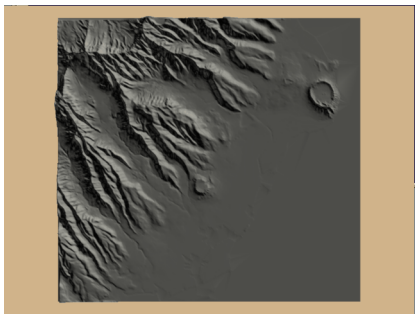
[17] Weiler, K., "Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments," *IEEE Computer Graphics and Applications*, Vol. 5, No. 1, pp. 21-40, January 1985.



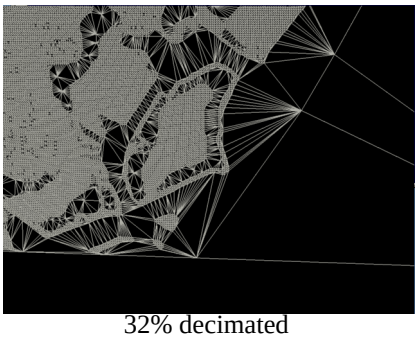
75% decimated
(425K flat shaded triangles)



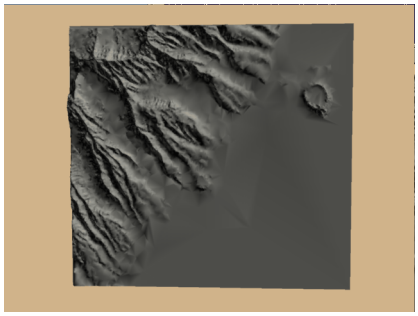
90% decimated
(170K flat shaded triangles)



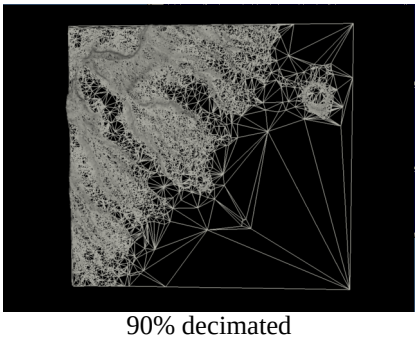
32% decimated
(276K flat shaded triangles)



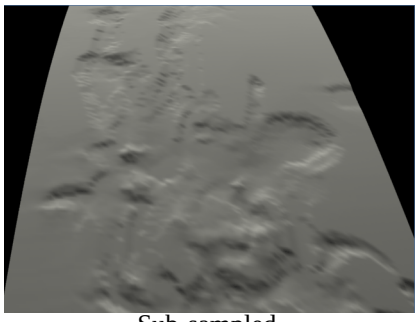
32% decimated
(shore line detail, wireframe)



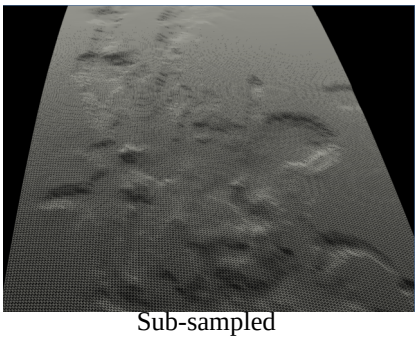
90% decimated
(40K Gouraud shaded triangles)



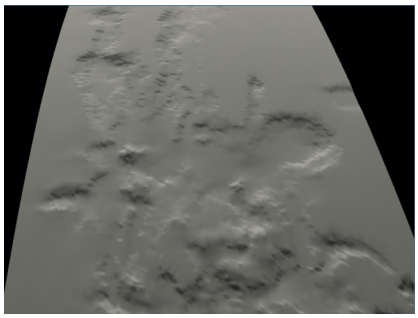
90% decimated
(40K wireframe)



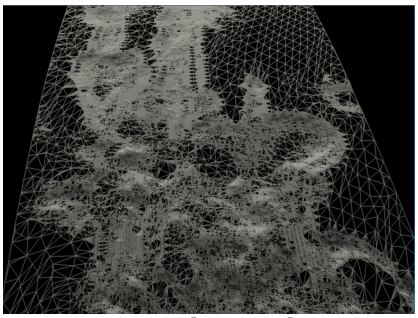
Sub-sampled
(68K Gouraud shaded triangles)



Sub-sampled
(68K wireframe)



77% decimated
(62K Gouraud shaded triangles)



77% decimated
(62K wireframe)

A Compact Cell Structure for Scientific Visualization

W.J. Schroeder
Boris Yamrom

GE Corporate Research & Development
Schenectady, NY 12301

Abstract

Well designed data structures and access methods are vital to developing efficient visualization algorithms. The cell structure is a compact, general data structure for representing n-dimensional topological constructs such as unstructured grids, polygonal, or triangle strip representations. The cell structure also provides constant time access methods for a wide variety of visualization algorithms. This paper describes the representation, access methods, and implementation of the cell structure. Sample algorithms such as decimation, triangle strip generation, and streamline propagation are used to illustrate its application.

1.0 Introduction

The bulk of the visualization literature is oriented towards algorithms and graphical representational schemes[1],[2],[3]. Data structures, if described at all, are often presented superficially. However it is only with well designed combinations of both algorithm and data structure that useful visualization techniques can be created[4].

Visualization data tends to have some particular characteristics: the data size is large and the data type is varied. Large data are simply the result of the basic goal of visualization - to transform large data into more comprehensible forms. The data is varied because visualization techniques are general. An iso-surface generation algorithm[5],[5] is just as useful applied to medical data as it is to financial visualization. Hence visual data structures must be both compact (i.e., small memory requirement) and general (i.e., represent a wide variety of data).

The cell structure is a compact and general data structure for representing cell topology. Cell topology consists of points plus and a particular ordering of points (i.e., a cell). One or more cells may share a given point as well as other topological features such as edges and faces. The most important feature of the cell structure is that it represents adjacency, or topological neighborhood information, with minimal memory requirement. The cell structure has also been designed with access methods that support a wide variety of visualization algorithms.

A number of similar data structures have been previously developed[7],[8]. The simplest structures are variations of display lists: lists of points, and polygon/element/cell connectivity. While these data structures are compact, performing operations requiring adjacency information results in algorithms of $O(n^2)$ time complexity since searching is required. A variation of this data structure uses a supplemental list to represent adjacency information[9],[10]. For each cell of particular type and dimension (e.g., hexahedron with six faces), a list of neighbors is maintained (e.g., the six face neighbors of the hexahedron). This structure is particularly useful when the cell type and topology is the same for all cells. However, when mixed cell type and topology is required, the structure becomes unwieldy. Also, in order to build this structure, an $O(n^2)$ search is initially required. More elaborate data structures include the hierarchical winged edge[11] and radial edge[12] structures. Hierarchical structures explicitly represent topology in terms of a hierarchy of increasing topological dimension: vertices, edges, faces, regions. Although extremely powerful constructs, fully elaborated hierarchical structures require large amounts of memory than the simpler ones just described.

The cell structure is a variation of the display list structure with additional hierarchical information. It can simultaneously represent cells of mixed topology, and provides constant time access to adjacency information. Moreover, the cell structure is more compact than hierarchical structures since its hierarchical information is implicitly represented.

2.0 Cell Structure

In this section the mathematical basis, representation, access methods, and implementation of the cell structure is described.

2.1 Mathematical Basis

The cell structure is based on the topological construct called a cell, Ci. A cell is an ordered sequence of

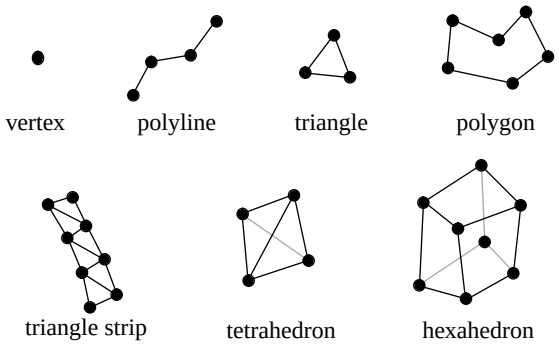


Figure 1. Some example cell types.

points $Ci = \{p_1, p_2, \ldots, p_n\}$ with $p_i \in P$ where P is a set of *n*-dimensional points. The particular meaning of the sequence of points, or cell topology, is determined by the *type* of cell. The number of points *n* defining the cell is the *size* of the cell.

Examples of cells (Figure 1) include points (0D topology), lines (1D topology), polygons and triangle strips (2D topology), and unstructured grid elements such as tetrahedron, hexahedron, pyramids, and triangular prisms (3D topology). Higher dimension cells are also possible, such as n-dimensional simplices.

A key concept of the cell structure is the "use" of a point by a cell. A cell C_i "uses" a point p_i when $p_i \in C_i$. Hence the "use set" $U(p_i)$ is the collection of all cells using p_i:

$$U(p_i) = \{C_i : p_i \in C_i\}$$

2.2 Representation

The cell structure expresses the relationship between points *P*, cells *C*, and use sets *U*. The representation of this information may take a variety of forms, but the preference is for a data structure that is simple, compact, and can be accessed in constant time. It is also desirable that the structure can be directly retrieved from or written to storage. That is, the use of pointers, or memory locations, is not directly evident in the data structure.

The implementation of the cell structure consists of four *dynamic arrays* (Figure 2).A dynamic array is simply an array (e.g., a contiguous, addressable memory space) that grows dynamically to accomodate new data. The elements of the array are addressed with a unique, non-negative integer id.

The first array represents a list of points *x-y-z* coordinates whose id is the array access id. The second array represents the cells: the size, type, and a list of point ids that define the cell. In the third dynamic array the use arrays for each point are expressed. Each use array consists of the number of cells using a particular point. That is, at use array position *i* the cells that use point *i* are listed. The final dynamic array provides storage and consists of id types in no particular order. This array is not required, but it provides a contiguous pool of memory from which the cell arrays and use arrays construct their

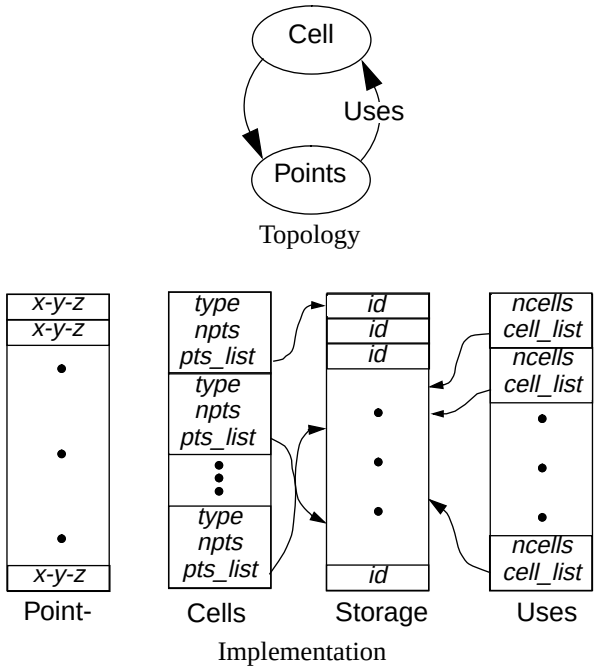


Figure 2. Cell representation.

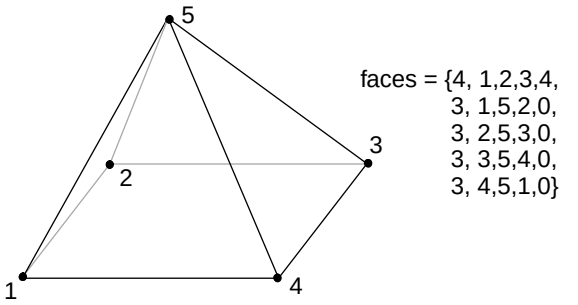


Figure 3. Accessing implicit topology

lists. Using the storage array reduces memory fragmentation greatly, and reduces the number of system calls to acquire memory.

A key feature of the cell structure is that it implicitly represents intermediate topology between the cell (*n*-dimensional) and the defining points (0-dimensional). As a result, supplemental information is required to address the $(n-2)$ layers of intermediate cell topology. If a polygon is represented as an ordered sequence of points, then pairs of adjacent points represent polygon edges. A more complex topology is a pyramid. The list shown in Figure 3 allows direct access to the five faces of the pyramid. The first number represents the number of points defining the face, followed by a list of ordered indices into the cells point list (1-offset). This implicit representation of topology is the reason why the cell structure can compactly represent adjacency information.

Other information can represent the relationship between the cell geometry and topology. A typical example is when traversing cells, such as streamline tracking.

Often the traversal is computed using parametric coordinates, and when a boundary is encountered (parametric coordinate value ± 1), it is necessary to obtain the corresponding topology (e.g., face or edge).

Another important capability of the cell structure is the ability to simultaneously represent cells of different type. For example, Figure 5 shows an unstructured grid consisting of hexadra, tetraheda, pyramids, triangular prisms (3D topology), polygons (2D topology), and lines (1D topology).

2.3 Access Methods

There are three categories of methods for manipulating the cell structure: *primitive methods, topological methods,* and *adjacency methods.* A description of these methods follow.

2.3.1 Primitive Methods

Primitive methods are used for creating, destroying, modifying, and traversing the cell structure. Sample methods include:

Cells = initialize ()
Create an empty cell structure and return a pointer *Cells* to the structure. Optional arguments for specifying initial storage size are possible.

create_point (Cells, pt_id, x)
Given a point id and x-y-z coordinate, create a point in the *Cells* structure.

create_cell (Cells, cell_id, type, npts, pts)
Given a cell id, a cell type, the number of points, and the point ids defining the cell, create a cell in the *Cells* structure.

build_uses ()
Using the current cells and points, build the use lists for the *Cells* structure.

npts = get_number_points (Cells)
Return the number of points in the structure.

ncells = get_numer_cells (Cells)
Return the number of cells in the structure.

destroy (Cells)
Release the *Cells* structure back to system memory.

Building the cell structure consists of creating points and cells, followed by the building the use lists. This process is of linear time complexity. For each cell in the structure, a traversal of the cell's point list is made. Then for each point in the cell's point list, the corresponding use list is updated. Once all cells are visited the data structure is complete. In some applications the *build_uses()* method need not be executed. Then the structure is basically a display list.

2.3.2 Topological Methods

Topological methods provide access to the topology of the cell. Some of these methods are as follows.

get_cell_pts (Cells, cell_id, pts, npts)
Return a list of point ids that define the given cell.

get_pt_cells (Cells, pt_id, cells, ncells)
Return a list of cell ids that use the specified point.

These operators return information directly from the cell structure. It is also possible to return other topological information using supplemental information associated with the type of the topology. For example, if the cell type is a pyramid, then accessing the faces of the pyramid can be implemented using the table of Figure 3 and the operator:

get_cell_face_pts (Cells, cell_id, face_id, npts, pts)
Return a list of point ids that define the specified face of the given cell.

2.3.3 Adjacency Methods

Adjacency methods are used to obtain information about the neighbors of a cell. A neighbor of a particular cell $\bar{C}$ is simply a cell that shares one or more points in common with $\bar{C}$. Examples of these methods follow.

get_cell_pt_nbr (Cells, cell_id, pt_id, nbrs, nnbrs)
Given a point and cell, return a list of neighboring cells that use the point.

get_cell_feature_nbr (Cells,cell_id,pts,npts,nbr,nnbrs)
Given a list of points from a cell, return a list of neighboring cells that each use all the specified points.

The *get_cell_feature_nbr()* method is useful for extracting adjacency information across topological features. For example, if the feature is specified by two points defining an edge, this method returns edge neighbors. Or, if the points define a face, this method returns a face neighbor.

The adjacency operators are simple set operations. For a particular cell $\bar{C}$ and point list $\bar{P} = (\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_n)$ with $\bar{P} \subset P$, where $\bar{P}$ typically corrsponds to a topological feature of the cell, the result of the *get_cell_feature_nbrs()* method is the adjacency set $A(\bar{C}, \bar{P})$. The adjacency set is simply the intersection of the use sets for each point, excluding the cell $\bar{C}$

$$A(\bar{C}, \bar{P}) = \left(\bigcap_{i=1}^{n} U(\bar{p}_i) \right) - \bar{C}$$

The adjacency set implicitly represents a variety of useful information. In a manifold object represented by a polyhedra, for example, each polygon must have exactly one edge neighbor for each of its edges. Edges that have no neighbors are boundary edges; edges that have more than one edge neighbor represent non-manifold topology.

Volume data sets that consist of 3D cells (e.g., unstructured grids) are topologically consistent only if for each cell there is exactly one face neighbor for each face. Faces that have no neighbors are on the boundary of the volume. More than one face neighbor implies that the neighbors are self-intersecting.

The construction of an adjacency set is of $O(n)$ time and space complexity provided that a constant bound can be placed on the number of uses of a point. Examining Equation 2 above, if $n \leq \text{MAX\_USE} \ll npts$, and MAX_USE is independent of *npts*, then the time complexity of the set operations are bounded by a fixed constant. It is possible to design pathological cases where a particular point is used by every cell, but in application such situations do not occur. Typically a point is used by 5-6 triangles in a triangle mesh, or eight hexahedra in a hexahedral mesh.

3.0 Algorithms

A variety of algorithms have been implemented using the cell structure. In the following subsections a few are expressed in pseudo-code using the cell access methods described earlier. The pseudo-code examples are simplified to demonstrate the use of the data structure.

3.1 Streamlne Propagation

A common vector field visualization technique is to generate streamlines. Streamlines are the path that a massless particle takes when moving through the vector field. Typical examples include visualizing fluid flow.

The cell structure provides a convenient structure to propagate the streamline through a computational grid (Figure 6). Computational grids are typically composed of many thousands or perhaps millions of cells in which numerical computation is carried out. The streamline traverses many of these cells, and the propagation algorithm requires tracking the streamline from cell to cell.

```
determine initial cell cell, position x, and velocity v;
while ( inside Cells) {
  x_{i+1} = x_i + v_i · Δt ;
  map x_{i+1} to cell coordinates (r,s,t);
  if ( (r,s,t) outside cell ) {
    find cell face f that streamline passed through;
    get_cell_face_nbrs (Cells, cell, f, nbrs, nnbrs);
    if (nnbrs < 1) outside Cells;
    else cell = nbrs[0];
  }
  evaluate velocity v_i in cell at (r,s,t);
  x_i = x_{i+1};
}
```

3.2 Decimation

The goal of the decimation algorithm is to reduce the number of polygons in a polygonal mesh, while maintaining the original topology of the mesh. Schroeder et al[15] have implemented this algorithm using the cell structure and have achieved reductions of greater than 90% on maximum model sizes of 1.7 million triangles. Figure 7 shows a 90% decimated model of a human face.

The algorithm repeatedly visits all non-decimated vertices, gathering the polygons surrounding each vertex into a list. These polygons are than evaluated against a local planarity (or edge) condition. If the condition is satisfied, then the vertex and using polygons are deleted, and the resulting “hole” is triangulated. The process repeats until an appropriate number of vertices are eliminated.

(pseudo-code)

3.3 Feature Normals

Realistic rendering of polygonal representations depends upon using vertex normals to smooth the transition from one polygon to the next. Frequently polygons are generated without normals, and techniques must be used to generate the normals from the polygon connectivity.

One naive approach to normal generation is to compute a vertex normal by averaging the normals of polygons using the vertex. This works well in situations where the dihedral angles between polygons is small, and when the polygons are all ordered consistently. In many cases, e.g., a cube, the angles between polygons are quite large, resulting in images that appear less than realistic.

In the algorithm that follows, normals are generated on a polygonal mesh that may or may not be consistently ordered. In addition, polygons whose dihedral angle is greater than a specified feature angle are separated along their common edges (Figure 8).

```
for each cell in Cells { /*make order consistent*/
  if (not visited cell) order (cell); /*recursively reorder*/
}
for each cell in Cells { /*compute cell normals*/
  get_cell_pts (Cells, cell, pts, npts);
  generate polygon normal;
}
for each point p in Cells { /*split feature edges*/
  if (not visited p) split(p); /*recursively split*/
}
for each point p in Cells{
  get_pt_cells (Cells, p, cells);
  determine average normal based on using cells;
}

order (cell) {/* recursive reorder function */
  mark cell visited;
  get_cell_pts (Cells, cell, pts, npts);
  for each edge (p1,p2) in cell pts {
    get_cell_edge_nbrs (Cells, cell, p1, p2, nbrs, nnbrs);
    if (nnbrs > 0 && nbrs[i] not visited) {
      if (nbrs[i] edge order not (p2,p1) reverse(nbrs[i]);
      order (nbrs[i]); /*recursive call*/
    }}}

split (p) {/* recursive edge splitting function */
  mark p visited;
  get_pt_cells (Cells, pt, cells, ncells);
  for each cell cells[i] {
    get_cell_pts (Cells, cells[i], pts, npts);
```

```
    for each cell edge (p,pts[i])
      get_cell_edge_nbrs (Cells,cells[i],p,pts[i],nbrs, nnbrs);{
      if ( nnbrs > 0 ) && (nnbrs > 1 or
            dihedral angle > feature angle ) {
        create new point pnew;
        replace p (in cells[i]) with pnew;
      } }
    if (not visited pts[i]) split (pts[i]);
  }}
```

3.4 Triangle Strip Generation

Triangle strips are compact representations of adjacent triangles. Most rendering hardware supports triangle strips directly as a high-performance graphics primitive. Unfortunately triangle strips are not typically generated in visualization algorithms. Instead, as this simple algorithm illustrates, triangle strips can be easily generated from triangles (or polygons) using the cell structure.

A typical application of this algorithm is shown in Figure 9. Here the original data of x polygons is stripped to produce y strips. The modest length of the strips is misleading: the result is a six-fold increase in rendering speed.

```
initialize list of triangle strips strips;
for each cell in cell list Cells {
  if cell not visited {
    mark cell visited;
    start new triangle strip strip;
    get_cell_pts (Cells, cell, pts, npts);
    for each cell edge p1,p2 { /* assumed triangles */
      get_cell_edge_nbrs (Cells, cell, p1, p2, nbrs, nnbrs);
      if (nnbrs>0 && (nbr=nbrs[0]) not visited) break;{
    }/* start growing strip */
    while ( nbr != NULL ) {
      add nbr to strip;
      mark nbr visited;
      get_cell_edge_nbrs (Cells, nbr, p1, p2, nbrs, nnbrs);
      if (nnbrs < 1) nbr = NULL else nbr=nbrs[0];
    }
    add strip to strips;
  }
}
```

3.5 Extrusion

A simple geometric construct is an extrusion or sweep. Here it is assumed that the starting point for this operation is a collection of points, lines, and polygons, and that the surface is swept along some path to create a "volume".

The use of the cell structure in this algorithm is to identify boundary edges, i.e., polygon edges that are used by only a single polygon. These edges when swept create the sides of the resulting polyhedron (Figure 10).

```
for each cell in cell list Cells {
  if cell type is POINT {
    extrude point to create line;
  } else if cell type is LINE {
    extrude line to create triangle strip;
  } else {
    for each edge (p1,p2) in cell {
      get_cell_edge_nbrs(Cells, cell, p1, p2, nbrs,nnbrs);
```

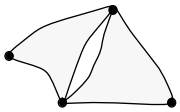


Figure 4. Curvilinear triangles

```
      if (nnbrs == 0) {
        extrude edge to create triangle strip;
      }
    }
  }
}
```

4.0 Conclusion

The cell structure is a compact, general data structure that has been used to implement a variety of visualization algorithms. The cell structure is particular useful when cell adjacency information is required. Such operations require *O(1)* time and space complexity. Hence the cell structure can be used to implement algorithms of *O(n)* time complexity.

As compared to more complex data structures, the cell structure is limited in two important ways. First, the cell structure does not represent "ordering" information. That is, given a topological feature such as an edge, the particular order of using faces around the edge. Second, because intermediate topology is implicit, certain types of topology requiring explicit information cannot be properly represented. Figure 4 is one such example. Here two curvilinear triangles share common vertices, put not edges. In such situations more explicit hierarchical structures, such as the winged edge or radial edge structures, are appropriate. It is possible to address these limitations by performing local geometric processing to order usage, or to add conditional hierarchial information to the cell structure.

References

[1] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in Scientific Computing. *Computer Graphics*, 21(6), Nov. 1987.

[2] N. M. Patrikalakis, editor. *Scientific Visualization of Physical Phenomena*. Springer-Verlag. Tokyo 1991.

[3] G. M. Nielson and B. Shriver, editors. *Visualization in Scientific Computing*. IEEE Computer Society Press. Los Alamitos, CA. 1990.

[4] A. V. Aho and J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company. Reading, MA. 1983.

[5] W. E. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163-169, July, 1987.

[6] G. Wyvill and C. McPheeters and B. Wyville. Data Structures for Soft Objects. *Visual Computer.* 2(4):227-234

[7] A. Paoluzzi and F. Bernardini and C. Cattani and V. Ferrucci. Dimension-Independent Modeling with Simplicial Complexes. *ACM Transactions on Graphics.* 12(1), January, 1993.

[8] E. Brisson. Representing geometric structures in d-dimensions: Topology and order. *ACM Symposium on Computational Geometry.* ACM Press, New York,1989.

[9] R. Haimes and M. Giles. VISUAL3: Interactive Unsteady Unstructured 3D Visualization. AIAA Report No. AIAA-91-0794. January, 1991.

[10] L. Gelberg, D. Kamins, D. Parker, and J. Stacks. Visualization Techniques for Structured and Unstructured Scientific Data. *SIGGRAPH `90 Course Notes for State of the Art Data Visualization*. August, 1990.

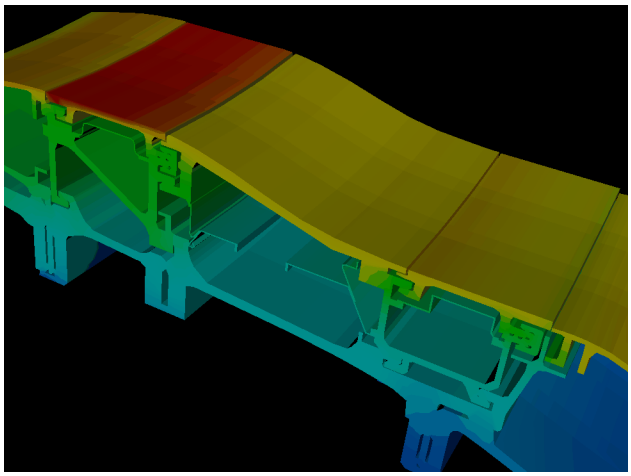


Figure 5. Unstructured grid representation.

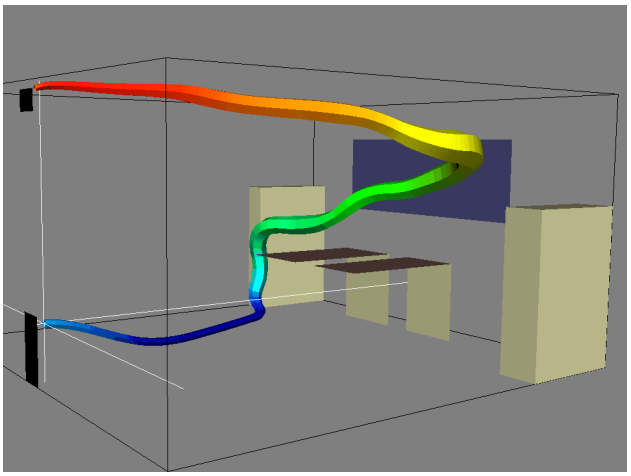


Figure 6. Stream tube propagation.

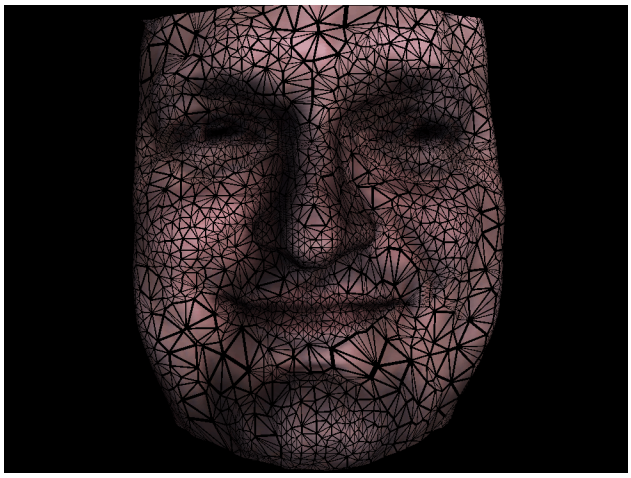


Figure 7. 90% decimated polygonal mesh, triangles shrunk to show shape.

Figure 8. Feature normal generation from decimated geometry of Figure 7.

Figure 9. Triangle strip generation from original range data. Every other triangle strip is turned off.

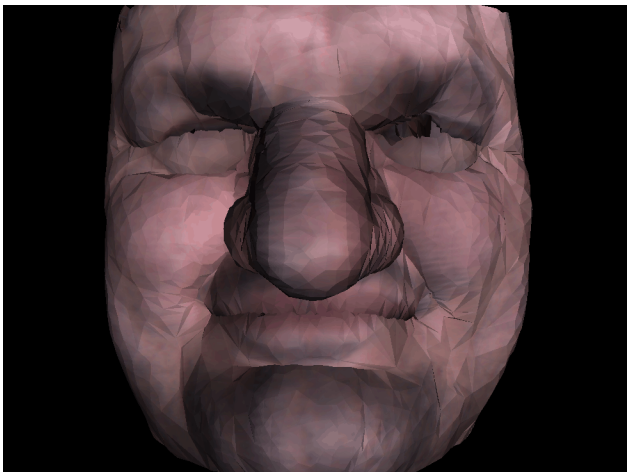


Figure 10. Extrusion (along surface normals) of surface from Figure 8 to create closed geometry.

A Topology Modifying Progressive Decimation Algorithm

William J. Schroeder
GE Corporate R&D Center

Abstract

Triangle decimation techniques reduce the number of triangles in a mesh, typically to improve interactive rendering performance or reduce data storage and transmission requirements. Most of these algorithms are designed to preserve the original topology of the mesh. Unfortunately, this characteristic is a strong limiting factor in overall reduction capability, since objects with a large number of holes or other topological constraints cannot be effectively reduced. In this paper we present an algorithm that yields a guaranteed reduction level, modifying topology as necessary to achieve the desired result. In addition, the algorithm is based on a fast local decimation technique, and its operations can be encoded for progressive storage, transmission, and reconstruction. In this paper we describe the new progressive decimation algorithm, introduce mesh splitting operations and show how they can be encoded as a progressive mesh. We also demonstrate the utility of the algorithm on models ranging in size from 1,132 to 1.68 million triangles and reduction ratios of up to 200:1.

1 Introduction

Even with the increasing speeds of computer hardware, interactive computer graphics and animation remains an elusive goal. Model size keeps growing, partly because of advances in data acquisition, and partly due to ever more sophisticated computer simulation and modeling techniques. Laser digitizers can generate nearly one million polygons in a 30 second span, while iso-surface generation can create 1-10 million polygons. Terrain models from high-altitude sources such as satellites are even larger: more than 10 million triangles is not uncommon.

To address this situation a variety of polygon reduction techniques have been developed to reduce model size. Siggraph '92 was a seminal year for polygon reduction with the presentation of two papers. Schroeder et al. [1] presented an algorithm called triangle decimation based on local vertex deletion followed by re-triangulation. Turk [2] described an algorithm based on dispersion of new points on top of the original mesh, followed by global re-triangulation. At Siggraph '93 Hoppe et al [3] presented a mathematically rigorous algorithm based on optimization techniques. Later that year Hincker and Hanson [4] described an algorithm based on merging regions of nearly co-planar polygons into a single large polygon, and then re-triangulating. Since that time other notable algorithms have been presented including methods that are guaranteed accurate within a global error bounds [11] or within a simplification envelope [8].

All of the algorithms described above are designed to preserve the original topology of the mesh. While this may be important for many applications (e.g., analysis or computational geometry), preserving topology introduces constraints into the reduction process. For example, Color Plate 1(a) shows a shell mesh with seven holes, and Plate 1(b) shows the result of reduction where topology is preserved. As shown in the figure, the topological constraint introduced by the holes clearly limits the ability of an algorithm to reduce the mesh.

In many applications, such as interactive navigation of geometric databases, preserving topology is not a critical constraint. Reduction algorithms are typically used to improve rendering speed or to minimize data size or compression requirements. In such applications topology-preserving reduction schemes are often incapable of achieving desired reduction levels. This results in unresponsive systems or the use of crude bounding boxes or bounding hulls to represent objects. Removing topological constraint can create large gains in reduction factors and therefore, system responsiveness and interactivity.

Another important development in the field of polygon reduction is the progressive mesh [7] introduced by Hoppe. A progressive mesh is one in which the original mesh can be decomposed into a simpler base mesh, where the base mesh is related to the original via a compact series of operations. In Hoppe's scheme, the single topology preserving operation *EdgeCollapse* (and its inverse *EdgeSplit*) is sufficient to transform the full resolution mesh into a simpler base mesh (and the base mesh back to the full resolution mesh). Progressive meshes offer many attractive properties, including the ability to compactly store and incrementally transmit and reconstruct geometry. Such capability is important for network based rendering and data transmission.

In this paper we present a new algorithm that can modify the topology of a mesh in order to guarantee a requested reduction level. Moreover, the algorithm creates progressive representations by extending the pair of operations *Edge Collapse/Split* with another pair of operators: *Vertex Split/Merge*. The algorithm is fast by virtue of a local decimation scheme similar to [1]. We begin by describing related background work, and then describe the algorithm from a high level, followed by a detailed look at each of its parts. We conclude by applying the algorithm to five data sets and report times to reduce the meshes at various levels of reduction.

2 Objectives

The motivation for this work is to support interactive visu-

alization of large geometric databases. These databases typically represent the design of industrial equipment such as aircraft engines or power generation equipment. Our approach is based on building level-of-detail (LOD) models for each part, ranging in complexity from full resolution to a bounding hull consisting of dozens of triangles. The size of our databases for a complete system may reach 100 million triangles and 25,000 separate parts, while an average database size is approximately 10 million triangles and 10,000 parts. Also, in an active design environment it is common to have a dozen or more variations of the same design, or multiple designs ongoing simultaneously. With this application in mind, we formulated the following objectives for our reduction algorithm:

- *Guaranteed reduction level*: Building a LOD database requires generating meshes of the correct complexity relative to every other level. In an automatic process with thousands of parts, only using an algorithm with a guaranteed reduction level will reliably construct a consistent database.
- *Modify topology:* Arbitrary reduction levels implies the need to modify topology. For example, to reduce a flat, square plate with a single hole that is represented by 1000 polygons to two triangles forming a square with no hole requires the elimination of the hole.
- *Progressive representation:* The decimation process must be encodable into a series of compact, incremental operations. This facilitates the transmission of the mesh across a network, and minimizes disk storage.
- *Fast:* We wish to construct an entire LOD database in a single day, or rapidly process parts as they are designed and submitted to the database. From our statistics cited previously, this requires a triangle processing rate of approximately 100 million triangles per day. At this rate we feel confident that we can manage a business-wide design process.
- *Robust*: Since the LOD database is built completely automatically, the algorithm must run without human intervention or error correction.

3 Background

In this section we briefly describe some related work that forms the basis of our progressive decimation algorithm.

3.1 Topology Modifying Algorithms

We began our work by investigating two important polygon reduction algorithms that modify topology. Rossignac [9] uses a vertex merge technique, where local vertices can be identified using an octree or 3D bin array. Vertices lying close to each other are merged into a single vertex, and the topology of the affected triangles are then updated. This may consist of either triangle deletion (if two or three of the vertices are merged), or updating the connectivity list of the triangle. He et al. [10] use a volume sampling approach, where the triangles are convolved into a volume to generate scalar values (e.g., scalar value may be distance from original mesh). Then an iso-surfacing technique such as marching cubes can be used to extract a surface approximating the original mesh.

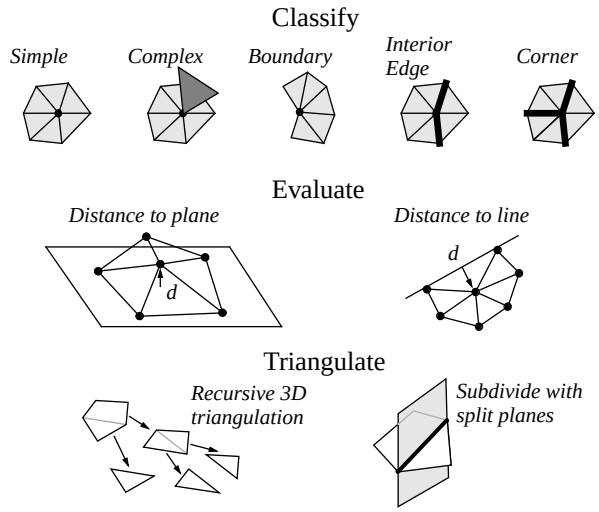


Figure 1. Overview of the decimation algorithm.

Rossignac's method is capable of generating representations at any reduction level. Unfortunately, it is not amenable to a compact progressive operator. For example, if merging neighboring vertices eliminates 1000 triangles, we need to keep track of each triangle modified and its relationship to the merged vertices, and do this for each merging step of the algorithm. Another problem with this method is that points can be merged regardless of surface coherence. For example. points that lie close to one another (measured via Euclidean distance) but are far apart (via surface distance) or even on separate surfaces may be merged.

The volume sampling technique can generate representations at almost any level by judicious selection of the sampling (or volume) dimensions. Unfortunately, it is not possible to recover the original mesh using this technique, since the extracted iso-surface has no direct relation to triangles in the original mesh. Also, there is no method to control the number of triangles produced by the method, and the number of triangles may vary unpredictably as the volume dimensions are changed.

3.2 Decimation

The decimation algorithm has been widely used because of its combination of desirable features: *O(n)* time complexity, speed, simplicity, and the ability to treat large meshes. In addition, it preserves high-frequency information such as sharp edges and corners, and creates reduced meshes whose vertices are a subset of the original vertex set, thereby eliminating the need to map vertex information. However, as originally presented the decimation algorithm is topology preserving and provides no progressive mesh representation.

The decimation algorithm proceeds by iteratively visiting each vertex in the triangle mesh. For each vertex, three basic steps are carried out. The first step classifies the local geometry and topology in the neighborhood of the vertex. The clas-

sification yields one of the five categories shown in Figure 1: *simple*, *boundary*, *complex*, *edge*, and *corner* vertex. Based on this classification, in the second step a local planarity measure is used to determine whether the vertex can be deleted. Although many different criterion are possible, distance to plane (for simple vertices) and distance to line (for edge and boundary vertices) has proven to be useful. If this decimation criterion is satisfied, in the third step the vertex is deleted (along with associated triangles), and the resulting hole is triangulated. The triangulation process is designed to capture sharp edges (for vertices classified *edge* and *corner*) and generally preserve the original surface geometry. (See [1] for additional details.)

An attractive feature of the decimation algorithm is its relative speed. We used the decimation algorithm as the basis on which to build our progressive algorithm.

3.3 Progressive Meshes

A progressive mesh is a series of triangle meshes $M^i = M^i(V, K)$ related by the operations

$$(\hat{M} = M^n) \rightarrow M^{n-1} \rightarrow \ldots \rightarrow M^1 \rightarrow M^0 \quad \textbf{(1)}$$

where $\hat{M}$ and M^n represent the mesh at full resolution, and M^0 is a simplified base mesh. Here the mesh is defined in terms of its geometry V, or vertex set in R^3, and K is the mesh topology specifying the connectivity of the mesh triangles.

It is possible to choose the progressive mesh operations in such a way to make them invertible. Then the operations can be played back (starting with the base mesh M^0)

$$M^0 \rightarrow M^1 \rightarrow \ldots \rightarrow M^{n-1} \rightarrow M^n \quad \textbf{(2)}$$

to obtain a mesh of desired reduction level (assuming that the reduction level is *less than* the base mesh M^0).

Hoppe's invertible operator is an edge collapse and its inverse is the edge split (*Edge Collapse/Split*) shown in Figure 5(a). Each collapse of an interior mesh edge results in the elimination of two triangles (or one triangle if the collapsed vertex is on a boundary). The operation is represented by five values

$$\text{Edge Collapse/Split } (v_s, v_t, v_l, v_r, A)$$

where v_s is the vertex to collapse/split, v_t is the vertex being collapsed to / split from, and v_l and v_r are two additional vertices to the left and right of the split edge. These two vertices in conjunction with v_s and v_t define the two triangles deleted or added. In Hoppe's presentation, A represents vertex attribute information, which at a minimum contains the coordinates x of the collapsed / split vertex v_s.

A nice feature of this scheme is that a sequence of these operators is compact (smaller than the original mesh representation), and it is relatively fast to move from one reduction level to another. One significant problem is that the reduction level is limited by the reduction value of the base mesh M^0. Since in our application we wish to realize *any* given reduction level, the base mesh contains no triangles

$$(\hat{M} = M^n) \rightarrow M^{n-1} \rightarrow \ldots \rightarrow M^1 \rightarrow (M^0 = M(V, \varnothing)) \quad \textbf{(3)}$$

(some vertices are necessary to initiate the edge split operations). Thus our progressive mesh representation consists of a series of invertible operations, without requiring any base mesh triangles.

4 Algorithm

Our proposed algorithm is based on two observations. First, we recognized that reduction operations (such as edge collapse) reduce the high-frequency content of the mesh, and second, topological constraint can only be removed by modifying the topology of the mesh. The implication of the first observation is that "holes" in the mesh (see Plates 1(a)-(c)) tend to close up during reduction. The only reason they do not close completely is because topological modification is prevented. The second observation led us to realize that we could modify the topology by "splitting" the mesh, i.e., replacing one vertex with another for a subset of the triangles using the original vertex. We also realized that any collection of triangles, whether manifold or non-manifold, could be simplified by splitting the mesh appropriately. Thus we developed our algorithm to use topology-preserving operators whenever possible, to allow hole closing (or non-manifold attachments to form), and to split the mesh when further reduction was not possible.

4.1 Overview

The algorithm is similar to the decimation algorithm. We begin by traversing all vertices, and classify each vertex according to its local topology and geometry. Based on this classification, an error value is computed for the vertex. The vertex is then inserted into a priority queue, where highest priority is given to vertices with smallest error values.

Next, vertices are extracted from the priority queue in order. The vertex is again classified, and if of appropriate classification, an attempt is then made to retriangulate the loop formed by the triangles surrounding the vertex. Unlike the decimation algorithm, where the hole is formed by deleting the vertex and triangles, in our algorithm the retriangulation is formed by an edge collapse. Note that in this process holes in the mesh may close, and non-manifold attachments may form.

If all allowable triangulations are performed and the desired reduction level is still not achieved, a mesh splitting operation is initiated. In this process, the mesh is separated along sharp edges, at corners, at non-manifold vertices, or wherever triangulation fails (due to no legal edge collapses). This process continues until the desired reduction rate is achieved.

4.2 Vertex Classification

Vertices are classified based on topological and geometric characteristics. Key topological characteristics are whether the vertex is manifold or non-manifold, or whether the vertex is on the boundary of the mesh. A man-

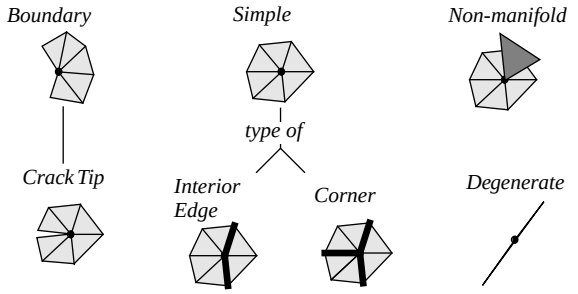


Figure 2. Vertex classification. Crack tip vertices shown displaced to emphasize topological disconnect.

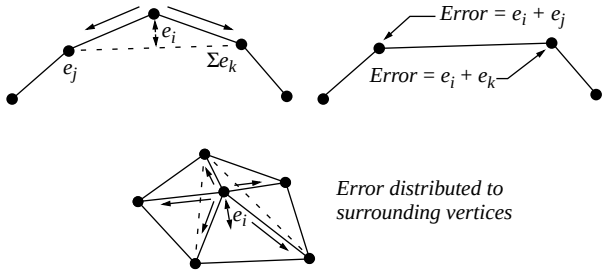


Figure 3. Computing and distributing error.

ifold vertex is one in which the triangles that use the vertex form a complete cycle around the vertex, and each edge attached to the vertex is used by exactly two triangles. A boundary vertex is used by a manifold semi-cycle (i.e., two of the edges on the mesh boundary are used by only one triangle). Note that a vertex with a single triangle attached to it is a boundary vertex. A vertex not falling into one of these two categories is classified non-manifold.

Geometric characteristics further refine the classification. A feature angle (angle between the normals of two edge-connected triangles) is specified by the user. This parameter is used to identify sharp edges or corners in the mesh, and guide the triangulation process. Whether a triangle is degenerate is another important geometric characteristic. A degenerate triangle is one that has zero area, either because two or more vertices are coincident, or because the vertices are co-linear. (Although not common in most meshes, many CAD systems do produce them.) Proper treatment of degenerate triangles is necessary to implement a robust algorithm.

Vertices are classified into seven separate categories as shown in Figure 2. There are three base types: simple, boundary, non-manifold; and four types derived from the three base types: corner, interior edge, crack-tip, and degenerate. Interior edge vertices have two feature edges, and corner vertices have three or more feature edges (if simple), or one or more feature edges (if a boundary vertex).

Crack-tip vertices are types of boundary vertices, except that the two vertices forming the boundary edges are coincident. These form as a side effect of some splitting operations. These types must be carefully treated during triangulation to prevent the propagation of the crack through the mesh.

4.3 Error Measures

The computation of vertex error is complicated by the fact that the topology of the mesh is changing. Researchers such as [8] and [11] have devised elaborate techniques to limit the global error of a reduced mesh, but these methods depend on the topology of the mesh remaining constant. When the topology changes, it is hard to build useful simplification envelopes or to measure a distance to the mesh surface. Because of these considerations, and because of our desire for rapid decimation rates, we choose to retain the distance to plane and distance to line error measure employed in the decimation algorithm (Figure 1). Our only modification to the process is to estimate the error of a vertex connected to a single triangle differently. Instead of using distance to line that a boundary vertex would use, we compute an vertex error e_i based on the triangle area a_i

$$e_i = \sqrt{a_i} \quad \textbf{(4)}$$

which is an effective edge length. This error measure has the effect of eliminating small, isolated triangles first.

One improvement we made to the process of error computation is to *distribute* and *accumulate* the error of each deleted vertex. The idea is as follows: we maintain an accumulated error value e_i for each vertex v_i. When a vertex v_j connected to v_i is deleted via an edge collapse, the error e_j is distributed to v_i using

$$e_i = e_i + e_j \quad \textbf{(5)}$$

Thus the total error e_i is a combination of the local error (i.e., distance to plane or distance to edge measure) *plus* the accumulated error value at that vertex. The error at each vertex is initially zero, but as the mesh is reduced, regions that are non-planar will generate errors, and therefore propagate the error to neighboring vertices. Figure 3 illustrates this process or a 1D polyline and 2D surface mesh.

A nice feature of this is approach is that it is relatively simple to compute a global error measure. As a vertex is deleted and the hole re-triangulated, the actual error $\hat{e}_j$ to the re-triangulated surface (versus the estimated error e_j) is computed. (The error $\hat{e}_j$ is computed by determining the minimum distance from the deleted vertex to the re-triangulated surface.) Then, the accumulated error is actually a global error bounds. This global error measure is conservative, but it can be used to terminate the algorithm for applications requiring a limit on surface error. (Note: setting a limit on maximum error may prevent the algorithm from achieving a specified reduction value.)

4.4 Priority Queue

A central feature of the algorithm is the use of a priority queue. We use an implementation similar to that described by [5], but modified to support the *Delete(id)* method,

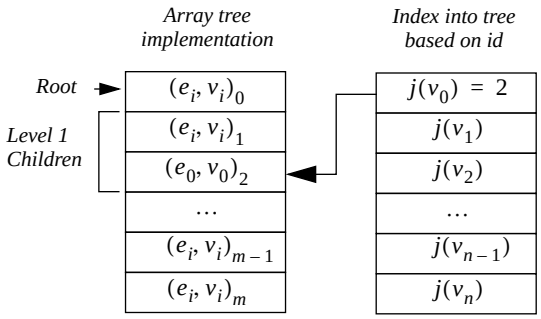


Figure 4. Modified priority queue implementation.

where *id* specifies a vertex not necessarily at the top of the queue. The implementation is based on a well-balanced, semi-sorted binary tree structure, represented in a contiguous array. In addition, we have another array indexed by vertex id, that keeps track of the location of a particular id in the priority queue array. Figure 4 illustrates the data structure. (Note that the introduction of the priority queue means the time complexity of the algorithm is *O(n log n)* as compared to the *O(n)* of the decimation algorithm.)

The addition of the *Delete(id)* method is necessary because of the incremental nature of the progressive decimation algorithm. When a vertex is deleted (via an edge collapse), both the local topology and geometry surrounding the vertex are modified. The effect is that the vertices directly surrounding the deleted vertex (i.e., those connected by an edge) have their topology and geometry modified as well. Thus the error values of these vertices must be recomputed. This means that the surrounding vertices are first deleted from the queue, the error is recomputed, and the vertices are reinserted back into the priority queue.

4.5 Triangulation

A vertex marked for deletion is eliminated by an edge collapse operation as shown in Figure 5(a). We identify the edge to collapse (and the vertex v_t) by identifying the shortest edge (v_s, v_t) that forms a valid split. If feature edges are present, we choose the shortest feature edge. This is not an optimal scheme, but when used with small feature angle values gives satisfactory results. We chose this simple approach because of our requirements on triangle processing rate.

A valid split is one that creates a valid local triangulation (i.e., triangles do not overlap or intersect). The edge collapse may modify the global topology of the mesh, either by closing a hole or introducing a non-manifold attachment. Non-manifold attachments may occur, for example, when a vertex at the entrance of a tunnel is deleted, and the tunnel entrance collapses to a line. (See Figure 5(c) and (d).)

We use a half-space comparison method to determine whether an edge collapse is valid. To determine if a vertex $v_1 = v_t$ forms (in conjunction with v_s) a valid split, we define the loop of vertices connected to v_s as $L_i = (v_1, v_2, \ldots, v_n)$. An edge collapse is valid when all

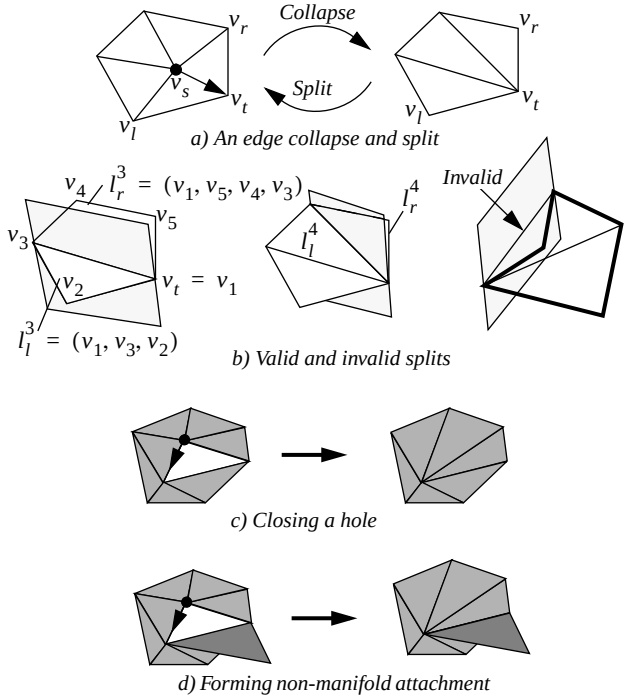


Figure 5. Triangulation via edge collapse

planes p_i passing through the vertices (v_t, v_j) for $3 \le j \le n-1$ and normal to the average plane normal N_i separate the vertex loop l_i into two non-overlapping sub-loops. This comparison can be computed by creating the $n-3$ split planes and evaluating the plane equation $p_i = N_i \cdot (v_j - v_t)$. The sub-loops are non-overlapping when all vertices in one sub-loop evaluate either positive or negative, and all vertices in the other loop evaluate to the opposite sign.

As we mentioned earlier, crack-tip vertices must be carefully triangulated to avoid propagating a crack through the mesh. Crack-tip vertices are treated like simple vertices by temporarily assuming that the two coincident vertices are one vertex. Then, if a valid split can be found, the two coincident vertices are merged with a *VertexMerge*, followed by an *EdgeCollapse*. Although this may reverse a previous *VertexSplit* operation, it does eliminate a vertex and two triangles and prevent the crack from growing. Eventually changes in the local topology and geometry will either force the crack to grow or to close up. The process will eventually terminate since the mesh will eventually be reduced to a desired level, or will be eliminated entirely.

4.6 Vertex Splits

A mesh split occurs when we replace vertex v_s with vertex v_t in the connectivity list of one or more triangles that originally used vertex v_s (Figure 6(a)). The new vertex v_t is given exactly the same coordinate value as v_s. Splits introduce a “crack” or “hole” into the mesh. We prefer not to split the mesh, but at high decimation rates this relieves topological constraint and enables further decimation.

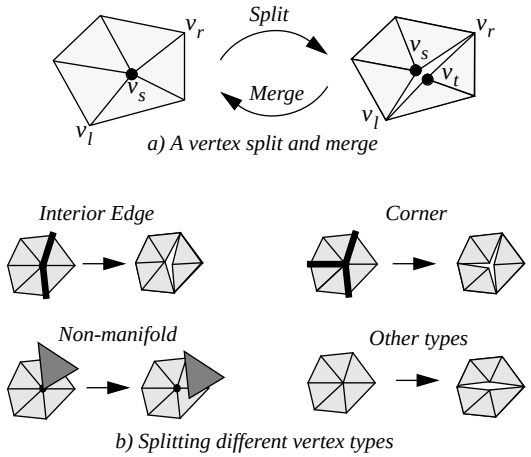


Figure 6. Mesh splitting operations. Splits are exaggerated.

Splitting is only invoked when a valid edge collapse is not available, or when a vertex cannot be triangulated (e.g., a non-manifold vertex). Once the split operation occurs, the vertices v_s and v_t are re-inserted into the priority queue.

Different splitting strategies are used depending on the classification of the vertex (Figure 6(b)). Interior edge vertices are split along the feature edges, as are corner vertices. Non-manifold vertices are split into separate manifold pieces. In any other type of vertex splitting occurs by arbitrarily separating the loop into two pieces. For example, if a simple vertex cannot be deleted because a valid edge collapse is not available, the loop of triangles will be arbitrarily divided in half (possibly in a recursive process).

Like the edge collapse/split, the vertex split/merge can also be represented as a compact operation. A vertex split/merge operation can be represented with four values

$$\text{Vertex Split/Merge } (v_s, v_t, v_l, v_r)$$

as shown in Figure 6(a). The vertices v_l and v_r define a sweep of triangles (from v_r to v_l) that are to be separated from the original vertex v_s (we adopt a counter-clockwise ordering convention to uniquely define the sweep of triangles).

4.7 Progressive Mesh Storage

The storage requirements for a progressive mesh are smaller than the standard triangle mesh representation schemes. We estimate the storage requirements as follows. A mesh generally consists of about n vertices and $2n$ triangles, for a total of $9n$ words of storage using a standard scheme of three vertex indices per triangle, and three coordinate values per vertex, each stored as one word of information. Using a progressive mesh representation, each edge split requires at a minimum the coordinates and vertex indices v_s, v_t, v_l, and v_r, and creates two triangles and one vertex, at a cost of $7n$ words of storage. A vertex split requires the four vertex indices v_s, v_t, v_l, and v_r, and typically no more than $n/4$ splits are required to reduce the mesh to M^0. We can then estimate the storage requirements to total $8n$ words, for a savings of 11% over the standard scheme. Note that even though the vertex split requires additional storage over edge collapse, it does allow us to virtually eliminate the cost of representing the base mesh.

Further savings are possible by carefully organizing the order of the playback operations. For example, by renumbering the vertices after the forward progression is complete, it is possible to eliminate the vertex index v_t.

As [7] describes, additional storage savings are possible based on the coherence of local topological operations, and by 16-bit quantization and compression of the coordinate values. We can also take into account the number of bits required for a vertex index and use smaller word sizes, if appropriate.

4.8 Error Inflection Points

An error inflection point E_i occurs when the ratio of the error is greater than a user-defined value E_r

$$\frac{e_{i+1}}{e_i} > E_r \qquad \textbf{(6)}$$

The importance of inflection points is that they mark abrupt transitions in the error of the mesh, and often correspond to significant changes in the appearance of the mesh. Thus, by tracking these inflection points, we can find natural points at which to generate LOD's for a particular part. For example, the first error inflection point always occurs at the point in which non-zero error is introduced. Geometrically, this corresponds to the point where all co-planar triangles have been removed from the interior of the mesh, and all co-linear vertices are removed from the boundary of the mesh. This is an important point in the reduction process, since at this point the reduced model is indistinguishable from the original mesh using typical surface-shading techniques.

We use error inflection points to build other levels in our sequence of LOD models. For a requested reduction level L_k, we use the mesh M_j with the closest inflection point

$$M_j : \; E_j - L_k \le E_i - L_k \;\text{ for all }\; k \le i,\, j \le n \qquad \textbf{(7)}$$

Typical values of E_r range from 10 to 100, and are empirically determined.

4.9 A Variation In Strategy

We found a variation of our splitting strategy to work well for a certain class of problems. In this variation, we *pre-split* the mesh along feature edges, at corners, and at non-manifold vertices. This strategy works well for objects with large, relatively flat regions, separated by thin, small surfaces. This is because splitting isolates regions from one another, thereby preventing the distortion of the mesh due to errors from edge collapse across the thin surfaces. An example of this behavior can be seen in Color Plate 2, where the triangles forming the thin edges of the plate disappear before the much larger triangles forming the faces of the plate.

Model	Number Triangles	Reduction 0.5			Reduction 0.75			Reduction 0.90			Reduction 1.00			Rate
		time	collapses	splits	time	collapses	splits	time	collapses	splits	time	collapses	splits	
Shell	1132	0.46	319	0	0.68	494	0	0.70	557	0	0.76	648	0	89,368
Plate	2624	1.07	656	0	1.41	984	0	1.68	1228	169	1.73	1457	221	91,006
Heat Ex.	11,006	3.27	2759	0	4.42	4139	0	5.36	5447	914	5.86	6355	1160	112,689
Turbine	314,393	158	98,401	43,011	202	157,877	69,019	229	202,702	83,214	241	232,281	88,184	78,272
Blade	1,683,472	747	421,042	0	825	631,606	0	914	758,056	0	1042	852,743	24,501	96,937

Figure 7. Results for five different meshes. Times shown are elapsed seconds. Number of edge collapses and vertex splits are also shown. The rate is the maximum number of triangles eliminated per elapsed minute.

5 Results

We implemented the algorithm as a C++ class in the *Visualization Toolkit* (vtk) system [6]. The algorithm is relatively compact requiring less than 2 KLines of code (including the priority queue implementation). No formal optimization of the code was attempted other than the usual compiler optimization. All test results are run on a 190 MHz R10000 processor (compiled and run in 32-bit mode) with 512 MBytes of memory.

We choose to test the algorithm on five different models. The first two are shown in Color Plates 1 and 2, and are a shell mesh and plate with seven holes. The next two models are extracted from CAD systems. The first is a heat exchanger with 11,006 original polygons. The next is a turbine shell consisting of 314,393 polygons. Finally, the last model is very detailed turbine blade consisting of 1.68 million triangles originally. The data was obtained by generating an isosurface from a 512 x 512 x 300 industrial CT scan of the part.

Figure 7 shows elapsed wall clock time in seconds for these five examples at reduction levels of 0.50, 0.75, 0.90, and 1.00 (elimination of all triangles). We also show the number of edge collapses and vertex splits required at each level of reduction.

Color Plates 3(a)-(f) show results for the heat exchanger. Plates 4(a)-(f) show results for the turbine shell. Plates 5(a)-(f) show the results for the turbine blade. Note that in each case the onset of vertex splitting is shown. In some of these color plates the red edges are used to indicate mesh boundaries, while light green indicates manifold, or interior edges.

The ability to modify topology provides us with reduction levels greater than could be achieved using any topology preserving algorithm. Color Plate 4 clearly demonstrates this since the maximum topology preserving reduction we could obtain was *r=0.404*. We were able to more than double the reduction level and still achieve a reasonable representation.

At extreme reduction levels the quality of the mesh varied greatly depending on the model. In some cases a few large triangles will “grow” and form nice approximations (e.g., the plate and shell). In other cases, the mesh is fragmented by the splitting process and does not generate larger triangles or good approximations. But in each case we were able to recognize the part being represented, which is sufficient for initial camera positioning and navigation. We were generally pleased with the results of the algorithm, since the creation of non-optimal meshes is well balanced by the algorithm’s speed, ability to process large meshes, and robustness.

6 Conclusion

We have described a topology modifying decimation algorithm capable of generating reductions at any level. The algorithm uses the invertible progressive operators *Edge Collapse/Split* and *Vertex Split/Merge* to construct compact progressive meshes. The algorithm has a high enough polygonal processing rate to support a large scale design and visualization process. We have found it to be an invaluable tool for creating LOD databases.

References

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

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

[3] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle. Mesh optimization. *Computer Graphics Proceedings* (SIGGRAPH ‘93), pp 19-26, August 1993.

[4] P. Hinker and C. Hansen. Geometric optimization. In *Proc. of Visualization ‘93*. pages 189-195, October 1993.

[5] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[6] W. J. Schroeder, K. M. Martin, W. E. Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics*. Prentice-Hall, 1996.

[7] H. Hoppe. Progressive Meshes. *Proc. SIGGRAPH ‘96*, pp 99-108, August 1996.

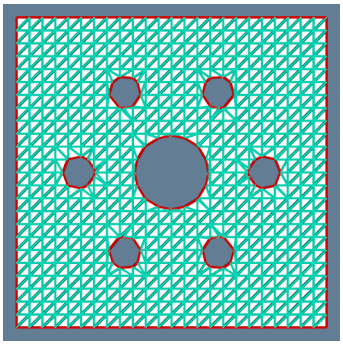
[8] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, W. Wright. Simplification envelopes. *Proc. SIGGRAPH ‘96*, pp 119-128, August 1996.

[9] J. Rossignac and P. Borel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pp. 455-465, Springer-Verlag, June-July 1993.

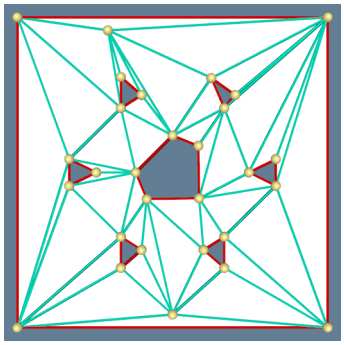
[10] T. He, L. Hong A. Kaufma, A. Varshney, S. Wang. Voxel based object simplification. In *Proc. of Visualization ‘95*, pp. 296-303, October 1995.

[11] R. Klein, G. Liebich, W. Strasser. Mesh reduction with error control. In *Proc. of Visualization ‘96*, pp. 311-318, October 1996.

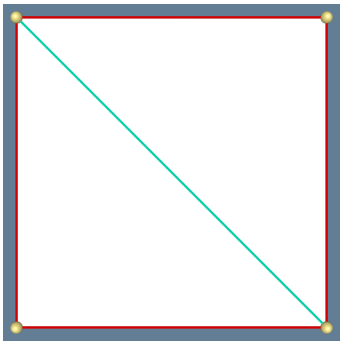
[12] P. Heckbert and M. Garland. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Carnegie Mellon University, August 1995.



a) Original model
1,132 triangles

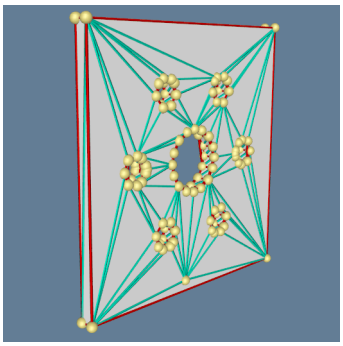


b) Topology unchanged
43 triangles

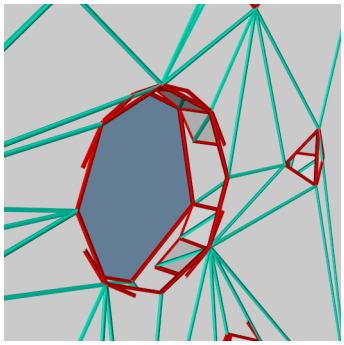


c) Topology modified

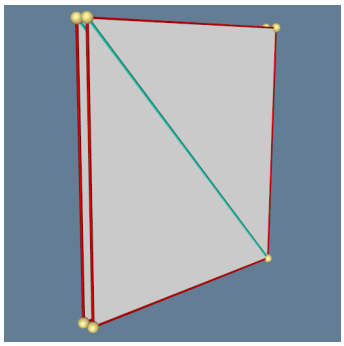
Plate 1 - Topological constraints prevent further reduction (shell with holes). Red edges are on the boundary of the mesh, light green edges are in the interior of the mesh.



a) Original model
2,624 triangles

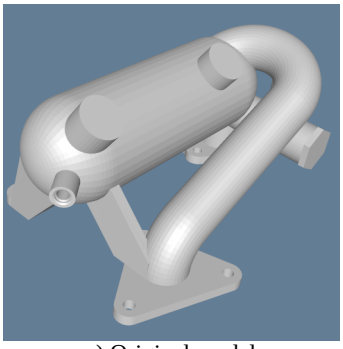


b) Shortly after splitting
157 triangles

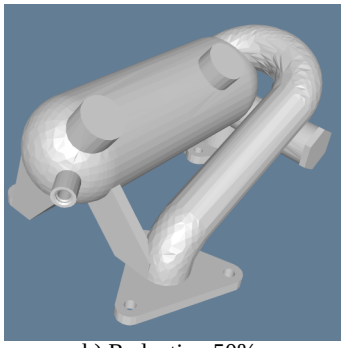


c) Final model
4 triangles

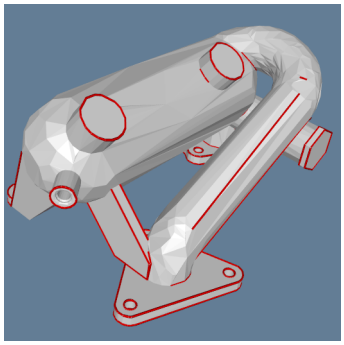
Plate 2 - Sharp edge splits and hole elimination (thin plate with holes). (b) is a close-up of the mesh showing the how the hole triangles are disconnected from the top plate.



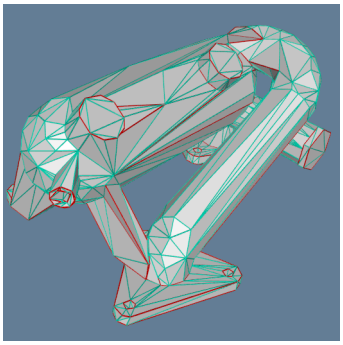
a) Original model
11,006 triangles



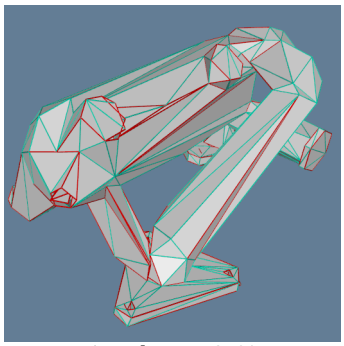
b) Reduction 50%,
5,518 triangles



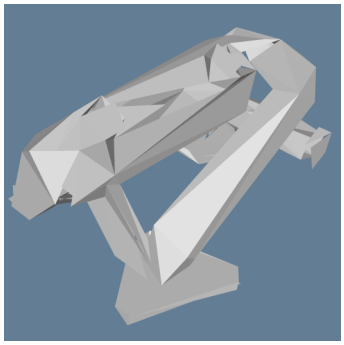
c) Just after mesh splitting
2,707 triangles, 75.4% reduction



d) Reduction 90%,
1,100 triangles

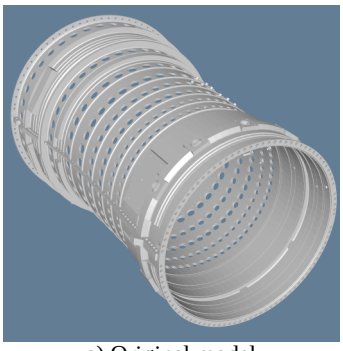


e) Reduction 95%,
550 triangles

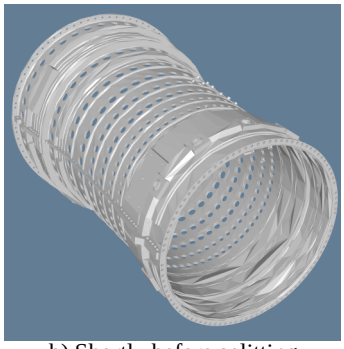


f) Reduction 98%,
220 triangles

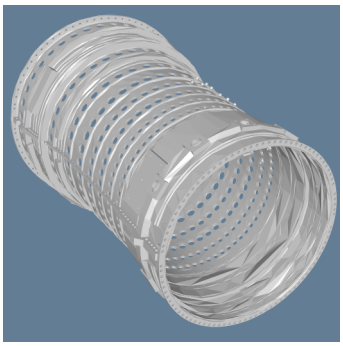
Plate 3 - Heat exchanger at different levels of reduction. (c) shows the edges that are split at the onset of edge splitting.



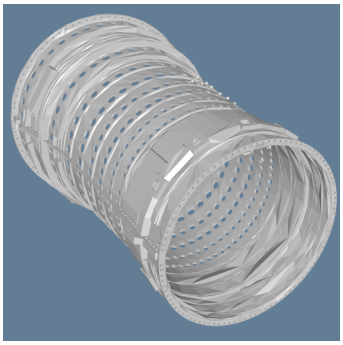
a) Original model
314,39 triangles



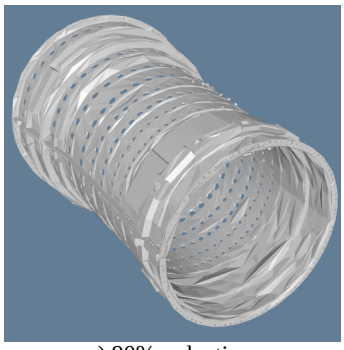
b) Shortly before splitting
187,449 triangles, reduction 40.4%



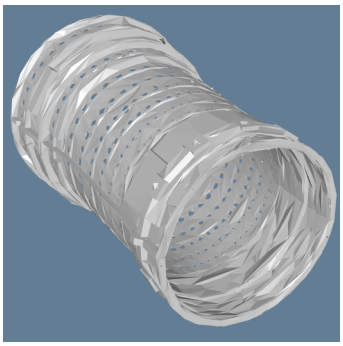
c) 50% reduction,
157,196 triangles



d) 75% reduction,
78,598 triangles

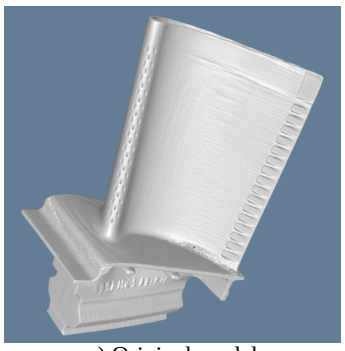


e) 90% reduction,
31,439 triangles

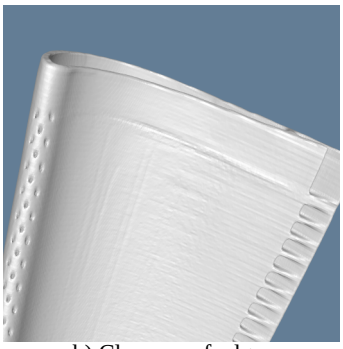


c) 95% reduction,
15,719 triangles

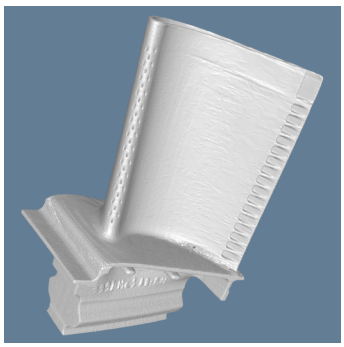
Plate 4 - Turbine shell shown at various reduction levels. Shell has thickness, and many features requiring vertex splitting.



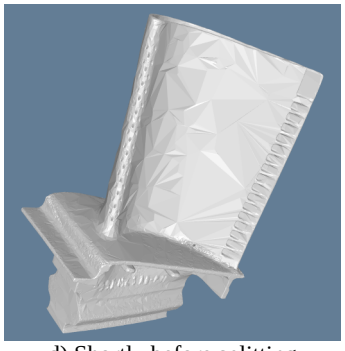
a) Original model
1,683,472 triangles



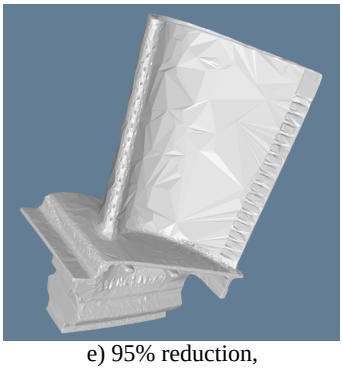
b) Close-up of edges
showing holes leading to interior



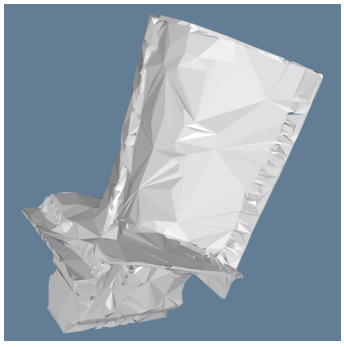
c) 75% reduction,
420,867triangles



d) Shortly before splitting
134,120 triangles, 92% reduction



e) 95% reduction,
84,173 triangles



f) 99.5% reduction,
8,417 triangles

Plate 5 - Turbine blade shown at various levels of reduction. Data derived from 512^2 by 300 CT scan.

Decimation of Triangle Meshes

Will Schroeder
GE Corporate R&D
Schenectady, NY

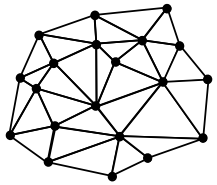
Outline

- Vertex Decimation Algorithms
  – Introduction
  – Taxonomy
  – Overview
  – Error Measures
  – Results
- Extensions
  – Modifying Topology
  – Progressive Meshes
  – Results
- Industrial Application
  – Scientific Visualization
  – Interactive Visualization
  – Terrain
  – Digitized Surfaces

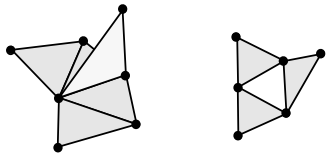
Terminology

Definition

- Polygonal mesh is a collection of non-intersecting polygons, possibly joined along common edges or at vertices
- Triangle mesh is a polygonal mesh whose polygons are triangles
- Meshes may be non-manifold



Triangle Mesh



Non-manifold forms

Requirements

Reduce number of triangles

- Form “good” approximation to original mesh
  – Visual approximations
  – Geometric approximations
  – Data approximations

Desirable qualities

- General applicability
- Treat large meshes (10^6 triangles)
- O(n) time complexity / high processing rates
- Reduced vertex set is a subset of original vertex set
  – preserve texture coordinates
  – normals
  – other vertex attributes
- Treat large meshes (10^6 triangles)
- Simple, compact, efficient algorithm

Data Sources

Digitizers:

- Laser range finders
- 3D coordinate mapping

Data Sources

Terrain:

- Satellite
- Radar Mapping
- Sonar

Data Sources

Visualization:

- Iso-Surface Generation
- Geometry Extraction Techniques
- Glyph Generation

Data Sources

Modeling / Tessellation:

- Stereo Lithography
- Graphics Representations
- Mesh Generation / Tessellation

Motivation

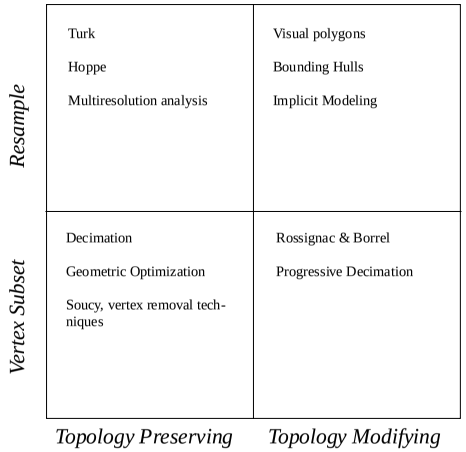
Triangle meshes are large:

- Tessellation algorithms -> 10^4 polygons
- Digital elevation data -> 10^5 to 10^7 polygons
- 3D Digitizers -> 10^5 to 10^6 polygons
- Iso-surface generation -> 10^6 triangles

Large meshes mean

- Slower rendering speeds
- Large memory requirements
- More expensive analysis

Taxonomy of Methods



Topology Preserving
Original topology of mesh maintained

Topology Modifying
Original topology of mesh may be modified

Vertex Subset
Set of reduced vertices is a subset of original vertices.

Resample
Reduced vertices do not belong to original set of vertices.

Simple Methods

- Visible polygons
  - Discard polygons not visible during viewing
- Bounding Hulls
  - Convex hulls (Delaunay triangulations. etc).
  - Shrink-wrapped polygonal surfaces
  - Bounding boxes; close fit bounding polyhedra
- Implicit modeling
  - Voxelize based on distance function
  - Use low-resolution volume
  - Extract iso-surface of distance value

Decimation Algorithm

Iterate over set of vertices; for each vertex:

- Characterize local topology and geometry
- Evaluate decimation criterion
- If criterion satisfied, remove vertices and associated triangles
- Triangulate resulting hole

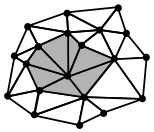
Features

- Fast O(n) time complexity
- Creates simplified mesh with subset of vertices
- Topology Preserved
- Flexible framework for different error measure / triangulation

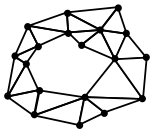
Decimation

Schroeder, Zarge, Lorensen, Proc. Siggraph '92.

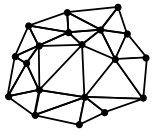
- Three basic steps (repeated for each vertex)



Evaluate local topology / geometry



Delete Vertex



Triangulate

Decimation

Evaluate local topology / geometry:

- Look for sharp corners / edges
- Look for non-manifold cases
- Evaluate decimation criterion:
  – vertex distance to plane (if interior vertex)
  – vertex distance to line (if boundary vertex or if sharp edge)
- *O(n)* for each pass

Decimation

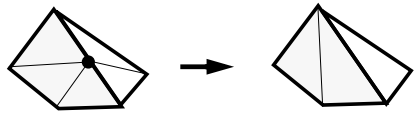
Delete vertex under following conditions:

- Vertex is not complex
- Vertex is not classified as "corner" vertex
- Vertex meets decimation criterion (i.e., distance to plane for interior vertices, distance to edge for boundary vertices or for vertices on sharp edge)
- Triangulation, including constraints on aspect ratio, is possible.

Decimation

Triangulation:

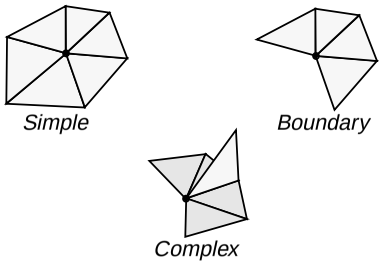
- Use 3D recursive loop splitting
- Polygons creased with sharp edge are triangulated along edge



- Aspect ratio controlled to insure good retriangulations

Vertex Classification

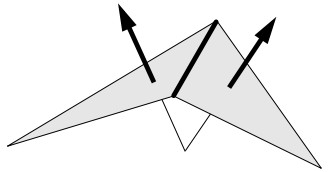
- Simple vertex
  – complete cycle of triangles
  – each triangle edge used twice
- Boundary vertex
  – semi-cycle of triangles
  – boundary edges used once
  – interior edges used twice
- Complex vertex
  – non-manifold



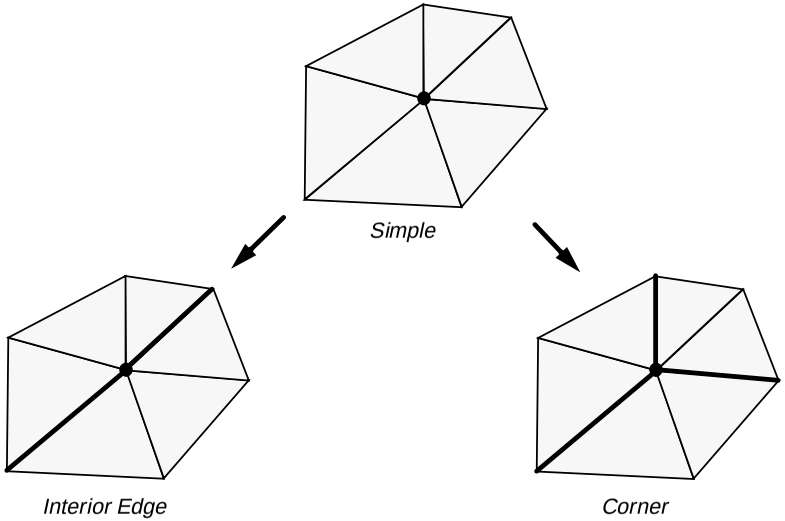
Simple Vertex Classification

Simple vertices are further classified based on geometry (i.e., evaluation of feature edges)

- Identify feature edges (specified dihedral angle)
- Interior edge vertex -> 2 feature edges
- Corner vertex -> 1 or 3 or more feature edges



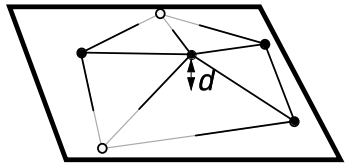
Simple Vertex Classification



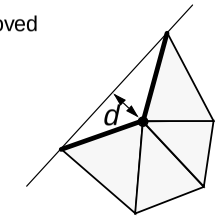
Decimation Criterion

Vertex removal based on decimation criteria

- Distance to plane
  – simple vertices
- Distance to line
  – boundary vertices
  – interior edge vertices
- Corner and complex vertices not removed



- Distance to “average” plane
  – simple vertices

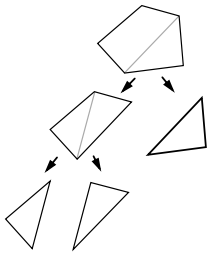


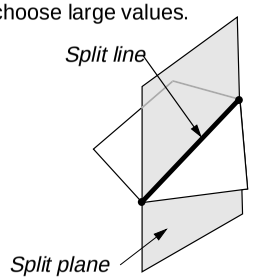
- Distance to line
  – boundary vertices
  – interior edge vertices

Triangulation

Vertex removal creates hole to be patched

- Hole is star-shaped
- Hole is generally non-planar
- Use recursive 3D loop splitting process
- Splits controlled by aspect ratio (maximum vertex distance from split line) / (length of split line) -> choose large values.

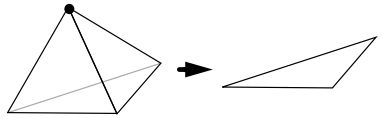




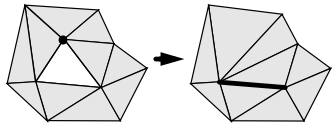
Triangulation

Triangulation Failure:

- Not possible to find split line
- Aspect ratio not satisfied
- Collapsing simple volumes



- Collapsing tunnels

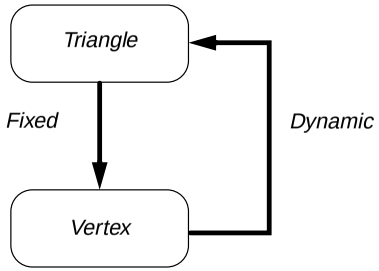


Non-manifold attachments from tunnels

Data Structure

Data structure is critical to algorithm efficiency

- Vertex list
  – x,y,z coordinates
- Triangle list
  – 3 defining vertices
- Vertex use list
  – triangles using vertex



(see included paper)

Results

- Medical
- Terrain
- Geometric Modeling
- Laser Digitizer

Characterizing Algorithms

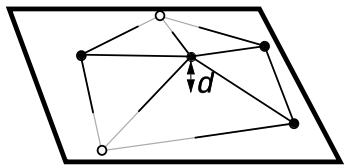
Error Measures:

- Distance measures
- Surface Area
- Removal Volume
- Energy Functions
- Hausdorff Distance

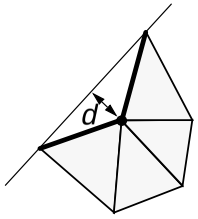
Error Measures

Distance Measures:

- Absolute
- Relative



- Distance to “average” plane



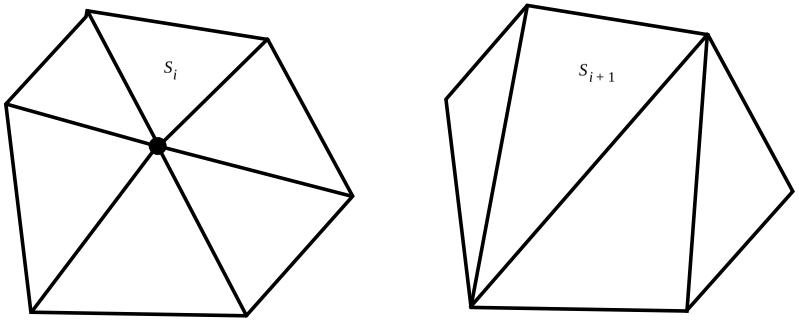
- Distance to line

Error Measures

Surface Area:

- Total area of mesh before and after reduction
- Can be computed incrementally (in most cases)

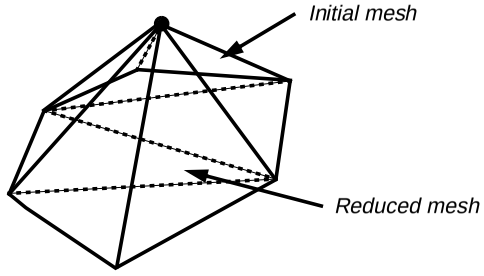
$$\Delta E_{i+1} = S_{i+1} - S_i$$



Error Measures

Removal Volume:

- Volume enclosed by initial and reduced mesh.
- Can be computed incrementally in some cases.



Error Measures

Energy Functions:

- Combinations of error functions, representational functions, and terms to guarantee minimum value.
- Error functions could include distance, surface area, and/or volume functions, as well as constraints on triangle shape.

Error Measures

Hausdorff Distance

- The maximum distance between two distance measures:
  – Distance of points in simplified mesh to original mesh
  – Distance of points in original mesh to simplified mesh
- Symmetric
- Klein, Liebich, Strasser, *Mesh Reduction with Error Control*, Proc. of Visualization ‘96.

Extensions To Decimation

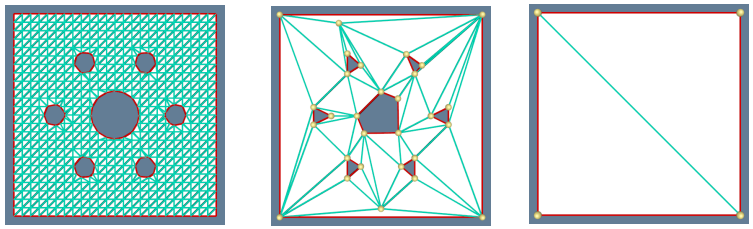
Going Beyond These Limitations

- Topology Constraints
  – Genus Preservation
  – Manifold vs. Non-Manifold Topology
- Progressive Meshes
  – Smooth transitions between reduction level
  – Encodable, compact operations on the mesh

Topological Constraints

Genus Preservation

- Collapsing holes



a) Original model
1,132 triangles

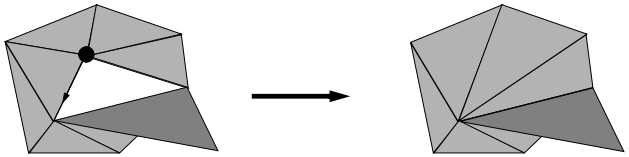
b) Topology unchanged
43 triangles

c) Topology modified
(2 triangles)

Topological Constraints

Genus Preservation

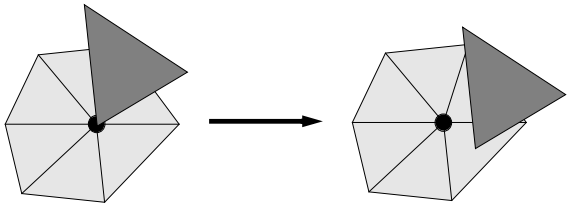
- Tunnel collapse can create non-manifold topology



Splitting Mesh

Non-manifold Vertices Can be Split

- Result is two or more manifold pieces



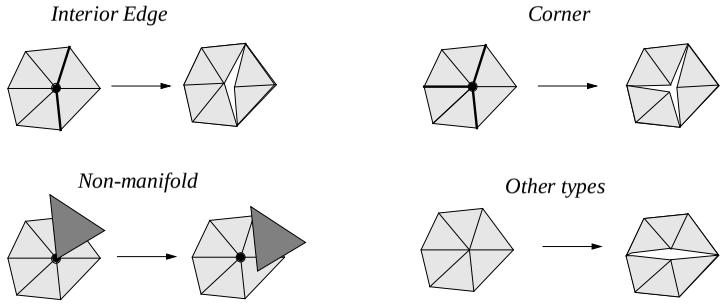
Extending Splitting

Use Splitting To Address Other Cases

- Along sharp edges
- Triangulate complex regions
- Recovering from mesh degeneracies

Split Operations

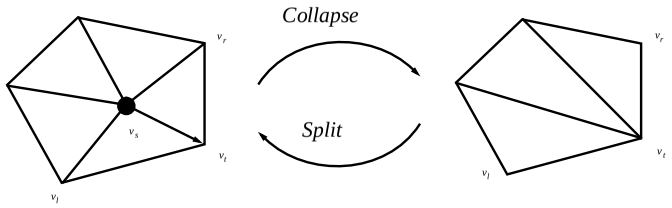
A Family of Operations



Progressive Meshes

Hoppe Siggraph '96

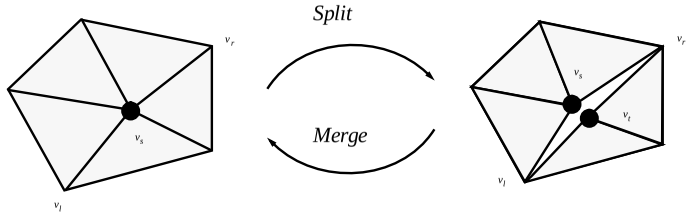
- Smooth transitions between reduction level
- Compact, encodable operations for efficient storage and transmission
- Original implementation was topology preserving edge collapse / edge split



Topology Modifying Operations

Progressive operations extended

- Vertex split - create identical vertex and update topology of mesh
- Vertex merge - merge coincident vertices



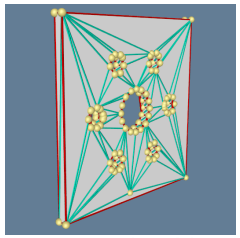
Progressive Decimation Algorithm

Extend decimation for topological modification

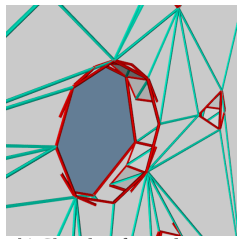
- Triangulation operation is altered to use edge collapse
- Mesh splitting is performed to resolve non-manifold / complex conditions
- Progressive operations for efficient storage and transmission; and smooth playback
- Guaranteed reduction level

Results

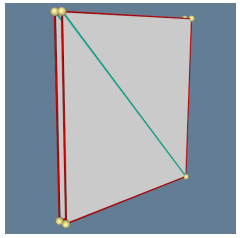
Progressive decimation



a) Original model
2,624 triangles



b) Shortly after splitting
157 triangles

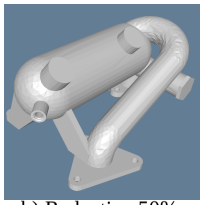


c) Final model
4 triangles

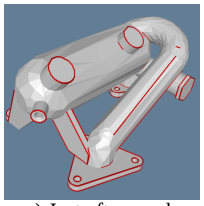
Results

Progressive decimation

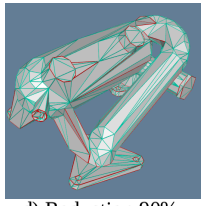
a) Original model
11,006 triangles



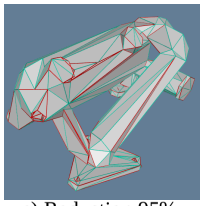
b) Reduction 50%,
5,518 triangles



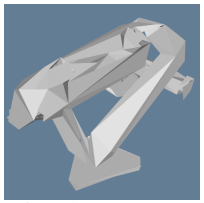
c) Just after mesh
splitting



d) Reduction 90%,
1,100 triangles



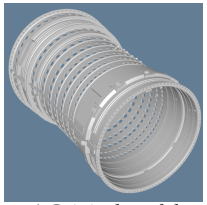
e) Reduction 95%,
550 triangles



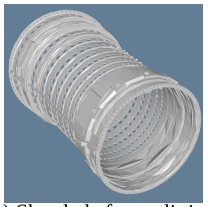
f) Reduction 98%,
220 triangles

Results

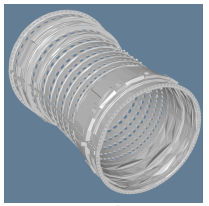
Progressive decimation



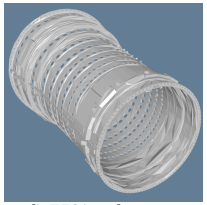
a) Original model
314,39 triangles



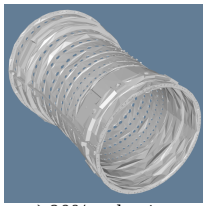
b) Shortly before splitting
187,449 triangles, reduc-
tion 40.4%



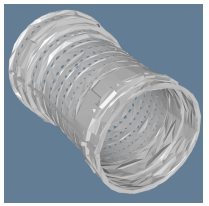
c) 50% reduction,
157,196 triangles



d) 75% reduction,
78,598 triangles



e) 90% reduction,
31,439 triangles

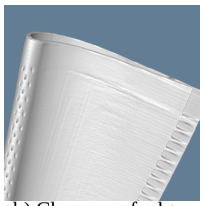


c) 95% reduction,
15,719 triangles

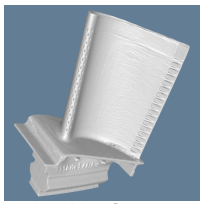
Results

Progressive decimation

a) Original model
1,683,472 triangles

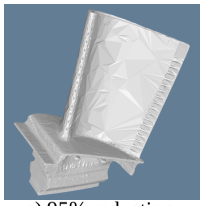


b) Close-up of edges
showing holes leading
to interior

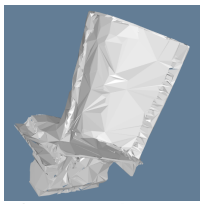


c) 75% reduction,
420,867triangles

d) Shortly before
splitting
134,120 triangles,



e) 95% reduction,
84,173 triangles



f) 99.5% reduction,
8,417 triangles

Results

Progressive Decimation

Model	Number Triangles	Reduction 0.5			Reduction 0.75			Reduction 0.90			Reduction 1.00			Rate
		time	collapses	splits	time	collapses	splits	time	collapses	splits	time	collapses	splits	
Shell	1132	0.46	319	0	0.68	494	0	0.70	557	0	0.76	648	0	89,368
Plate	2624	1.07	656	0	1.41	984	0	1.68	1228	169	1.73	1457	221	91,006
Heat Ex.	11,006	3.27	2759	0	4.42	41.39	0	5.36	5447	914	5.86	6355	1160	112,689
Turbine	314,393	158	98,401	43,011	202	157,877	69,019	229	202,702	83,214	241	232,281	88,184	78,272
Blade	1,683,472	747	421,042	0	825	631,606	0	914	758,056	0	1042	852,743	24,501	96,937

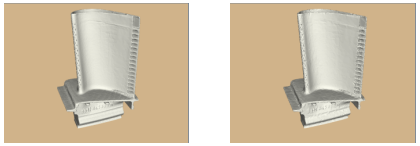
Figure 1. Results for five different meshes. Times shown are elapsed seconds. Number of edge collapses and vertex splits are also shown. The rate is the maximum number of triangles eliminated per elapsed minute.

- Nearly 100,000 triangles / elapsed minute rate
- Can process 100 million triangles / day
- Trade-off between fidelity and speed

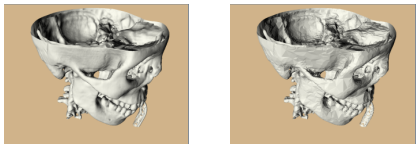
Industrial Application

Scientific Visualization

- Isosurface from industrial CT: 1.68 million triangles



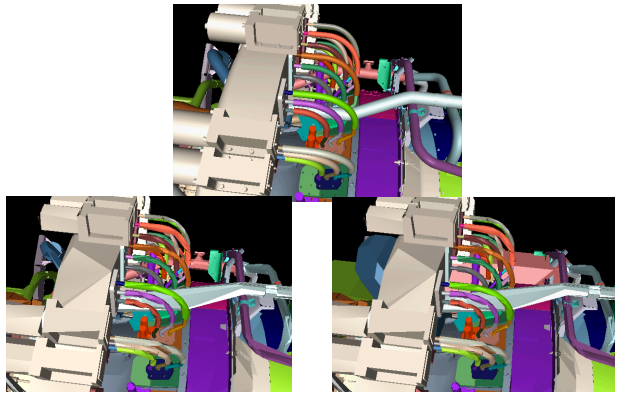
- Isosurface from medical CT:90% reduction



Industrial Application

Interactive Visualization

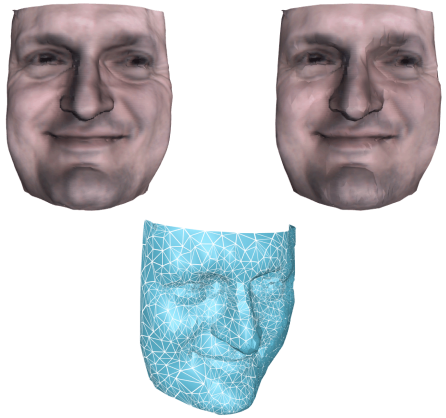
- Datasets on the order of 100 million triangles
- Need to process data within a single day



Industrial Application

Digitized Surfaces

- 95% reduction with texture



InnovMetric's Multiresolution Modeling Algorithms

Marc Soucy, Ph.D.

InnovMetric Software Inc.
2014 Jean-Talon Nord
Ste-Foy, Qué., Canada, G1N 4N6
Tel.: (418) 688-2061 Fax: (418) 688-3001
http://www.innovmetric.com

Table of Contents

1. Why create detailed polygonal models?

There exists surface-generation processes that produce very detailed polygonal descriptions:

- Laser digitizing (with or without color information)
- CAT/SCAN
- Digital terrain modeling
- Tessellation of CAD models

Why is it preferable to produce a detailed description?

1. To ensure that no features are missing. Adaptive reconstruction of an unknown surface is a VERY complex issue.

2. In some cases, we WANT the details. Archeologists and museum curators want highly-detailed models for visually inspecting the fine 3-D texture of their artifacts. They also need reduced representations to rapidly access areas of interest.

By computing a high-resolution representation in the first place, we have the potential to generate any desired reduced representation.

2. IMCompress: A vertex decimation technique that preserves 3-D tolerances

2.1 Development history

1989-1991

First implementation of the vertex decimation technique.

1992

M. Soucy, A. Croteau, D. Laurendeau, "A multi-resolution surface model for compact representation of range images", in *Proc. of IEEE Intl. Conf. on Robotics and Automation*, pp. 1701-1706, May 10-15, 1992.

M. Soucy, D. Laurendeau, "Multi-resolution surface modeling from multiple range views", in *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 348-353, June 15-18, 1992.

1994

Second implementation (IMCompress 1.0). A nearly *O(n)* time complexity has been achieved.

1997

IMCompress 2.0 offers support for grouping information, material properties, and texture coordinates.

2.2 Purpose of a vertex decimation technique

Given an initial triangular mesh, a vertex decimation technique generates reduced triangular meshes that a) are anchored on a subset of the initial vertices, and b) optimize fidelity with respect to the original shape.

❑ **Input**

A dense triangulation model describing a manifold surface embedded in 3-D space. Ideally, the model vertices should be evenly distributed over the surface.

❑ **Output**

One or several reduced triangulations having the following characteristics:

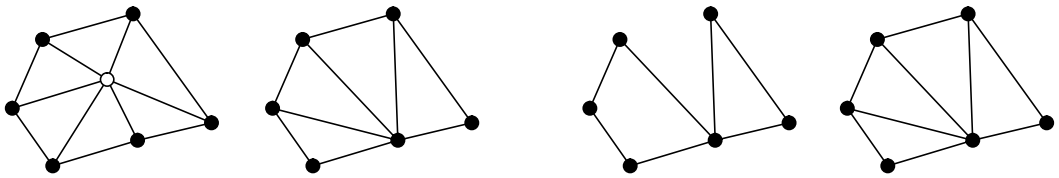
- The set of vertices forming the reduced triangulation is a subset of the original set of vertices.
- The deviation of the reduced model surface with respect to the original model is known and characterized by a metric.
- Important features (edges, corners) are automatically preserved.

2.3 Algorithm outline and properties

2.3.1 Definitions

❑ **Preservation of local topology**

Removing a vertex locally modifies a triangulation. A set of triangles is deleted, and new triangles are computed and inscribed in the triangulation. Local topology is preserved if the two sets of deleted and newly-created triangles have the same topology.



Topology is preserved *Topology is not preserved*

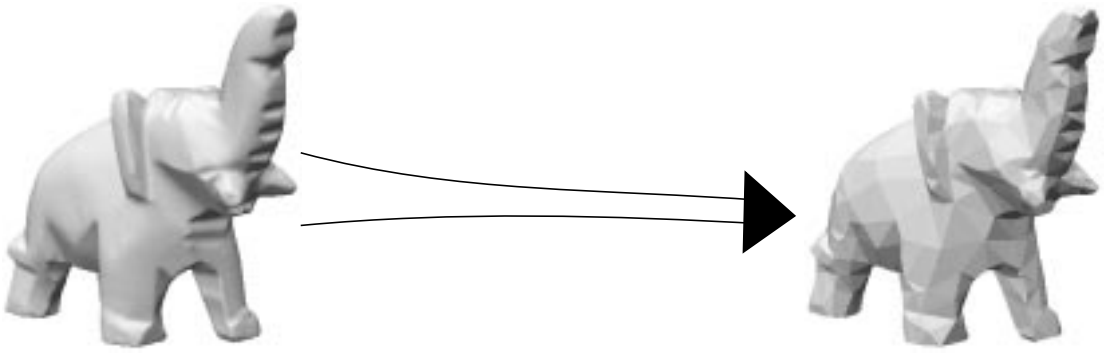
❑ **3-D tolerance**

The maximum 3-D distance between a reduced triangulation and ALL the original model vertices.

❑ **Mapping of a vertex onto a reduced triangulation**

An operation that associates a vertex with a unique corresponding point located on a triangle of a reduced triangulation.

2.3.2 Algorithm strategy

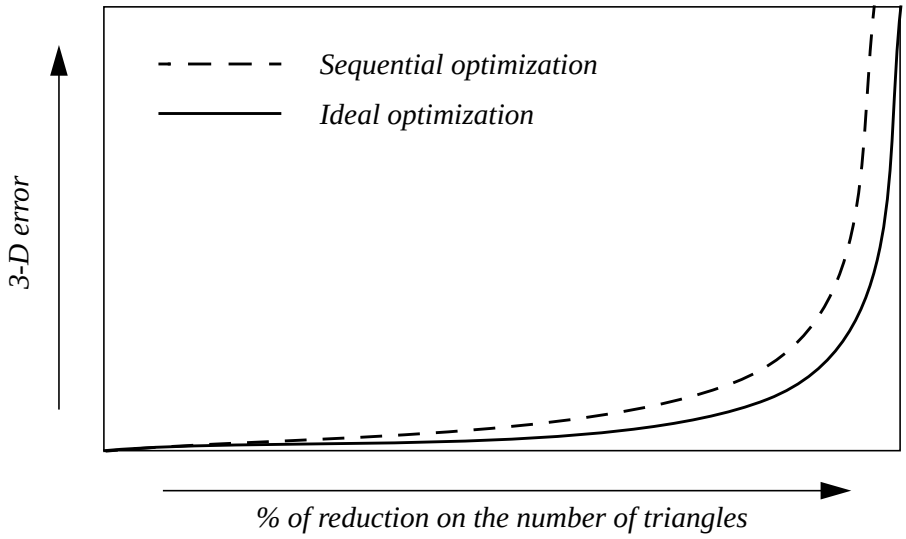


169 764 triangles *1216 triangles*

Directly computing the optimal 1216 triangle model that optimizes fidelity with respect to the original model is a VERY difficult problem.

A more practical approach consists of implementing a sequential optimization process. At each iteration, one can find the optimal vertex that should be removed.

As the number of iterations increases, the sequential optimization process tends to deviate from the ideal optimization process. Therefore, we need a very good optimization function to use sequential optimization.



❑ **Vertex decimation based on a sequential optimization process**

Algorithm: At each iteration, remove the vertex that minimizes the 3-D tolerance.

Pseudocode:

```
/* Initialization of the algorithm */
for (i=0; i< number_vertices; i++) {
     compute_retriangulation_vertex_i;
     compute_retriangulation_error_vertex_i;
     }
built_list_ordered_vertices;

/* Sequential optimization process */
while (maximum_error < THRESHOLD) {
     find_vertex_minimizing_error;
     find_neighbors_vertex;
     remove_vertex;
     delete_old_triangles;
     inscribe_new_triangles;
     for (i=0; i< number_neighbors; i++) {
          compute_retriangulation_neighbor_i;
          compute_retriangulation_error_neighbor_i;
          modify_list_of_vertices;
          }
     }
```

2.3.3 Algorithm geometrical properties

❑ **Preservation of local surface topology**

❑ **Preservation of surface edges**

The devised sequential optimization process naturally preserves important orientation discontinuities.

❑ **Equiangularity of the reduced triangulations**

❑ **Mapping of the original vertices onto the reduced triangulations**

This property allows the mapping of RGB colors onto a reduced model.

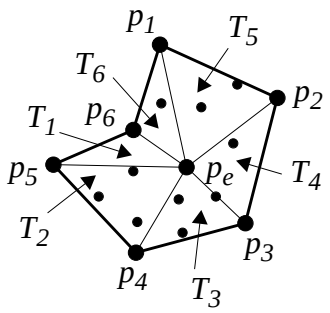
❑ **Maximum error norm**

The maximum 3-D distance between the original vertices and a reduced model is bounded by a 3-D tolerance value.

2.4 A closer look at the vertex removal operation

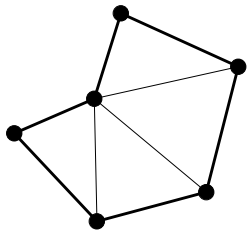
A)

- Vertex p_e is to be removed.
- Triangles T_1 to T_6 are connected to p_e.
- Vertices p_1 to p_6 are connected to p_e.
- Small dots represent removed vertices.



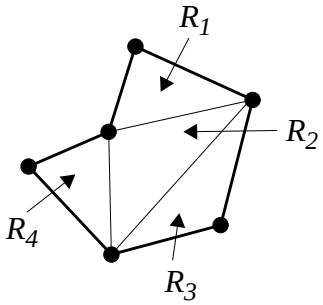
B)

- Project vertices p_1 to p_6 onto a retriangulating plane orthogonal to the surface normal at p_e.
- Verify that triangles T_1 to T_6 form a valid triangulation once projected onto the retriangulating plane.
- Compute 2-D Delaunay triangulation.



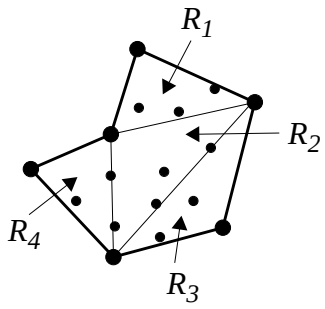
C)

- Optimize the equiangularity of the new triangles in 3-D space.



D)

- Compute 3-D distances between removed vertices and the new surface.
- Find the largest 3-D distance.



❑ **Interior vertices and contour vertices**

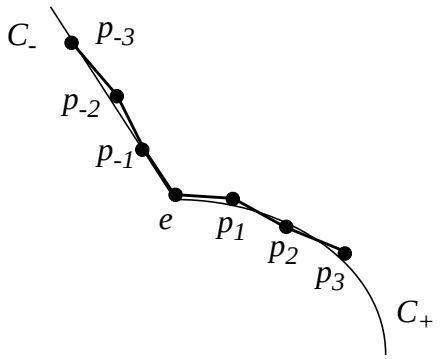
Interior vertices are completely surrounded by triangles. Contour vertices are connected to a contour edge (an edge connected to a single triangle). Additional constraints are used to remove a contour vertex:

- A contour vertex must be connected to only two contour edges.
- Interior vertices must remain inside the triangulation when a contour vertex is removed.
- The error of a removed contour vertex is the smallest 3-D distance between the vertex and the nearest contour edge.

2.5 Preservation of surface orientation discontinuities

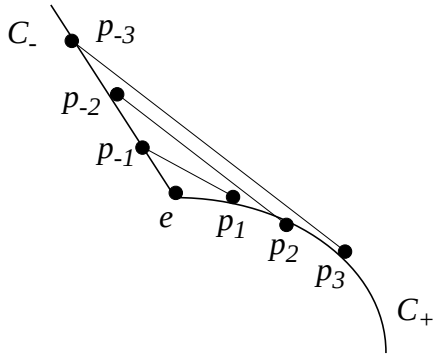
❑ **A 2-D demonstration**

- Two continuous curves C_- and C_+ are connected. The tangent is discontinuous at their intersection.
- The curves are sampled and modeled by a set of line segments.
- One of the sampled vertex (e) is located near the tangent discontinuity.



- If the discontinuity is locally significant, p_1 and p_{-1} will be removed BEFORE *e*.
- As a result, the maximum error resulting from the removal of *e* almost doubles.

  Before: *distance(e, (p_{-1},p_1))*
  After: *distance(e, (p_{-2},p_2))*
- The sequential optimization process gradually strenghtens the tangent discontinuity, such that vertex *e* will remain in the linear approximation.



❑ **Preservation of surface edges**

The previous 2-D example demonstrates the existence of a sequential strenghtening phenomenon. This phenomenon is also observed in 3-D. Vertices located near a surface edge are strenghtened by the removal of vertices located on both sides of the edge.

An edge compression phenomenon is also observed. A vertex *e* located on an orientation discontinuity may be removed if the retriangulation contains an edge joining the predecessor and successor of *e* on the edge. Therefore, surface edges are preserved and reduced, as the 3-D tolerance gradually increases.

2.6 IMCompress implementation

2.6.1 Memory requirements

IMCompress requires approximately 115 bytes per triangle.

A custom memory manager has been developed to provide garbage collection capabilities.

2.6.2 Time complexity: A nearly O(n) implementation

```
/* Sequential optimization process */
while (maximum_error < THRESHOLD) {
      find_vertex_minimizing_error;
      find_neighbors_vertex;
      remove_vertex;
      delete_old_triangles;
      inscribe_new_triangles;
      for (i=0; i< number_neighbors; i++) {
            compute_retriangulation_neighbor_i;
            compute_retriangulation_error_neighbor_i;
            modify_list_of_vertices;
            }
      }
```

The pseudocode in bold represents the time consuming operations.

- The number of neighbors is constant on average.
- The time required to compute the retriangulation of a fixed number of vertices is constant on average.
- The time required to compute the maximum retriangulation error gradually increases as vertices are removed. All removed vertices are considered in this computation.
- The time required to modify the list of vertices is constant on average.

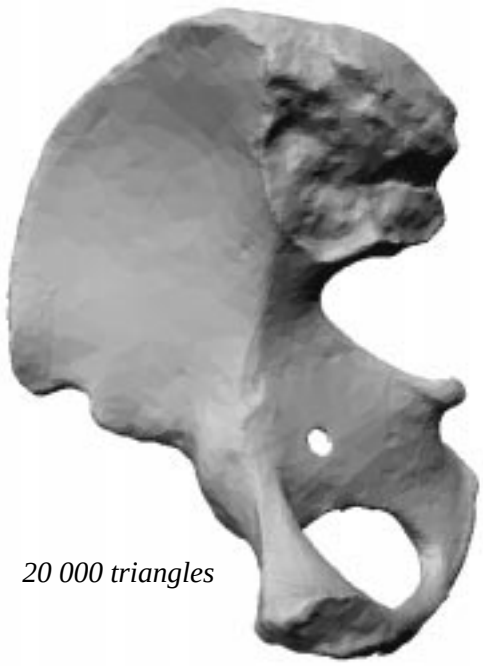
Therefore, time complexity has a linear and an exponential component. In order to achieve a nearly-linear time complexity, we have heavily optimized the maximum error computation. A 500 000 triangle model is completely reduced in 30 minutes on a 250 MHz workstation.

2.7 Results

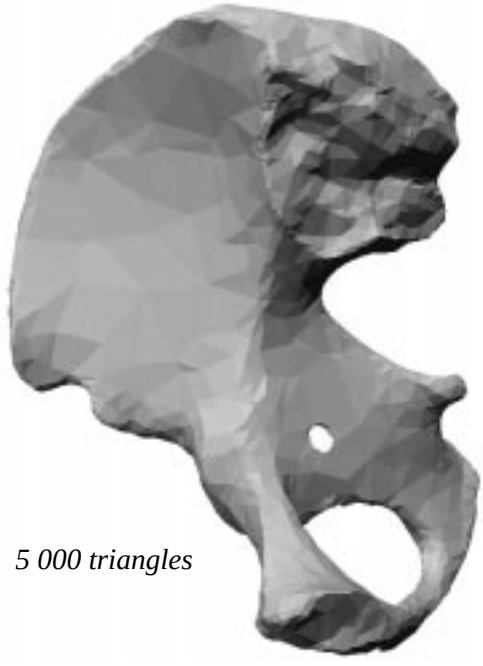
2.7.1 Hipbone model

The hipbone model has been generated using 15 Cyberware scans and InnovMetric's POLYWORKS 3-D modeling software. The original high-resolution model contains 339 478 triangles.

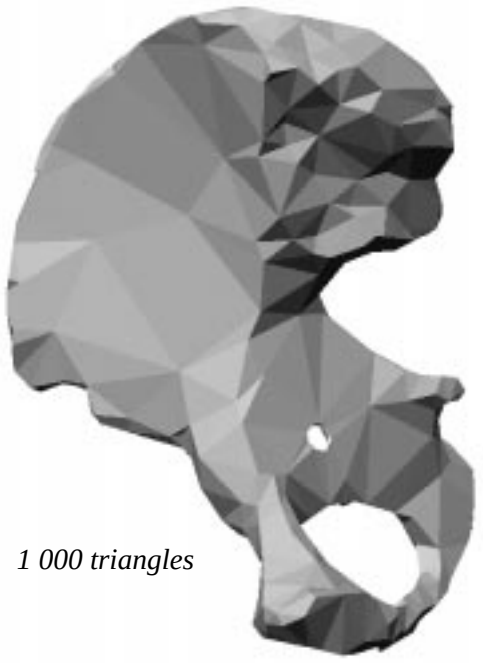
339 478 triangles



20 000 triangles



5 000 triangles

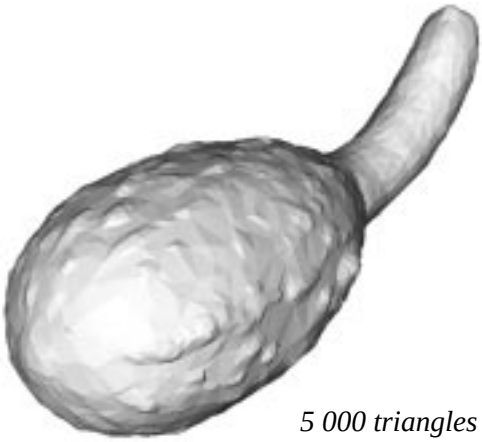


1 000 triangles

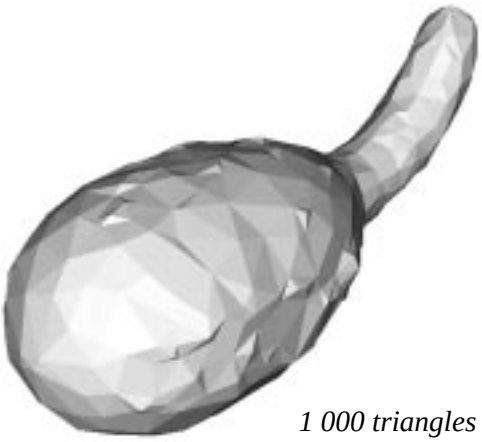
2.7.2 Squash model

The squash model has been generated using 6 scans from the National Research Council of Canada's color laser rangefinder and InnovMetric's POLYWORKS 3-D modeling software. The original high-resolution model contains 419 616 triangles. The squash is a courtesy of our local grocer.

419 616 triangles



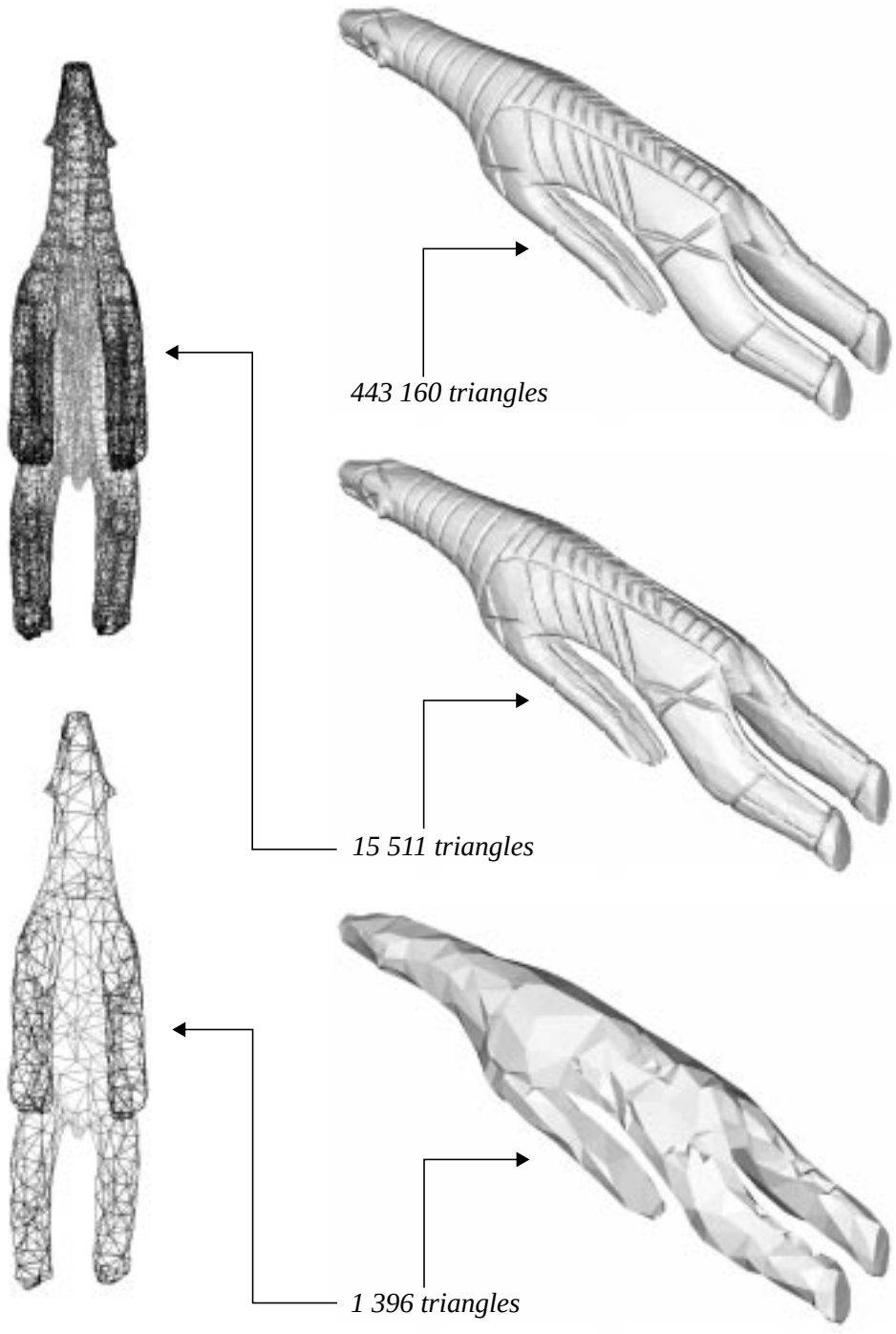
5 000 triangles



1 000 triangles

2.7.3 Ivory bear model

The ivory bear model has been generated using 20 NRCC's color laser scans and InnovMetric's POLYWORKS 3-D modeling software. The original high-resolution model contains 443 160 triangles. The ivory bear has been carved about 2 000 years ago in the central Arctic.



2.7.4 Additional remarks

All reduced models have been generated in a single pass of the IMCompress program. No hidden thresholds have been set to obtain these results. For the first two objects, the compression criteria have been specified as numbers of triangles. For the last object, the compression criteria have been specified as 3-D tolerances (1.0 and 2.0 millimeters).

The three objects have been completely reduced in 41, 56, and 59 minutes respectively on a Silicon Graphics Indigo^2 workstation equipped with a 100 MHz CPU.

3. IMTexture: Generating coarse texture-mapped models from accurate color 3-D models

3.1 Development history

1993-1994

First implementation of the texture mapping algorithm.

1994

Second implementation (IMTexture 1.0). The generated models must be rendered using the nearest texel texture-rendering mode.

1996

M. Soucy, G. Godin, R. Baribeau, F. Blais, M. Rioux, “Sensors and algorithms for the construction of digital 3-D colour models of real objects”, in *Proc. of the International Conference on Image Processing*, pp. 409-412, September 16-19, 1996.

1997

IMTexture 2.0 generates texture-mapped models that can be rendered using either the nearest texel or bilinear texture-rendering mode.

3.2 Purpose of the algorithm

Using color laser digitizers, we can produce high-precision surface triangulations in which RGB colors are computed for each triangulation vertex. Large triangulation models with color per vertex information are very realistic, but they cannot be displayed within a real-time visualization environment. The purpose of the texture-mapping algorithm is to produce coarse texture-mapped models that look similar to high-resolution models, but contain very small numbers of triangles.

❑ **Input**

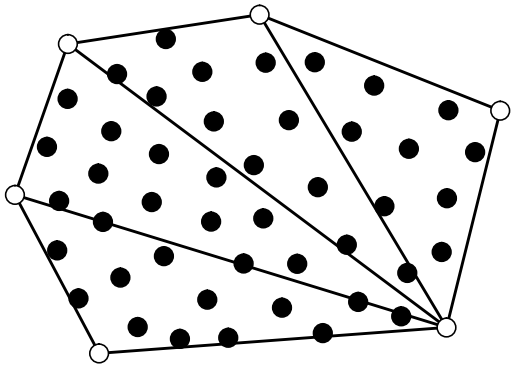
- A reduced triangulation.
- The set of original vertices, their RGB colors, and their mapping positions on the reduced triangulation.

- Texture image width and height.

❑ **Example of input data**

○ *Vertex of the reduced triangulation*

● *Removed vertex mapped onto the reduced triangulation*



❑ **Output**

The algorithm generates:

- A texture image.
- Texture coordinates for the triangles of the reduced triangulation.

3.3 Algorithm outline

Strictly speaking, IMTexture is not a tool that maps textures onto a surface. IMTexture is an algorithm that generates a geometric representation that can be displayed by texture-mapping rendering software.

❑ **First step: Tessellating the texture image**

The texture image is tessellated to accomodate all the triangles of the reduced triangulation.

❑ **Second step: Interpolating the texture image**

The high-resolution color vertices are mapped onto the texture image, and RGB colors are computed for all texels.

❑ **Third step: Enforcing texture continuity**

Constraints are applied to ensure texture continuity on the 3-D surface.

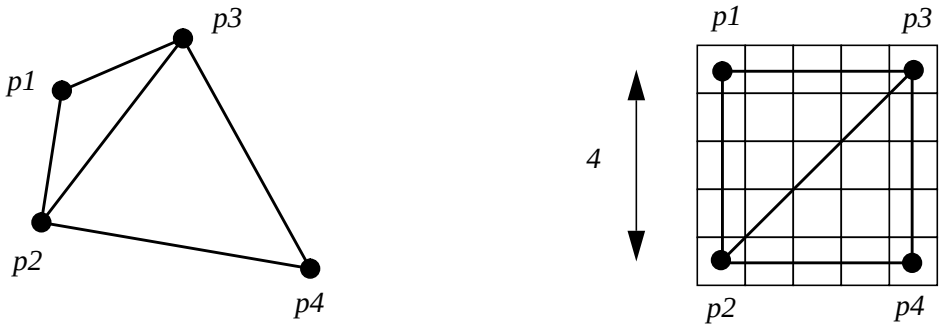
3.4 Tessellating the texture image

The user specifies the texture image width and height. The IMTexture algorithm has to distribute this texture space among all triangles. The following issues must be considered when devising a tessellation algorithm:

- The input triangulation topology should not be constrained.
- The input triangles have different 3-D sizes.
- Texture continuity must be achieved over the 3-D surface.

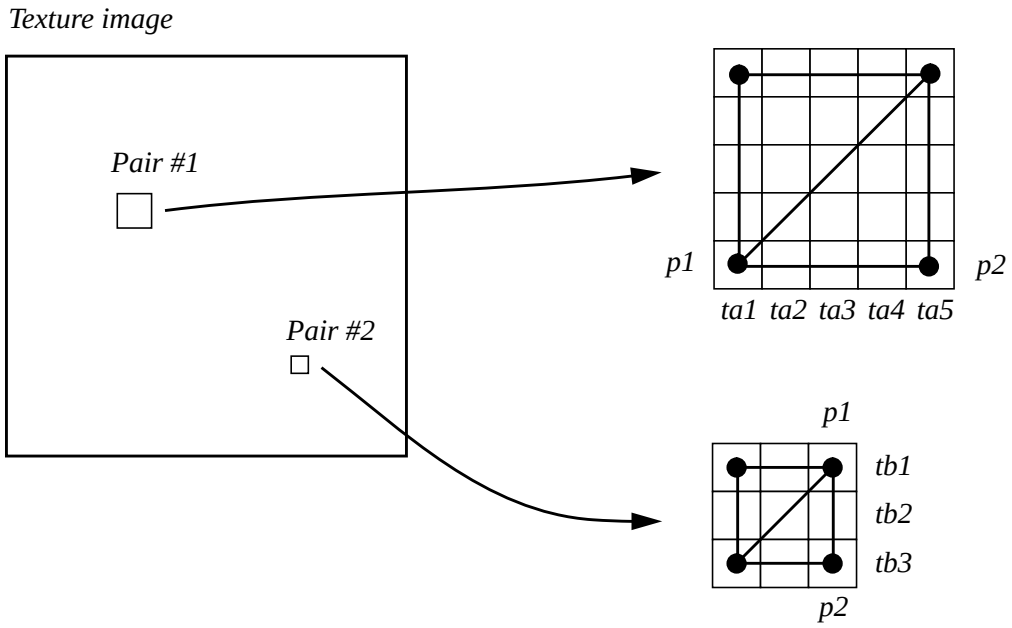
❑ **Tessellation strategy**

- The input triangulation is segmented into pairs of triangles using a recursive method. A few triangles remain orphan.
- Pairs of 3-D triangles and orphan triangles are mapped as texture squares.
- The dimensions of texture squares are powers of two.
- The four texture square corners are anchored onto texel centers.



Mapping a pair of triangles onto a 4x4 texture square

❑ **Achieving texture continuity on a triangle edge**

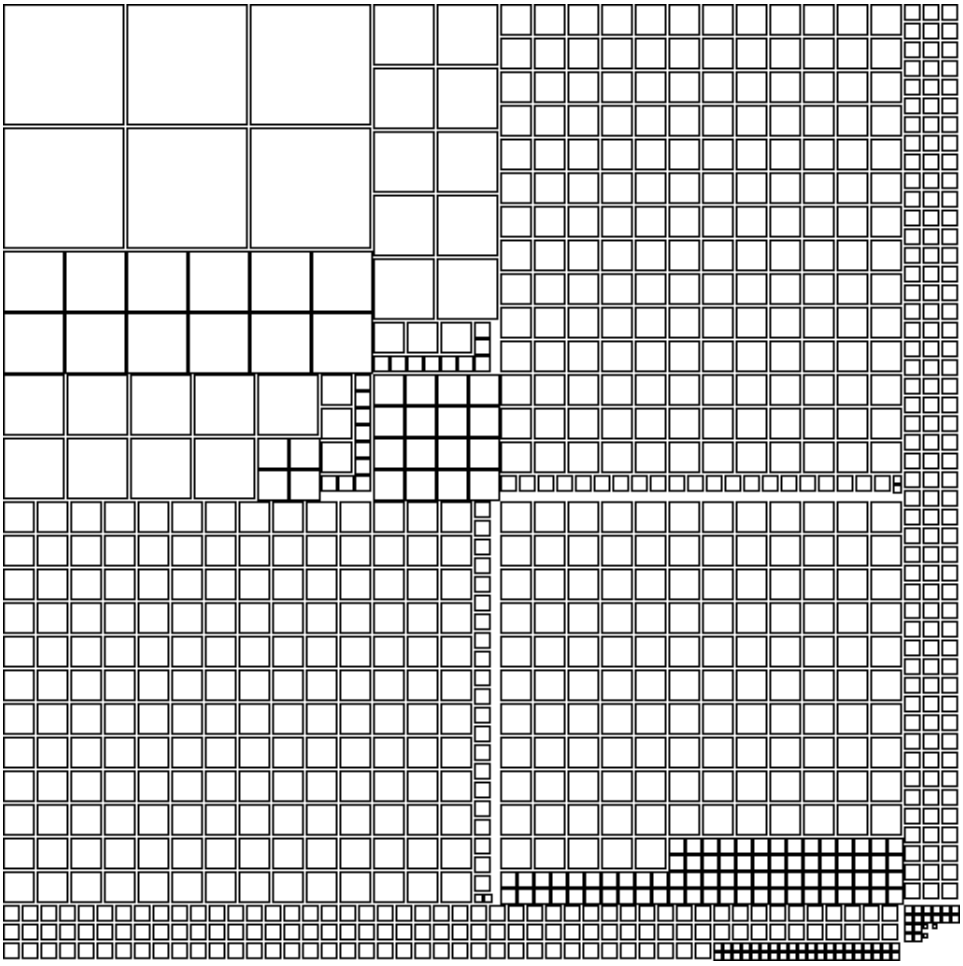


In the bilinear texture-rendering mode, texture will be continuous on the *p1-p2* edge if the following constraints are verified in the RGB space:

- *ta1= tb1 ta3= tb2 ta5= tb3*
- *ta2= 0.5*(ta1+ta3) ta4= 0.5*(ta3+ta5)*

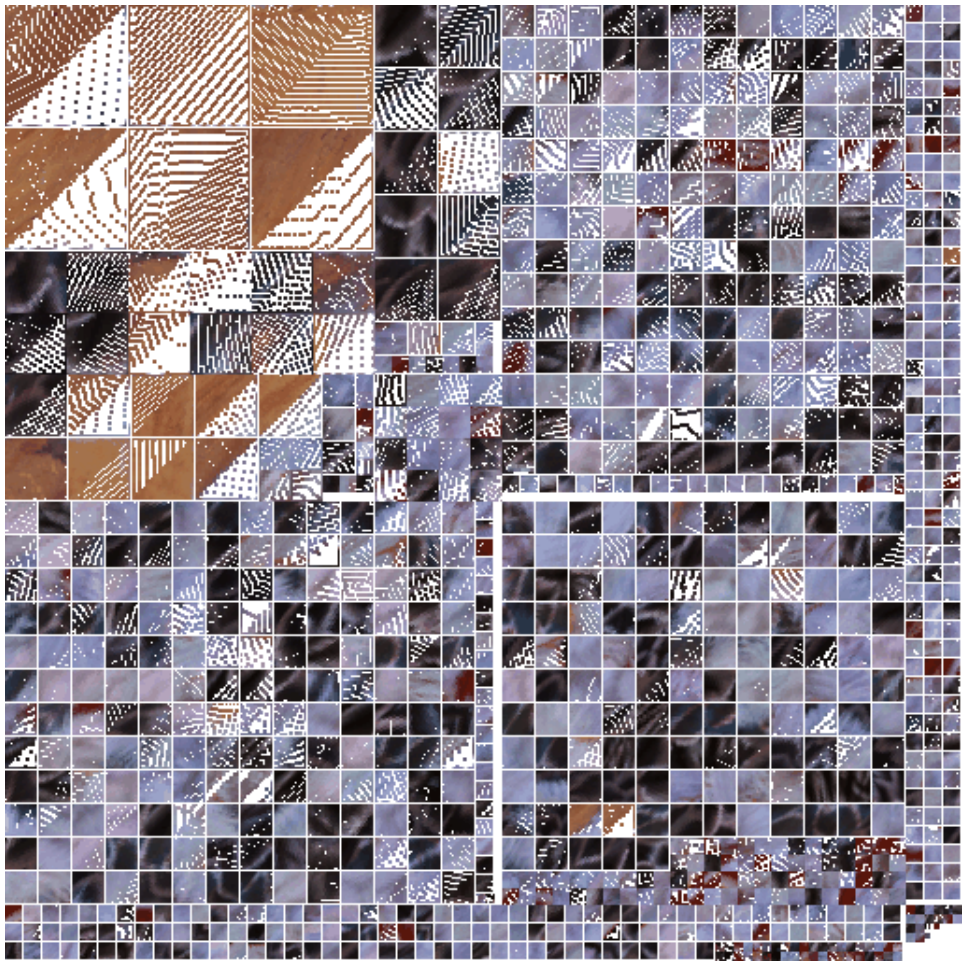
❑ **Tessellation algorithm**

An algorithm has been devised to automatically determine the dimensions and positions of all texture squares. The image below illustrates a 512x512 texture image tessellated into 1000 texture squares whose dimensions are powers of two. It should be noted that a 2^n by 2^n texture square requires 2^n+1 by 2^n+1 texels.



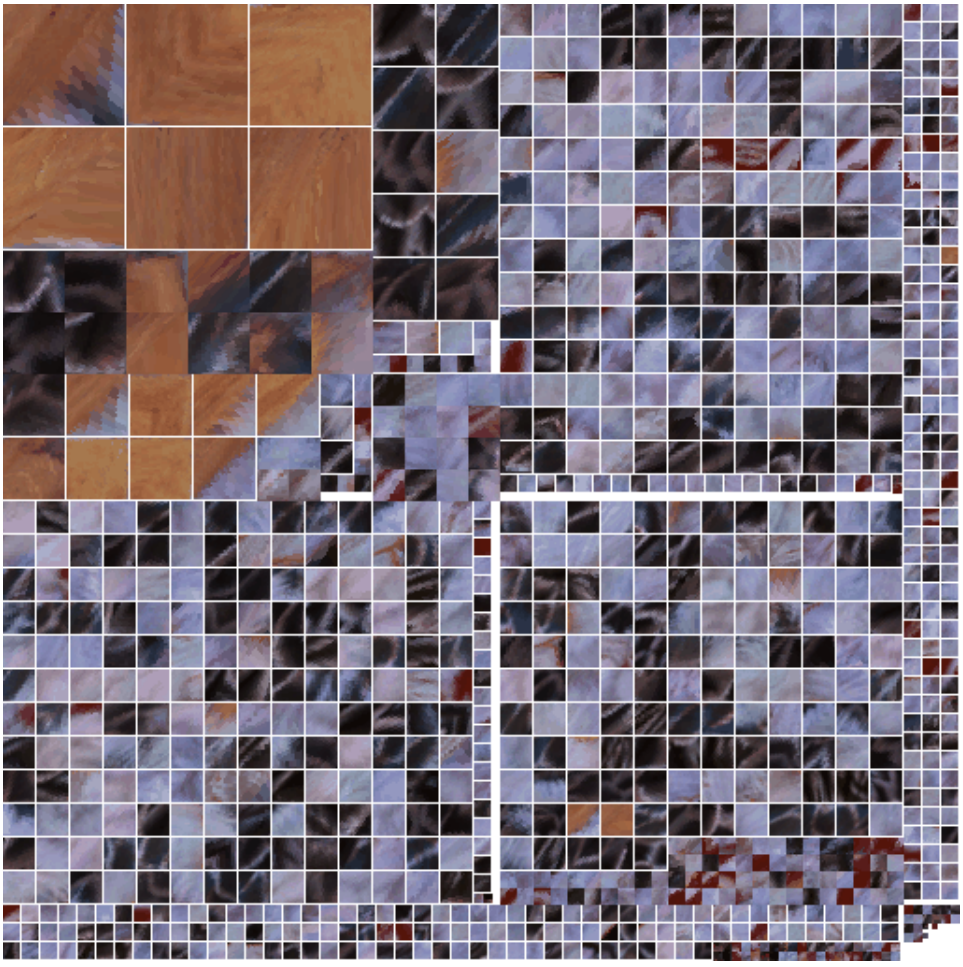
3.5 Interpolating the texture image

First, the original vertices are mapped onto the texture image, and their RGB values are attributed to the 4 neighboring texels. The contribution of a vertex is weighted using bilinear interpolation weights. The image below illustrates a 512x512 texture image after the original color data has been mapped onto it.



❑ **Assigning RGB values to all texels**

One may observe that some texels did not receive any color contribution. An additional step is thus required to assign RGB values to these empty texels. A 3-D nearest-neighbor interpolation algorithm based on a modified fast distance transform is used for this purpose. The texture image shown below results from this final interpolating step.



3.6 Enforcing texture continuity

The last processing step consists of enforcing texture continuity over the 3-D surface:

- A vertex is constrained to have a unique texture. All the texels whose centers correspond to a particular vertex must have the same RGB values.
- Texture must be continuous across an edge shared by two triangles. Texels located on triangle edges are constrained (refer to Section 3.4).

3.7 Results

3.7.1 Procedure for creating a texture-mapped model

The following procedure is used to generate a texture-mapped model from color 3-D data:

- First, a high-resolution polygonal model with color per vertex information is created from several color 3-D scans.
- IMCompress is then used to reduce the high-resolution model. The mapping information of each vertex is preserved in a file.
- IMTexture is finally invoked to generate the texture map and texture coordinates for each triangle. The user specifies the texture map dimensions. The texture map width and height must be powers of two.

Generating a 1024x1024 texture map requires approximately 1 minute.

3.7.2 Squash model

The squash model has already been shown in Section 2.7.2 without color information. The high-resolution model made of 419 616 triangles is displayed using a color per vertex rendering mode. The two reduced models are displayed using the nearest texel texture-rendering mode. The light sources and materials have been specified to ensure that the two rendering modes produce equivalent colors.

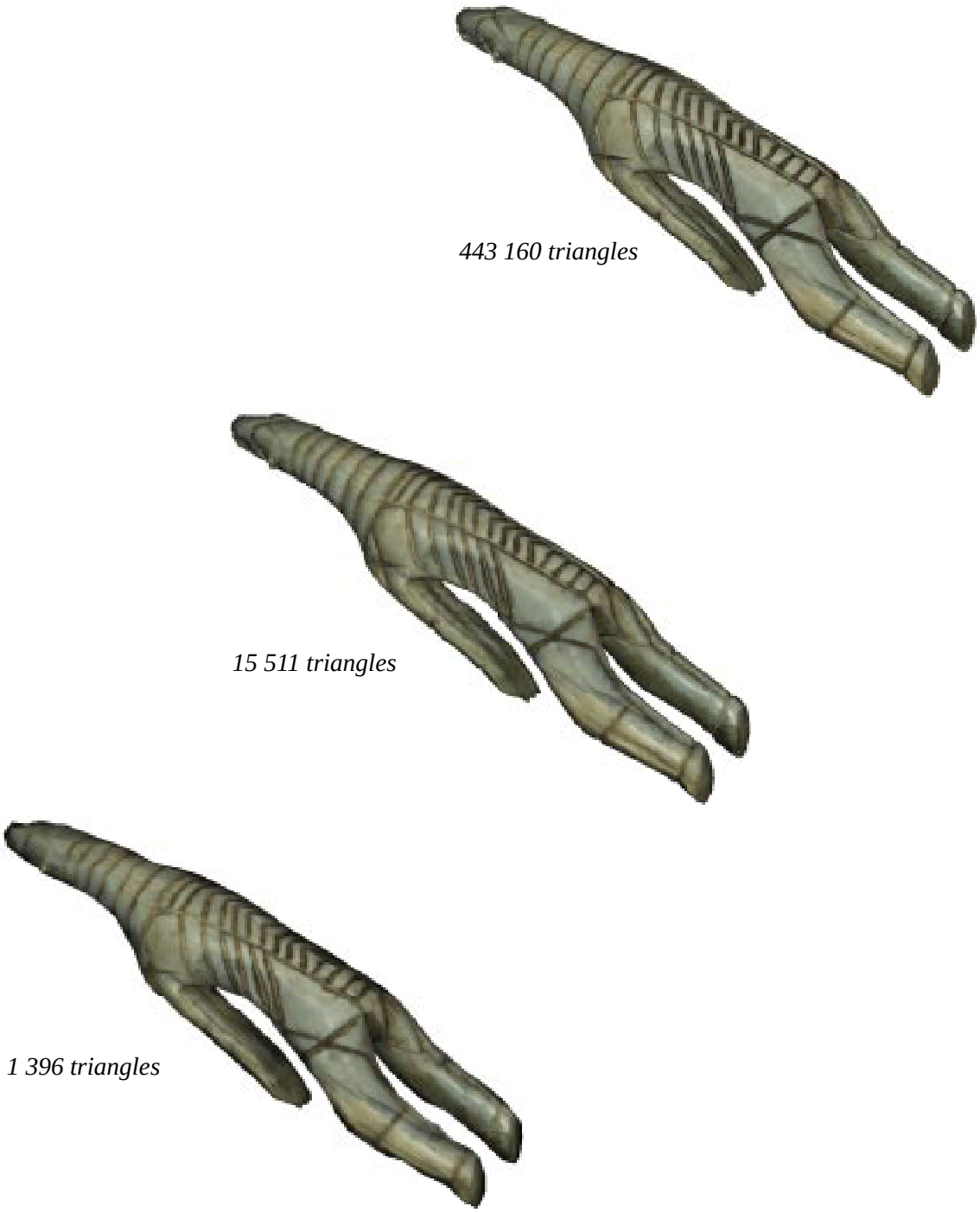
419 616 triangles

5 000 triangles

1 000 triangles

3.7.3 Ivory bear model

The ivory bear model has already been shown in Section 2.7.3 without color information. The high-resolution model made of 443 160 triangles is displayed using a color per vertex rendering mode. The two reduced models are displayed using the bilinear texture-rendering mode.



443 160 triangles

15 511 triangles

1 396 triangles

4. Packaging commercial polygon reduction applications

4.1 Topological anomalies found in polygonal models

A surface-based polygon reduction algorithm must be immune to the following anomalies, frequently encountered in tessellated models:

- **A) Degenerate triangles**

  When two or three vertices of a triangle are equal, the triangle becomes an edge or a single point and is therefore degenerate.

- **B) Duplicate triangles**

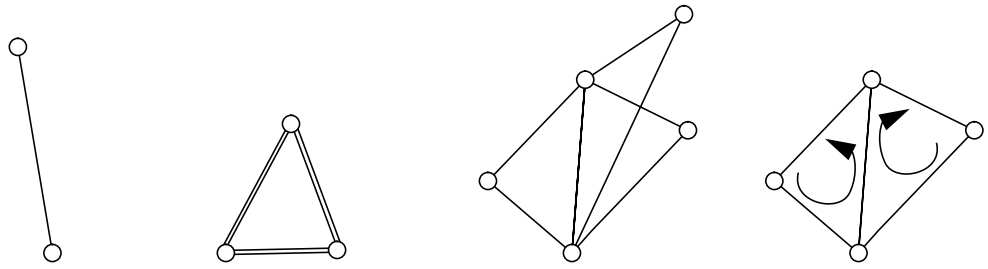
  This problem arises when there are several copies of the same triangle in a model.

- **C) Degenerate edges**

  A degenerate edge is shared by more than two triangles.

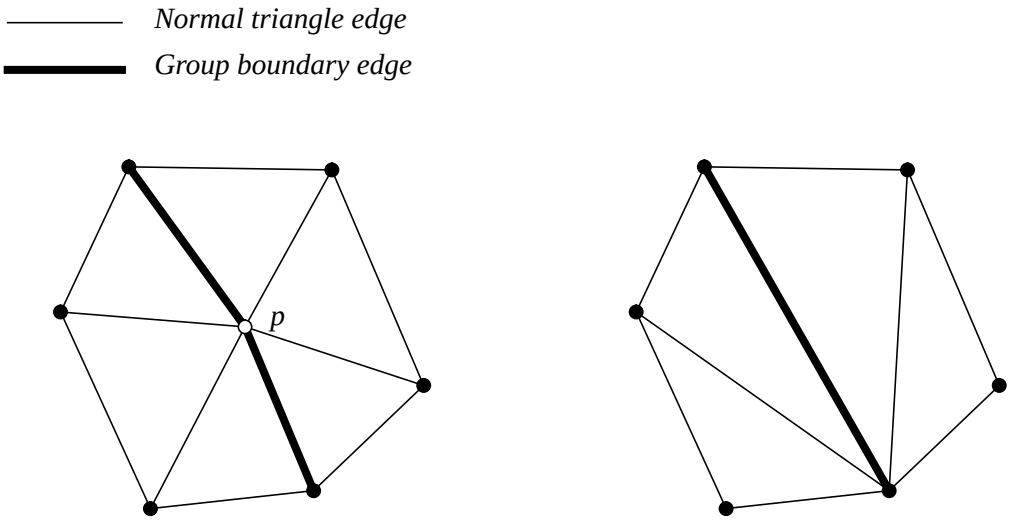
- **D) Inconsistent edges**

  An inconsistent edge is shared by two adjacent triangles that have opposite orientations. One triangle is oriented in counter-clockwise order and the other in clockwise order.



A) **B)** **C)** **D)**

4.2 Preserving grouping information



The group boundary has been preserved while vertex p has been removed

IMCompress preserves grouping information during the polygon reduction process:

- Groups of triangles are identified in the input file.
- Boundaries between adjacent groups are detected.
- A vertex located on a group boundary can only be removed if the local boundary is preserved.

4.3 Interactive polygon reduction

Interactive polygon reduction is implemented within a 3-D rendering window by:

- Letting the user select a 3-D area on a polygonal model
- Letting the user select important vertices that must be preserved
- Invoking a polygon reduction algorithm

Interactive polygon reduction lets the user apply different compression criteria to different parts of a model.

5. Conclusion

5.1 Reduction of detailed polygonal models

We have presented a unique vertex decimation technique providing superior polygon reduction capabilities for detailed polygonal models. The algorithm:

- Is fully automated
- Guarantees 3-D tolerances
- Produces equiangular triangulations
- Naturally preserves important features
- Is fast
- Preserves a mapping between the original and reduced models

5.2 Future developments

One important application which is not addressed by IMCompress is the reduction of typical CAD models generated by solid modeling software. Generating coarse LOD representations of tessellated solid models is a very important issue. In our opinion, an algorithm performing this task should:

- Be based on a volumetric method
- Be entirely automatic
- Detect and preserve edges and corners
- Be immune to topological anomalies

A Hierarchy of Techniques for Simplifying Polygonal Models

Amitabh Varshney
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
varshney@cs.sunysb.edu

1 Introduction

It is evident from these course notes and their references that there is a tremendous interest in the general area of simplification of polygonal objects. There are several ways in which one can classify the various simplification techniques. One of these is by examining the nature of the performed simplifications. This section of the course will overview some representative techniques from a hierarchy of increasingly aggressive simplification techniques. In Section 2 we overview a popular approach to compress the connectivity information for polygonal objects – by using triangle strips. Section 3 overviews *Simplification Envelopes* and their features. In Section 4 we overview some of our research in simplifying the genus of an object. Constructing a multiresolution hierarchy is only part of the solution in a level-of-detail-based rendering scheme. Switching levels of detail among different objects and at different regions within the same object is quite important for view-dependent level-of-detail rendering and is discussed in Section 5. Section 6 concludes this part of the course by overviewing some of our experiences in writing software for simplification of real-life objects.

2 Connectivity Compression: Triangle Strips

The speed of high-performance rendering engines on triangular meshes in computer graphics can be bounded by the rate at which triangulation data is sent into the machine. Obviously, each triangle can be specified by three vertices, but to maximize the use of the available data bandwidth, it is desirable to order the triangles so that consecutive triangles share an edge. Using such an ordering, only the incremental change of one vertex per triangle need be specified, potentially reducing the rendering time by a factor of three by avoiding redundant clipping and transformation computations. Besides, such an approach also has obvious benefits in compression for storing and transmitting models.

Consider the triangulation in Figure 2. Without using triangle strips, we would have to specify the five triangles with three vertices each. By using triangle strips, as supported by the OpenGL

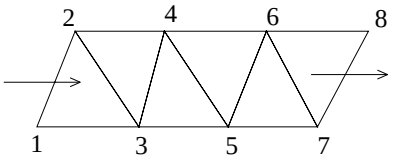


Figure 1: A Triangle Strip

graphics library (Ope93; ONDW93), we can describe the triangulation using the strip $(1, 2, 3, 4, 5, 6, 7, 8)$, and assuming the convention that the ith triangle is described by the ith, $(i + 1)$st, and $(i + 2)$nd vertices of the *sequential* strip. Such a sequential strip can reduce the cost to transmit n triangles from $3n$ to $n + 2$ vertices.

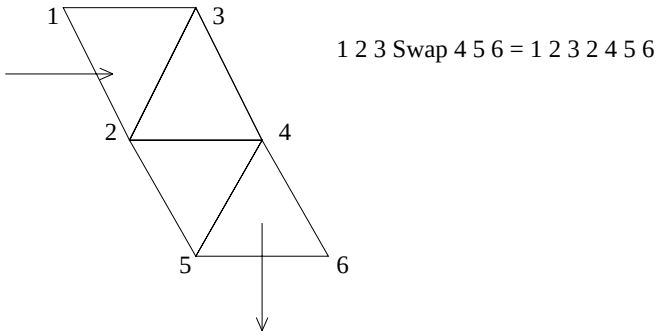


Figure 2: Replacing a swap requires an extra vertex.

To allow greater freedom in the creation of triangle strips, a "swap" command permits one to alter the first-in-first-out queuing discipline in a triangle strip (Sil91). A swap command swaps the order of the two latest vertices in the buffer so that the instead of vertex i replacing the vertex $(i - 2)$ in a buffer of size 2, vertex i replaces the vertex $(i - 1)$. This allows for a single triangle strip representation of the collection of triangles shown in Figure 2, as $(1, 2, 3, SWAP, 4, 5, 6)$. This form of a triangle strip that includes swap commands is referred to as a *generalized triangle strip*.

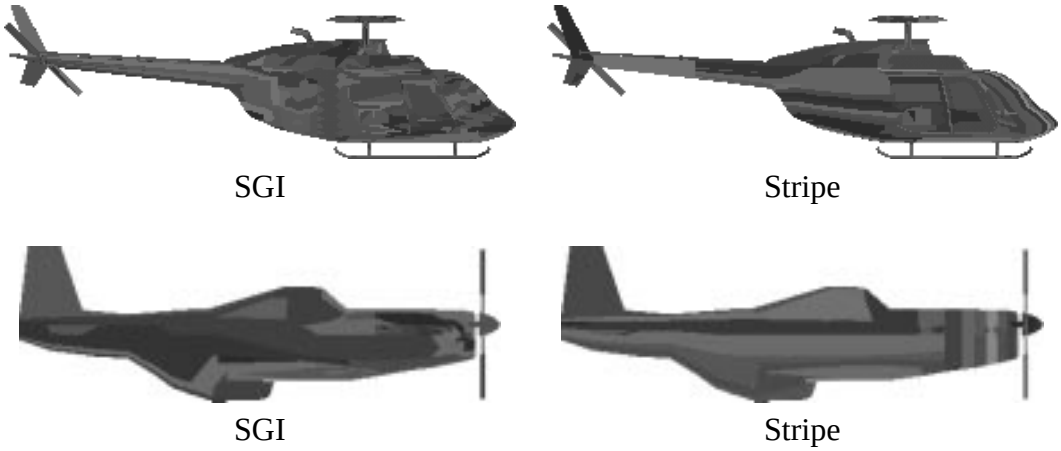


Figure 3: Visual Comparison of Triangle Strips Generated by SGI and Stripe

Although the swap command is supported in the GL graphics library (Sil91), keeping portability considerations in mind it was decided to not support it in OpenGL. With OpenGL gaining rapid acceptance in the graphics software community, the one-bit-per-vertex cost model that was appropriate for a swap command in GL is now outdated. A more appropriate cost for such a swap command under the OpenGL model is a penalty of one vertex as explained next. One can simulate a swap command in OpenGL by re-transmitting the vertex that had to be swapped. This results in an empty triangle two of whose vertices are the same. This is illustrated in Figure 2, where we simulate $(1, 2, 3, SWAP, 4, 5, 6)$ by $(1, 2, 3, 2, 4, 5, 6)$. Note that, even though a swap costs one vertex in the OpenGL model, it is still cheaper than starting a new triangle strip that costs two vertices.

We have considered the problem of constructing good triangle strips from polygonal models. Often such models are not fully triangulated, and contain quadrilaterals and other non-triangular faces, which must be triangulated prior to rendering. The choice of triangulation can significantly impact the cost of the resulting strips. We have recently proved that the problem of triangulating a polygonal model for optimal strips is NP-complete (ESV97). Our work on efficient triangle strips therefore explores several heuristics that we have empirically observed to produce good triangle strips on real-life models (ESV96). Our linear-time algorithm manages to achieve this by exploiting both the local and the global structure of the model. Our analysis of the global structure of a geometric model is done via a non-geometric technique we term *patchification*, which we believe is of general interest as an efficient tool for logically partitioning polygonal models.

The best previous code for constructing triangle strips which we are aware of is (AHB90), implementing what we will call the SGI algorithm. The SGI algorithm seeks to create strips that tend to minimize leaving isolated triangles. It is a greedy algorithm, which always chooses as the next triangle in a strip the triangle that is adjacent to the least number of neighbors.

We have experimented with several variants of local and global algorithms; the details are available in (ESV96). For our local approaches there were ten different options for each data file that we ran our experiments on. Similarly, for our global approaches there were ten different options for each data file that we ran our experiments on. After comparing the results we had from the above-mentioned 20 different approaches on over 200 datasets, we found that the best option was to use the the global row or column strips with a patch cutoff size of 5. In this approach we first partition the model into regions that have collections of $m \times n$ quadrilaterals arranged in m rows and n columns, which we refer to as a *patch*. Each patch whose number of quadrilaterals, mn, is greater than a specified cutoff, in our case 5, is converted into one strip, at a cost of three swaps per turn. Further, every such strip is extended backwards from the starting quadrilateral and forwards from the ending quadrilateral of the patch to the extent possible. For extension, we use an algorithm similar to the SGI algorithm. However, we triangulate our faces "on the fly", which gives us more freedom in producing triangle strips. We have implemented this option in our tool, *Stripe*. This utility is available from our web-site `http://www.cs.sunysb.edu/~evans/stripe.html` and is free for non-commercial use.

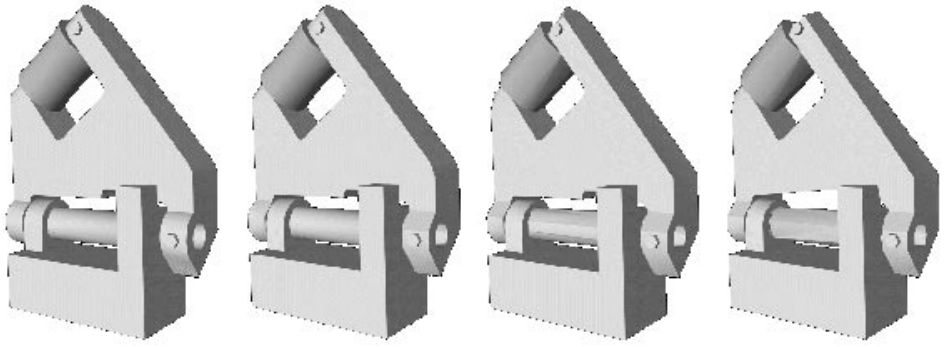
The times for execution of our algorithms behaved linearly with respect the input size. When rendering the models with the triangle strips that were produced by each algorithm, the savings in transmission time to the renderer did prove to be a significant savings in rendering time. The triangle strips produced by our code were up to 27% faster to draw than those produced by the SGI algorithm, and were up to 65% faster to draw than without using triangle strips at all. Figure 3

provides visual comparison of the results obtained by our tool Stripe and those obtained by the earlier algorithm being used by SGI.

3 Simplification Envelopes

In (Var94; CVM+96) we have presented the framework of *simplification envelopes* for computing various levels of detail of a given polygonal model. Simplification envelopes are a generalization of *offset surfaces*. Given a polygonal representation of an object, they allow the generation of minimal approximations that are guaranteed not to deviate from the original by more than a user-specifiable amount while preserving global topology. We surround the original polygonal surface with two envelopes, then perform simplification within this volume. We have demonstrated this technique in conjunction with two algorithms, one global, the other local. The global algorithm computes the solution by generating and considering approximating triangles in a globally exhaustive manner. The local algorithm provides a fast method for generating approximations to large input meshes (at least hundreds of thousands of triangles). A sample application of the algorithms we describe can be seen in Figure 4, where four levels of details of a torpedo roller in a notional submarine are substituted at increasing distances in (a) and are shown side-by-side in (b). The four levels of detail of the torpedo rollers in Figure 4(b) consist of 2346, 1180, 676, and 514 triangles from left to right.

(a) Receding



(b) Upclose

Figure 4: Visual Comparison of Four Levels-of-Detail of a Torpedo Roller

Such an approach has several benefits in computer graphics. First, one can very precisely quantify the amount of approximation that is tolerable under given circumstances. Given a user-specified error in number of pixels of deviation of an object's silhouette, it is possible to choose which level of detail to view from a particular distance to maintain that pixel error bound. Second, this approach allows one a fine control over which regions of an object should be approximated more and which ones less. This could be used for selectively preserving those features of an object that are *perceptually* important. Third, the user-specifiable tolerance for approximation is the only parameter required to obtain the approximations; fine tweaking of several parameters depending upon the object to be approximated is not required. Thus, this approach is quite useful for *automating the process* of genus-preserving simplifications of a large number of objects.

Our simplification envelopes approach guarantees non-self-intersecting approximations and allows the user to do adaptive approximations by simply editing the simplification envelopes (either manually or automatically) in the regions of interest. It allows for a global error tolerance, preservation of the input genus of the object, and preservation of sharp edges. Our approach requires only one user-specifiable parameter, allowing it to work on large collections of objects with no manual intervention if so desired. It is rotationally and translationally invariant, and can elegantly handle holes and bordered surfaces through the use of cylindrical tubes. Simplification envelopes are general enough to permit both simplification algorithms with good theoretical properties such as our global algorithm, as well as fast, practical, and robust implementations like our local algorithm. Additionally, envelopes permit easy generation of correspondences across several levels of detail. For details, the interested reader can refer to (Var94; CVM+96).

4 Genus Simplifications

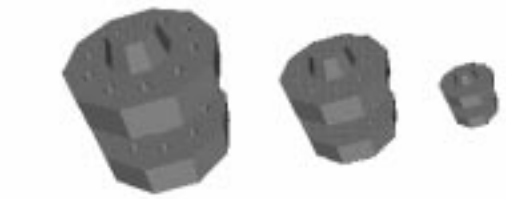
A constraint common to most existing work on automatic generation of multi-resolution object hierarchies has been the genus preservation criterion. Genus is related to the number of holes in an object – a sphere has genus 0, a torus has genus 1, and digit 8 has genus 2. We have developed methods to simplify the genus of an object in a *controlled* fashion. Preservation of topology (or the genus) is crucial for certain applications such as study of tolerances in mechanical CAD. Clearly, if the target application demands topology preservation, then the simplification algorithm should adhere to it. However, if the goal is fast and realistic rendering, such as for virtual reality CAD visualization, the topology preservation criterion could stand in the way of efficient simplification. Let us consider a flythrough in a virtual reality CAD model. A tiny hole on the surface of a mechanical part in this model will gradually disappear as the observer moves away from the part. However, genus preserving simplification of this object will retain such features, thereby reducing frame rates (due to limits on the amount of geometry-simplification that one can achieve while preserving topology) and increasing image-space aliasing (due to undersampling, especially in perspective viewing).

We view the simplification of an object of arbitrary topological type as a two-stage process – simplification of the topology (i.e. reduction of the genus) and simplification of the geometry (i.e. reduction of the number of vertices, edges, and faces). One can mix the execution of these two stages in any desired order. We have observed that genus reductions by small amounts can lead to large overall simplifications. This observation together with the fact that genus-reducing sim-

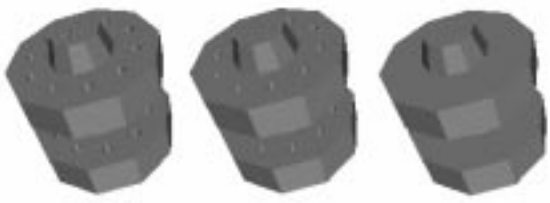
plifications are usually faster than genus-preserving simplifications, makes such an approach quite attractive for generating multiresolution hierarchies. Our research extends the underlying concepts of α-hulls(EM94) to polygonal datasets and allows one to perform genus-reducing simplifications. The primary targets of our research are the interactive three-dimensional graphics and visualization applications where topology can be sacrificed if (a) it does not directly impact the application underlying the visualization and (b) produces no visual artifacts. Both of these goals are easier to achieve if the simplification of the topology is finely *controlled* and has a sound mathematical basis.

We first developed a method to perform genus simplifications in the volumetric domain (HHVW96). We have recently also developed an algorithm that can directly perform genus-reducing simplifications in the polygonal domain. The intuitive idea underlying our approach is to simplify the genus of a polygonal object by rolling a sphere of radius α over it and filling up all the regions that are not accessible to the sphere. This also happens to be the underlying idea behind α-hulls over point-sets. The problem of planning the motion of a sphere amidst polyhedral obstacles in three-dimensions has been very nicely worked out (BK88). We use these ideas in our approach. Let us first assume that our polygonal dataset consists of only triangles; if not, we can triangulate the individual polygons (Sei91; NM95). Planning the motion of a sphere of radius α, say $S(\alpha)$, amongst triangles is equivalent to planning the motion of a point amongst "grown" triangles. Mathematically, a grown triangle $T_i(\alpha)$ is the Minkowski sum of the original triangle t_i with the sphere $S(\alpha)$. Formally, $T_i(\alpha) = t_i \oplus S(\alpha)$, where $\oplus$ denotes the Minkowski sum which is equivalent to the convolution operation. Thus, our problem reduces to efficiently and robustly computing the union of the grown triangles $T_i(\alpha)$. The boundary of this union, $\partial \bigcup_{i=1}^{n} T_i(\alpha)$, where n is the number of triangles in the dataset, will represent the locus of the center of the sphere as it is rolled in contact with one or more triangles and can be used to guide the genus simplification stage.

We have tested our approach on several real-life datasets and have achieved encouraging results. Results of our approach on a couple of polygonal mechanical CAD objects are shown in figures 5 and 6.

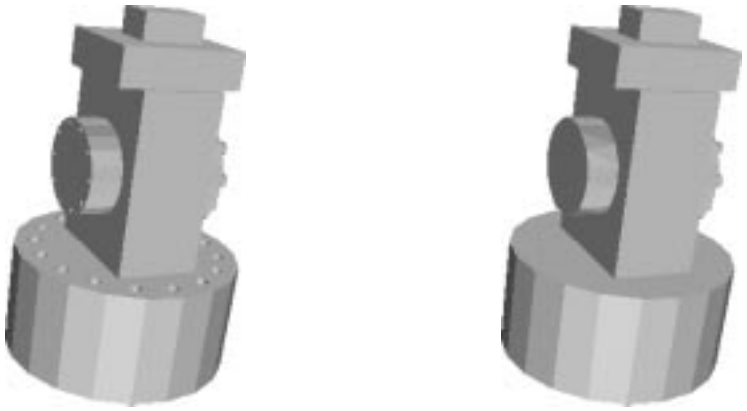


(a)



(b)

Figure 5: Hierarchical simplifications of the genus



(a) (b)

Figure 6: Genus-reducing simplifications for an industrial CAD part

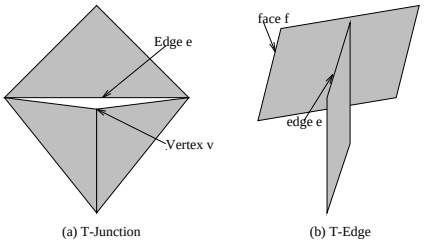


Figure 7: Permissible non-manifold degeneracies

In addition to two-manifold polygonal objects our approach also works in presence of some limited cases of non-manifold polygonal objects including those that have T-junctions and T-edges (shown in Figure 7). Such degeneracies are quite common in mechanical CAD datasets from the manufacturing industry due to numerical inaccuracies in computing surface-surface intersections and in conversion from B-reps to polygonal representations.

5 View-Dependent Level-of-Detail Rendering

Constructing a fixed number of levels of detail, is well-suited for virtual reality walkthroughs and flythroughs of large and complex structures with several thousands of objects. Examples of such environments include architectural buildings, airplane and submarine interiors, and factory layouts. However, for scientific visualization applications where the goal often is to visualize one or two highly detailed objects at close range, most of the previous work is not directly applicable. For instance, consider a biochemist visualizing the surface of a molecule or a physician inspecting the iso-surface of a human head extracted from a volume dataset. It is very likely during such a visualization session, that the object being visualized will not move adequately far away from the viewer to allow the rendering algorithm to switch to a lower level of detail. What is desirable in such a scenario is an algorithm that can allow several different levels of details to co-exist across different regions of the same object. Such a scheme needs to satisfy the following two important criteria:

- It should be possible to select the appropriate levels of detail across different regions of the same object in real time.
- Different levels of detail in different regions across an object should merge seamlessly with one another without introducing any cracks and other discontinuities.

Level-of-detail-based rendering has thus far emphasized object-space simplifications with minimal feedback from the image space. The feedback from the image space has been in the form of very crude heuristics such as the ratio of the screen-space area of the bounding box of the object to the distance of the object from the viewer. As a result, one witnesses coarse image-space artifacts such as the distracting "popping" effect when the object representation changes from one level of detail to the next (Hel95). Attempts such as alpha-blending between the old and the new levels of detail during such transitions serve to minimize the distraction at the cost of rendering two representations. However alpha blending is not the solution to this problem since it does not address the real cause – lack of sufficient image-space feedback to select the appropriate local level of detail in the object space; it merely tries to cover-up the distracting artifacts.

Increasing the feedback from the image space allows one to make better choices regarding the level of detail selection in the object-space. We next outline some of the ways in which image-space feedback can influence the level of detail selection in the object-space.

5.1 Local Illumination

Increasing detail in a direction perpendicular to, and proportional to, the illumination gradient across the surface is a good heuristic (ARB90). This allows one to have more detail in the regions where the illumination changes sharply and therefore one can represent the highlights and the sharp shadows well. Since surface normals play an important role in local illumination one can take advantage of the coherence in the surface normals to build a hierarchy over a continuous resolution model that allows one to capture the local illumination effects well.

5.2 Screen-Space Projections

Decision to keep or collapse an edge should depend upon the length of its screen-space projection instead of its object-space length. At a first glance this might seem very hard to accomplish in real-time since this could mean checking for the projected lengths of all edges at every frame. However, usually there is a significant coherence in the ratio of the image-space length to the object-space length of edges across the surface of an object and from one frame to the next. This makes it possible to take advantage of a hierarchy built upon the the object-space edge lengths for an object. We use an approximation to the screen-space projected edge length that is computed from the object-space edge length.

5.3 Visibility Culling

During interactive display of any model there is usually a significant coherence between the visible regions from one frame to the next. This is especially true of the back-facing polygons that account for almost half the total number of polygons and do not contribute anything to the visual realism.

A hierarchy over a continuous resolution representation of an object allows one to significantly simplify the invisible regions of an object, especially the back-facing ones. This view-dependent visibility culling can be implemented in a straightforward manner using the hierarchy on vertex normals.

5.4 Silhouette boundaries

Silhouettes play a very important role in perception of detail. Screen-space projected lengths of silhouette edges (i.e., edges for which one of the adjacent triangles is visible and the other is invisible), can be used to very precisely quantify the amount of smoothness of the silhouette boundaries. A hierarchy built upon a continuous-resolution representation of a object allows one to do this efficiently.

In contrast to the traditional approaches of precomputing a fixed number of level-of-detail representations for a given object we have developed an approach that involves statically generating a continuous level-of-detail representation for the object (XESV97; XV96). This representation is then used at run-time to guide the selection of appropriate triangles for display. The list of displayed triangles is updated incrementally from one frame to the next. Our scheme can construct seamless and adaptive level-of-detail representations on-the-fly for polygonal objects. Since these representations are view-dependent, they take advantage of view-dependent illumination, visibility, and frame-to-frame coherence to maximize visual realism and minimize the time taken to construct and draw such objects. Our approach shows how one can adaptively define such levels of detail based on (a) scalar attributes such as distance from the viewpoint and (b) vector attributes such as the direction of vertex normals. For the details of this approach, the interested reader is referred to (XESV97).

6 Conclusions

We have overviewed several representative approaches for building and using different kinds of simplification schemes for polygonal models. These can be thought of as belonging to a hierarchy of progressively aggressive simplification techniques – from the lossless compression of connectivity as provided by triangle strips, to genus-preserving simplifications from Simplification Envelopes, to genus-reducing simplifications. An orthogonal component of multiresolution hierarchy generation is the selection of various levels of detail. We have outlined how view-dependent selection of detail incorporates various image-space and object-space criteria to achieve a good balance between speed and visual realism.

There are various problems in using the multiresolution techniques outlined in this course and elsewhere. These have not received much attention in the simplification research community thus far. Some of these have been outlined below:

Data Degeneracies: Almost all datasets that we have worked on have had various manifestations of geometric degeneracies, such as T-junctions, cracks, T-edges, and coincident polygons. Such dataset deformities have their sources in both human as well as computer-generated errors. Almost all multiresolution techniques assume that such degeneracies do not exist in their input. Bridging this gap should be a very fruitful area for further research.

Numbers of Objects: Large numbers of (relatively) low complexity objects versus small numbers of high complexity objects are best rendered using different techniques for simplification as well as level-of-detail-based renderings. For instance, if an approach requires even one user-specified constant per object, it becomes difficult to automatically generate multiresolution hierarchies for say, a hundred thousand objects, each with ten thousand polygons. Most of the current multiresolution techniques are optimized for small numbers of high representation complexity objects.

Geometric Debugging: Current tools that can help debug geometric software are woefully inadequate for the task. It will be useful to build tools that actively assist in geometric debugging.

Acknowledgements

Several people have influenced this research over the last eight years through their insightful comments, critiques, enthusiasm, and hard work. These include Fred Brooks, Pankaj Agarwal, John Airey, Jon Cohen, Bruno Costa, Lucia Darsa, Jihad El-Sana, Francine Evans, Taosong He, Lichan Hong, Arie Kaufman, Dinesh Manocha, Jai Menon, Joe Mitchell, Steve Skiena, Greg Turk, Sidney Wang, Hans Weber, Bill Wright, and Julie Xia. The objects that appear in Figures 4 and 6 are parts of the dataset of a notional submarine provided to us by the Electric Boat Division of General Dynamics. The objects in Figure 3 have been provided to us by Viewpoint DataLabs.

References

K. Akeley, P. Haeberli, and D. Burns. tomesh.c : C Program on SGI Developer's Toolbox CD, 1990.

J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, No. 2, pages 41–50, March 1990.

C. Bajaj and M.-S. Kim. Generation of configuration space obstacles: the case of a moving sphere. *IEEE Journal of Robotics and Automation*, 4, No. 1:94–99, February 1988.

J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. V. Wright. Simplification envelopes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 119 – 128. ACM SIGGRAPH, ACM Press, August 1996.

H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13(1):43–72, January 1994.

F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *IEEE Visualization '96 Proceedings*, pages 319 – 326. ACM/SIGGRAPH Press, October 1996.

F. Evans, S. Skiena, and A. Varshney. Efficient triangle strips for fast rendering. *ACM Transactions on Graphics*, 1997. (submitted).

J. Helman. Graphics techniques for walkthrough applications. In *Interactive Walkthrough of Large Geometric Databases, Course Notes 32, SIGGRAPH '95*, pages B1–B25, 1995.

T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.

A. Narkhede and D. Manocha. Fast polygon triangulation based on seidel's algorithm. *Graphics Gems 5*, pages 394–397, 1995.

Open GL Architecture Review Board, J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, Reading, MA, 1993.

Open GL Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley Publishing Company, Reading, MA, 1993.

R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1:51–64, 1991.

Silicon Graphics, Inc. Graphics Library Programming Guide, 1991.

A. Varshney. Hierarchical geometric approximations. Ph.D. Thesis TR-050-1994, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994.

J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, June 1997. (to appear).

J. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96 Proceedings*, pages 327 – 334. ACM/SIGGRAPH Press, October 1996.

Optimizing Triangle Strips for Fast Rendering

Francine Evans Steven Skiena Amitabh Varshney

State University of New York at Stony Brook

Abstract

Almost all scientific visualization involving surfaces is currently done via triangles. The speed at which such triangulated surfaces can be displayed is crucial to interactive visualization and is bounded by the rate at which triangulated data can be sent to the graphics subsystem for rendering. Partitioning polygonal models into triangle strips can significantly reduce rendering times over transmitting each triangle individually.

In this paper, we present new and efficient algorithms for constructing triangle strips from partially triangulated models, and experimental results showing these strips are on average 15% better than those from previous codes. Further, we study the impact of larger buffer sizes and various queuing disciplines on the effectiveness of triangle strips.

1 Introduction

Interactive display rates are crucial to exploratory scientific visualization and virtual reality. The speed of high-performance rendering engines on triangular meshes in computer graphics can be bounded by the rate at which triangulation data is sent into the machine. Obviously, each triangle can be specified by three vertices, but to maximize the use of the available data bandwidth, it is desirable to order the triangles so that consecutive triangles share an edge. Using such an ordering, only the incremental change of one vertex per triangle need be specified, potentially reducing the rendering time by a factor of three by avoiding redundant lighting and transformation computations. Besides, such an approach also has obvious benefits in compression for storing and transmitting models.

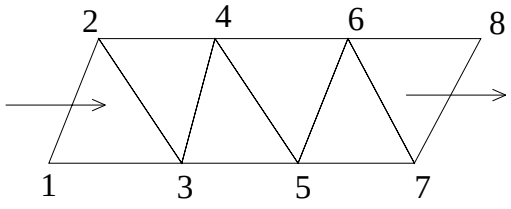


Figure 1: A Triangle Strip

Consider the triangulation in Figure 1. Without using triangle strips, we would have to specify the six triangles with three vertices each. By using triangle strips, as supported by the OpenGL graphics library [11, 12], we can describe the triangulation using the strip $(1, 2, 3, 4, 5, 6, 7, 8)$, and assuming the convention that the ith triangle is described by the ith, $(i + 1)$st, and $(i + 2)$nd vertices of the *sequential* strip. Such a sequential strip can reduce the cost to transmit n triangles from $3n$ to $n + 2$ vertices.

In this paper, we consider the problem of constructing good triangle strips from polygonal models. Often such models are not fully triangulated, and contain quadrilaterals and other non-triangular faces, which must be triangulated prior to rendering. The choice of triangulation can significantly impact the cost of the resulting strips. For example, Figure 2 demonstrates that one triangle strip suffices to represent a cube, provided it is triangulated in a particular manner. Although we have shown that the problem of triangulating a polygonal model for optimal strips is NP-complete [7], here we provide heuristics which exploit the freedom to triangulate these faces to produce strips that are on average 15% better than those of previous codes. Our linear-time algorithm manages to achieve this by exploiting both the local and the global structure of the model. Our analysis of the global structure of a geometric model is done via a non-geometric technique we term *patchification*, which we believe is of general interest as an efficient tool for logically partitioning polygonal models.

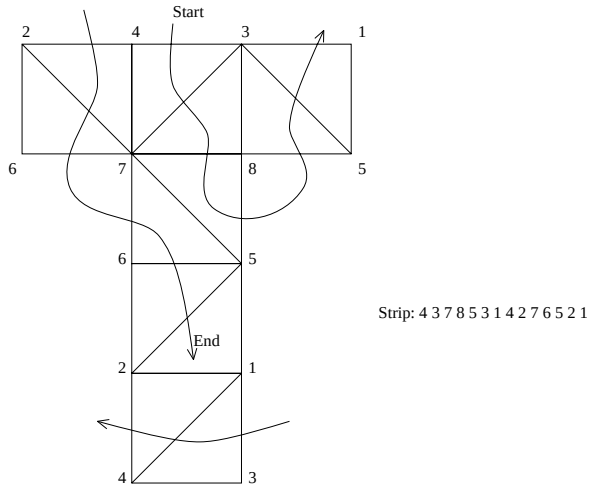


Figure 2: Triangulating a cube for one sequential strip.

To allow greater freedom in the creation of triangle strips, a "swap" command permits one to alter the FIFO (first-in, first-out) queuing discipline in a triangle strip [13]. A swap command swaps the order of the two latest vertices in the buffer so that instead of vertex i replacing the vertex $(i - 2)$ in a buffer of size 2, vertex i replaces the vertex $(i - 1)$. This allows for a single triangle strip representation of the collection of triangles shown in Figure 3, as $(1, 2, 3, SWAP, 4, 5, 6)$. This form of a triangle strip that includes swap commands is referred to as a *generalized triangle strip*.

The swap command gives greater freedom in the creation of triangle strips at the cost of one bit per vertex. Although the swap command is supported in the GL graphics library [13], keeping portability considerations in mind it was decided to not support it in OpenGL [8]. With OpenGL gaining rapid acceptance in the graphics software community, the one-bit-per-vertex cost model that was appropriate for a swap command in GL is now outdated. A more appropriate cost for such a swap command under the OpenGL model is a penalty of one vertex as explained next. One can simulate a swap command in OpenGL by re-transmitting the vertex that had

to be swapped. This results in an empty triangle two of whose vertices are the same. This is illustrated in Figure 3, where we simulate $(1, 2, 3, SWAP, 4, 5, 6)$ by $(1, 2, 3, 2, 4, 5, 6)$. Note that, even though a swap costs one vertex in the OpenGL model, it is still cheaper than starting a new triangle strip that costs two vertices. In this paper, we evaluate all algorithms for both the GL and OpenGL cost models.

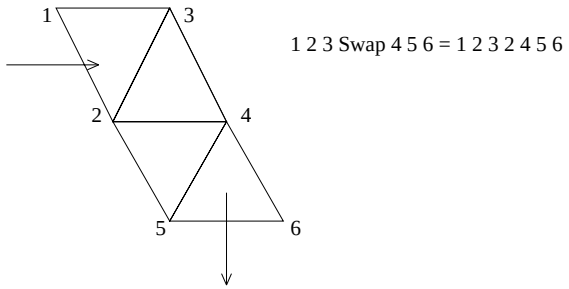


Figure 3: Replacing a swap requires an extra vertex.

Special-purpose rendering hardware is needed to fully exploit the advantages of triangle strips, by maintaining a buffer with the k previously transmitted vertices as determined by a certain queuing discipline. Although current rendering engines use a buffer of size of $k = 2$ and FIFO queuing discipline, there has been recent interest in studying the impact of larger buffer sizes, for both rendering [3] and geometric compression [6]. The decomposition of a triangular mesh into a triangle strip data structure that back-references the previous k vertices, $k \geq 2$ is referred to as a *generalized triangle mesh* [6]. Towards this end, we provide extensive analysis of the impact of buffer size and queuing discipline on triangle strip performance. We demonstrate that relatively small buffer sizes are sufficient to achieve most of the potential benefits of triangle strips, making for a desirable tradeoff between increasing hardware cost versus the speedup in rendering time.

In Section 2, we summarize previous work on triangular strips. In Section 3, we describe our local and global algorithms for constructing quality triangle strips from polygonal meshes. Experimental results are presented in Section 4. In Section 5, we study the impact of buffer size on triangle strip performance. Conclusions and plans for future work are discussed in Section 6.

2 Previous Work

The problem of constructing quality triangle strips has received attention from both the graphics and the computational geometry communities.

Akeley, Haeberli, and Burns have written a program that converts triangle meshes to triangle strips [1]. We discuss the approach in this program in greater details in Section 3. Deering has proposed the use of generalized triangle meshes for compressing connectivity information in geometric polygonal models [6]. He has proposed maintaining a stack of size $k = 16$ to store 16 previous vertices. A vertex for a new triangle is specified either through back-referencing one of the existing vertices on the stack, or by reading-in a new vertex and replacing an existing vertex on the stack. Although a novel idea, no algorithms have been proposed there to suggest how one can decompose polygonal models into generalized triangle meshes for a given buffer size k. An interesting alternative to compressing connectivity information is presented by Hoppe in [9] where vertex-split/edge-collapse information is encoded efficiently with respect to its neighbors. Although not as efficient as generalized triangle meshes for a single resolution model, this approach has the advantage of being able to encode multiresolution models compactly.

Within computational geometry, interest has focused on constructing and recognizing Hamiltonian and sequential triangulations. A triangulation is *Hamiltonian* if its dual graph contains a Hamiltonian cycle. Hamiltonian triangulations can be represented by using generalized triangle strips (triangle strips with swaps). Arkin, et.al. [2] proved that every point set has a Hamiltonian triangulation. Further, they showed that the problem of testing whether a triangulation is Hamiltonian is NP-complete. They gave an $O(n^2)$ algorithm for constructing a Hamiltonian triangulation of a polygon that has since been improved to $O(n \lg n)$ by Narasimhan [10].

A triangulation is *sequential* if its dual graph contains a Hamiltonian cycle whose turns alternate left-right. Sequential triangulations can be represented by using one triangle strip without any swaps. A Hamiltonian triangulation is sequential if three consecutive edges do not share a common vertex. Arkin, et.al. [2] proved that for any $n \geq 9$ there exists a set of n points in general position that do not admit a sequential triangulation. Although linear time suffices to test whether a triangulation is sequential, we [7] have shown that problem of finding a sequential triangulation of a partially triangulated surface is NP-complete using a reduction from 3-satisfiability. Hence, heuristics such as those described in this paper are required to find good sequential strips.

A simple path in the dual of a triangulation identifies a sequence of triangles that form a "strip" or a (triangular) "ribbon" . Bhattacharya and Rosenfeld [4] have studied geometric and topological properties of ribbons. The Hamiltonian triangulation problem can be considered that of identifying if a set of points or a polygon has a triangulation that consists of a single strip (triangular ribbon).

Bose and Toussaint [5] have recently studied a set of problems involving *quadrangulation* of point sets, and have obtained several interesting results. A quadrangulation of a point set S is a decomposition of the convex hull into quadrilaterals, such that each point of S is a vertex of some quadrilateral. In particular, they have applied the notion of Hamiltonian triangulations to this problem, and they have obtained an alternate method of computing Hamiltonian path triangulations.

By Euler's theorem on graphs, the number of triangles in a triangulation is at most twice the number of vertices, and on average we will have to send each vertex twice to the renderer using sequential triangle strips and a buffer of size 2. Bar-Yehuda and Gotsman [3] studied the extent to which we can increase the stack (buffer) size to reduce this duplication of vertices. This yields a time-versus-space tradeoff; for as we increase memory usage, rendering time will decrease. Bar-Yehuda and Gotsman have shown that a buffer of size $13.35\sqrt{n}$ is sufficient to render any mesh on n vertices in the optimal time n, and that a buffer size of $1.649\sqrt{n}$ is necessary for optimal rendering in the worst-case. They show the problem of minimizing the buffer-size for a given mesh is NP-hard, using a reduction from the problem of finding minimum separators of a planar graph.

3 Constructing Triangle Strips

In this section, we propose several heuristics for constructing triangle strips from polygonal models. There are at least three different objectives such heuristics might reasonably seek to achieve:

- *Maximize the length of each strip* – since each strip of length s represents $s - 2$ triangles, maximizing strip length minimizes this overhead.

- *Minimizing swaps* – since each swap costs one additional vertex in the OpenGL cost model.

- *Minimizing the number of singleton strips* – since each triangle left isolated after removing a strip creates a singleton strip, we should seek to begin and end our strips on low-degree faces of the triangulation.

The best previous code for constructing triangle strips which we are aware of is [1], implementing what we will call the SGI algorithm. The SGI algorithm seeks to create strips that tend to minimize leaving isolated triangles. It is a greedy algorithm, which always chooses as the next triangle in a strip the triangle that is adjacent to the least number of neighbors (i.e. minimizes the number of *adjacencies*). When there is more than one triangle with the same, least number of neighbors, the algorithm looks one level ahead to its neighbors' neighbors, and chooses the direction of minimum degree, choosing arbitrarily if there is again a tie. After starting from an arbitrary lowest degree triangle, it extends its strips in both directions, so that each strip is as long as possible. There is no reluctance to generate swaps, and understandably so, since this algorithm was aimed at generating triangle strips for Iris GL. A fast, linear-time implementation is obtained by using hash tables to store the adjacency information, linked to a priority queue maintaining strip length to choose which triangle starts a new strip.

Figure 4 illustrates how the algorithm breaks ties. Starting with a face of lowest adjacency (of degree 1 on the upper center of the figure), the algorithm always selects the lower degree face as the next triangle in the strip to peel off the marked strip. At the face of degree 3 it turns left because a neighbor to the left adjacent face is of degree 1 as opposed to 2.

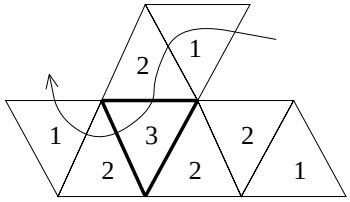


Figure 4: Adjacency counts in the SGI algorithm

The SGI algorithm uses strictly local adjacency information in constructing the triangle strips. However, fully exploiting the freedom to triangulate quads seems to require a more global approach. We have experimented with several variants of local and global algorithms, as discussed in the following two sections.

3.1 Local Algorithms

Our class of local heuristics starts from the same basic idea as the SGI algorithm – to use least adjacencies as the basis for choosing the next face in a strip. However, we have tried to improve upon their algorithm by dynamic triangulation and alternate tie-breaking procedures.

We have considered three different approaches to triangulating faces:

- *Static triangulation* – In this approach, we triangulate all quads and larger faces in our model as a pre-processing step before we begin finding strips. We use alternate left-right turns, as shown in Figure 5(b) because such a triangulation is inherently sequential, as opposed to the simpler and more conventional fan triangulation. The SGI algorithm accepts only triangulated models as input. Therefore, to compare their approach with ours we pre-triangulate all non-triangulated models using this static triangulation approach and then run their algorithm.

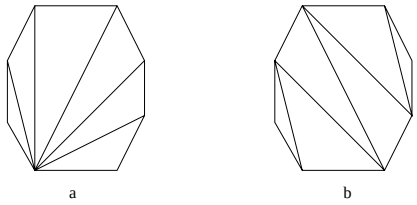


Figure 5: Fan vs sequential triangulation of a polygonal face.

- *Dynamic whole-face triangulation* – A second approach completely triangulates each face when we first enter it via some edge on a strip. After using one of the tie-breaking procedures described below to determine the exit edge e, we can triangulate the face as sequentially as possible while exiting at e. If the surface normals do not vary across a face, then whole face triangulation has the additional advantage of encoding fewer normal transitions.
- *Dynamic partial-face triangulation* – Partial-face triangulation provides the freedom to triangulate and walk only part of a face before exiting it. This approach can under certain conditions provably perform better than the whole-face triangulation, as is seen in the example where we represent a cube using a single sequential triangle strip. After identifying the exit edge e of the face with the minimum number of adjacencies, we sequentially triangulate the smallest portion possible of the face from the input edge to exit at e. This is illustrated in Figure 6.

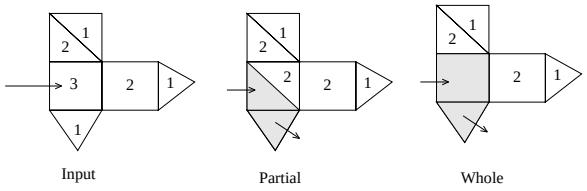


Figure 6: Examples of partial and whole-face triangulation.

We have considered several different approaches in breaking ties when there is more than one polygon that has the least number of adjacencies to the current face. Such ties often occur since the possible number of adjacencies ranges only over 1, 2, and 3. In particular, we tried:

- *Arbitrary* – meaning that we use the first face found among the low-adjacency faces.
- *Look-ahead* – this is the same approach that SGI algorithm takes, as described above.
- *Alternate* – this rule tries to alternate directions in choosing the next polygonal face. To motivate this option, note that sequential strips alternate directions.
- *Random* – chooses the next face randomly from those that were tied.
- *Sequential* – chooses the next face that will not produce a swap, and picks randomly if there is no such face.

To quickly identify the lowest adjacency face to start from, we maintain a priority queue ordered by the number of adjacent polygons to each face. The faces in the priority queue are linked to the adjacency list data structure representing the dual graph of the triangulation. This enables fast lookup to find and delete faces when forming the triangle strips.

3.2 Global Algorithms

Although the problem of finding the strip-minimal triangulation is NP-complete, we perform a global analysis of the structure of a polygonal model using a technique we call *patchification*, which we believe is of independent interest.

In typical polyhedral models, there are many quadrilateral faces, often arranged in large connected regions. We attempt to find large "patches", rectangular regions consisting only of quadrilaterals, as illustrated in Figure 7. Figure 8 shows the largest patches in a typical model. These patches can be triangulated sequentially along each row or column, although there is a cost of either 3 swaps per turn or 2 vertices to stop and restart each strip at the end of a row or column.

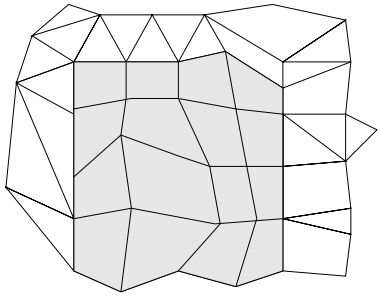


Figure 7: A rectangular patch of quadrilaterals.

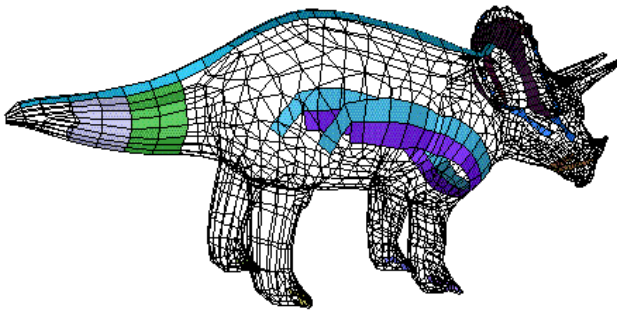


Figure 8: The six largest patches in a triceratops model.

Efficient patchification requires computing the number of polygons to the east, west, north, and south of each face, and making sure that when forming the patches, the polygons in the patch are all adjacent. Hence, we have to "walk" through the faces and calculate the number of adjacent polygons to them in each orientation. Each "walk" only visits each face exactly 2 times: once for the north-south direction and once for the east-west direction; once we visit a face in a walk, that face does not require visiting again. To avoid generating too many small patches, we keep a *patch cutoff size* which is the area of the smallest patch we would like to generate. Since we generate patches in decreasing order of size, we can conveniently stop the process once the areas of the patches being generated falls below this cutoff size. This approach takes us time $O(pn)$ where p is the number of patches found. In our studies p was much smaller than n and therefore this approach demonstrated a linear behavior.

We tried two different approaches for exploiting the coherence identified in large patches:

- *Row or column strips* – After selecting all patches whose size was greater than a specified cutoff size, we partitioned the patches into sequential strips along rows or columns (whichever direction yielded larger strips) and deleted them from the model. Next, a local algorithm (using whole-face triangulation) was used on the remaining model. By generating one strip along each row or column, we minimize the number of swaps needed.

- *Full-patch strips* – Each patch larger than the cutoff size was converted into one strip, at a cost of 3 swaps per turn. Further, every such strip was extended backwards from the starting quadrilateral and forwards from the ending quadrilateral of the patch to the extent possible. As before, the local algorithm was used on the model left after removing the patches and their forward and backward extensions.

4 Experimental Results

We have exhaustively tested our local and global algorithms on several datsets and compared them with the best known triangle strip code [1]. For our local approaches there were ten different options for each data file that we ran our experiments on: (a) whole-face triangulation and (b) partial-face triangulation, for each of the five tie breaking methods – (i) arbitrary, (ii) look-ahead, (iii) alternate, (iv) random, and (v) sequential. For our global approaches there were ten different options for each data file that we ran our experiments on: (a) row/column strips and (b) full-patch strips, for each of five different patch cutoff sizes of – 5, 10, 15, 20, and 25.

Table 1 shows the results of comparison of our best option, which was the global row/column strips with a patch cutoff size of 5, against the SGI algorithm. The cost columns show the total number of vertices required to represent the dataset in a generalized triangle strip representation under the OpenGL cost model (we are counting each vertex and swap that needs to be sent to the renderer).

Data File	Num Verts	Num Tris	Cost SGI	Cost Ours	Savings
plane	1508	2992	4005	3509	12%
skyscraper	2022	3692	5621	4616	18%
triceratops	2832	5660	8267	6911	16%
power lines	4091	8966	12147	10621	13%
porsche	5247	10425	14227	12367	11%
honda	7106	13594	16599	15075	9%
bell ranger	7105	14168	19941	16456	17%
dodge	8477	16646	20561	18515	10%
general	11361	22262	31652	27702	12%

Table 1: Comparison of triangle strip algorithms.

Figure 9 shows the performance comparisons between our best local and best global algorithms against the SGI algorithm for (a) GL and (b) OpenGL cost models. The models sorted by number of triangles are along the x-axis and the cost of generalized triangle strip representation is along the y-axis in this figure.

Observations include:

- Little if any savings seems possible by sophisticated algorithms under the GL model. However, under the more realistic model the combined local/global algorithm can save up to about 20% over the SGI algorithm.
- Our results are close to the theoretical lower bound of the number of triangles + (the number of connected components in the model * 2), so there is limited potential for better algorithms.
- Although the number of swaps required is sensitive to the composition of the model, the total cost grows linear in the size of the model.

Our times for execution of these algorithms behaved linearly with respect to the input size. The timings for our local algorithms were about a factor of two slower than those generated by SGI. Thus, for example, dynamic partial-face method with sequential triangulation took around 8 seconds on the $22K$ triangle model `general` whereas the SGI code took around 4 seconds.

For local algorithms under the GL cost model whole-face triangulations worked better than those with partial-face triangulations; under the OpenGL cost model the reverse was true. Partial-face triangulations produce less swaps than whole-face triangulations because the former have a greater choice in selecting the next face in a strip, and are therefore more likely to be able to select faces that do not require a swap. For global algorithms, full-patch strips with cutoff size of 25 have the best performance under the GL cost model whereas row/column strips with a cutoff size of 5 have the best performance under the OpenGL cost model. This is because a cutoff size of 5 generates more patches than a cutoff size of 25 and more patches means lesser number of swaps.

5 Impact of Buffer Size

The benefits realized by using triangle strips could be further enhanced by special-purpose hardware that has additional buffer space (beyond the usual storage for two vertices) and alternate queuing disciplines. In this section, we study the impact of such resources on performance, to provide guidance for future hardware design.

Increasing the buffer size from a capacity of two vertices naturally decreases the cost of transmission, since we can now specify which of the previous k vertices in the buffer defines the next triangle. The cost of specification becomes $\lceil \lg k \rceil$ bits, instead of number of bits representing one vertex, thus enabling us to potentially represent polygonal models at a cost of less than one vertex per triangle. In our paper, we will ignore the costs of these index bits, since we only seek to determine an upper bound potential improvement in rendering time to assess whether it might be worth the increase in hardware costs.

We considered two different queuing disciplines for maintaining the buffer:

- *First-in, first-out (FIFO)* – This implies that there is no rearrangement of the vertices in the buffer, excluding swaps. FIFO is easiest to implement in hardware, and would thus be preferable if performance is comparable.
- *Least recently used (LRU)* – LRU dynamically rearranges the vertices in the buffer, by placing a vertex that was used most recently into the spot in the buffer that holds the most recently admitted vertex. The least recently used vertex is eliminated when a new vertex is added to the queue. LRU provides the benefit that popular vertices are held in the buffer in the hope that they will likely be used in the near future.

The results of running our tests on several datasets using the whole-face local triangulation method with buffer size of $k \geq 2$ are presented in Figure 10. A larger buffer size implies that we are reusing more of the vertices that were previously transmitted. These figures show the cost of the LRU and FIFO queuing disciplines versus the dataset sizes. As can be seen the advantages to be gained from larger buffer sizes diminish rapidly beyond a buffer size of about 8. For buffer sizes less than 8, LRU performs better than the FIFO scheme by a factor of about 10%.

6 Conclusions and Future Work

We have explored a total of twenty different local and global algorithms in our quest for an effective triangle strip generation algorithm that can perform well under the prevalent OpenGL cost model. Our conclusion is that the best approach for the OpenGL cost model is global row/column strips with a patch cutoff size of 5.

As can be seen from the results of Table 1, we are able to outperform the SGI algorithm significantly. We typically produce a significantly lower number of strips than they do (usually 60%-80% less strips using the local whole-triangulation algorithm), resulting in an average cost savings of about 15% less than SGI algorithm under the OpenGL model. Further, our cost averages just 10% more than the theoretical minimum of using one sequential strip with no swaps, when using the global full-patch strips algorithm with a patch cutoff size of 5, as shown in Figure 9.

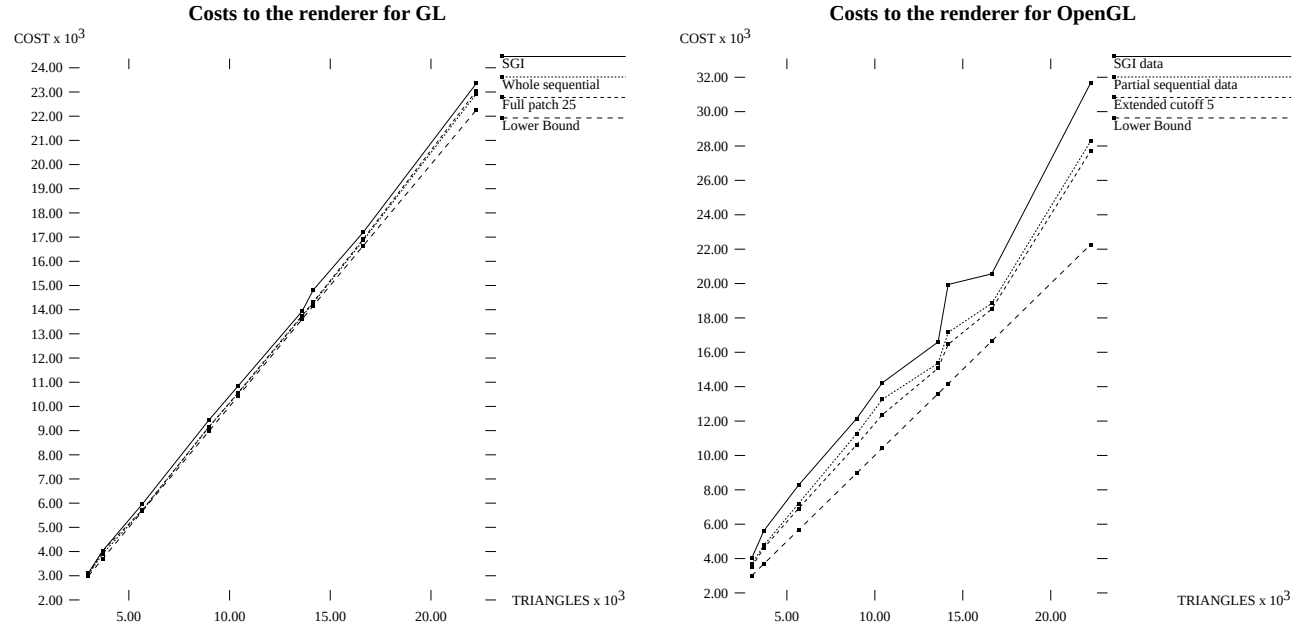
We have found that using global algorithms for detecting large strips of quads proves very effective for reducing swaps. This has proved to be quite useful for generating efficient triangle strips for the OpenGL cost model where every swap costs one vertex.

All our algorithms run in linear time. Although the SGI algorithm does have a slightly better running time, we do not believe this to be a serious drawback of our approach since the triangle-strip generation phase is typically done off-line before interactive visualization.

The results of our experiments with larger buffer sizes offer only limited room for optimism. As we increase the buffer-size the savings do increase, however the improvements diminish very quickly. LRU seems to work much better than FIFO in the smaller buffers, although this must be contrasted with the time and hardware needed to maintain a LRU buffer. The theoretical minimum of using larger buffers is the number of vertices in the model, since each vertex would only have to be transmitted exactly one time, and then could remain in the buffer forever to be used again, provided the buffer is large enough. However, in our implementation we had been assuming that the buffer gets flushed between renderings of different generalized triangle meshes, i.e. a generalized triangle mesh cannot take advantage of the buffer references left behind by a previous mesh. Even if we do not make this assumption, achieving close to the minimum requires a prohibitively large buffer, which is not feasible for hardware implementation. Further, as the result of Bar-Yehuda and Gotsman [3] shows, to achieve this minimum for a mesh of size n a buffer of size $1.649\sqrt{n}$ is necessary, thus making the size of the buffer depend on the size of the input mesh. All of these factors combined with our results seem to make a choice of a small buffer size, say around 8, attractive.

Future work includes:

- Investigate other ways to globally analyze a model prior to finding triangle strips. Currently we only find patches consisting of quadrilaterals, however we can also seek large sequential patches of other polygons, such as triangles. We can experiment with running other local options on the remaining model, although we predict that there will only be slight differences.



(a) GL Cost Model (b) OpenGL Cost Model

Figure 9: GL and OpenGL cost model comparisons.

- Creating and distributing a robust and efficient utility for creating strips for polygonal models, based on the algorithms described in this paper.
- Perform a careful study of algorithms for constructing triangle strips from fully triangulated models, since this work exploits freedom which is not present in this common situation.
- Our current cost function has been motivated by systems that are bandwidth limited or perform all transformations sequentially. However, on many multi-processor graphics systems the triangles/second curve levels-off as the system approaches its parallel processing and cache memory limits. For such a system, if most of the peak performance is achieved by strips of length, say 16 triangles, then rendering two strips of lengths 30 and 2 will be slower than rendering two strips of lengths 16 each. Our current cost model does not account for this and we plan to explore this further.

Acknowledgements

We would like to acknowledge several valuable discussions we have had on triangle strips with Joe Mitchell, Martin Held, Estie Arkin, Jarek Rossignac, Josh Mittleman, and Jim Helman. We would also like to thank the anonymous referees for their helpful comments. The datasets that we have used have been provided by Viewpoint DataLabs. Francine Evans is supported in part by a NSF Graduate Fellowship and a Northrop Grumman Fellowship. Steven Skiena is supported by ONR award 400x116yip01. Amitabh Varshney is supported in part by NSF Career Award CCR-9502239.

References

[1] K. Akeley, P. Haeberli, and D. Burns. tomesh.c : C Program on SGI Developer's Toolbox CD, 1990.

[2] E. Arkin, M. Held, J. Mitchell, and S. Skiena. Hamiltonian triangulations for fast rendering. In *Second Annual European Symposium on Algorithms*, volume 855, pages 36–47. Springer-Verlag Lecture Notes in Computer Science, 1994.

[3] R. Bar-Yehuda and C. Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 1996. (to appear).

[4] P. Bhattacharya and A. Rosenfeld. Polygonal ribbons in two and three dimensions. Technical report, Department of Computer Science, University of Maryland, 1994.

[5] J. Bose and G. Toussaint. No quadrangulation is extremely odd. Technical Report 95-03, Department of Computer Science, University of British Columbia, 1995.

[6] M. Deering. Geometry compression. *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH*, pages 13–20, 1995.

[7] F. Evans, S. Skiena, and A. Varshney. Completing sequential triangulations is hard. Technical report, Dept of Computer Science, State University of New York at Stony Brook, NY 11794-4400, 1996.

[8] J. Helman. Personal Communication.

[9] H. Hoppe. Progressive meshes. *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH*, 1996. (to appear).

[10] G. Narasimhan. On hamiltonian triangulations in simple polygons. In *Proceedings of the Fifth MSI-Stony Brook Workshop on Computational Geometry*, page 15, October 1995.

[11] Open GL Architecture Review Board. *OpenGL Reference Manual*. Addison-Wesley Publishing Company, Reading, MA, 1993.

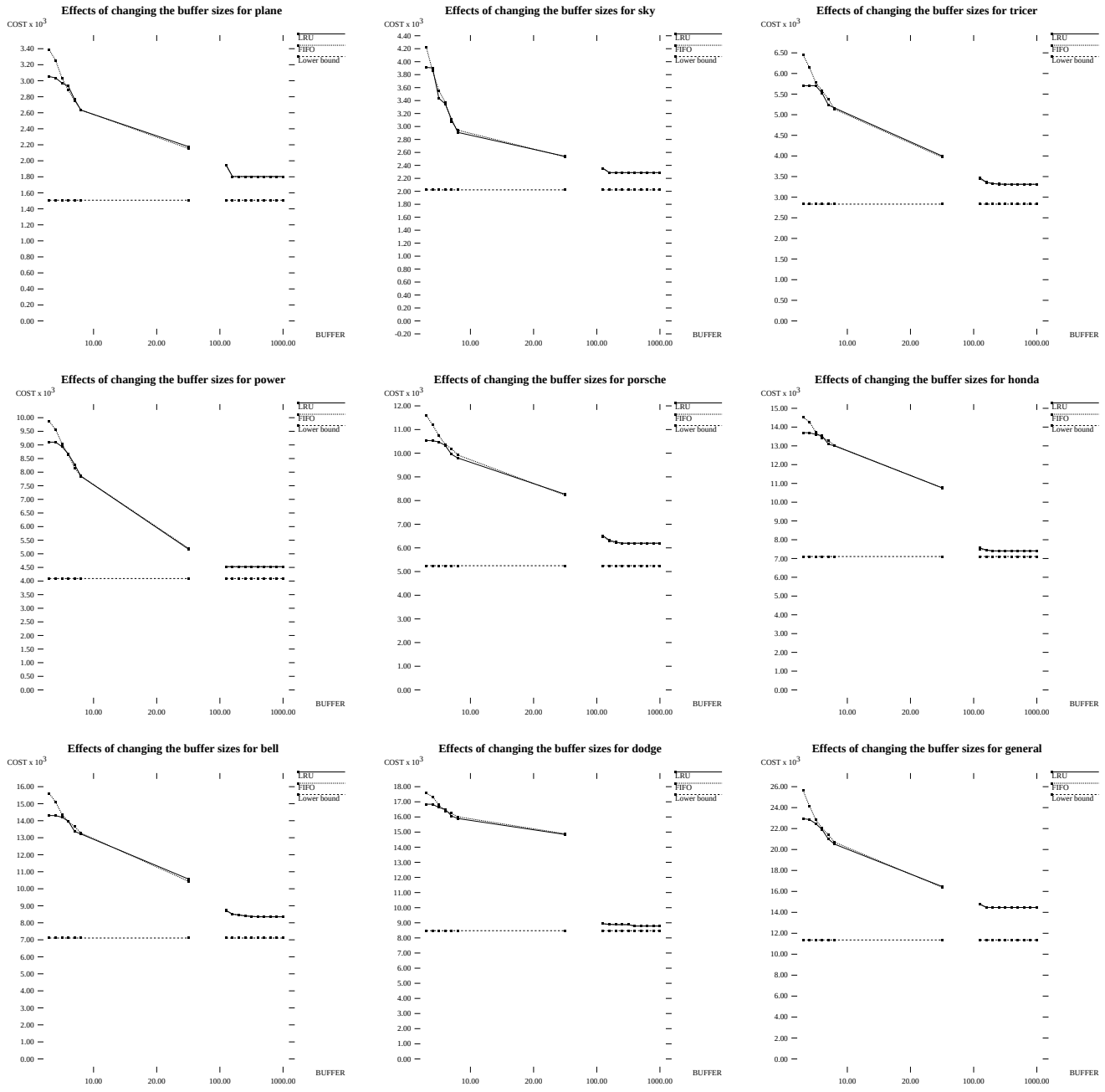


Figure 10: Cost versus buffer size for nine models.

[12] Open GL Architecture Review Board, J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, Reading, MA, 1993.

[13] Silicon Graphics, Inc. Graphics Library Programming Guide, 1991.

Simplification Envelopes

Jonathan Cohen* Amitabh Varshney† Dinesh Manocha* Greg Turk* Hans Weber*
Pankaj Agarwal‡ Frederick Brooks* William Wright*
http://www.cs.unc.edu/~geom/envelope.html

Abstract

We propose the idea of *simplification envelopes* for generating a hierarchy of level-of-detail approximations for a given polygonal model. Our approach guarantees that all points of an approximation are within a user-specifiable distance ϵ from the original model and that all points of the original model are within a distance ϵ from the approximation. Simplification envelopes provide a general framework within which a large collection of existing simplification algorithms can run. We demonstrate this technique in conjunction with two algorithms, one local, the other global. The local algorithm provides a fast method for generating approximations to large input meshes (at least hundreds of thousands of triangles). The global algorithm provides the opportunity to avoid local "minima" and possibly achieve better simplifications as a result.

Each approximation attempts to minimize the total number of polygons required to satisfy the above ϵ constraint. The key advantages of our approach are:

- General technique providing guaranteed error bounds for genus-preserving simplification
- Automation of both the simplification process and the selection of appropriate viewing distances
- Prevention of self-intersection
- Preservation of sharp features
- Allows variation of approximation distance across different portions of a model

CR Categories and Subject Descriptors: I.3.3 **[Computer Graphics]:** Picture/Image Generation — Display algorithms; I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling — Curve, surface, solid, and object representations.
Additional Key Words and Phrases: hierarchical approximation, model simplification, levels-of-detail generation, shape approximation, geometric modeling, offsets.

*Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175.
{cohenj,weberh,manocha,turk,brooks,wright}@cs.unc.edu
†Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400. varshney@cs.sunysb.edu
‡Department of Computer Science, Duke University, Durham, NC 27708-0129. pankaj@cs.duke.edu

1 Introduction

We present the framework of *simplification envelopes* for computing various levels of detail of a given polygonal model. These hierarchical representations of an object can be used in several ways in computer graphics. Some of these are:

- Use in a level-of-detail-based rendering algorithm for providing desired frame update rates [4, 9].
- Simplifying traditionally over-sampled models such as those generated from volume datasets, laser scanners, and satellites for storage and reducing CPU cycles during processing, with relatively few or no disadvantages [10, 11, 13, 15, 21, 23].
- Using low-detail approximations of objects for illumination algorithms, especially radiosity [19].

Simplification envelopes are a generalization of *offset surfaces*. Given a polygonal representation of an object, they allow the generation of minimal approximations that are guaranteed not to deviate from the original by more than a user-specifiable amount while preserving global topology. We surround the original polygonal surface with two envelopes, then perform simplification within this volume. A sample application of the algorithms we describe can be seen in Figure 1.

Figure 1: *A level-of-detail hierarchy for the rotor from a brake assembly.*

Such an approach has several benefits in computer graphics. First, one can very precisely quantify the amount of approximation that is tolerable under given circumstances. Given a user-specified error in number of pixels of deviation of an object's silhouette, it is possible to choose which level of detail to view from a particular distance to maintain that pixel error bound. Second, this approach allows one a fine control over which regions of an object should be approximated more and which ones less. This could be used for selectively preserving those features of an object that are *perceptually* important. Third, the user-specifiable tolerance for approximation is the only parameter required to obtain the approximations; fine tweaking of several parameters depending upon the object to be approximated is not required. Thus, this approach is quite useful for *automating the process* of topology-preserving simplifications of a large number of objects. This problem of *scalability* is important for any simplification algorithm. One of our main goals is to create a method for simplification which is not only automatic for large datasets, but tends to preserve the shapes of the original models.

The rest of the paper is organized in the following manner: we survey the related work in Section 2, explain our assumptions and terminology in Section 3, describe the envelope and approximation computations in Sections 4 and 5, present some useful extentions to and properties of the approximation algorithms in Section 6, and explain our implementation and results in Section 7.

2 Background

Approximation algorithms for polygonal models can be classified into two broad categories:

- **Min-# Approximations**: For this version of the approximation problem, given some error bound ϵ, the objective is to minimize the number of vertices such that no point of the approximation $\mathcal{A}$ is farther than ϵ distance away from the input model $\mathcal{I}$.
- **Min-ϵ Approximations**: Here we are given the number of vertices of the approximation $\mathcal{A}$ and the objective is to minimize the error, or the difference, between $\mathcal{A}$ and $\mathcal{I}$.

Previous work in the area of min-# approximations has been done by [6, 20] where they adaptively subdivide a series of bicubic patches and polygons over a surface until they fit the data within the tolerance levels.

In the second category, work has been done by several groups. Turk [23] first distributes a given number of vertices over the surface depending on the curvature and then re-triangulates them to obtain the final mesh. Schroeder et al. [21] and Hinker and Hansen [13] operate on a set of local rules — such as deleting edges or vertices from almost coplanar adjacent faces, followed by local re-triangulation. These rules are applied iteratively till they are no longer applicable. A somewhat different local approach is taken in [18] where vertices that are close to each other are clustered and a new vertex is generated to represent them. The mesh is suitably updated to reflect this.

Hoppe et al. [14] proceed by iteratively optimizing an energy function over a mesh to minimize both the distance of the approximating mesh from the original, as well as the number of approximating vertices. An interesting and elegant solution to the problem of polygonal simplification by using wavelets has been presented in [7, 8] where arbitrary polygonal meshes are first subdivided into patches with *subdivision connectivity* and then multiresolution wavelet analysis is used over each patch. This wavelet approach preserves global topology.

In computational geometry, it has been shown that computing the minimal-facet ϵ-approximation is NP-hard for both convex polytopes [5] and polyhedral terrains [1]. Thus, algorithms to solve these problems have evolved around finding polynomial-time approximations that are *close* to the optimal.

Let k_o be the size of a min-# approximation. An algorithm has been given in [16] for computing an ϵ-approximation of size $O(k_o \log n)$ for convex polytopes. This has recently been improved by Clarkson in [3]; he proposes a randomized algorithm for computing an approximation of size $O(k_o \log k_o)$ in expected time $O(k_o n^{1+\delta})$ for any $\delta > 0$ (the constant of proportionality depends on δ, and tends to $+\infty$ as δ tends to 0). In [2] Brönnimann and Goodrich observed that a variant of Clarkson's algorithm yields a polynomial-time deterministic algorithm that computes an approximation of size $O(k_0)$. Working with polyhedral terrains, [1] present a polynomial-time algorithm that computes an ϵ-approximation of size $O(k_o \log k_o)$ to a polyhedral terrain.

Our work is different from the above in that it allows adaptive, genus-preserving, ϵ-approximation of arbitrary polygonal objects. Additionally, we can simplify bordered meshes and meshes with holes. In terms of direct comparison with the global topology preserving approach presented in [7, 8], for a given ϵ our algorithm has been *empirically* able to obtain "reduced" simplifications, which are much smaller in output size (as demonstrated in Section 7). The algorithm in [18] also guarantees a bound in terms of the Hausdorff metric. However, it is not guaranteed to preserve the genus of the original model.

3 Terminology and Assumptions

Let us assume that $\mathcal{I}$ is a three-dimensional compact and orientable object whose polygonal representation $\mathcal{P}$ has been given to us. Our objective is to compute a piecewise-linear approximation $\mathcal{A}$ of $\mathcal{P}$. Given two piecewise linear objects $\mathcal{P}$ and $\mathcal{Q}$, we say that $\mathcal{P}$ and $\mathcal{Q}$ are *ϵ-approximations* of each other iff every point on $\mathcal{P}$ is within a distance ϵ of some point of $\mathcal{Q}$ and every point on $\mathcal{Q}$ is within a distance ϵ of some point of $\mathcal{P}$. Our goal is to outline a method to generate two envelope surfaces surrounding $\mathcal{P}$ and demonstrate how the envelopes can be used to solve the following polygonal approximation problem. Given a polygonal representation $\mathcal{P}$ of an object and an approximation parameter ϵ, generate a genus-preserving ϵ-approximation $\mathcal{A}$ with minimal number of polygons such that the vertices of $\mathcal{A}$ are a subset of vertices of $\mathcal{P}$.

We assume that all polygons in $\mathcal{P}$ are triangles and that $\mathcal{P}$ is a well-behaved polygonal model, i.e., every edge has either one or two adjacent triangles, no two triangles interpenetrate, there are no unintentional "cracks" in the model, no T-junctions, etc.

We also assume that each vertex of $\mathcal{P}$ has a single normal vector, which must lie within 90° of the normal of each of its surrounding triangles. For the purpose of rendering, each vertex may have either a single normal or multiple normals. For the purpose of generating envelope surfaces, we shall compute a single vertex normal as a combination of the normals of the surrounding triangles.

The three-dimensional ϵ-offset surface for a parametric surface

$$\mathbf{f}(s,t) = (f_1(s,t), f_2(s,t), f_3(s,t)),$$

whose unit normal to $\mathbf{f}$ is

$$\mathbf{n}(s,t) = (n_1(s,t), n_2(s,t), n_3(s,t)),$$

is defined as $\mathbf{f}^\epsilon(s,t) = (f_1^\epsilon(s,t), f_2^\epsilon(s,t), f_3^\epsilon(s,t))$, where

$$f_i^\epsilon(s,t) = f_i(s,t) + \epsilon n_i(s,t).$$

Note that offset surfaces for a polygonal object can self-intersect and may contain non-linear elements. We define a simplification envelope $\mathcal{P}(+\epsilon)$ (respectively $\mathcal{P}(-\epsilon)$) for an object $\mathcal{I}$ to be a *polygonal* surface that lies *within* a distance of ϵ from every point p on $\mathcal{I}$ in the same (respectively opposite) direction as the normal to $\mathcal{I}$ at p. Thus, the simplification envelopes can be thought of as an approximation to offset surfaces. Henceforth we shall refer to simplification envelope by simply envelope.

Let us refer to the triangles of the given polygonal representation $\mathcal{P}$ as the *fundamental triangles*. Let $e = (v_1, v_2)$ be an edge of $\mathcal{P}$. If the normals $\mathbf{n}_1, \mathbf{n}_2$ to $\mathcal{I}$ at both v_1 and v_2, respectively, are identical, then we can construct a plane π_e that passes through the edge e and has a normal that is perpendicular to that of v_1. Thus v_1, v_2 and their normals all lie along π_e. Such a plane defines two half-spaces for edge e, say π_e^+ and π_e^- (see Fig 2(a)). However, in general the normals $\mathbf{n}_1$ and $\mathbf{n}_2$ at the vertices v_1 and v_2 need not be identical, in which case it is not clear how to define the two half-spaces for an edge. One choice is to use a *bilinear patch* that passes through v_1 and v_2 and has a tangent $\mathbf{n}_1$ at v_1 and $\mathbf{n}_2$ at v_2. Let us call such a bilinear patch for e as the *edge half-space* β_e. Let the two half-spaces for the edge e in this case be β_e^+ and β_e^-. This is shown in Fig 2(b).

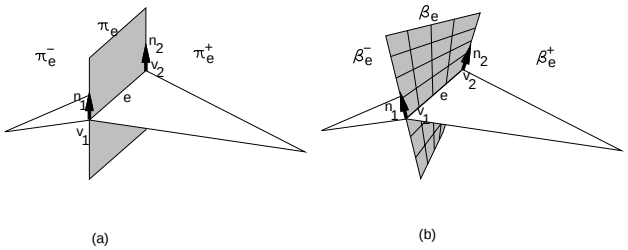


Figure 2: *Edge Half-spaces*

Let the vertices of a fundamental triangle be v_1, v_2, and v_3. Let the coordinates and the normal of each vertex v be represented as $\mathbf{c}(v)$ and $\mathbf{n}(v)$, respectively. The coordinates and the normal of a $(+\epsilon)$-offset vertex v_i^+ for a vertex v_i are: $\mathbf{c}(v_i^+) = \mathbf{c}(v_i) + \epsilon\mathbf{n}(v_i)$, and $\mathbf{n}(v_i^+) = \mathbf{n}(v_i)$. The $(-\epsilon)$-offset vertex can be similarly defined in the opposite direction. These offset vertices for a fundamental triangle are shown in Figure 3.

Now consider the closed object defined by v_i^+ and v_i^-, $i = 1, 2, 3$. It is defined by two triangles, at the top and bottom, and three edge half-spaces. This object contains the fundamental triangle (shown shaded in Figure 3) and we will henceforth refer to it as the *fundamental prism*.

4 Envelope Computation

In order to preserve the input topology of $\mathcal{P}$, we desire that the simplification envelopes do not self-intersect. To meet this criterion we reduce our level of approximation at certain places. In other words, to guarantee that no intersections amongst the envelopes occur, we have to be

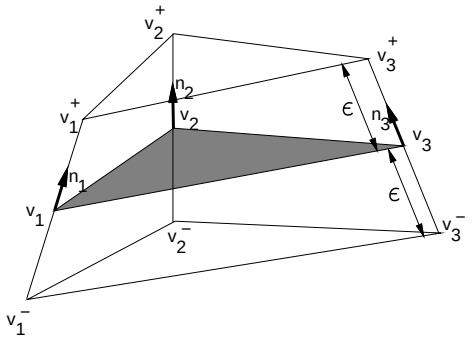


Figure 3: *The Fundamental Prism*

content at certain places with the distance between $\mathcal{P}$ and the envelope being smaller than ϵ. This is how simplification envelopes differ from offset surfaces.

We construct our envelope such that each of its triangles corresponds to a fundamental triangle. We offset each vertex of the original surface in the direction of its normal vector to transform the fundamental triangles into those of the envelope.

If we offset each vertex v_i by the same amount ϵ, to get the offset vertices v_i^+ and v_i^-, the resulting envelopes, $\mathcal{P}(+\epsilon)$ and $\mathcal{P}(-\epsilon)$, may contain self-intersections because one or more offset vertices are closer to some non-adjacent fundamental triangle. In other words, if we define a Voronoi diagram over the fundamental triangles of the model, the condition for the envelopes to intersect is that there be at least one offset vertex lying in the Voronoi region of some non-adjacent fundamental triangle. This is shown in Figure 4 by means of a two-dimensional example. In the figure, the offset vertices b^+ and c^+ are in the Voronoi regions of edges other than their own, thus causing self-intersection of the envelope.

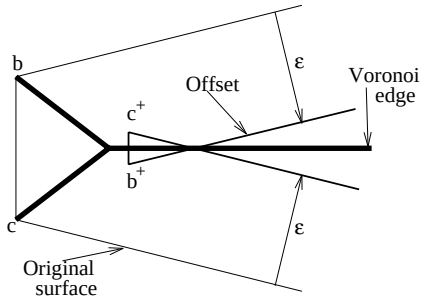


Figure 4: *Offset Surfaces*

Once we make this observation, the solution to avoid self-intersections becomes quite simple — just do not allow an offset vertex to go beyond the Voronoi regions of its adjacent fundamental triangles. In other words, determine the positive and negative ϵ for each vertex v_i such that the vertices v_i^+ and v_i^- determined with this new ϵ do not lie in the Voronoi regions of the non-adjacent fundamental triangles.

While this works in theory, efficient and robust computation of the three-dimensional Voronoi diagram of the fundamental triangles is non-trivial. We now present two methods for computing the reduced ϵ for each vertex, the first method analytical, and the second numerical.

4.1 Analytical ϵ Computation

We adopt a conservative approach for recomputing the ϵ at each vertex. This approach underestimates the values for the positive and negative ϵ. In other words, it guarantees the envelope surfaces not to intersect, but it does not guarantee that the ϵ at each vertex is the largest permissible ϵ. We next discuss this approach for the case of computing the positive ϵ for each vertex. Computation of negative ϵ follows similarly.

Consider a fundamental triangle t. We define a prism t_p for t, which is conceptually the same as its fundamental prism, but uses a value of 2ϵ instead of ϵ for defining the envelope vertices. Next, consider all triangles Δ_i that do not share a vertex with t. If Δ_i intersects t_p above t (the directions above and below t are determined by the direction of the normal to t, above is in the same direction as the normal to t), we find the point on Δ_i that lies within t_p and is closest to t. This point would be either a vertex of Δ_i, or the intersection point of one of its edges with the three sides of the prism t_p. Once we find the point of closest approach, we compute the distance δ_i of this point from t. This is shown in Figure 5.

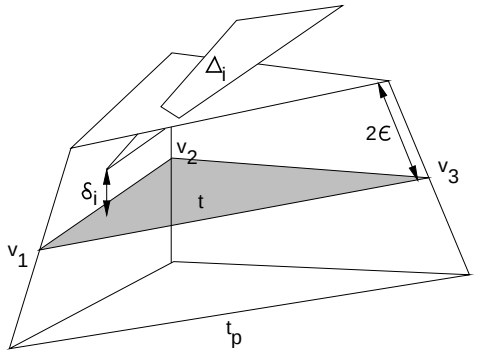


Figure 5: *Computation of δ_i*

Once we have done this for all Δ_i, we compute the new value of the positive ϵ for the triangle t as $\epsilon_{new} = \frac{1}{2}\min_i \delta_i$. If the vertices for this triangle t have this value of positive ϵ, their positive envelope surface will not self-intersect. Once the $\epsilon_{new}(t)$ values for all the triangles t have been computed, the $\epsilon_{new}(v)$ for each vertex v is set to be the minimum of the $\epsilon_{new}(t)$ values for all its adjacent triangles.

We use an octree in our implementation to speed up the identification of triangles Δ_i that intersect a given prism.

4.2 Numerical ϵ Computation

To compute an envelope surface numerically, we take an iterative approach. Our envelope surface is initially identical to the input model surface. In each iteration, we sequentially attempt to move each envelope vertex a fraction of the ϵ distance in the direction of its normal vector (or the opposite direction, for the inner envelope). This effectively stretches or contracts all the triangles adjacent to the vertex. We test each of these adjacent triangles for intersection with each other and the rest of the model. If no such intersections are found, we accept the step, leaving the vertex in this new position. Otherwise we reject it, moving the vertex back to its previous position. The iteration terminates when all vertices have either moved ϵ or can no longer move.

In an attempt to guarantee that each vertex gets to move a reasonable amount of its potential distance, we use an adaptive step size. We encourage a vertex to move at least K (an arbitrary constant which is scaled with respect to ϵ and the size of the object) steps by allowing it to reduce its step size. If a vertex has moved less than K steps and its move is been rejected, it divides its step size in half and tries again (with some maximum number of divides allowed on any particular step). Notice that if a vertex moves i steps and is rejected on the $(i+1)$st step, we know it has moved at least $i/(i+1)$ % of its potential distance, so $K/(K+1)$ which is a lower bound of sorts. It is possible, though rare, for a vertex to move less than K steps, if its current position is already quite close to another triangle.

Each vertex also has its own initial step size. We first choose a global, maximum step size based on a global property: either some small percentage of the object's bounding box diagonal length or ϵ/K, whichever is smaller. Now for each vertex, we calculate a local step size. This local step size is some percentage of the vertex's shortest incident edge (only those edges within 90^o of the offset direction are considered). We set the vertex's step size to the minimum of the global step size and its local step size. This makes it likely that each vertex's step size is appropriate for a step given the initial mesh configuration.

This approach to computing an envelope surface is robust, simple to implement (if difficult to explain), and fair to all the vertices. It tends to maximize the minimum offset distance amongst the envelope vertices. It works fairly well in practice, though there may still be some room for improvement in generating maximal offsets for thin objects. Figure 6 shows internal and external envelopes computed for three values of ϵ using this approach.

As in the analytical approach, a simple octree data structure makes these intersection tests reasonably efficient, especially for models with evenly sized triangles.

5 Generation of Approximation

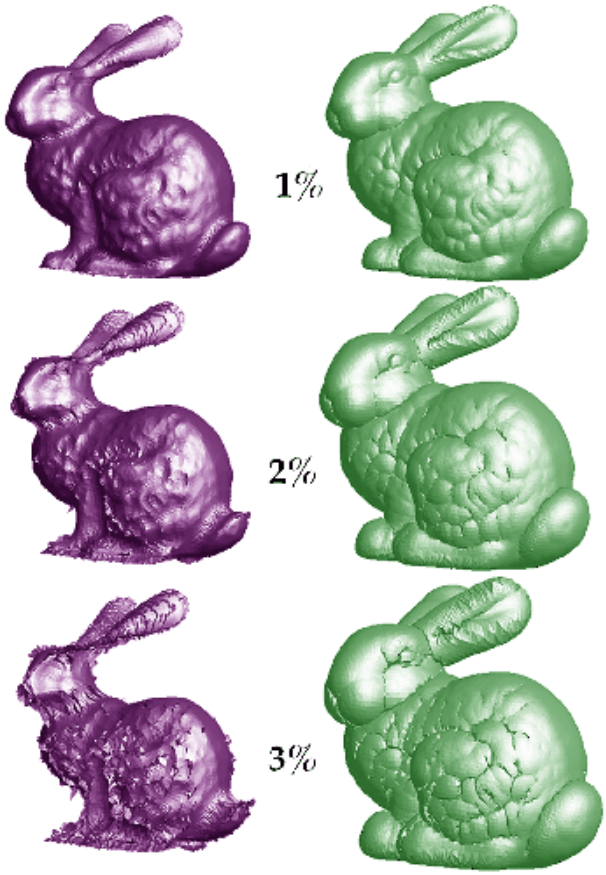
Generating a surface approximation typically involves starting with the input surface and iteratively making modifications to ultimately reduce its complexity. This process may be broken into two main stages: *hole creation*, and *hole filling*. We first create a hole by removing some connected set of triangles from the surface mesh. Then we fill the hole with a smaller set of triangles, resulting in some reduction of the mesh complexity.

We demonstrate the generality of the simplification envelope approach by designing two algorithms. The hole filling stages of these algorithms are quite similar, but their hole creation stages are quite different. The first algorithm makes only local choices, creating relatively small holes, while the second algorithm uses global information about the surface to create maximally-sized holes. These design choices produce algorithms with very different properties.

We begin by describing the envelope validity test used to determine whether a *candidate triangle* is valid for inclusion in the approximation surface. We then proceed to the two example simplification algorithms and a description of their relative merits.

5.1 Validity Test

A *candidate triangle* is one which we are considering for inclusion in an approximation to the input surface. Valid candidate triangles must lie between the two envelopes. Because we construct candidate triangles from the vertices of the original model, we know its vertices lie between the two envelopes. Therefore, it is sufficient to test the candidate triangle for intersections with the two envelope



Inner Envelopes ϵ **Outer Envelopes**

Figure 6: *Simplification envelopes for various* ϵ

surfaces. We can perform such tests efficiently using a space-partitioning data structure such as an octree.

A valid candidate triangle must also not cause a self-intersection in our surface, Therefore, it must not intersect any triangle of the current approximation surface.

5.2 Local Algorithm

To handle large models efficiently within the framework of simplification envelopes we construct a vertex-removal-based local algorithm. This algorithm draws heavily on the work of [21], [23], and [14]. Its main contributions are the use of envelopes to provide global error bounds as well as topology preservation and non-self-intersection. We have also explored the use of a more exhaustive hole-filling approach than any previous work we have seen.

The local algorithm begins by placing all vertices in a queue for removal processing. For each vertex in the queue, we attempt to remove it by creating a hole (removing the vertex's adjacent triangles) and attempting to fill it. If we can successfully fill the hole, the mesh modification is accepted, the vertex is removed from the queue, and its neighbors are placed back in the queue. If not, the vertex is removed from the queue and the mesh remains unchanged. This process terminates when the global error bounds eventually prevent the removal of any more vertices. Once the vertex queue is empty we have our simplified mesh.

For a given vertex, we first create a hole by removing all adjacent triangles. We begin the hole-filling process by generating all possible triangles formed by combinations of the vertices on the hole boundary. This is not strictly necessary, but it allows us to use a greedy strategy to favor triangles with nice aspect ratios. We fill the hole by choosing a triangle, testing its validity, and recursively filling the three (or fewer) smaller holes created by adding that triangle into the hole (see figure 7). If a hole cannot be filled at any level of the recursion, the entire hole filling attempt is considered a failure. Note that this is a single-pass hole filling strategy; we do not backtrack or undo selection of a triangle chosen for filling a hole. Thus, this approach does not guarantee that if a triangulation of a hole exists we will find it. However, it is quite fast and works very well in practice.

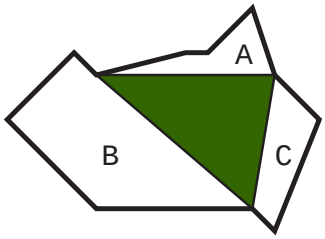


Figure 7: *Hole filling: adding a triangle into a hole creates up to three smaller holes*

We have compared the above approach with an exhaustive approach in which we tried all possible hole-filling triangulations. For simplifications resulting in the removal of hundreds of vertices (like highly oversampled laser-scanned models), the exhaustive pass yielded only a small improvement over the single-pass heuristic. This sort of confirmation reassures us that the single-pass heuristic works well in practice.

5.3 Global Algorithm

This algorithm extends the algorithm presented in [3] to non-convex surfaces. Our major contribution is the use of simplification envelopes to bound the error on a non-convex polygonal surface and the use of fundamental prisms to provide a generalized projection mechanism for testing for regions of multiple covering (overlaps). We present only a sketch of the algorithm here ; see [24] for the full details.

We begin by generating all possible candidate triangles for our approximation surface. These triangles are all 3-tuples of the input vertices which do not intersect either of the offset surfaces. Next we determine how many vertices each triangle *covers*. We rank the candidate triangles in order of decreasing covering.

We then choose from this list of candidate triangles in a greedy fashion. For each triangle we choose, we create a large hole in the current approximation surface, removing all triangles which *overlap* this candidate triangle. Now we begin the recursive hole-filling process by placing this candidate triangle into the hole and filling all the subholes with other triangles, if possible. One further restriction in this process is that the candidate triangle we are testing should not overlap any of the candidate triangles we have previously accepted.

5.4 Algorithm Comparison

The local simplification algorithm is fast and robust enough to be applied to large models. The global strategy is theoretically elegant. While the global algorithm works well for small models, its complexity rises at least quadratically,

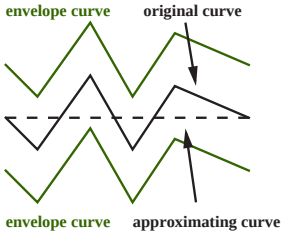


Figure 8: *Curve at local minimum of approximation*

making it prohibitive for larger models. We can think of the simplification problem as an optimization problem as well. A purely local algorithm may get trapped in a local "minimum" of simplification, while an ideal global algorithm will avoid all such minima.

Figure 8 shows a two-dimensional example of a curve for which a local vertex removal algorithm might fail, but an algorithm that globally searches the solution space will succeed in finding a valid approximation. Any of the interior vertices we remove would cause a new edge to penetrate an envelope curve. But if we remove all of the interior vertices, the resulting edge is perfectly acceptable.

This observation motivates a wide range of algorithms of which our local and global examples are the two extremes. We can easily imagine an algorithm that chooses nearby groups of vertices to remove simultaneously rather than sequentially. This could potentially lead to increased speed and simplification performance. However, choosing such sets of vertices remains a challenging problem.

6 Additional Features

Envelope surfaces used in conjunction with simplification algorithms are powerful, general-purpose tools. As we will now describe, they implicitly preserve sharp edges and can be extended to deal with bordered surfaces and perform adaptive approximations.

6.1 Preserving Sharp Edges

One of the important properties in any approximation scheme is the way it preserves any sharp edges or normal discontinuities present in the input model. Simplification envelopes deal gracefully with sharp edges (those with a small angle between their adjacent faces). When the ϵ tolerance is small, there is not enough room to simplify across these sharp edges, so they are automatically preserved. As the tolerance is increased, it will eventually be possible to simplify across the edges (which should no longer be visible from the appropriate distance). Notice that it is not necessary to explicitly recognize these sharp edges.

6.2 Bordered Surfaces

A bordered surface is one containing points that are homeomorphic to a half-disc. For polygonal models, this corresponds to edges that are adjacent to a single face rather than two faces. Depending on the context, we may naturally think of this as the boundary of some plane-like piece of a surface, or a hole in an otherwise closed surface.

The algorithms described in 5 are sufficient for closed triangle meshes, but they will not guarantee our global error bound for meshes with borders. While the envelopes constrain our error with respect to the normal direction of the surface, bordered surfaces require some additional constraints to hold the approximation border close to the original border. Without such constraints, the border of the approximation surface may "creep in," possibly shrinking the surface out of existence.

In many cases, the complexity of a surface's border curves may become a limiting factor in how much we can simplify the surface, so it is unacceptable to forgo simplifying these borders.

We construct a set of border tubes to constrain the error in deviation of the border curves. Each border is actually a cyclic polyline. Intuitively speaking, a border tube is a smooth, non-self-intersecting surface around one of these polylines. Removing a border vertex causes a pair of border edges to be replaced by a single border edge. If this new border edge does not intersect the relevant border tube, we may safely attempt to remove the border vertex.

To construct a tube we define a plane passing through each vertex of the polyline. We choose a coordinate system on this plane and use that to define a circular set of vertices. We connect these vertices for consecutive planes to construct our tube. Our initial tubes have a very narrow radius to minimize the likelihood of self-intersections. We then expand these narrow tubes using the same technique we used previously to construct our simplification envelopes.

The difficult task is to define a coordinate system at each polyline vertex which encourages smooth, non-self-intersecting tubes. The most obvious approach might be to use the idea of Frenet frames from differential geometry to define a set of coordinate systems for the polyline vertices. However, Frenet frames are meant for smooth curves. For a jagged polyline, a tube so constructed often has many self-intersections.

Instead, we use a projection method to minimize the twist between consecutive frames. Like the Frenet frame method, we place the plane at each vertex so that the normal to the plane approximates the tangent to the polyline. This is called the *normal plane*.

At the first vertex, we choose an arbitrary orthogonal pair of axes for our coordinate system in the normal plane. For subsequent vertices, we project the coordinate system from the previous normal plane onto the current normal frame. We then orthogonalize this projected coordinate system in the plane. For the normal plane of the final polyline vertex, we average the projected coordinate systems of the previous normal plane and the initial normal plane to minimize any twist in the final tube segment.

6.3 Adaptive Approximation

For certain classes of objects it is desirable to perform an adaptive approximation. For instance, consider large terrain datasets, models of spaceships, or submarines. One would like to have more detail near the observer and less detail further away. A possible solution could be to subdivide the model into various spatial cells and use a different ϵ-approximation for each cell. However, problems would arise at the boundaries of such cells where the ϵ-approximation for one cell, say at a value ϵ_1 need not necessarily be continuous with the ϵ-approximation for the neighboring cell, say at a different value ϵ_2.

Since all candidate triangles generated are constrained to lie within the two envelopes, manipulation of these envelopes provides one way to smoothly control the level of approximation. Thus, one could specify the ϵ at a given vertex to be a function of its distance from the observer — the larger the distance, the greater is the ϵ.

As another possibility, consider the case where certain

features of a model are very important and are not to be approximated beyond a certain level. Such features might have human perception as a basis for their definition or they might have mathematical descriptions such as regions of high curvature. In either case, a user can vary the ϵ associated with a region to increase or decrease the level of approximation. The bunny in Figure 9 illustrates such an approximation.

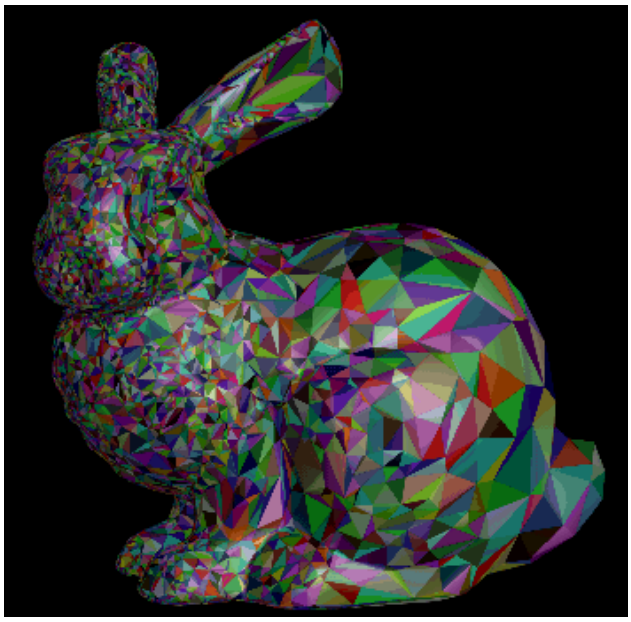


Figure 9: *An adaptive simplification for the bunny model. ϵ varies from 1/64% at the nose to 1% at the tail.*

7 Implementation and Results

We have implemented both algorithms and tried out the local algorithm on several thousand objects. We will first discuss some of the implementation issues and then present some results.

7.1 Implementation Issues

The first important implementation issue is what sort of input model to accept. We chose to accept only manifold triangle meshes (or bordered manifolds). This means that each edge is adjacent to two (one in the case of a border) triangles and that each vertex is surrounded by a single ring of triangles.

We also do not accept other forms of degenerate meshes. Many mesh degeneracies are not apparent on casual inspection, so we have implemented an automatic degeneracy detection program. This program detects non-manifold vertices, non-manifold edges, sliver triangles, coincident triangles, T-junctions, and intersecting triangles in a proposed input mesh. Note that correcting these degeneracies is more difficult than detecting them.

Robustness issues are important for implementations of any geometric algorithms. For instance, the analytical method for envelope computation involves the use of bilinear patches and the computation of intersection points. The computation of intersection points, even for linear elements, is difficult to perform robustly. The numerical method for envelope computation is much more robust because it involves only linear elements. Furthermore, it requires an intersection test but not the calculation of intersection points. We perform all such intersection tests in a conservative manner, using fuzzy intersection tests that may report intersections even for some close but non-intersecting elements.

Another important issue is the use of a space-partitioning scheme to speed up intersection tests. We chose to use an octree because of its simplicity. Our current octree implementation deals only with the bounding boxes of the elements stored. This works well for models with triangles that are evenly sized and shaped. For CAD models, which may contain long, skinny, non-axis-aligned triangles, a simple octree does not always provide enough of a speed-up, and it may be necessary to choose a more appropriate space-partitioning scheme.

7.2 Results

We have simplified a total of 2636 objects from the auxiliary machine room (AMR) of a submarine dataset, pictured in Figure 10 to test and validate our algorithm. We reproduce the timings and simplifications achieved by our implementation for the AMR and a few other models in Table 1. All simplifications were performed on a Hewlett-Packard 735/125 with 80 MB of main memory. Images of these simplifications appear in Figures 11 and 12. It is interesting to compare the results on the bunny and phone models with those of [7, 8]. For the same error bound, we are able to obtain much improved solutions.

We have automated the process which sets the ϵ value for each object by assigning it to be a percentage of the diagonal of its bounding box. We obtained the reductions presented in Table 1 for the AMR and Figures 11 and 12 by using this heuristic.

For the rotor and AMR models in the above results, the i^{th} level of detail was obtained by simplifying the $i-1^{th}$ level of detail. This causes to total ϵ to be the sum of all previous ϵ's, so choosing $\epsilon's$ of 1, 2, 4, and 8 percent results in total $\epsilon's$ of 1, 3, 7, and 15 percent. There are two advantages to this scheme:
(a) It allows one to proceed incrementally, taking advantage of the work done in previous simplifications.
(b) It builds a hierarchy of detail in which the vertices at the i^{th} level of detail are a subset of the vertices at the $i-1^{th}$ level of detail.

One of the advantages of the setting ϵ to a percent of the object size is that it provides an a way to automate the selection of switching points used to transition between the various representations. To eliminate visual artifacts, we switch to a more faithful representation of an object when ϵ projects to more than some user-specified number of pixels on the screen. This is a function of the ϵ for that approximation, the output display resolution, and the corresponding maximum tolerable visible error in pixels.

8 Future Work

There are still several areas to be explored in this research. We believe the most important of these to be the generation of correspondences between levels of detail and the moving of vertices within the envelope volume.

Bunny			Phone			Rotor			AMR		
ϵ %	# Polys	Time	ϵ %	# Polys	Time	ϵ %	# Polys	Time	ϵ %	# Polys	Time
0	69,451	N/A	0	165,936	N/A	0	4,735	N/A	0	436,402	N/A
1/64	44,621	9	1/64	43,537	31	1/8	2,146	3	1	195,446	171
1/32	23,581	10	1/32	12,364	35	1/4	1,514	2	3	143,728	61
1/16	10,793	11	1/16	4,891	38	3/4	1,266	2	7	110,090	61
1/8	4,838	11	1/8	2,201	32	1 3/4	850	1	15	87,476	68
1/4	2,204	11	1/4	1,032	35	3 3/4	716	1	31	75,434	84
1/2	1,004	11	1/2	544	33	7 3/4	688	1			
1	575	11	1	412	30	15 3/4	674	1			

Table 1: *Simplification ϵ's and run times in minutes*

8.1 Generating Correspondences

A true geometric hierarchy should contain not only representations of an object at various levels of detail, but also some correspondence information about the relationship between adjacent levels. These relationships are necessary for propagating local information from one level to the next. For instance, this information would be helpful for using the hierarchical geometric representation to perform radiosity calculations. It is also necessary for performing geometric interpolation between the models when using the levels of detail for rendering. Note that the envelope technique preserves silhouettes when rendering, so it is also a good candidate for alpha blending rather than geometric interpolation to smooth out transitions between levels of detail.

We can determine which elements of a higher level of detail surface are covered by an element of a lower level of detail representation by noting which fundamental prisms this element intersects. This is non-trivial only because of the bilinear patches that are the sides of a fundamental prism. We can approximate these patches by two or more triangles and then tetrahedralize each prism. Given this tetrahedralization of the envelope volume, it is possible to stab each edge of the lower level-of-detail model through the tetrahedrons to determine which ones they intersect, and thus which triangles are covered by each lower level-of-detail triangle.

8.2 Moving Vertices

The output mesh generated by either of the algorithms we have presented has the property that its set of vertices is a subset of the set of vertices of the original mesh. If we can afford to relax this constraint somewhat, we may be able to reduce the output size even further. If we allow the vertices to slide along their normal vectors, we should be able to simplify parts of the surface that might otherwise be impossible to simplify for some choices of epsilon. We are currently working on a goal-based approach to moving vertices within the envelope volume. For each vertex we want to remove, we slide its neighboring vertices along their normals to make them lie as closely as possible to a tangent plane of the original vertex. Intuitively, this should increase the likelihood of successfully removing the vertex. During this whole process, we must ensure that none of the neighboring triangles ever violates the envelopes. This approach should make it possible to simplify surfaces using smaller epsilons than previously possible. In fact, it may even enable us to use the original surface and a single envelope as our constraint surfaces rather than two envelopes. This is important for objects with areas of high maximal curvature, like thin cylinders.

9 Conclusion

We have outlined the notion of simplification envelopes and how they can be used for generation of multiresolution hierarchies for polygonal objects. Our approach guarantees non-self-intersecting approximations and allows the user to do adaptive approximations by simply editing the simplification envelopes (either manually or automatically) in the regions of interest. It allows for a global error tolerance, preservation of the input genus of the object, and preservation of sharp edges. Our approach requires only one user-specifiable parameter, allowing it to work on large collections of objects with no manual intervention if so desired. It is rotationally and translationally invariant, and can elegantly handle holes and bordered surfaces through the use of cylindrical tubes. Simplification envelopes are general enough to permit both simplification algorithms with good theoretical properties such as our global algorithm, as well as fast, practical, and robust implementations like our local algorithm. Additionally, envelopes permit easy generation of correspondences across several levels of detail.

10 Acknowledgements

Thanks to Greg Angelini, Jim Boudreaux, and Ken Fast at Electric Boat for the submarine model, Rich Riesenfeld and Elaine Cohen of the Alpha_1 group at the University of Utah for the rotor model, and the Stanford Computer Graphics Laboratory for the bunny and telephone models.Thanks to Carl Mueller, Marc Olano, and Bill Yakowenko for many useful suggestions, and to the rest of the UNC Simplification Group (Rui Bastos, Carl Erikson, Merlin Hughes, and David Luebke) for providing a great forum for discussing ideas. The funding for this work was provide by a Link Foundation Fellowship, Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, ARO MURI grant DAAH04-96-1-0013, NSF Grant CCR-9319957, NSF Grant CCR-9301259, NSF Career Award CCR-9502239, ONR Contract N00014-94-1-0738, ARPA Contract DABT63-93-C-0048, NSF/ARPA Center for Computer Graphics and Scientific Visualization, NIH Grant RR02170, an NYI award with matching funds from Xerox Corp, and a U.S.-Israeli Binational Science Foundation grant.

References

[1] P. Agarwal and S. Suri. Surface approximation and geometric partitions. In *Proceedings Fifth Symposium on Discrete Algorithms*, pages 24–33, 1994.

[2] H. Brönnimann and M. Goodrich. Almost optimal set covers in finite VC-dimension. In *Proceedings Tenth ACM Symposium on Computational Geometry*, pages 293–302, 1994.

[3] K. L. Clarkson. Algorithms for polytope covering and approximation. In *Proc. 3rd Workshop Algorithms Data Struct.*, Lecture Notes in Computer Science, 1993.

[4] M. Cosman and R. Schumacker. System strategies to optimize CIG image content. In *Proceedings of the Image II Conference*, Scottsdale, Arizona, June 10–12 1981.

[5] G. Das and D. Joseph. The complexity of minimum convex nested polyhedra. In *Proc. 2nd Canad. Conf. Comput. Geom.*, pages 296–301, 1990.

[6] M. J. DeHaemer, Jr. and M. J. Zyda. Simplification of objects rendered by polygonal approximations. *Computers & Graphics*, 15(2):175–184, 1991.

[7] T. D. DeRose, M. Lounsbery, and J. Warren. Multiresolution analysis for surface of arbitrary topological type. Report 93-10-05, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[8] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. *Computer Graphics: Proceedings of SIGGRAPH'95*, pages 173–182, 1995.

[9] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[10] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Computer Graphics: Proceedings of SIGGRAPH 1993*, pages 231–238. ACM SIGGRAPH, 1993.

[11] T. He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. Voxel-based object simplification. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 296–303, 1995.

[12] P. Heckbert and M. Garland. Multiresolution modeling for fast rendering. *Proceedings of Graphics Interface*, 1994.

[13] P. Hinker and C. Hansen. Geometric optimization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings Visualization '93*, pages 189–195, October 1993.

[14] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 19–26, August 1993.

[15] A. D. Kalvin and R. H. Taylor. Superfaces: Polyhedral approximation with bounded error. Technical Report RC 19135 (#82286), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10958, 1993.

[16] J. Mitchell and S. Suri. Separation and approximation of polyhedral surfaces. In *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 296–306, 1992.

[17] Kevin J. Renze and J. H. Oliver. Generalized surface and volume decimation for unstructured tessellated domains. In *Proceedings of SIVE'95*, 1995.

[18] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.

[19] H. E. Rushmeier, C. Patterson, and A. Veerasamy. Geometric simplification for indirect illumination calculations. In *Proceedings Graphics Interface '93*, pages 227–236, 1993.

[20] F. J. Schmitt, B. A. Barsky, and W. Du. An adaptive subdivision method for surface-fitting from sampled data. *Computer Graphics (SIGGRAPH '86 Proceedings)*, 20(4):179–188, 1986.

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

[22] G. Taubin. A signal processing approach to fair surface design. In *Proc. of ACM Siggraph*, pages 351–358, 1995.

[23] G. Turk. Re-tiling polygonal surfaces. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 55–64, July 1992.

[24] A. Varshney. Hierarchical geometric approximations. Ph.D. Thesis TR-050-1994, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994.

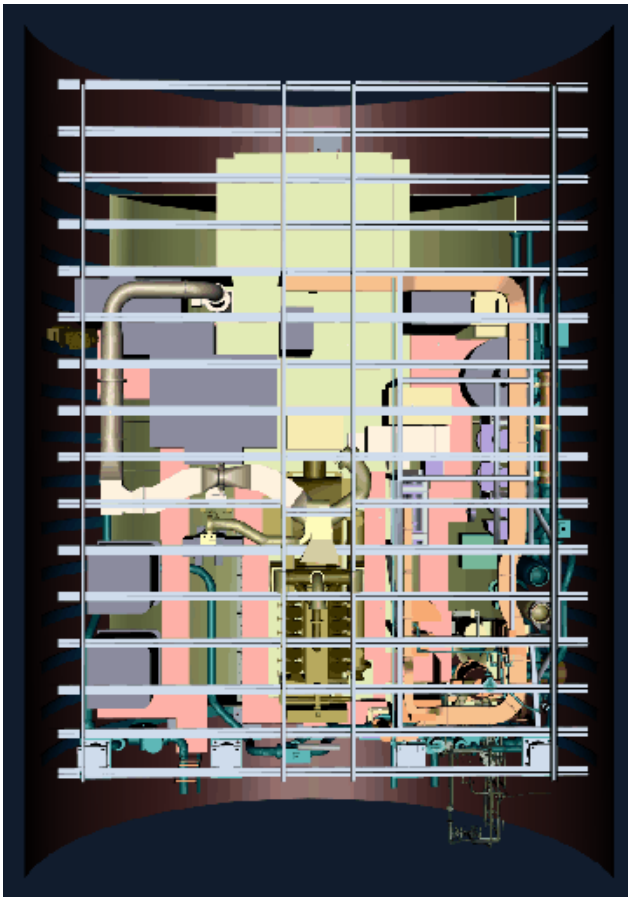


Figure 10: *Looking down into the auxiliary machine room (AMR) of a submarine model. This model contains nearly 3,000 objects, for a total of over half a million triangles. We have simplified over 2,600 of these objects, for a total of over 430,000 triangles.*

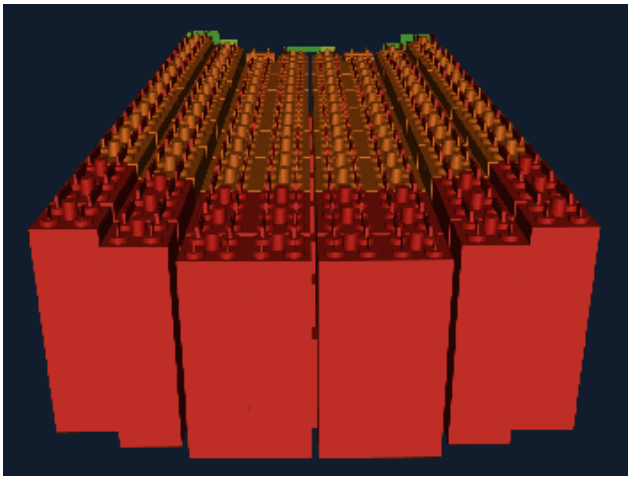
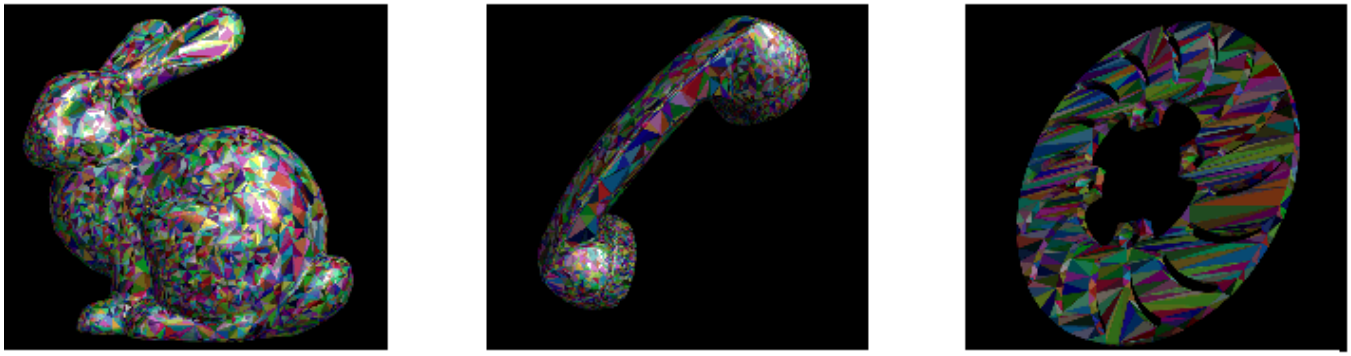
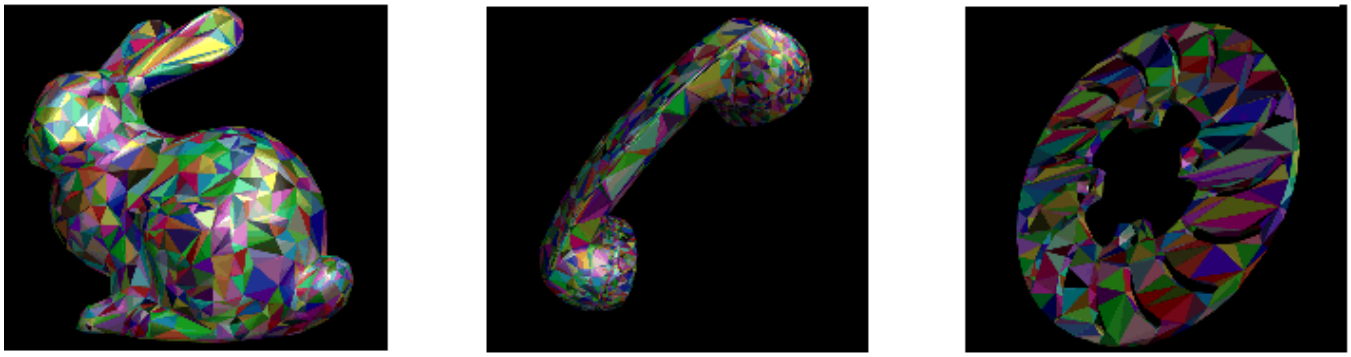


Figure 11: *An array of batteries from the AMR. All parts but the red are simplified representations. At full resolution, this array requires 87,000 triangles. At this distance, allowing 4 pixels of error in screen space, we have reduced it to 45,000 triangles.*

(a) bunny model: 69,451 triangles (e) phone model: 165,936 triangles (i) rotor model: 4,736 triangles



(b) $\epsilon = 1/16\%$, 10,793 triangles (f) $\epsilon = 1/32\%$, 12,364 triangles (j) $\epsilon = 1/8\%$, 2,146 triangles



(c) $\epsilon = 1/4\%$, 2,204 triangles (g) $\epsilon = 1/16\%$, 4,891 triangles (k) $\epsilon = 3/4\%$, 1,266 triangles

(d) $\epsilon = 1\%$, 575 triangles (h) $\epsilon = 1\%$, 412 triangles (l) $\epsilon = 3\ 3/4\%$, 716 triangles

Figure 12: *Level-of-detail hierarchies for three models. The approximation distance, ϵ, is taken as a percentage of the bounding box diagonal.*

Controlled Topology Simplification

Taosong He[‡], Lichan Hong[‡], Amitabh Varshney[‡], and Sidney Wang[*]

[‡]Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400, U.S.A.

[*]Sony-Kihara Research Center, Inc.
1-14-10 Higashigotanda
Shinagawa-ku, Tokyo, 141 Japan

Abstract

We present a simple, robust, and practical method for object simplification for applications where gradual elimination of high frequency details is desired. This is accomplished by converting an object into multi-resolution volume rasters using a controlled filtering and sampling technique. A multi-resolution triangle-mesh hierarchy can then be generated by applying the Marching Cubes algorithm. We further propose an adaptive surface generation algorithm to reduce the number of triangles generated by the standard Marching Cubes. Our method simplifies the topology of objects in a controlled fashion. In addition, at each level of detail, multi-layered meshes can be used for an efficient antialiased rendering.

1. Introduction

Interactive and realistic rendering is of importance in many applications such as scientific visualization and virtual reality. However, there has always been a conflict between the ever larger datasets and the limited rendering capabilities of graphics engines. Object simplification provides one way to reconcile scene realism with interactivity. The basic idea is to use object simplification to automatically generate a multi-resolution object hierarchy, and perform level-of-detail-based rendering. A level-of-detail-based rendering scheme uses the perceptual importance of a given object in the scene to select its appropriate level of representation in the multi-resolution object hierarchy [6, 8, 15]. Thus, higher detail representations are used when the object is perceptually more important and lower detail representations are used when the object is perceptually less significant. This method allows one to achieve higher frame update rates while maintaining good visual realism.

Most existing algorithms in the area of object simplification preserve the object topology [10, 12, 21, 35, 37, 38]. Topology in this context means the properties such as the holes, tunnels, and cavities of an object. Preservation of topology is crucial for certain applications, such as molecular surface modeling, where the presence (or absence) of interior tunnels and cavities in a molecule conveys important structural and chemical information to the biochemist. Clearly, if the target application demands topology preservation, then the simplification algorithm should adhere to it. However, if the primary goal is fast and realistic rendering, such as for virtual reality or some other time critical applications, the topology preservation criterion could stand in the way of efficient simplification.

Let us consider a virtual fly-through in a CAD model. A tiny hole on the surface of a mechanical part in this model will gradually disappear as the observer moves away from the part. However, topology-preserving simplification of this object will retain such features, thereby reducing simplification rates due to limits on the amount of geometry-simplification one can achieve while preserving topology. Another disadvantage is that rendering of a simplified object retaining high

frequency details would increase image-space aliasing due to undersampling, especially in perspective viewing, thereby causing distracting effects such as flickering. On the other hand, by appropriately simplifying the topology of the model, both simplification rates and visual realism can be increased. This idea was previously demonstrated in [8], where a chair was shown at three levels of detail with no preservation of the topology across them.

We therefore classify an object simplification process into the following two stages:
(a) geometry simplification, in which the number of geometry primitives, such as vertices, edges, and faces, is reduced;
(b) topology simplification, in which the number of holes, tunnels, cavities, as well as the number of geometry primitives, is reduced.
Depending upon the target application, these two stages should be performed either independently or jointly. For example, topology simplification itself naturally includes the reduction of geometry primitives, while geometry simplification can be applied on a topology simplified model to further reduce its complexity. However, most of the existing work for object simplification deals exclusively with geometry simplification, and the extension to topology simplification is usually difficult and complicated. The primary goal of our research is to address the topology simplification stage in a simple and robust way, and thereby also help the geometry simplification algorithms to achieve better results for certain applications.

In this paper we present a voxel-based topology simplification algorithm for the generation of multi-resolution object hierarchy, with gradual elimination of high-frequency features including, but not limited to, tiny holes, tunnels, and cavities. In our approach all formats of input objects are first converted into three-dimensional volume rasters by applying a controlled filtering and sampling technique, which is an extension of the volume sampling method proposed by Wang and Kaufman [39]. Then a surface-fitting technique such as Marching Cubes can be applied on the volume rasters to produce simplified polygon meshes. By simply adjusting the size of each voxel, thereby adjusting the resolution of the volume raster, the desired level of detail can be achieved, and consequently, a multi-resolution hierarchy of polygon meshes can be generated.

An earlier version of this work has been presented in [17]. One of the major potential problems left unresolved in [17] is that surface-fitting techniques, such as Marching Cubes, could generate a large number of redundant triangles in the regions of low surface curvature. To alleviate this problem, we adopt the idea of adaptive subdivision of volume space [30], and present in this paper an adaptive Marching Cubes algorithm. Since surface extraction from the multi-resolution volume rasters should preserve the already simplified topology of the model, our adaptive Marching Cubes algorithm guarantees that the simplified mesh is within a given bound of the mesh generated by the standard Marching Cubes.

Although our controlled filtering and sampling technique effectively eliminates the object-space aliasing in the multi-resolution volume representations, both image-space and object-space aliasings are re-introduced when the binary surface-fitting technique is applied. To solve this problem, we have developed a multi-layered triangle mesh rendering algorithm. Our idea is to smooth out the transition between the boundary of an object and empty space surrounding it by using multiple layers of triangle mesh with increasing translucency from the innermost layer to the outermost one. Unlike the earlier antialiasing techniques presented in [1, 5], the prefiltering of the projected objects in image-space is replaced here by a view-independent filtering in object space, which is performed only once in a pre-rendering stage.

The rest of the paper is organized as follows. We first summarize the previous work on object simplification in Section 2. We then present our voxel-based topology simplification in Section 3. We discuss the adaptive Marching Cubes in Section 4, and introduce the multi-layered Marching Cubes for antialiasing in Section 5. We have implemented our algorithm and tested it on several kinds of objects, and we summarize our results in Section 6. Conclusions and some ideas on future work appear in Section 7.

2. Previous Work on Object Simplification

The last few years have seen extensive research in the area of object simplification for level-of-detail-based rendering. Thus far no single algorithm works well for all kinds of objects under all conditions. Some algorithms work best on smooth objects with no sharp edges, whereas others work best for objects that have large areas that are almost coplanar, and yet others are fine-tuned to exploit special object properties such as convexity. It is therefore natural that research on hierarchy generation has evolved around different classes of objects. These are mainly convex polytopes, polyhedral terrains, and arbitrary three-dimensional polygonal objects.

Convex Objects: Automatic simplification of convex objects is now a relatively well-understood subject, due mainly to some recent seminal papers on this topic. It has been shown that computing the minimal-facet approximation within a certain error bound is NP-hard for convex polytopes [9]. Thus algorithms for approximation of convex objects focus mainly on fast heuristics that produce approximations close to the optimal [3, 7, 28].

Polyhedral Terrains: Simplification of polyhedral terrains has been an active area of research for almost two decades because of its considerable importance to the GIS (Geographical Information System) community. It is impossible to do full justice to such a vast area in a mere section. We would, however, like to point interested readers to a recent paper on this topic by Heckbert and Garland [18], for a comprehensive survey of the field.

General Three-Dimensional Objects: Research on simplification of general (non-convex, non-terrain, possibly high genus) three-dimensional objects has spanned the entire gamut of highly local to purely global algorithms, with several approaches in between that have both local and global steps. Local algorithms work by applying a set of local rules, which primarily work under some definition of a *local neighborhood*, for simplifying an object. The local rules are iteratively applied under a set of constraints, and the algorithm terminates when it is no longer possible to apply the local rule without violating some constraint. The global algorithms optimize the simplification process over the whole object, and are not necessarily limited to the small neighborhood regions on the object.

Some of the local rules that have appeared in the literature are mentioned below:

• Vertex Deletion: Delete a vertex with its adjacent triangles and retriangulate the resulting hole. This is used by Schroeder et al. in [35] with some very good results. A generalization of this, deleting several neighboring vertices at once and retriangulating the resulting hole, is proposed by Varshney [38].

• Vertex Collapsing: Merge all the vertices that satisfy a given criterion into one vertex. This is used in conjunction with a global grid by Rossignac and Borrel [32].

• Edge Collapsing: Merge the two vertices of an edge into one, thereby deleting the two adjacent triangles of the edge. This is used as a subroutine in the mesh optimization algorithm by Hoppe et al. [21].

• Polygon Merging: Merge the adjacent coplanar polygons into larger polygons [20].

Some of the global rules that have been used are:

• Uniform Distribution of Fewer Vertices: In a re-tiling approach to simplification outlined by Turk [37], the program first distributes a given number of vertices over the surface and then repositions them based on a global repulsion method to uniformly spread them as a function of the curvature. These new vertices are then retriangulated and the old vertices deleted to obtain the approximation mesh.

• Minimization of Energy Function: Hoppe et al. [21] globally optimize the energy function representing (a) the sum of squared distances from the mesh, (b) the number of vertices, and (c) the edge lengths over the vertices of the newer mesh. The overall optimization procedure alternates between local and global optimization steps.

• Minimization by Set Partitioning: Varshney [38] uses a greedy set-partitioning-based minimization approach to reducing the number of triangles. His method can relate the quality of the approximation produced to the optimal for the same ε tolerance.

• Wavelets: An interesting solution to the problem of polygonal simplification by using wavelets is presented in [12, 26], where arbitrary polygonal meshes are first subdivided into patches with *subdivision connectivity* and then multi-resolution wavelet analysis is used over each patch.

The issue of preservation or simplification of topology is independent of whether an algorithm uses local rules, or global rules, or both to simplify. With the exception of Rossignac and Borrel [32], all other papers cited above preserve the topology of the input object. Preservation of input topology is mathematically elegant and aesthetically pleasing. However, if interactivity is the bottomline, as is often the case in interactive three-dimensional graphics and visualization applications, topology can and should be sacrificed if the topology simplification (a) does not directly impact the application underlying the visualization and (b) does not decrease visual realism. Both of these goals are easier to achieve if the simplification of the topology is finely *controlled* and has a sound mathematical basis. In the next section we outline our approach, which has these properties and is global in nature.

3. Controlled Topology Simplification

In order to better motivate our approach to object simplification, we turn to the classic example of rendering a tilted checkerboard, in which the top of the checkerboard is placed further away from the viewer than the bottom of the board. Once rendered, the Moire patterns are especially noticeable at the top of the image, where too many details from the checkerboard are forced into too few image pixels. This is mainly due to the pyramidal viewing frustum of perspective projection. The same problem occurs when highly detailed objects far from the viewer are rendered. Therefore, our goals for object simplification are twofold. First, we would like to avoid the Moire patterns by gradually eliminating detailed features of an object as it moves away from the viewer. Second, as in the case of most existing level-of-detail algorithms, we would like to increase the frame rate by establishing a multi-resolution object representation, and using simplified models for distant objects.

A flow diagram illustrating our overall object simplification algorithm is outlined in Fig. 1. The algorithm starts by first overlaying the object with a three-dimensional grid and applying a three-dimensional low-pass filter at each grid point. A three-dimensional volume raster data-structure is used to store these filtered grid-point values. Once the filtering and sampling process is completed, a reconstruction process is employed to generate the detail-eliminated object from the set of filtered sample points represented in the volume raster. In this section, we first explain the controlled filtering and sampling process, then discuss the establishment of the hierarchical representation and the smooth transition between levels of detail. The reconstruction process will be presented in Section 4.

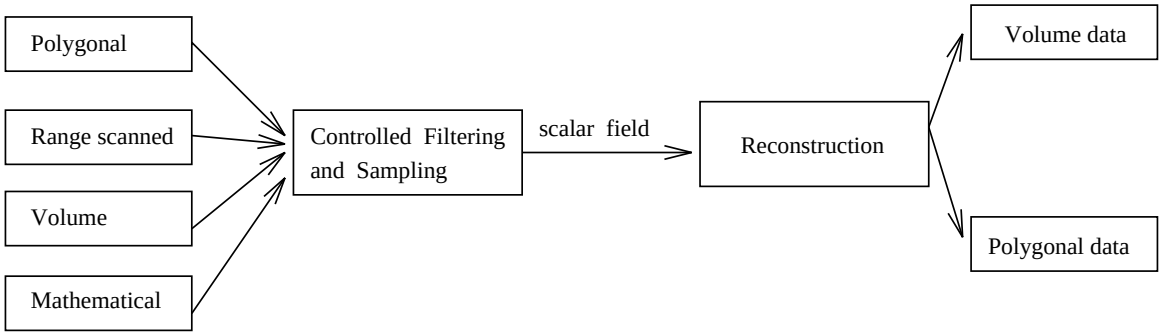


Fig. 1: *Pipeline for controlled topology simplification.*

3.1. Controlled filtering and sampling

To simplify the topology in a controlled fashion, we adopt a signal-processing approach to object detail-elimination by low-pass filtering the object to gradually remove the high frequencies (i.e., detailed features) from the object. The class of input objects that our algorithm can accept and process includes polygonal meshes, volume datasets, objects derived from range-scanners, and algebraic mathematical functions such as fractals. Our algorithm is backed by a sound and elegant mathematical framework of sampling and filtering theory. In fact, a similar sampling and filtering principle has been used extensively in the image processing communities to reduce noise and smooth sharp features in 2D images. Wang and Kaufman [39] generalized the concept into 3D to remove aliasing in volume-based modeling of geometric objects. Our approach utilizes their volume sampling approach, and extends it to incorporate more precise control over the filtering and sampling process. In the following discussion we assume readers are familiar with the basic concepts of sampling and filtering theory. A good reference text is [43].

Fourier analysis tells us that a signal's (or an object's) shape is determined by its frequency spectrum. The more details the signal contains, the richer it is in high-frequency components of its spectrum. Therefore, to gradually eliminate the detail features from an object, we create a smoother signal by removing the offending high frequencies from the original signal. This process is known as low-pass filtering, or band-limiting the signal, and is described mathematically in frequency domain as:

$$FT(f_{new}) = FT(f_{orig}) \cdot H(\nu) \tag{1}$$

where

$$H(\nu) = \begin{cases} 1 & -k \le \nu \le k \\ 0 & otherwise \end{cases}. \tag{2}$$

The more high frequencies we remove (i.e., the smaller the k), the more details that are eliminated from the object. Since the multiplication in the frequency domain corresponds exactly to convolution in the spatial domain, the equations can be rewritten to operate in the spatial domain as:

$$f_{new} = f_{orig} * sinc \tag{3}$$

where $*$ is the convolution operator, and *sinc* is the ideal low-pass filter. Although analytic evaluation of Equation 3 is sometimes possible for objects which are represented by algebraic mathematical functions, for general mathematical functions, polygon meshes, or volume datasets, an analytical solution either does not exist or is too expensive to be calculated. For such cases, a discrete approximation must be used. To minimize the discrete approximation error, the original object is kept in its continuous form while the three-dimensional filter is divided into a number of bins, each having a precomputed filter weight. Note that the resolutions of the bins should be finer than that of the sampling grid to achieve good approximation. These weights are computed by evaluating the filter function at these discrete bin positions and multiplying them by a normalization factor to ensure that the sum of all weights equals unity. Thus, during convolution, a lookup table is used to obtain the corresponding set of weights, which is then applied to the intersected region between the filter kernel and the object. That is, for a grid point (i, j, k) in the volume raster, the resulting filtered density $f(i, j, k)$ is calculated as:

$$f(i, j, k) = \int\int\int h(i-\alpha, j-\beta, k-\gamma)\, S(\alpha, \beta, \gamma)\, d\gamma\, d\beta\, d\alpha \tag{4}$$

where h is a low-pass filter of choice and $S(\alpha, \beta, \gamma)$ is a binary function defined as:

$$S(\alpha, \beta, \gamma) = \begin{cases} 1 & \text{if point } (\alpha, \beta, \gamma) \in \text{object} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Thus an important criterion of our input object is that for a given point (α, β, γ), it can be determined whether that point is inside or outside of the object. For example, only those polygonal meshes that form the boundary of a solid can be treated by the algorithm.

The issues that still remain to be addressed are the determination of the appropriate resolution of the sampling volume raster and the appropriate size of the filter support. From Shannon's sampling theorem, these two variables are directly related to each other. That is, the volume raster must be sampled at a frequency that is greater than twice f_h, the highest frequency component in the signal. This lower bound on the sampling rate is known as the Nyquist rate, or Nyquist frequency *NF*. Suppose that we have a volume raster consisting of $X \times Y \times Z$ sampling resolution, which is used to represent a rectangular volume region of $p \times q \times r$ $unit^3$ of space; then the Nyquist frequency and the sampling frequency f_v of the volume raster are:

$$NF = f_v = \left\{ f_{v_x},\ f_{v_y},\ f_{v_z} \right\} = \left\{ \frac{X}{p}, \frac{Y}{q}, \frac{Z}{r} \right\} \tag{6}$$

Hence, ideally, the cut-off frequency f_g of the low-pass filter must be set to $NF/2$ in order to filter out all offending high frequencies. However, in practice, since the ideal low-pass filter is rarely used, f_h is usually set far less than $NF/2$. In our experiments, approximate filters such as Gaussian filters and hyper-cone filters are often employed [17]. The use of these non-ideal filters results in a combination of frequency leakages and non-unity gains. Fortunately, substantial improvement can be made by filtering out more high frequencies from the objects to allow some error margins caused by the non-ideal filters. This is achieved by the following calculation of f_h:

$$f_h = \frac{NF}{2} \cdot error_{non\_ideal} \tag{7}$$

where $0 < error_{non\_ideal} < 1$. Generally, depending on the type of non-ideal filter used, an $error_{non\_ideal}$

factor between 0.5 and 0.85 achieves satisfactory results. More in-depth discussions on the selection and error estimation of three-dimensional filters can be found in [4, 27]. Given the cut-off frequency f_g, the size of the corresponding low-pass filter support is determined accordingly. High f_h, therefore high resolution of the volume raster, corresponds to small support in the spatial domain, and vice versa.

3.2. Multi-resolution hierarchy

Now that we have established the direct correspondence between the size of the filter support and the resolution of the volume raster, a hierarchical level-of-detail object representation can be easily constructed by controlling the amount of high frequency removed from the spectrum, as one goes from one level of the hierarchy to another. In frequency domain, that is,

$$FT(f_i) = FT(f_{orig}) \cdot H_i(\nu), \qquad 0 \le i \le L \tag{8}$$

and

$$H_i(\nu) = \begin{cases} 1 & -k - \delta \cdot (L - i) \le \nu \le k + \delta \cdot (L - i) \\ 0 & otherwise \end{cases} \tag{9}$$

where f_i represents the i-th level of the level-of-detail hierarchy and L represents the total number of levels.

The base of the proposed hierarchy contains the most detailed and the highest resolution version of the object, and the top contains the blurriest and lowest resolution version of the object. Thus, during volume raster hierarchy construction, as one moves up the hierarchy, the sampling resolution of the volume raster decreases. Consequently, a low-pass filter with wider support is applied. Finally, if desired, a surface-fitting technique can be used to reconstruct a polygon mesh model for each level of the volume raster hierarchy. During rendering, the appropriate level of the hierarchy is selected for each object in the scene. The heuristic that we have used is that the footprint of each filtered sample point covers approximately one and a half times the area of a pixel.

Furthermore, in order to reduce temporal aliasing during animation, smooth interpolation between two adjacent resolution meshes should be generated, which is generally a non-trivial task. However, in our algorithm, an arbitrary integer resolution volume raster can be generated by adjusting the low-pass filter support. It is also straightforward and efficient to directly interpolate between two adjacent resolution volume rasters to generate an in-between resolution volume raster. This is achieved by first linearly interpolating the resolution of the volume rasters at adjacent levels. Then the density at a grid point of the in-between volume raster is decided by linearly interpolating the densities at the corresponding positions of the two adjacent resolution volume rasters. A two-dimensional example of this process is demonstrated in Fig. 2.

The topology simplification algorithm presented above is simple, robust, and widely applicable. By continuously adjusting the filter support, the user can control the elimination of appropriate amount of high frequency in the model. The desired passband for filtering can be precisely calculated according to the distance from the model to the eyepoint. At first glance, it might seem somewhat similar to the clustering scheme [32] or the three-dimensional "mip-map" approach [24, 33]. However, our approach follows a control-based filtering for gradual elimination of high frequencies, which is different from the locality-based clustering of geometry as presented in [32]. As for the three-dimensional "mip-map" approach, every level of the hierarchy is formed by averaging several voxels

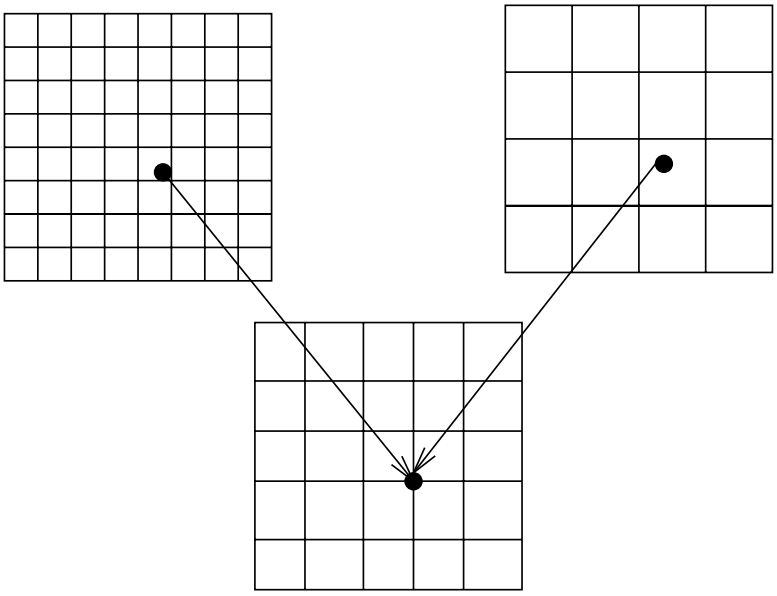


Fig. 2: *Interpolation between two adjacent resolution volume rasters*.

from the previous level. In our approach, every level of the volume raster hierarchy is created by convolving the original object with a low-pass filter of an appropriate support, whose size can theoretically be any positive real number. Thus, errors caused by a non-ideal filter do not propagate and accumulate from level to level. Furthermore, the sampling resolution of our volume raster hierarchy need not be the same for all three axes nor even be required to be a power of two.

4. Surface Reconstruction

Once the filtered value for each volume raster grid point is generated, surface-fitting techniques can be used to reconstruct the isodensity surfaces, if desired. Alternatively, if a polygonal model is not required for the simplification result, one can delay the reconstruction process until rendering, instead of reconstructing the isodensity surface using polygonal elements. In other words, if the direct volume rendering [23, 40] technique is employed to render the volume raster of filtered values, then the reconstruction process is done during rendering, without resorting to intermediate polygonal representation. In this paper, we focus on the polygon reconstruction.

Marching Cubes, originally proposed by Lorensen and Cline [25], has been considered the standard approach to surface extraction from a volume raster of scalar values. In this algorithm, an isodensity surface is approximated by determining its intersections with edges of every voxel in the volume raster. Up to five triangles are used to approximate the surface within a voxel, depending on the configurations of the voxel with respect to an isodensity value. One advantage of Marching Cubes is that it can be efficiently implemented using a precomputed lookup table for the various arrangements of surface-voxel intersections. However, despite its extensive applications, the original algorithm proposed by Lorensen and Cline [25] has some particular problems, in turn provoking substantial research aimed at the solutions. One of the problems is that the 15 basic configurations proposed in [25] are incomplete, and could generate topology inconsistent surfaces due to the ambiguities [11]. Several solutions have been proposed to add additional configurations [31, 41].

Recently, Schroeder, Martin, and Lorensen have published the Visualization Toolkit [36]. It contains an implementation of a topology-consistent Marching Cubes based on a complete set of 256 configurations. Since this implementation is simple and available, it has been adopted in this paper. Another problem of Marching Cubes is that the time of computation spent for empty voxels with no surface intersection could be considerable. It can be solved by avoiding the visiting and testing of empty regions [42].

For the purpose of object simplification, the number of triangles generated by Marching Cubes is particularly important. Since the maximum size of the triangles is limited by the regular grid spacing of the volume raster, there could be excessive fragmentation of the output data even in the area of low curvature. The solutions proposed to solve this problem can be classified into either filter-based or adaptive techniques. A filter-based technique starts with a large number of primitives and removes or replaces them to reduce the model size. For example, Montani et al. [29] discretize the intersection points between the surface and the edges of voxel and merge the coplanar facets. Schroeder et al. [35] have proposed a decimation algorithm based on multiple filtering passes and vertex deletion. It analyzes the geometry and topology of a triangle mesh locally and recursively removes vertices that pass a minimum distance or curvature-angle criterion. The advantage of this approach is that any level of reduction can be obtained, on the condition that a sufficiently coarse approximation threshold is set. Kalvin and Taylor [22] have proposed a *superface* algorithm by merging faces. It guarantees a bounded approximation and can be applied on any polyhedral mesh that is a valid manifold. On the other hand, adaptive techniques produce more primitives in selected areas, such as an area with highly detailed features. For example, Schmitt [34] starts with a rough bi-cubic patch approximation to sample data, and then subdivides those patches that are not sufficiently close to the underlying samples. Adaptive techniques have been used for terrains [13], implicit modeling [2], and general polygon meshes [10].

As mentioned in the introduction, the surface-fitting technique used for our algorithm must preserve the topology of the model, since the topology simplification has been appropriately achieved in the stage of controlled filtering and sampling. One method is to first generate a triangle mesh using the standard Marching Cubes, and then apply the existing topology preserving geometry simplification algorithm on the mesh. As an alternative, we adopt the adaptive idea and propose a simple algorithm based directly on Marching Cubes. The basic idea is to adapt the size of the generated triangles and hence their number to the shape of the isodensity surface. To achieve this, Muller and Stark [30] have proposed a Splitting-box algorithm. Our *adaptive Marching Cubes* is based on Splitting-box, but with some major improvements. In this section, we first introduce the Splitting-box algorithm, following the description in [30], and then discuss the difference between it and our algorithm.

4.1. Splitting-box algorithm

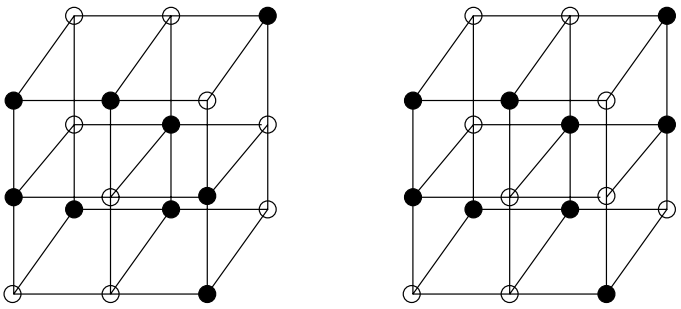
The input of the Splitting-box algorithm, like Marching Cubes, consists of a regular 3D grid of scalar values and an isodensity value for the surface. A vertex of the grid is called *black* if its value is greater than or equal to the isodensity, and *white* otherwise. A box is a rectangular parallelepiped, whose edges are induced by linear sequences of vertices of the grid. For Marching Cubes, the assumption is that the box edge length is always 2, and the surface has one and only one intersection at the box edges whose vertices are of different colors. For Splitting-box, a box can have longer edges, but the second assumption still holds. An edge is called *MC* if the color changes at most once along its grid vertex sequences. A face is called *MC* if its four edges are *MC* edges, and a box is *MC* if its six

faces are *MC* faces., An example of an *MC* box is shown in Fig. 3a.

Splitting-box starts with the box given by the input grid. This box is bisected perpendicular to its longest edge into two sub-boxes. The process of bisection is recursively performed until a 2×2×2 box is reached. Meanwhile, the bisection is postponed for the boxes arising during the bisection process if they are recognized *MC*. Instead, triangle chains are generated for such boxes, according to the rules of Marching Cubes configurations. One important point here is that a triangle vertex on an *MC* edge is interpolated between the pair of consecutive vertices of different colors on the edge similar to Marching Cubes. Thus, the vertex coincides exactly with the one generated by Marching Cubes. After the generation of the triangle chains for an *MC* box, the bisection is continued to check the quality of approximation of the triangle chains with respect to the true triangle chains both on the faces of and inside the *MC* box. If the approximation is acceptable, it will be part of the output. Otherwise, the chain is discarded, and a new approximation is tested in the sub-boxes. A satisfactory approximation is generated, at worst, at the level of a 2×2×2 box.

The purpose of checking the quality of the approximation of the triangle chains is to preserve the topology. With the bisection of an *MC* box B into two sub-boxes B_1 and B_2, if one of B_1 and B_2 is not *MC*, then essentially the approximation of B will not be acceptable. If both B_1 and B_2 are *MC* boxes, then a satisfactory approximation of B must satisfy the following two criteria. First, the intersections between the chains in B and the new edges on the common face of B_1 and B_2 must lie between a pair of consecutive vertices with different colors. If this condition is satisfied, the respective triangle vertices in B_1 and B_2 are replaced by the intersection point. Second, the triangle chain of B must be geometrically coincident with those of B_1 and B_2. Using these criteria, Splitting-box preserves the exact separation of black and white vertices, and guarantees that the topology of the surface coincides with and is not more than the sampling distance apart from the Marching Cubes solution.

Splitting-box provides a simple and practical framework for reducing the number of triangles generated by Marching Cubes. One of the problems of this approach, however, is that the algorithm achieves only a fixed-bound approximation. In other words, the Splitting-box mesh is always within sampling distance of the Marching Cubes mesh, and this bound can not be changed by the users. This is due to the requirement to preserve the exact separation of black and white vertices. This requirement also limits the possible reduction of polygons, even with the sampling distance bound.



(a) (b)

Fig. 3: *(a) An MC box for Splitting Box. (b) An AMC box for adaptive Marching Cubes*.

4.2. Adaptive Marching Cubes

To solve this problem, we propose an adaptive Marching Cubes algorithm. By slightly change the definition of *MC* of Splitting-box, we propose the concept of *AMC*. The definition of *AMC* edges and *AMC* faces are the same as of Splitting-box. However, a box in a 3D grid is *AMC* if and only if all the edges induced by linear sequences of grid vertices on the face and inside the box are *AMC* (Fig. 3b). In other words, when we define an *AMC* box, we not only consider the faces, but also the interior. Therefore, together with the quality checking process discussed below, we guarantee that a satisfactory approximation in an *AMC* box does not affect the topology of the model.

The adaptive Marching Cubes algorithm follows a similar bisection process to Splitting-box. The process is recursively performed, but postponed to generate the triangle chain according to Marching Cubes configurations when a box is recognized *AMC*. The quality of the triangle chain approximation of this *AMC* box is then checked, and bisection is continued if the approximation is not satisfactory.

However, there are several important differences between the two algorithms. First, for Splitting-box, a box is always bisected perpendicular to its longest edge. In some situations, this could cause unnecessary bisections. For example, if all the edges along a certain axis are *AMC* in a non-*AMC* box, bisecting along this axis would still generate *non-AMC* boxes. On the other hand, for adaptive Marching Cubes, only an *AMC* box is still bisected perpendicular to its longest edges. For non-*AMC*, we first find those axes along which there is at least one non-*AMC* edge, then bisect perpendicular to the longest edges along those axes. An example is shown in Fig. 4, where more than three bisections must be performed to make all the sub-boxes *AMC* using the Splitting-box approach, while only one bisection is needed perpendicular to the shortest edge along the Z axis using the adaptive Marching Cubes approach.

The major difference between the two algorithms lies in the quality checking process. Given a user-specified bound ε, the goal of our algorithm is to satisfy the following conditions:
(1) The set of adaptive Marching Cubes generated mesh vertices is a subset of the set of Marching Cubes generated mesh vertices.
(2) The topology of the standard Marching Cubes generated mesh is preserved in the *adaptive Marching Cubes* generated mesh.
(3) All the vertices of the standard Marching Cubes generated mesh are within ε distance of the

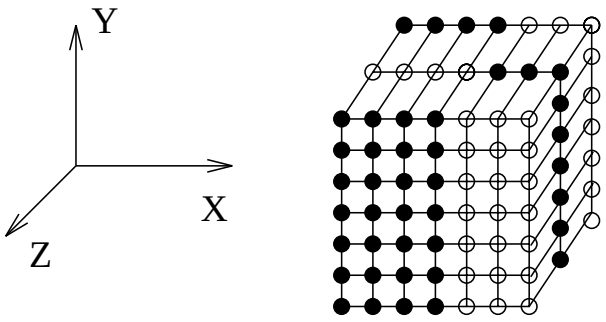


Fig. 4: *Bisecting of a non-AMC Box*.

adaptive Marching Cubes generated mesh.

The first condition is met through the method of generating the approximating triangle chain in an *AMC* box, and the second condition is satisfied by approximating only the mesh in an *AMC* box. To satisfy the third condition, for each *AMC* box *B*, we test the distance between all the Marching Cubes vertices to the approximating chain. To check whether the distance between a point and a triangle is within a certain bound, we conservatively approximate the distance by calculating the distance between the point and the triangle along X, Y, and Z axes. Since quality checking is always performed between the approximating chain and the Marching Cubes vertices, it can be achieved without the recursive bisection of Splitting-box.

A potential problem for adaptive approximation is the cracks generated on the common face between different levels of the hierarchy. An example of the possible cracks is shown in Fig. 5a. To make things clear, we present only triangles related to the cracks in Fig. 5. For Splitting-box, the crack is first detected, then eliminated by exploiting the restriction that the intersections between the approximation chains in an *MC* box and the common faces of the sub-boxes lie between a pair of consecutive vertices with different colors. Since this restriction is released for adaptive Marching Cubes, we develop a simple *stitching* algorithm for the elimination of the cracks. The basic idea is to first find those potential crack edges on the common faces, then retriangulate them to stitch the cracks. The edge on a common face between B_1 and B_2 is defined as *good* if it appears in more than one triangle in the union of B_1 chain and B_2 chain, and *crack* otherwise. The retriangulation is performed among the triangles with at least one crack edge. An example of such retriangulation is shown in Fig. 5b, where one triangle in the coarse resolution *AMC* box on the right is split into two triangles. Notice that many additional triangles could be generated using this simple stitching algorithm. Another possible retriangulation with fewer triangles but more complicated implementations is shown in Fig. 5c, where the position of the corresponding vertices of the triangles in the fine resolution *AMC* box on the left is moved.

5. Multi-Layered Marching Cubes Rendering

The controlled filtering and sampling technique discussed in Section 3 effectively eliminates the high frequencies above the Nyquist frequency in the multi-resolution volume rasters. They can be appropriately rendered through antialiased volume rendering. However, as discussed above, surface-fitting techniques, such as our adaptive Marching Cubes, are sometimes needed to generate an isodensity surface from the volume representations. Generally, these techniques apply a binary surface classification to extract an isodensity surface from the 3D grid. Although the isodensity surface is considered to be good from the point of view of modeling, it does introduce infinitely high frequencies. Since these frequencies cannot be fully represented by a discrete image, they can cause image-space aliasing. As an alternative to the commonly used hardware-supported antialiasing for rendering, we have developed a multi-layered Marching Cubes antialiased rendering algorithm. This approach takes advantage of the low-pass filtering applied during the controlled filtering and sampling stage. The non-binary surface classifier that we have used permits surfaces to be associated with a continuous range of densities, thereby allowing a smooth transition from the object-boundary to the empty space. Another motivation for this approach is to more appropriately represent the low-pass-filtered models using polygon meshes. Since a simplified model is represented in a volume raster with densities ranging continuously from 0 to 1, a single layer of surface with one isodensity sometimes cannot always produce a good simplified effect, especially when the object is far away and a very low

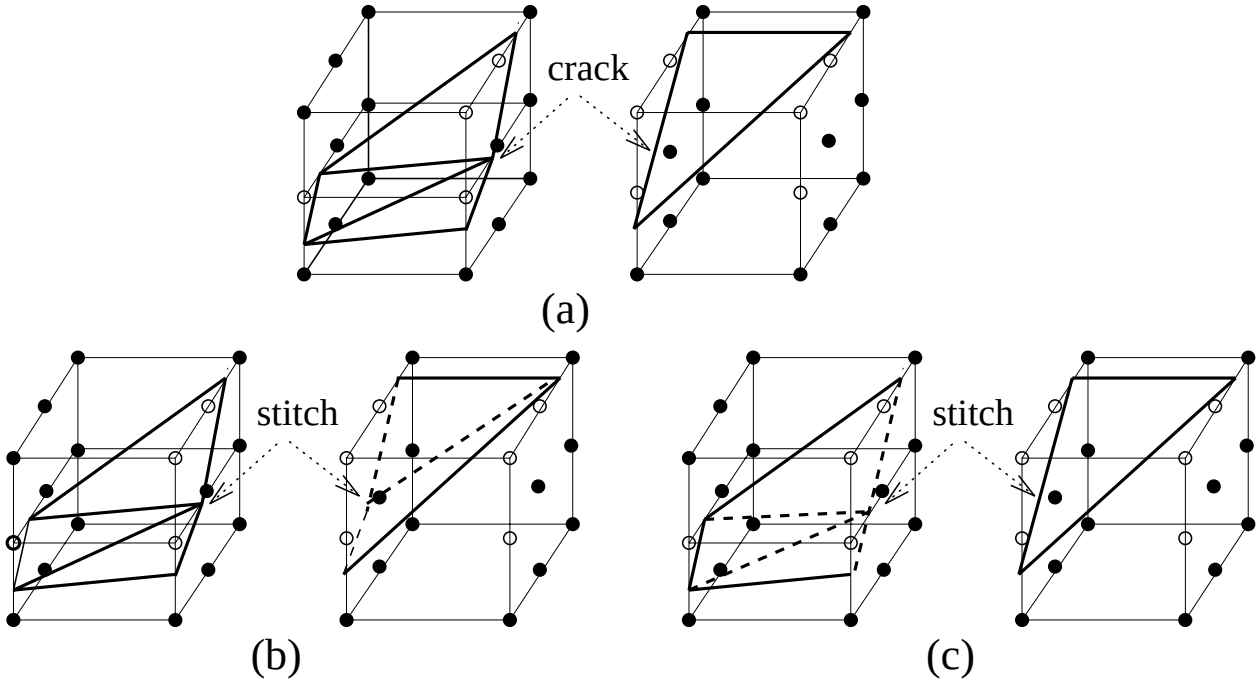


Fig. 5: *Crack Stitching: (a) Crack, (b, c) Examples of solutions.*

resolution volume raster is used. For these situations, object-space aliasing introduced by surface-fitting needs to be reduced.

Besides our work in [17], the idea of utilizing multiple layers of triangle meshes generated by Marching Cubes has been independently proposed by Heidrich et al. [19] for the purpose of interactive maximum projection. However, their algorithm is not concerned about the composition of semi-transparent layers. Fujishiro et al. [14] have generalized Marching Cubes to handle an interval volume with isodensities falling into a close interval $[\alpha, \beta]$. Their algorithm generates polyhedra, instead of triangles, but still cannot handle semi-transparent interval volumes. Guo [16] has proposed using

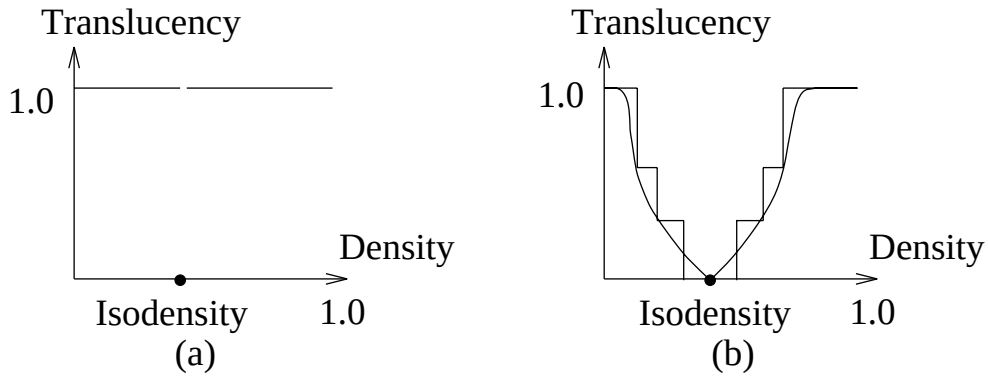


Fig. 6: *(a) Binary surface classification. (b) Continuous surface classification and discrete approximation.*

α-shape to approximate the interval volumes. The advantage of his approach is that the α-shape can be rendered as RGBA clouds, thereby producing a correct semi-transparent effect. However, too many tetrahedra would be generated if the interval $[\alpha, \beta]$ is large.

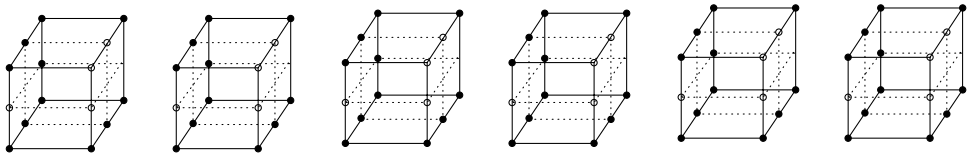
The basic idea of our algorithm is to discretely approximate the surface boundary by generating several layers of triangle meshes using Marching Cubes, with increasing translucency from the innermost layer to the outermost one (Fig. 6b). Then, by appropriately compositing these layers of triangle meshes using hardware-assisted blending, a high frame rate of antialiased rendering can be achieved. The accuracy of the discrete approximation of the continuous surface classification is determined by the number of mesh layers used and their corresponding isodensities. Better approximation can be achieved with more layers, at the cost of increased storage space and rendering time. The minimum number of layers needed for the approximation to be within a user-specified error bound depends upon both the low-pass filter employed to generate the volume raster and the geometry of the original polygon mesh (e.g., Fig. 7). However, it can be computed approximately by the following method.

First, it is assumed that the translucency of a point with a certain density d is

$$1 - \frac{d}{m} \tag{10}$$

where m is the maximum isodensity value associated with the innermost layer M of the multi-layered surfaces. Therefore, the problem of approximating a translucency function is simplified into the approximation of a density function. Then, by assuming that the density of a point is decided solely by its distance to M, we can approximate the density at every point. Mathematically, centering the low-pass filter h with support R at a point with distance r from M, and assuming the filter intersects a planar surface (Fig. 7a), the density of this point is:

$$d(r) = \int_r^R \int_{-\sqrt{R^2-\alpha^2}}^{\sqrt{R^2-\alpha^2}} \int_{-\sqrt{R^2-\alpha^2-\beta^2}}^{\sqrt{R^2-\alpha^2-\beta^2}} h(\alpha, \beta, \gamma) d\gamma \, d\beta \, d\alpha \tag{11}$$



• ○

Fig. 7: *Different intersections between a surface (solid) and the filter (dashed).*

Therefore, given an error bound ε, the minimum number of layers needed and their corresponding isodensities are decided by a piecewise constant function p with a minimum number of segments which satisfies:

$$\int_0^R |p(x) - d(x)|\, dx \le \varepsilon \tag{12}$$

The optimal piecewise constant function p might not be analytically derivable for certain filters. However, by discretely approximating d, and using the heuristic that more layers should be placed where the change of the function d is high, sub-optimal p can be recursively generated. The number of layers of triangle meshes is then equal to the number of segments in the function p, and the isodensities of the meshes are the corresponding constants of that function. In addition, the corresponding translucency can be computed using Equation 10.

In order to generate the correct composition of semi-transparent meshes, the triangles should be projected in a back-to-front or a front-to-back order, either of which generally involves an expensive sorting process. A nice property of a Marching Cubes generated mesh is that it is associated with a volume raster. As a result, sorting can be accomplished by traversing only the surface-intersected voxels in a slice-by-slice fashion. However, because of the generation of multiple layer meshes with different isodensities, when adaptive Marching Cubes is applied, the sorting is not always possible. Projection of multiple objects is even more complicated. One simple solution is to perform the sorting on bounding boxes of the volume rasters associated with the objects. A more accurate and still efficient sorting algorithm takes advantage of the volume rasters associated with the meshes of these objects, since intersections among the regularly partitioned volume rasters are easy to compute. Therefore, all the surface-intersected voxels can be rapidly traversed in an almost correct order.

6. Results

We have implemented our controlled topology simplification algorithm and applied it on a variety of objects. The results have been very encouraging and are summarized below. All the experiments were conducted on a Silicon Graphics Onyx VTX, equipped with two 100Mhz R4400 processors and 128MB of RAM. Only one of the processors was used.

An interesting feature of our voxel-based topology simplification algorithm is that it can simplify not only individual objects but also collections of objects. This is achieved by filtering and sampling the object cluster into one volume raster hierarchy. Fig. 8 illustrates the triangle-mesh hierarchy of a fractal ellipsoid-flake with 820 ellipsoids in the original model. The original triangle mesh, shown in Fig. 8a, is reconstructed from a high resolution volume raster to preserve the details. By convolving the original fractal functions with Gaussian filters with different radius supports, we decrease the resolution of volume rasters accordingly, and the resulting number of triangles in the simplified mesh is reduced. The simplification results are presented in Table 1, with the index specifying the corresponding image in Fig. 8. The discrete approximations of the applied Gaussian filters are at resolution $11 \times 11 \times 11$. The surfaces have been reconstructed using standard Marching Cubes from multi-resolution volume rasters using an isodensity of 0.5 on a normalized scale of 0 to 1. The running time is from several seconds to several minutes.

To further reduce the number of triangles in the simplified model, we have applied our adaptive Marching Cubes on the multi-resolution ellipsoid-flake volume rasters with different approximation

Table 1: *Simplification of a fractal ellipsoid-flake*

Index	Resolution	Triangles
a	200×200×200	320455
b	100×100×100	64687
c	50×50×50	13589
d	30×30×30	3746
e	15×15×15	640
f	5×5×5	8

bounds. Fig. 9 illustrates the results of applying this geometry simplification algorithm on the volume raster with 200×200×200 resolution. The approximation bound is 0.0 in Fig. 9a, 0.5 in Fig. 9b, 1.0 in Fig. 9c, and 2.0 in Fig. 9d, and the running times are 190sec, 260sec, 190sec, and 163sec, respectively. The simplification results of all the multi-resolution ellipsoid-flake volume rasters are presented in Table 2. In this example we use the simple stitching algorithm as presented in Fig. 5b.

Fig. 10 demonstrates a mechanical part generated by CSG operations using volume-sampled voxelized primitives [39]. The level-of-detail meshes established by applying Gaussian filters of different radius supports are presented in Table 3, with the index specifying the corresponding image in Fig. 10. The surfaces have been reconstructed with Marching Cubes from multi-resolution volume rasters using an isodensity of 0.5 on a normalized scale of 0 to 1. These images have been rendered using a solid steel texture. From these results, it can be seen that our algorithm provides a controlled way to gradually reduce the genus and small features. Fig. 11 presents the effect of the simplification on an assembly of identical mechanical parts at different resolutions, as shown in Fig. 10. The selection of resolution depends on the distance of the parts from the viewpoint. To reduce the temporal aliasing, we apply the smooth interpolation algorithm as described in Section 3. The effect is illustrated in Fig. 12, where the interpolation is performed between the 100×100×60 volume raster (top left) and 50×50×30 volume raster (bottom right). It should be noted that to generate the polygon mesh from interpolated volume rasters using Marching Cubes, only those voxels which might contain surfaces are examined. An interpolated voxel might contain a surface only if at least one of the corresponding regions in the two volume rasters contains a surface, or exactly one of the

Table 2: *Triangle number of a multi-resolution fractal ellipsoid-flake mesh using adaptive Marching Cubes*

Bound	0.0	0.5	1.0	2.0
Resolution				
200×200×200	321400	108103	79338	71364
100×100×100	64460	22519	16117	13267
50×50×50	13348	6500	4740	4199
30×30×30	3692	1813	1175	1033
15×15×15	620	314	217	205
5×5×5	4	4	4	4

corresponding regions in the two adjacent resolution volume rasters is inside the surface. Such voxels can be efficiently generated since the regions in the two volume rasters satisfying above conditions can be pre-computed.

We have also applied our algorithm on volumetric datasets. Fig. 13 presents the result of simplifying the head and neck of the Visible Man fresh CT data [44]. The data is first aligned and downsampled from 512×512×217×16 bits to 256×256×117×8 bits. Then, the simplified meshes reconstructed using Marching Cubes are presented in Table 4, with the index specifying the corresponding images in Fig. 13.

Unlike the volume raster generated from a solid object, a sampled or simulated volumetric dataset generally does not have a well-defined surface. However, for a given point, it is still possible to test whether this point is inside or outside the surface by tri-linearly interpolating the point value from the neighboring eight vertices and comparing it to the isodensity, and therefore Equations 4 and 5 can still be applied. Another method of simplifying volumetric datasets without well-defined surfaces is to directly apply the reconstruction filters with different radius supports to the original volumes. The application of 3D reconstruction filter for volumetric datasets has been previously discussed for volume rendering [40].

We also tested our adaptive Marching Cubes algorithm on the multi-resolution head and neck volume rasters with different approximation bounds. Fig. 14 illustrates the results of applying the algorithm on the original model with 256×256×225 resolution. The approximation bound is 0.0 in Fig. 14a, 0.5 in Fig. 14b, 1.0 in Fig. 14c, and 2.0 in Fig. 14d, and the running times are 223sec, 416sec, 312sec, and 280sec, respectively. The simplification results of all the multi-resolution volume

Table 3: *Simplification of a CSG mechanical part*

Index	Resolution	Triangles
a	200×200×120	271504
b	100×100×60	64344
c	50×50×30	13292
d	40×40×24	8660
e	20×20×12	1508
f	5×5×3	88

Table 4: *Simplification of the head and neck of Visible Man Fresh CT*

Index	Resolution	Triangles
a	256×256×117	334564
b	192×192×88	180996
c	128×128×59	76088
d	64×64×30	16852
e	32×32×15	3284
f	16×16×8	568

rasters are presented in Table 5. In this example, we apply the complicated stitching algorithm as presented in Fig. 5c.

The effect of our antialiasing algorithm is demonstrated by employing five layers of meshes on a bolt, shown in the bottom half of Fig. 15, and contrasted with the aliased result of applying the traditional algorithm with binary surface classification shown at the top half of Fig. 15. The antialiased effect can be clearly seen in the zoom view. Fig. 16 presents another example of applying the multi-layered Marching Cubes rendering on a lamp cover. It should be emphasized that the multi-layered Marching Cubes rendering generally requires more memory, and the rendering speed might be slower than other hardware-supported antialiasing algorithms. However, it provides a competitive object-space antialiasing method, and is quite useful when a high-quality antialiasing effect is required. It also helps the algorithm more appropriately represent a filtered model.

7. Conclusions and Future Work

Object simplification is an important research area for interactive applications. While most of the existing work focuses on geometry simplification, we have outlined in this paper a practical and robust method for topology simplification by controlled filtering and sampling an object into alias-free multi-resolution volume rasters. The strengths of our method are that it (a) works for a wide variety of objects; (b) simplifies the object topology in a controlled way; (c) is relatively easy to implement; and (d) is based on the robust theoretical foundation of signal-processing theory. To reduce the potentially large number of redundant triangles generated by the traditional surface-fitting algorithms, we have presented an adaptive Marching Cubes, which adheres to the topology preservation criterion, and guarantees that the generated mesh is within the user-specified error bound of the mesh generated by the standard Marching Cubes. To overcome the problems caused by the binary surface classification, we have further introduced a multi-layered Marching Cubes algorithm for hardware-assisted antialiasing.

Surface generation from multi-resolution volume rasters is an important step in our object simplification process. Our adaptive Marching Cubes, as well as the other existing geometry simplification algorithms, can be used to simplify the geometry of a model as a postprocess of the topology simplification stage. As part of our ongoing research in this area, we are currently developing a method of controlled low-pass filtering and sampling a polygon mesh or other formats of object into

Table 5: *Triangle number of a multi-resolution head and neck mesh using adaptive Marching Cubes*

Bound	0.0	0.5	1.0	2.0
Resolution				
256×256×256	333259	102253	75739	68859
192×192×88	180310	64181	47357	42853
128×128×59	75790	30739	23278	21452
64×64×30	16784	7935	6254	5820
32×32×15	3236	1654	1326	1273
16×16×8	536	276	248	242

volume raster with adaptive size voxels, where the high curvature areas are represented by small voxels and the smooth areas by large voxels. The adaptive Marching Cubes is then modified to combine the topology and geometry simplification into one stage.

One of the restrictions of our algorithm, as mentioned in Section 3.1, is that it only works properly for closed surfaces. A possible solution for open polygonal patches is to first close them with dummy patches. Another area that promises to be of interest, and one that we are currently exploring, is the use of multi-resolution object hierarchies for collision detection. The idea here is to recursively perform collision detection among the multi-resolution descriptions of objects, starting from the lowest resolution representations and moving up to the higher resolutions only when an intersection is suspected. This approach works because every time a low-pass filter is applied with a larger support, the area affected by it becomes a superset since a larger filter support is applied. Thus, computation time is saved by avoiding intersection detection in regions that cannot possibly collide. Furthermore, this hierarchical approach can be interrupted, allowing users to trade accuracy for speed.

Acknowledgments

This work has been partially supported by the National Science Foundation under grants CCR-9205047 and CCR-9502239 and by the Department of Energy under the PICS grant. We thank Arie Kaufman for his contribution to the original ideas of this project. The source of the Visible Human data set is the National Library of Medicine and the Visible Human Project.

References

1. Amanatides, J., ‘‘Ray Tracing with Cones’’, *Computer Graphics (SIGGRAPH ’84 Proceedings)*, **18**, 3 (July 1984), 129-135.
2. Bloomenthal, J., ‘‘Polygonization of Implicit Surfaces’’, *Computer Aided Geometric Design*, **5**, (1988), 341-355.
3. Bronnimann, H. and Goodrich, M., ‘‘Almost optimal set covers in finite VC-dimension’’, *Proceedings Tenth ACM Symposium on Computational Geometry*, 1994, 293-302.
4. Carlbom, I., ‘‘Optimal Filter Design for Volume Reconstruction and Visualization’’, *IEEE Visualization’93 Proceedings*, San Jose, CA, October 1993, 54-61.
5. Carpenter, L., ‘‘The A-buffer, an Antialiased Hidden Surface Method’’, *Computer Graphics (SIGGRAPH ’84 Proceedings)*, **18**, 3 (July 1984), 103-108.
6. Clark, J., ‘‘Hierarchical Geometric Models for Visible Surface Algorithms’’, *Communications of the ACM*, **19**, 10 (1976), 547-554.
7. Clarkson, K. L., ‘‘Algorithms for Polytope Covering and Approximation’’, *Proc. 3rd Workshop Algorithms Data Structure, Lecture Notes in Computer Science*, 1993.
8. Crow, F. C., ‘‘A More Flexible Image Generation Environment’’, *Computer Graphics (SIGGRAPH ’82 Proceedings)*, **16**, 3 (1982), 9-18.
9. Das, G. and Joseph, D., ‘‘The complexity of minimum convex nested polyhedra’’, *Proc. 2nd Canad. Conf. Comput. Geom.*, 1990, 296-301.
10. DeHaemer, Jr., M. and Zyda, M. J., ‘‘Simplification of objects rendered by Polygonal Approximations’’, *Computers and Graphics*, **15**, 2 (1991), 175-184.

11. Durst, M., ‘‘Letters: Additional Reference to Marching Cubes’’, *Computer Graphics*, **22**, 2 (1988), .

12. Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M. and Stuetzle, W., ‘‘Multiresolution Analysis of Arbitrary Meshes’’, *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH*, August 1995, 173-182.

13. Fowler, R. and Little, J., ‘‘Automatic Extraction of Irregular Network Digital Terrain Models’’, *Computer Graphics*, **13**, 2 (August 1979), 199-207.

14. Fujishiro, I., Maeda, Y. and Sato, H., ‘‘Interval Volume: A Solid Fitting Technique for Volumetric Data Display and Analysis’’, *IEEE Visualization’95 Proceedings*, Atlanta, GA, October 1995, 151-158.

15. Funkhouser, T. A. and Sequin, C. H., ‘‘Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments’’, *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH*, August 1993, 247-254.

16. Guo, B., ‘‘Interval Set: A Volume Rendering Technique Generalizing Isosurface Extraction’’, *IEEE Visualization’95 Proceedings*, Atlanta, GA, October 1995, 3-10.

17. He, T., Hong, L., Kaufman, A., Varshney, A. and Wang, S., ‘‘Voxel-Based Object Simplification’’, *IEEE Visualization’95 Proceedings*, Atlanta, GA, October 1995, 296-303.

18. Heckbert, P. and Garland, M., ‘‘Fast Polygonal Approximation of Terrains and Height Fields’’, *Technical Report CMU-CS-95-181*, September 1995.

19. Heidrich, W., McCool, M. and Stevens, J., ‘‘Interactive Maximum Projection Volume Rendering’’, *IEEE Visualization’95 Proceedings*, Atlanta, GA, October 1995, 11-18.

20. Hinker, P. and Hansen, C., ‘‘Geometric Optimization’’, *IEEE Visualization’93 Proceedings*, San Jose, CA, October 1993, 189-195.

21. Hoppe, H., DeRose, T., Duchamp, T., McDonald, J. and Stuetzle, W., ‘‘Mesh Optimization’’, *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH*, August 1993, 19-26.

22. Kalvin, A. D. and Taylor, R. H., ‘‘SuperFaces: Polyhedral Approximation with Bounded Error’’, *Technical Report RC 19808*, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10958, 1994.

23. Levoy, M., ‘‘Display of Surfaces from Volume Data’’, *IEEE Computer Graphics and Applications*, **8**, 5 (May 1988), 29-37.

24. Levoy, M. and Whitaker, R., ‘‘Gaze-Directed Volume Rendering’’, *Computer Graphics (Proc. 1990 Symposium on Interactive 3D Graphics)*, **24**, 2 (March 1990), 217-223.

25. Lorensen, W. E. and Cline, H. E., ‘‘Marching Cubes: A High Resolution 3D Surface Construction Algorithm’’, *Computer Graphics (SIGGRAPH ’87 Proceedings)*, **21**, 4 (July 1987), 163-169.

26. Lounsbery, M., DeRose, T. D. and Warren, J., ‘‘Multiresolution Analysis for Surfaces of Arbitrary Topological Type’’, *Tech. Rep. 93-10-05B*, University of Washington at Seattle, January 1994.

27. Machiraju, R. and Yagel, R., ‘‘Accuracy Control of Reconstruction Errors in Volume Slicing’’, *Biomedical Visualization Proceedings’95*, Atlanta, GA, October 1995, 50-57.

28. Mitchell, J. and Suri, S., ‘‘Separation and approximation of polyhedral surfaces’’, *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms*, 1992, 296-306.
29. Montani, C., Scateni, R. and Scopigno, R., ‘‘Discretized Marching Cubes’’, *IEEE Visualization’94 Proceedings*, Washington, D.C., 1994, 281-287.
30. Muller, H. and Stark, M., ‘‘Adaptive generation of surface in volume data’’, *The Visual Computer*, 1993, 182-199.
31. Payne, B. and Toga, A., ‘‘Surface Mapping Brain Functions on 3D models’’, *IEEE Computer Graphics and Applications*, **10**, 2 (July 1992), 41-53.
32. Rossignac, J. and Borrel, P., ‘‘Multi-Resolution 3D Approximations for Rendering Complex Scenes’’, in *Modeling in Computer Graphics*, B. Falcidieno and T. L. Kunni, (eds.), Springer-Verlag, 1993, 455-465.
33. Sakas, G. and Hartig, J., ‘‘Interactive Visualization of Large Scalar Voxel Fields’’, *IEEE Visualization’92 Proceedings*, Boston, MA, October 1992, 29-36.
34. Schmitt, F. J., Barsky, B. A. and Du, W., ‘‘An Adaptive Subdivision Method for Surface-fitting from Sample Data’’, *Computer Graphics (SIGGRAPH ’86 Proceedings)*, **20**, 4 (1986), 179-188.
35. Schroeder, W., Zarge, J. and Lorensen, W., ‘‘Decimation of Triangle Meshes’’, *Computer Graphics (SIGGRAPH ’92 Proceedings)*, **26**, 2 (July 1992), 65-70.
36. Schroeder, W., Martin, K. and Lorensen, W., *The Visualization Toolkit*, Prentice Hall, 1996.
37. Turk, G., ‘‘Re-Tiling Polygonal Surfaces’’, *Computer Graphics (SIGGRAPH ’92 Proceedings)*, **26**, 2 (July 1992), 55-64.
38. Varshney, A., ‘‘Hierarchical Geometric Approximations’’, Doctoral Dissertation, Department of Computer Science, Tech. Rep.-050-1994, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994.
39. Wang, S. W. and Kaufman, A. E., ‘‘Volume-Sampled 3D Modeling’’, *IEEE Computer Graphics & Applications*, **14**, 5 (September 1994), 26-32.
40. Westover, L., ‘‘Footprint Evaluation for Volume Rendering’’, *Computer Graphics (SIGGRAPH’90 Proceedings)*, **24**, 4 (August 1990), 367-376.
41. Wilhelms, J. and Van Gelder, A., ‘‘Topological Consideration in Isosurface Generation’’, *ACM Computer Graphics*, **24**, 5 (November 1990), 79-86.
42. Wilhelms, J. and Gelder, A. V., ‘‘Octree for Faster Isosurface Generation’’, *ACM Computer Graphics*, **24**, 5 (November 1990), 57-62.
43. Wolberg, G., *Digital Image Warping*, IEEE Computer Science Press, 1990.
44. ‘‘National Library of Medicine. Electronic Imageings: Report of the Board of Regions.’’, *NIH Publications 90-2197*, National Insistitute of Health, 1990.

Adaptive Real-Time Level-of-detail-based Rendering for Polygonal Models

Julie C. Xia Jihad El-Sana Amitabh Varshney

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

We present an algorithm for performing adaptive real-time level-of-detail-based rendering for triangulated polygonal models. The simplifications are dependent on viewing direction, lighting, and visibility and are performed by taking advantage of image-space, object-space, and frame-to-frame coherences. In contrast to the traditional approaches of precomputing a fixed number of level-of-detail representations for a given object our approach involves statically generating a continuous level-of-detail representation for the object. This representation is then used at run-time to guide the selection of appropriate triangles for display. The list of displayed triangles is updated incrementally from one frame to the next. Our approach is more effective than the current level-of-detail-based rendering approaches for most scientific visualization applications where there are a limited number of highly complex objects that stay relatively close to the viewer. Our approach is applicable for scalar (such as distance from the viewer) as well as vector (such as normal direction) attributes.

1 Introduction

The scientific visualization and virtual reality communities have always faced the problem that their "desirable" visualization dataset sizes are one or more orders of magnitude larger than what the hardware can display at interactive rates. Recent research on graphics acceleration for the navigation of such three-dimensional environments has been motivated by attempts to bridge the gap between the desired and the actual hardware performance, through algorithmic and software techniques. This research has involved reducing the geometric and rendering complexities of the scene by using

- statically computed level-of-detail hierarchies [35, 32, 31, 10, 25, 14, 22],
- visibility-based culling that is statically computed [1, 34] and dynamically computed [18, 27, 17],

- various levels of complexity in shading and illumination models[4],
- texture mapping [6, 5], and
- image-based rendering [8, 7, 29, 33, 13].

In this paper we will focus on reducing the geometric complexity of a three-dimensional environment by using dynamically computed level-of-detail hierarchies. Research on simplification of general three-dimensional polygonal objects (non-convex, non-terrain, possibly high genus) has spanned the entire gamut of highly local to global algorithms, with several approaches in between that have both local and global steps.

Local algorithms work by applying a set of local rules, which primarily work under some definition of a *local neighborhood*, for simplifying an object. The local rules are iteratively applied under a set of constraints and the algorithm terminates when it is no longer possible to apply the local rule without violating some constraint. The global algorithms optimize the simplification process over the whole object, and are not necessarily limited to the small neighborhood regions on the object. Some of the local approaches have been – vertex deletion by Schroeder *et al* [32], vertex collapsing by Rossignac and Borrel [31], edge collapsing by Hoppe *et al* [26] and Guéziec [20], triangle collapsing by Hamann [21], and polygon merging by Hinker and Hansen [24]. Some of the global approaches have been – redistributing vertices over the surface by Turk [35], minimizing global energy functions by Hoppe *et al* [26], using simplification envelopes by Varshney [36] and Cohen *et al* [10], and wavelets by DeRose *et al* [14]. The issue of preservation or simplification of the genus of the object is independent of whether an algorithm uses local rules, or global rules, or both, to simplify. Recent work by He *et al* [22] provides a method to perform a controlled simplification of the genus of an object.

Simplification algorithms such as those mentioned above are iteratively applied to obtain a hierarchy of successively coarser approximations to the input object. Such multiresolution hierarchies have been used in level-of-detail-based rendering schemes to achieve higher frame update rates while maintaining good visual realism. These hierarchies usually have a number of distinct levels of detail, usually 5 to 10, for a given object. At run time, the perceptual importance of a given object in the scene is used to select its appropriate level of representation from the hierarchy [9, 11, 12, 16, 30, 28]. Thus, higher detail representations are used when the object is perceptually more important and lower detail representations are used when the object is perceptually less significant. Transitions from one level of detail to the next are typically based on simple image-space metrics such as the ratio of the image-space area of the object (usually implemented by using the projected area of the bounding box of the object) to the distance of the object from the viewer.

Previous work, as outlined above, is well-suited for virtual reality walkthroughs and flythroughs of large and complex structures with several thousands of objects. Examples of such environments include architectural buildings, airplane and submarine interiors, and factory layouts. However, for scientific visualization applications where the goal often is to visualize one or two highly detailed objects at close range, most of the previous work is not directly applicable. For instance, consider a biochemist visualizing the surface of a molecule or a physician inspecting the iso-surface of a human head extracted from a volume dataset. It is very likely during such a visualization session, that the object being visualized will not move adequately far away from the viewer to allow the rendering algorithm to switch to a lower level of detail. What is desirable in such a scenario is an

algorithm that can allow several different levels of details to co-exist across different regions of the same object. Such a scheme needs to satisfy the following two important criteria:

- It should be possible to select the appropriate levels of detail across different regions of the same object in real time.
- Different levels of detail in different regions across an object should merge seamlessly with one another without introducing any cracks and other discontinuities.

In this paper we present a general scheme that can construct such seamless and adaptive level-of-detail representations on-the-fly for polygonal objects. Since these representations are view-dependent, they take advantage of view-dependent illumination, visibility, and frame-to-frame coherence to maximize visual realism and minimize the time taken to construct and draw such objects. Our approach shows how one can adaptively define such levels of detail based on (a) scalar attributes such as distance from the viewpoint and (b) vector attributes such as the direction of vertex normals. An example using our approach is shown in Figure 1.

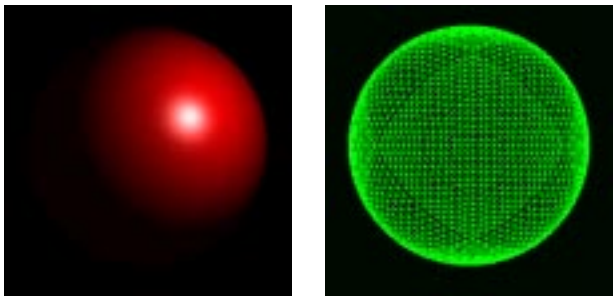
2 Previous Work

Adaptive levels of detail have been used in terrains by Gross *et al* [19] by using a wavelet decomposition of the input data samples. They define wavelet space filters that allow changes to the quality of the surface approximations in locally-defined regions. Thus, the level of detail around any region can adaptively refine in real-time. This work provides a very elegant solution for terrains and other datasets that are defined on a regular grid.

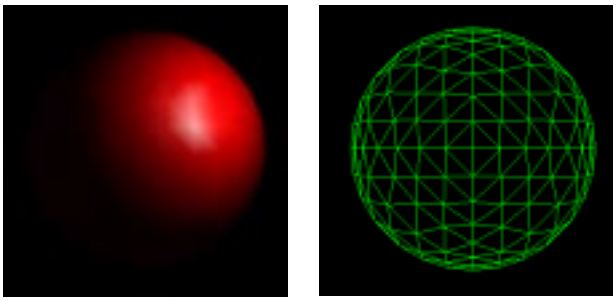
Some of the previous work in the area of general surface simplification has addressed the issue of adaptive approximation of general polygonal objects. Turk [35] and Hamann [21] have proposed curvature-guided adaptive simplification with lesser simplification in the areas of higher surface curvature. In [36, 10], adaptive surface approximation is proposed with different amounts of approximation over different regions of the object. Guéziec [20] proposes adaptive approximation by changing the tolerance volume in different regions of the object. However in all of these cases, once the level of approximation has been fixed for a given region of the object, a discrete level of detail corresponding to such an approximation is statically generated. No methods have been proposed there that allow free intermixing of different levels of detail across an object in real time in response to changing viewing directions.

Work on surface simplification using wavelets [14, 15] and progressive meshes [25] goes a step further. These methods produce a continuous level-of-detail representation for an object in contrast to a set of discrete number of levels of detail. In particular, Hoppe [25] outlines a method for selective refinement – i.e. refinement of a particular region of the object based upon view frustum, silhouette edges, and projected screen-space area of the faces. Since the work on progressive meshes by Hoppe [25] is somewhat similar to our work we overview his method next and discuss how our method extends it.

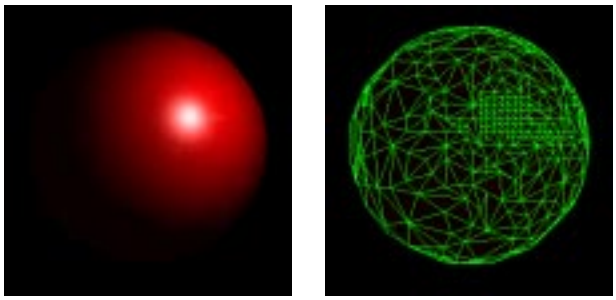
Progressive meshes offer an elegant solution for a continuous resolution representation of polygonal meshes. A polygonal mesh $\hat{M} = M^k$ is simplified into successively coarser meshes M^i by applying a sequence of edge collapses. An edge collapse transformation and its dual, the vertex split transformation, is shown in Figure 2.



(a) Sphere with 8192 triangles (uniform LOD)



(b) Sphere with 512 triangles (uniform LOD)



(c) Sphere with 537 triangles (adaptive LOD)

Figure 1: Uniform and adaptive levels of detail

Thus, a sequence of k successive edge collapse transformations yields a sequence of successively simpler meshes:

$$M^k \overset{collapse_{k-1}}{\rightarrow} M^{k-1} \overset{collapse_{k-2}}{\rightarrow} \ldots M^1 \overset{collapse_0}{\rightarrow} M^0 \tag{1}$$

We can retrieve the successively higher detail meshes from the simplest mesh M^0 by using a sequence of vertex-split transformations that are dual to the corresponding edge collapse transformations:

$$M^0 \overset{split_0}{\rightarrow} M^1 \overset{split_1}{\rightarrow} \ldots M^{k-1} \overset{split_{k-1}}{\rightarrow} (\hat{M} = M^k) \tag{2}$$

Hoppe [25] refers to $(M^0, \{split_0, split_1, \ldots, split_{k-1}\})$ as a *progressive mesh* representation. Progressive meshes present a novel approach to storing, rendering, and transmitting meshes by using a continuous-resolution representation. However we feel that there is some room for improve-

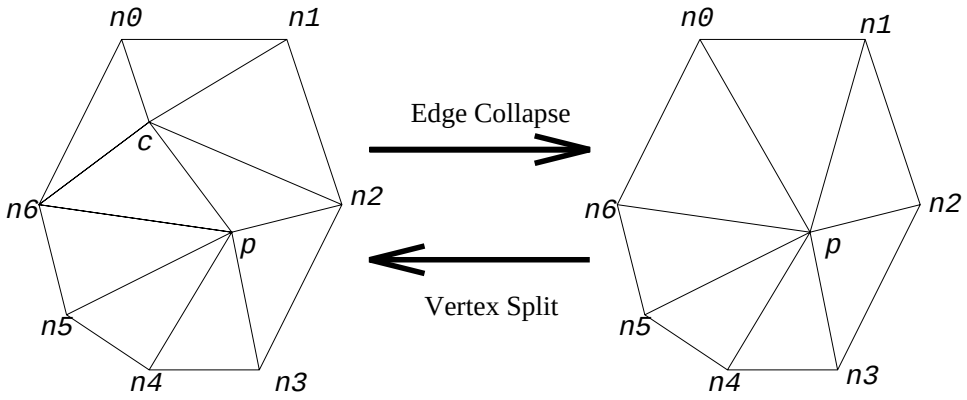


Figure 2: Edge collapse and vertex split

ment in adapting them for performing selective refinement in an efficient manner. In particular, following issues have not yet been addressed by progressive meshes:

- The sequence of edge collapses is aimed at providing good approximations M^i to $(\hat{M} = M^k)$. However, if a sequence of meshes M^i are good approximations to $\hat{M}$ under some distance metric, it does not necessarily mean that they also provide a "good" sequence of edge collapse transformations for selective refinement. Let us consider a two-dimensional analogy of a simple polygon as shown in Figure 3. Assume that vertices $v_0, v_6, v_7,$ and v_8 are "important" vertices (under say some perceptual criteria) and can not be deleted. An approach that generates approximations based on minimizing distances to the original polygon will collapse vertices in the order $v_1 \rightarrow v_2, v_2 \rightarrow v_3, v_3 \rightarrow v_4, v_4 \rightarrow v_5, v_5 \rightarrow v_6$ to get a coarse polygon (v_0, v_6, v_7, v_8). Then if selective refinement is desired around vertex v_1, vertices v_6, v_5, v_4, v_3, v_2 will need to be split in that order before one can get to vertex v_1. An approach that was more oriented towards selective refinement might have collapsed $v_1 \rightarrow v_2, v_3 \rightarrow v_4, v_5 \rightarrow v_6, v_2 \rightarrow v_4, v_4 \rightarrow v_6$ for better adaptive results, even though the successive approximations are not as good as the previous ones under the distance metric.

- Since the edge collapses are defined in a linear sequence, the total number of child links to be traversed before reaching the desired node is $O(n)$.

- No efficient method for incrementally updating the selective refinements from one frame to the next is given. The reverse problem of selective refinement – selective simplification too is not dealt with.

In this paper we provide a solution to the above issues with the aim of performing real-time adaptive simplifications and refinements. We define a criterion for performing edge collapses that permits adaptive refinement around any vertex. Instead of constructing a series of sequential edge collapses we construct a *merge tree* over the vertices of mesh $\hat{M}$ so that one can reach any child vertex in $O(\log n)$ links. We then describe how one can perform incremental updates within this tree to exploit frame-to-frame coherence, view-dependent illumination, and visibility computations using both scalar and vector attributes.

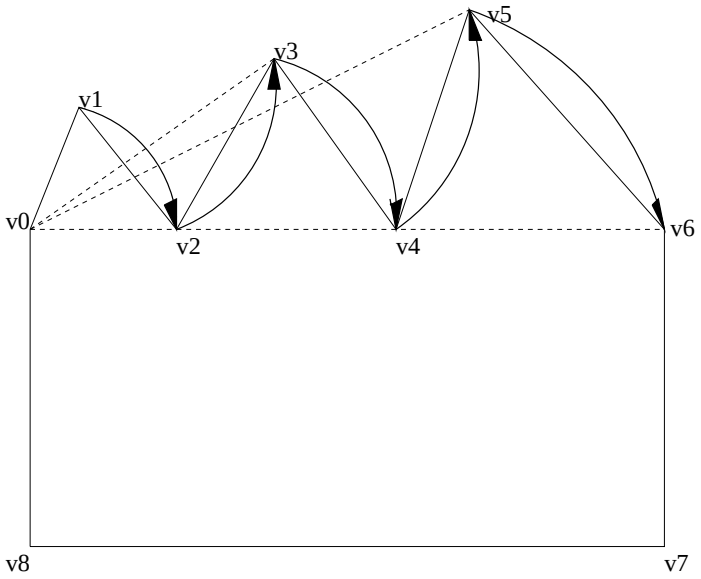


Figure 3: Good versus efficient selective simplification

3 Simplification with Image-Space Feedback

Level-of-detail-based rendering has thus far emphasized object-space simplifications with minimal feedback from the image space. The feedback from the image space has been in the form of very crude heuristics such as the ratio of the screen-space area of the bounding box of the object to the distance of the object from the viewer. As a result, one witnesses coarse image-space artifacts such as the distracting "popping" effect when the object representation changes from one level of detail to the next [23]. Attempts such as alpha-blending between the old and the new levels of detail during such transitions serve to minimize the distraction at the cost of rendering two representations. However alpha blending is not the solution to this problem since it does not address the real cause – lack of sufficient image-space feedback to select the appropriate local level of detail in the object space; it merely tries to cover-up the distracting artifacts.

Increasing the feedback from the image space allows one to make better choices regarding the level of detail selection in the object-space. We next outline some of the ways in which image-space feedback can influence the level of detail selection in the object-space.

3.1 Local Illumination

Increasing detail in a direction perpendicular to, and proportional to, the illumination gradient across the surface is a good heuristic [2]. This allows one to have more detail in the regions where the illumination changes sharply and therefore one can represent the highlights and the sharp shadows well. Since surface normals play an important role in local illumination one can take advantage of the coherence in the surface normals to build a hierarchy over a continuous resolution model that allows one to capture the local illumination effects well. We outline in Section 4.3 how one can build such a hierarchy.

3.2 Screen-Space Projections

Decision to keep or collapse an edge should depend upon the length of its screen-space projection instead of its object-space length. At a first glance this might seem very hard to accomplish in real-time since this could mean checking for the projected lengths of all edges at every frame. However, usually there is a significant coherence in the ratio of the image-space length to the object-space length of edges across the surface of an object and from one frame to the next. This makes it possible to take advantage of a hierarchy built upon the the object-space edge lengths for an object. We use an approximation to the screen-space projected edge length that is computed from the object-space edge length. We outline in Section 4.2 how one can build such a hierarchy.

3.3 Visibility Culling

During interactive display of any model there is usually a significant coherence between the visible regions from one frame to the next. This is especially true of the back-facing polygons that account for almost half the total number of polygons and do not contribute anything to the visual realism. A hierarchy over a continuous resolution representation of an object allows one to significantly simplify the invisible regions of an object, especially the back-facing ones. This view-dependent visibility culling can be implemented in a straightforward manner using the hierarchy on vertex normals discussed in Section 4.3.

3.4 Silhouette boundaries

Silhouettes play a very important role in perception of detail. Screen-space projected lengths of silhouette edges (i.e., edges for which one of the adjacent triangles is visible and the other is invisible), can be used to very precisely quantify the amount of smoothness of the silhouette boundaries. A hierarchy built upon a continuous-resolution representation of a object allows one to do this efficiently.

4 Construction of Merge Tree

We would like to create a hierarchy that provides us a continuous-resolution representation of an object and allows us to perform real-time adaptive simplifications over the surface of an object based upon the image-space feedback mechanisms mentioned in Section 3. Towards this end we implement a *merge tree* over the vertices of the original model. In our current implementation, the merge tree stores the edge collapses in a hierarchical manner. However, as we discuss in Section 7 the concept of a merge tree is a very general one and it can be used with other local simplification approaches as well. Note that the merge tree construction is done as an off-line preprocessing step before the interactive visualization.

4.1 Basic Approach

In Figure 2, the vertex c is merged with the vertex p as a result of collapsing the edge (pc). Conversely, during a vertex split the vertex c is created from the vertex p. We shall henceforth refer to

c as the child vertex of the parent vertex p. The merge tree is constructed upwards from the high-detail mesh $\hat{M}$ to a low-detail mesh M^0 by storing these parent-child relationships in a hierarchical manner over the surface of an object.

At each level l of the tree we determine parent-child relationships amongst as many vertices at level l as possible. In other words, we try to determine all vertices that can be safely merged based on criterion defined in Section 4.4. The vertices that are determined to be the children remain at level l and all the other vertices at level l are promoted to level $l+1$. Note that the vertices promoted to level $l+1$ are a proper superset of the parents of the children left behind at level l. This is because there are vertices at level l that are neither parents nor children. We discuss this in greater detail in the context of *regions of influence* later in this section. We apply the above procedure recursively at every level until either (a) we are left with a user-specified minimum number of vertices, or (b) we cannot establish any parent-child relationships amongst the vertices at a given level. Case (b) can arise because in determining a parent-child relationship we are essentially collapsing an edge and not all edge collapses are considered legal. For a detailed discussion on legality of edge collapses the interested reader can refer to [26]. Since in an edge collapse only one vertex merges with another, our merge tree is currently implemented as a binary tree.

To construct a balanced merge tree we note that the effects of an edge collapse are local. Let us define the *region of influence* of an edge (v_0, v_1) to be the union of triangles that are adjacent to either v_0 or v_1 or both. The region of influence of an edge is the set of triangles that can change as an edge is gradually collapsed to a vertex, for example, in a morphing. Thus, in Figure 2 as vertex c merges to vertex p, (or p splits to c), the changes to the mesh are all limited to within the region of influence of edge (pc) enclosed by $n_0, n_1, \ldots n_6$. Note that all the triangles in region of influence will change if vertices p and c are merged to form an intermediate vertex, say $(p + c)/2$. In our current implementation, the position of the intermediate vertex is the same as the position of the parent vertex p. However our data-structures can support other values of the intermediate vertex too. Such values could be used, for example, in creating intermediate morphs between two level-of-detail representations.

To create a reasonably balanced merge tree we try to collapse as many edges as possible at each level such that there are no common triangles in their respective regions of influence. Since this step involves only local checks, we can accomplish this step in time linear in the number of triangles at this level. If we assume that the average degree (i.e. the number of neighboring triangles) of a vertex is 6, we can expect the number of triangles in an edge's region of influence to be 10. After the collapse this number of triangles reduces to 8. Thus the number of triangles can be expected to reduce roughly by a factor of $4/5$ from a higher-detail level to a lower-detail level. Thus, in an ideal situation, the total time to build the tree will be given by $n + \frac{4n}{5} + \frac{16n}{25} + \ldots = O(n)$. However, this assumes that we arbitrarily choose the edges to be collapsed. A better alternative is to sort the edges by their edge lengths and collapse the shortest edges first. Collapsing an edge causes the neighboring edges to change their lengths. However as mentioned above, since changes are local we can maintain the sorted edge lengths in a heap for efficient updates. With this strategy one can build the merge tree in time $O(n \log n)$.

4.2 Scalar Subtree Attributes

To allow real-time refinement and simplification we can store at every parent node (i.e. a node that splits off a child vertex) of the merge tree, a range of scalar attributes of the children in the subtree below it. Then image-space feedback can be used to determine if this range of scalar attributes merits a refinement of this node or not. We explain this process of incremental refinement and simplification in greater details in Section 5.1.

In our current implementation every merge tree node v stores the Euclidean distances to its child and parent that determine when v's child will merge into v and when v will merge into its parent. The former is called the *downswitch distance* and the latter is called the *upswitch distance*. These distances are built up during the merge tree creation stage. If the maximum possible screen-space projection of the downswitch distance at the vertex v in the object space is greater than some pre-set threshold, we permit refinement at v. However, if the maximum possible screen-space projection of the upswitch distance at v in the object space is less than the threshold, it means that this region occupies very little screen space and can be simplified.

4.3 Vector Subtree Attributes

Our implementation also allows incremental simplification and refinement based upon the coherences of the surface normals. This allows us to implement view-dependent real-time simplifications based on local illumination and visibility. The regions with low intensity gradients are drawn in lower detail, while the regions with high intensity gradients are drawn in higher detail. Similarly, regions of the object that are back-facing are drawn at a much lower detail then the front-facing regions.

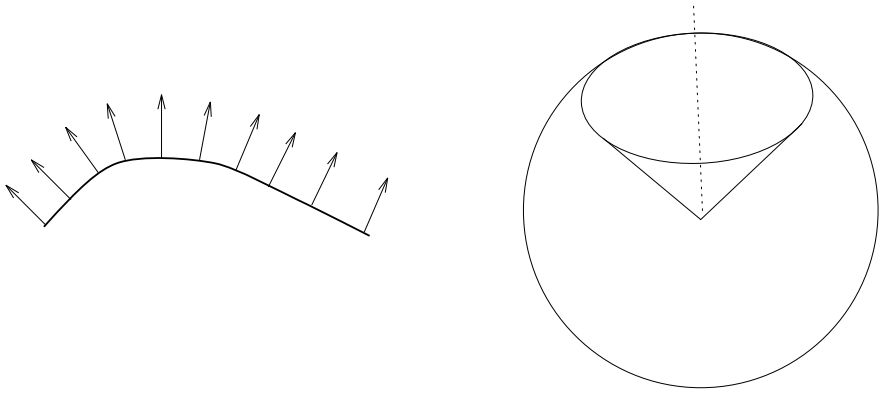
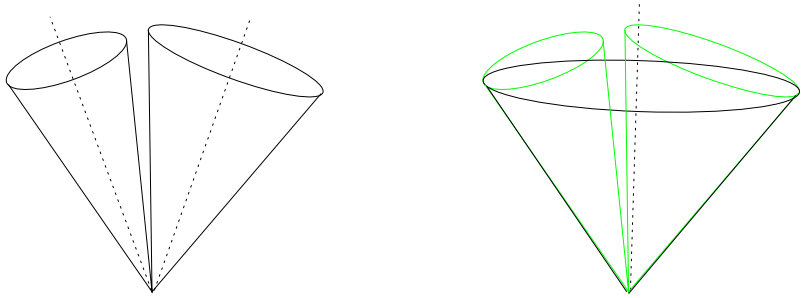


Figure 4: Bounding cone for normal vectors

Since we are using frame-to-frame coherences in computing the levels of detail we need to adopt a data-structure that represents the variation in the normal vectors amongst all the descendents of any given vertex. To identify a possible representation, let us consider the idea behind a Gauss map. A Gauss map is a mapping of the unit normals to the corresponding points on the surface of a unit sphere. Thus, all the normal variations in a subtree will be represented by a closed and connected region on the surface of a sphere using a Gauss map. To simplify the computations involved, we have decided to approximate such regions by circles on the surface of the unit sphere,

i.e. bounding cones containing all the subtree normal vectors. This is demonstrated in Figure 4 where the normal vectors in the surface shown on the left are contained within the cone (i.e. a circle on the Gauss map) on the right.

At the leaf-level, each vertex is associated with a normal-cone whose axis is given by its normal vector and whose angle is zero. As two vertices merge, the cones of the child and parent vertices are combined into a new normal cone that belongs to the parent vertex at the higher level. The idea behind this merging of cones is shown in Figure 5.



a) Child Cones b)Merged Parent Cone

Figure 5: Cone merging

4.4 Merge Tree Dependencies

By using techniques outlined in Section 5.1, one can determine which subset of vertices is sufficient to reconstruct an adaptive level-of-detail for a given object. However, it is not simple to define a triangulation over these vertices and guarantee that the triangulation will not "fold back" on itself or otherwise represent a non-manifold surface (even when the original was not so). Figure 6 shows an example of how an undesirable folding in the adaptive mesh can arise even though all the edge collapses that were determined statically were correct. A shows the initial state of the mesh. While constructing the merge tree, we first collapsed vertex v_2 to v_1 to get mesh B and then collapsed vertex v_3 to v_4 to get mesh C. Now suppose at run-time we determined that we needed to display vertices v_1, v_2, and v_4 and could possibly collapse vertex v_3 to v_4. However, if we collapse v_3 to v_4 directly, as in mesh D, we get a mesh fold where there should have been none. One could devise elaborate procedures for checking and preventing such mesh fold-overs at run-time. However, such checks involve several floating-point operations and are too expensive to be performed on-the-fly.

To solve the above problem we introduce the notion of dependencies amongst the nodes of a merge tree. Thus, the collapse of an edge e is permitted only when all the vertices defining the boundary of the region of influence of the edge e exist and are adjacent to the edge e. As an example, consider Figure 2. Vertex c can merge with vertex p only when the vertices $n_0, n_1, \ldots, n_k$ exist and are adjacent to p and c. From this we determine the following edge collapse dependencies, restricting the level difference between adjacent vertices:

1. c can collapse to p, only when n_0, n_1, ..., n_k are present as neighbors of p and c for display.

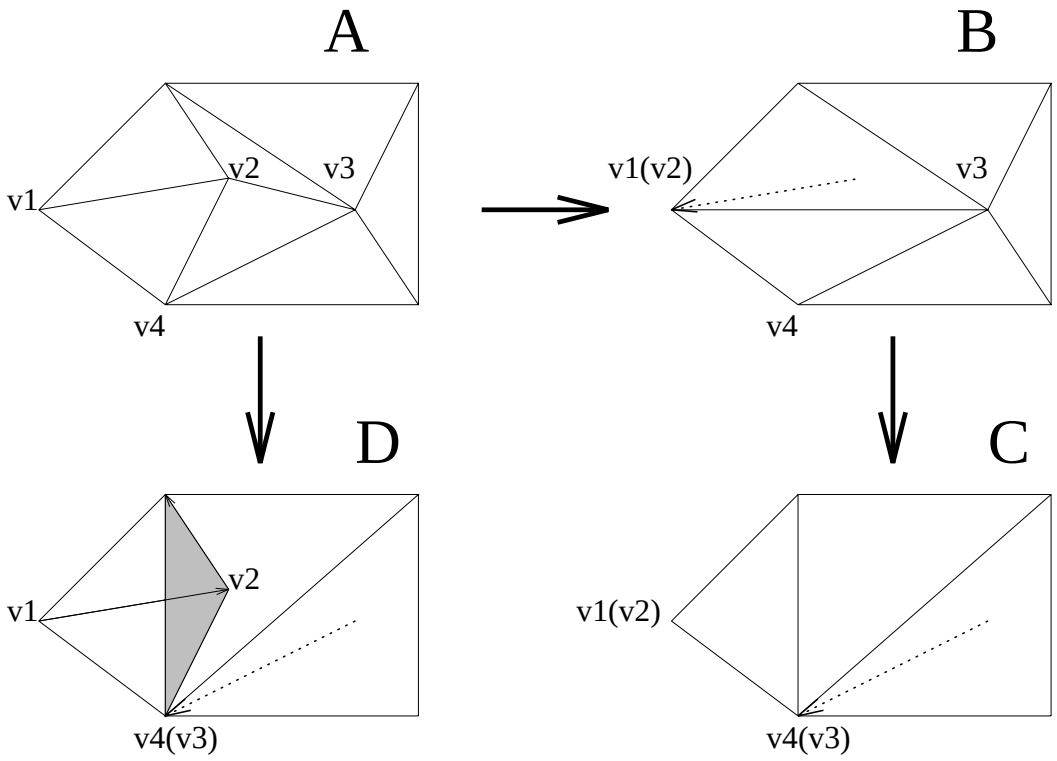


Figure 6: Mesh folding problem

2. $n_0, n_1, \ldots, n_k$ can not merge with other vertices, unless c first merges with p.

Similarly, to make a safe split from p to p and c, we determine the following vertex split dependency:

1. p can split to c and p, only when $n_0, n_1, \ldots, n_k$ are present as neighbors of p for display.

2. $n_0, n_1, \ldots, n_k$ can not split, unless p first splits to p and c.

The above dependencies are followed during each vertex-split or edge collapse during real-time simplification. These dependencies are easily identified and stored in the merge tree during its creation. Considering Figure 6 again, we can now see that collapse of vertex v_3 to v_4 depends upon the adjacency of vertex v_1 to v_3. If vertex v_2 is present then v_1 will not be adjacent to v_3 and therefore v_3 will not collapse to v_4. Although having dependencies might sometimes give lesser simplification than otherwise, it does have the advantage of eliminating the expensive floating-point run-time checks entirely. The basic idea behind merge tree dependencies has a strong resemblance to creating *balanced subdivisions* of quad-trees as presented by Baum *et al* in [3] where only a gradual change is permitted from regions of high simplifications to low simplifications. Details of how these merge tree dependencies are used during run-time are given in Section 5.1.

The pseudocode outlining the data-structure for a merge tree node is given in Figure 7. The pseudocode for building and traversing the merge tree is given in Figure 8. We are representing the triangular mesh by the winged-edge data-structure to maintain the adjacency information.

```
struct NODE {
        struct VERTEX  *vert   ;     /* associated vertex */
        struct NODE    *parent ;     /* parent node for merging   */
        struct NODE    *child[2] ;   /* child nodes for refinement */
        float          upswitch  ;   /* threshold to merge */
        float          downswitch;   /* threshold to refine */
        struct CONE    *cone     ;   /* range of subtree normals */
        struct VERTEX  **adj_vert ;  /* adjacent vertices */
        int            adj_num    ;  /* number of adjacent vertices */
        struct VERTEX  **depend_vert;/* dependency list for merge */
        int            depend_num   ;/* number of vertices in the */
};                                    /* dependency list */
```

Figure 7: Data-structure for a merge tree node

5 Real-Time Triangulation

Once the merge tree with dependencies has been constructed off-line it is easy to construct an adaptive level-of-detail mesh representation at run-time. Real-time adaptive mesh reconstruction involves two phases – determination of vertices that will be needed for reconstruction and determination of the triangulation amongst them. We shall refer to the vertices selected for display at a given frame as *display vertices* and triangles for display as *display triangles*. The phases for determination of display vertices and triangles are discussed next.

5.1 Determination of display vertices

In this section we outline how we determine the display vertices using the scalar and vector attribute ranges stored with the nodes of the merge tree. We first determine the *primary display vertices* using the screen-space projections and the normal vector cones associated with merge tree nodes. These are the only vertices that would be displayed if there were no triangulation constraints or mesh-folding problems. Next, from these primary display vertices we determine the *secondary display vertices* that are the vertices that need to be displayed due to merge tree dependencies to avoid the mesh fold-overs in run-time triangulations.

5.1.1 Primary Display Vertices

Screen-Space Projection

As mentioned earlier, every merge tree node v stores a Euclidean distance for splitting a vertex to its child (downswitch distance) as well as the distance at which it will merge to its parent (upswitch distance). If the maximum possible screen-space projection of the downswitch distance at the vertex v in the object space is greater than some pre-set threshold T, we permit refinement at v and recursively check the children of v. However, if the maximum possible screen-space projection of the upswitch distance at v in the object space is less than the threshold T, it means

```
/* Given a mesh, build_mergetree() constructs a list of merge trees - one
 * for every vertex at the coarsest level of detail.
 */
build_mergetree(struct MESH *mesh, struct NODE **roots)
{ struct HEAP  *current_heap, *next_heap ;
  int level ;

  current_heap = InitHeap(mesh);
  next_heap = InitHeap(nil) ;
  for ( level = 0 ; HeapSize(current_heap) > MIN_RESOLUTION_SIZE; level ++
  { while ( HeapSize(current_heap) > 0 )
       { edge = ExtractMinEdge(current_heap);
         node = CreatNode(edge);
         SetDependencies(node);
         SetCone(node);            /* Set vector attributes */
         SetSwitchDistances(node); /* Set scalar attributes */
         InsertHeap(next_heap, node);
       }
    FreeHeap(current_heap);
    current_heap = next_heap ;
    next_heap = InitHeap(nil);
  }
  FlattenHeap(roots, current_heap);
}

/* Given a list of nodes of the merge tree that were active in the previous
 * frame, traverse_mergetree() constructs a list of new merge tree nodes by
 * either refining or simplifying each of the active merge tree nodes.
 */
traverse_mergetree(struct NODE **current_list,
                   struct VIEW view, struct LIGHTS *lights )
{ int switch ;

  for each node in current_list do
  { switch = EvalSwitch(node, view, lights);
    if ( switch == REFINE )
       RefineNode(node);
    else if ( switch == SIMPLIFY )
       MergeNode(node);
  }
}
```

Figure 8: Pseudocode for building and traversing the merge tree

that this region occupies very little screen space and can be simplified, so we mark v as *inactive* for display.

Normal Vectors

We need to determine the direction and the extent of the normal vector orientation within the subtree rooted at a display vertex, with respect to the viewing direction as well as light source, to accomplish view-dependent local illumination and visibility-based culling.

To determine silhouettes and the back-facing regions of an object, we check to see if the normal vector cone at a vertex lies entirely in a direction away from the viewer. If so, this vertex can be marked inactive for display. If not, this vertex is a display vertex and is a candidate for further refinement based on other criteria such as screen-space projection, illumination gradient, and silhouette smoothness. In such a case we recursively check its children. The three possible cases are shown in Figure 9.

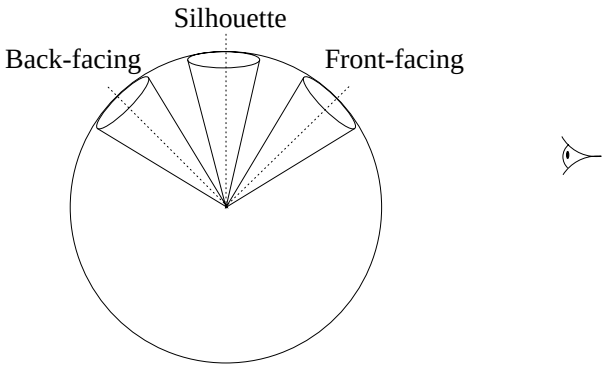


Figure 9: Selective refinement and simplification using normal cones

Similarly, for normal-based local illumination, such as Phong illumination, we use the range of the reflection vectors and determine whether they contain the view direction or not to determine whether to simplify or refine a given node of the merge tree.

We follow the procedures outlined above to select all those vertices for display that either (a) are leaf nodes and none of their parents have been marked as inactive, or (b) have their immediate child marked as inactive. This determines the list of primary display vertices.

5.1.2 Secondary Display Vertices

We follow the merge dependencies from the list of primary display vertices to select the final set of display vertices in the following manner. If a vertex v is in the initial list of display vertices and for it to be created (via a vertex split), the vertices $v_{d_0}, v_{d_1}, \ldots, v_{d_k}$ had to be present, we add the vertices $v_{d_0}, v_{d_1}, \ldots, v_{d_k}$ to the list of display vertices and recursively consider their dependencies. We continue this process until no new vertices are added.

When determining the vertices for display in frame $i+1$ we start from the vertex list for display used in frame i. We have found a substantial frame-to-frame coherence and the vertex display list does not change substantially from one frame to the next. There are minor local changes in the display list on account of vertices either refining or merging with other vertices. These are easily captured by either traversing the merge tree up or down from the current vertex position. The

scalar and vector attribute ranges stored in merge tree nodes can be used to guide refinements if the difference in the display vertex lists from one frame to the next becomes non-local for any reason. We compute the list of display vertices for first frame by initializing the list of display vertices for frame 0 to be all the vertices in the model and then proceeding as above.

5.2 Determination of display triangles

If the display triangles for frame i are known, determination of the display triangles for frame $i+1$ proceeds in an interleaved fashion with the determination of display vertices for frame $i+1$ from frame i. Every time a display vertex of frame i merges in frame $i+1$ we simply delete and add appropriate triangles to the list of display triangles as shown in Figure 10. The case where a display vertex in frame i splits for frame $i+1$ is handled analogously. Incremental determination of display triangles in this manner is possible because of the dependency conditions mentioned in Section 4.4. The list of display triangles for the first frame is obtained by initializing the list for frame 0 to be all the triangles in the model and then following the above procedure.

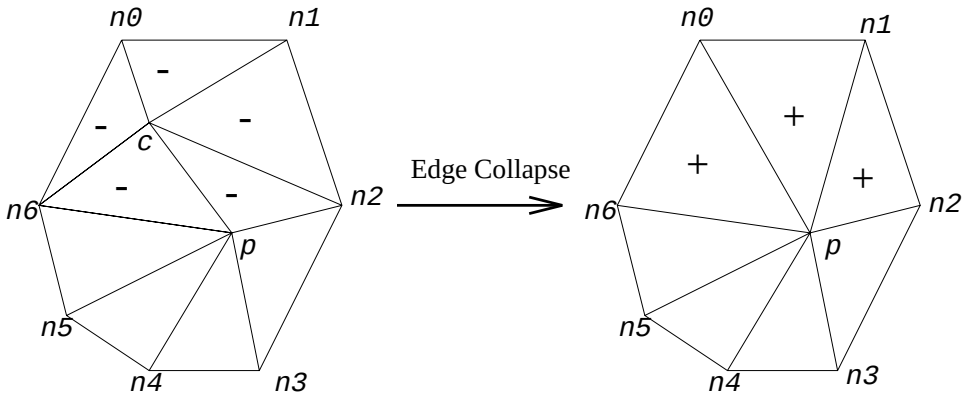
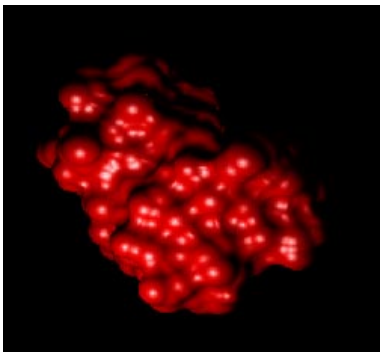


Figure 10: Display triangle determination

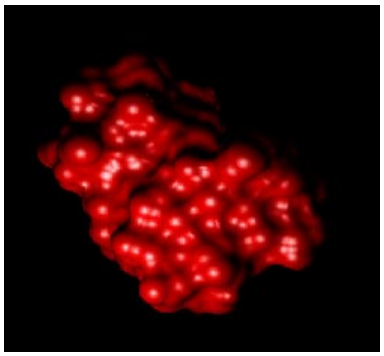
6 Results and Discussion

We have tried our implementation on several large triangulated models and have achieved encouraging results. These are summarized in Table 1 The images of teapot, bunny, crambin, phone, sphere, buddha, and dragon models that were produced for the times in Table 1 are shown in Figures 1, 11, 13, and 14 respectively. All of these timings are in milliseconds on a Silicon Graphics Onyx with RE2 graphics, a 194MHz R10000 processor, and 640MB RAM. It is easy to see that the time to traverse the merge tree and construct the list of triangles to be displayed from frame to frame is relatively small. This is because of our incremental computations that exploit image-space, object-space, and frame-to-frame coherences. The above times hold as the user moves through the model or moves the lights around. The triangulation of the model changes dynamically to track the highlights as well as the screen-space projections of the faces.

As can be seen from the merge tree depths, the trees are not perfectly balanced. However, they are still within a small factor of the optimal depths. This factor is the price that has to be paid

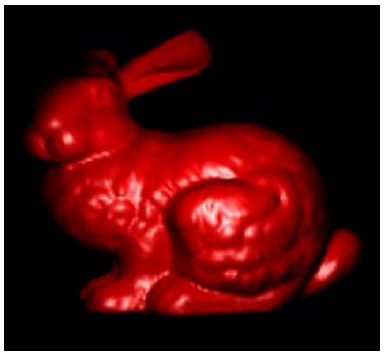


Highest detail Crambin surface



Simplified Crambin surface

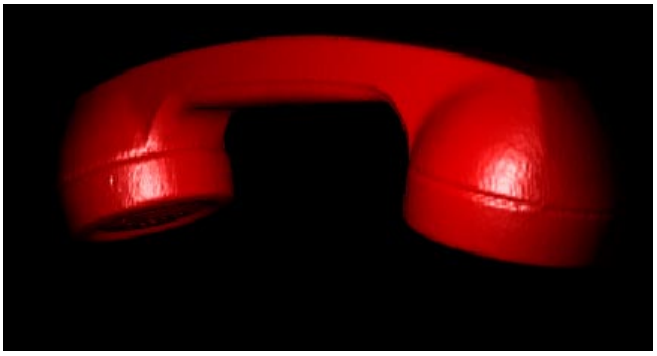
Highest detail Bunny



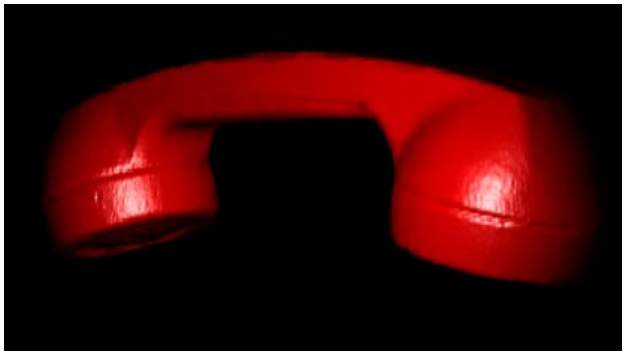
Simplified Bunny

Highest detail Teapot

Simplified Teapot

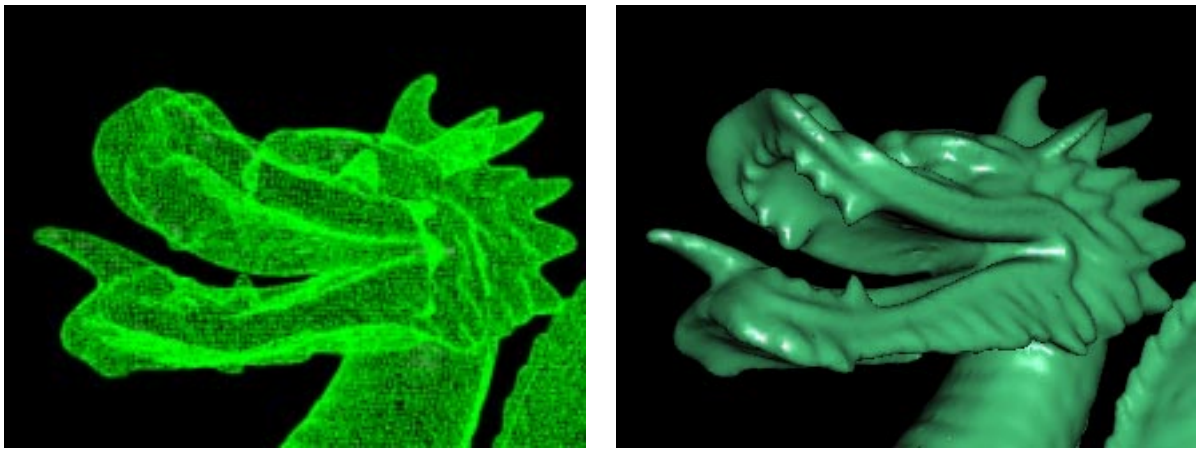


Highest detail Phone

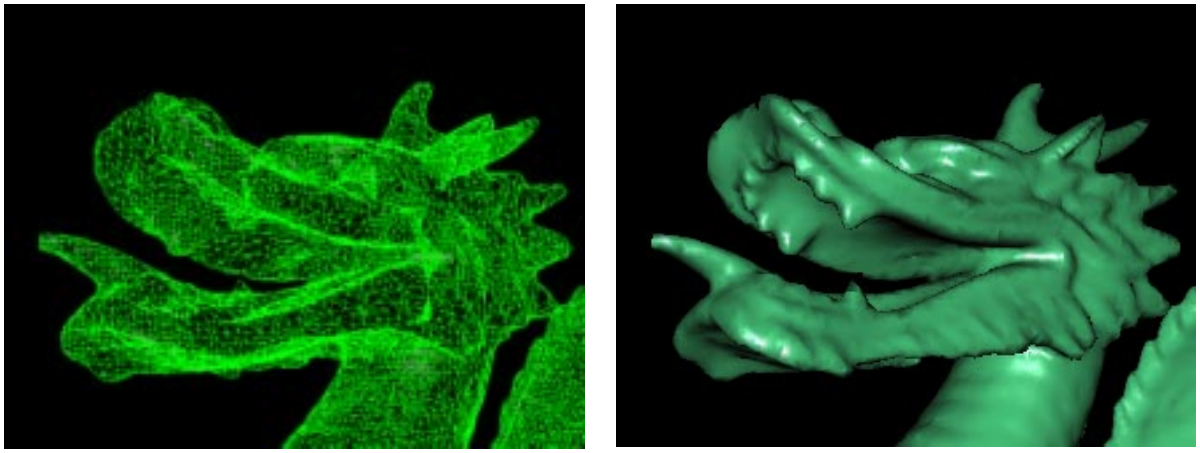


Simplified Phone

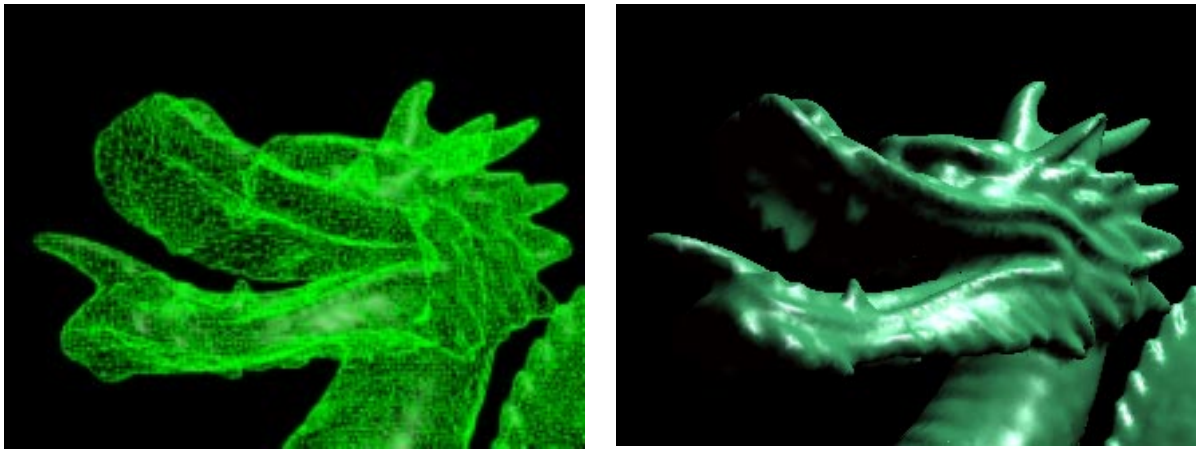
Figure 11: Dynamic adaptive simplification



Highest detail model – bottom light source



Dynamic adaptive simplification – top light source

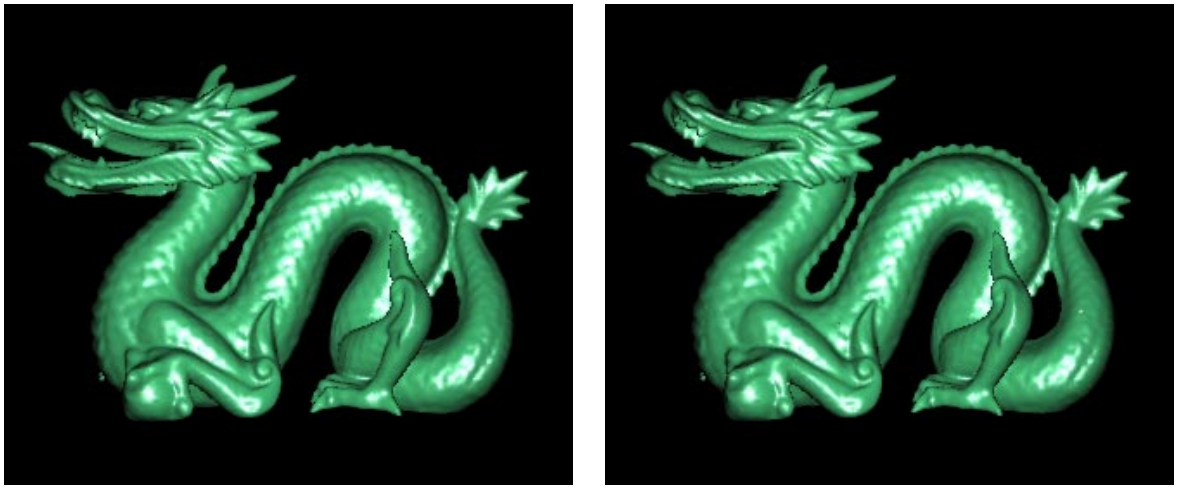


Dynamic adaptive simplification – top light source

Figure 12: Dynamic adaptive simplification for the head of the Dragon

Highest detail Simplified

Figure 13: Dynamic adaptive simplification for the Buddha



Highest detail Simplified

Figure 14: Dynamic adaptive simplification for the Dragon

Dataset	Highest Detail		Adaptive Detail					Reduction Ratio	
	Display Tris	Display Time	Display Tris	Tree Levels	Traverse Tree	Display Time	Total Time	Display Tris	Display Time
Teapot	3751	57	1203	36	10	17	27	32.0%	47.3 %
Sphere	8192	115	994	42	8	16	24	12.1%	20.8 %
Bunny	69451	1189	13696	65	157	128	285	19.7%	23.9 %
Crambin	109884	1832	19360	61	160	194	354	17.6%	19.3 %
Phone	165963	2629	14914	63	112	144	256	8.9 %	9.7 %
Dragon	202520	3248	49771	66	447	394	842	24.5%	25.9 %
Buddha	293232	4618	68173	69	681	546	1227	23.2%	26.5 %

Table 1: Adaptive level of detail generation times

to incorporate dependencies and avoid the expensive run-time floating-point checks for ensuring good triangulations. For each dataset, we continued the merge tree construction till 8 or fewer vertices were left. As expected, the factor by which the number of vertices decreases from one level to the next tapers off as we reach lower-detail levels since there are now fewer alternatives left to counter the remaining dependency constraints. As an example, for sphere, only 64 vertices were present at level 30 and it took another 12 levels to bring down the number to 8. If the tree depth becomes a concern one can stop sooner, trading-off the tree traversal time for the display time.

An interesting aspect of allowing dependencies in the merge tree is that one can now influence the characteristics of the run-time triangulation based upon static edge-collapse decisions during pre-processing. As an example, we have implemented avoidance of slivery (long and thin) triangles in the run-time triangulation. As Guéziec [20], we quantify the quality of a triangle with area a and lengths of the three sides $l_0, l_1,$ and l_2 based on the following formula:

$$\text{Quality} = \frac{4\sqrt{3}a}{l_0^2 + l_1^2 + l_2^2} \tag{3}$$

Using Equation 3 the quality of a degenerate triangle evaluates to 0 and that of an equilateral triangle to 1. We classify all edge collapses that result in slivery triangles to be invalid, trading-off quantity (amount of simplification) for quality.

One of the advantages of using normal cones for back-face simplification and silhouette definition is that it allows the graphics to focus more on the regions of the object that are perceptually more important. Thus, for instance, for generating the given image of the molecule crambin, 8729 front-facing vertices were traversed as compared to 3361 backfacing vertices; 1372 were classified as silhouette vertices. Similarly, for the model of phone, our approach traversed 6552 front-facing vertices compared to only 1300 backfacing vertices; 900 were classified as silhouette vertices.

Clearly, there is a tradeoff here between image-quality and amount of simplification achieved. The results for our simplifications given in this section correspond to the images that we thought were comparable to the images from the original highest detail models. Higher levels of simplifications (that are faster to incrementally compute and display) with correspondingly lower quality

images are obviously possible, allowing easy implementations of progressive refinement for display.

7 Conclusions and Future Work

We have outlined a simple approach to maintain dynamically adaptive level of detail triangulations. Crucial to this approach is the notion of merge trees that are computed statically and are used during run-time to take advantage of the incremental changes in the triangulation. In our current implementation we are using the method of edge collapses. However the idea behind merge trees is pretty general and can be used in conjunction with other local heuristics for simplification such as vertex deletion and vertex collapsing. We plan to study some of these other heuristics in the future and compare them with our current implementation that uses edge collapses.

At present we do not store color ranges at the nodes of the merge tree. Storing and using these should improve the quality of the visualizations produced using merge trees even further. Also of some interest will be techniques that create better balanced merge trees while still incorporating dependencies. We plan to investigate these issues further.

Of course, our approach also makes dynamically-specified manual simplifications possible, where the user can interactively specify the amounts of approximation desired at various regions of the object. Using this, certain parts of the object can be rendered at lower or higher details than otherwise. However, in this paper we have only considered automatic object simplifications during interactive display.

Acknowledgements

We would like to acknowledge several useful discussions with Arie Kaufman and Greg Turk. We should like to thank Greg Turk, Marc Levoy, and the Stanford University Computer Graphics laboratory for generously sharing models of the bunny, the phone, the dragon, and the happy Buddha. We should also like to acknowledge the several useful suggestions made by the anonymous reviewers that have helped improve the presentation of this paper. This work has been supported in part by the National Science Foundation CAREER award CCR-9502239 and a fellowship from the Fulbright/Israeli Arab Scholarship Program.

References

[1] J. M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, Chapel Hill, NC 27599-3175, 1990.

[2] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. In Rich Riesenfeld and Carlo Sequin, editors, *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, No. 2, pages 41–50, March 1990.

[3] D. R. Baum, Mann S., Smith K. P., and Winget J. M. Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. *Computer Graphics: Proceedings of SIGGRAPH'91*, 25, No. 4:51–60, 1991.

[4] L. Bergman, H. Fuchs, E. Grant, and S. Spach. Image rendering by adaptive refinement. In *Computer Graphics: Proceedings of SIGGRAPH'86*, volume 20, No. 4, pages 29–37. ACM SIGGRAPH, 1986.

[5] Jim F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78*, pages 286–292. ACM, 1978.

[6] Jim F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *CACM*, 19(10):542–547, October 1976.

[7] S. Chen. Quicktime VR – an image-based approach to virtual environment navigation. In *Computer Graphics Annual Conference Series (SIGGRAPH '95)*, pages 29–38. ACM, 1995.

[8] S. Chen and L. Williams. View interpolation for image synthesis. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, August 1993.

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

[10] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks, Jr., and W. V. Wright. Simplification envelopes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 119 – 128. ACM SIGGRAPH, ACM Press, August 1996.

[11] M. Cosman and R. Schumacker. System strategies to optimize CIG image content. In *Proceedings of the Image II Conference*, Scottsdale, Arizona, June 10–12 1981.

[12] F. C. Crow. A more flexible image generation environment. In *Computer Graphics: Proceedings of SIGGRAPH'82*, volume 16, No. 3, pages 9–18. ACM SIGGRAPH, 1982.

[13] L. Darsa, B. Costa, and A. Varshney. Navigating static environments using image-space simplification and morphing. In *Proceedings, 1997 Symposium on Interactive 3D Graphics*, 1997.

[14] T. D. DeRose, M. Lounsbery, and J. Warren. Multiresolution analysis for surface of arbitrary topological type. Report 93-10-05, Department of Computer Science, University of Washington, Seattle, WA, 1993.

[15] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of SIGGRAPH 95 (Los Angeles, California, August 6–11, 1995)*, Computer Graphics Proceedings, Annual Conference Series, pages 173–182. ACM SIGGRAPH, August 1995.

[16] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993)*, Computer Graphics Proceedings, Annual Conference Series, pages 247–254. ACM SIGGRAPH, August 1993.

[17] N. Greene. Hierarchical polygon tiling with coverage masks. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 65 – 74. ACM Siggraph, ACM Press, August 1996.

[18] Ned Greene and M. Kass. Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240, 1993.

[19] M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In G. M. Nielson and D. Silver, editors, *IEEE Visualization '95 Proceedings*, pages 135–142, 1995.

[20] A. Guéziec. Surface simplification with variable tolerance. In *Proceedings of the Second International Symposium on Medical Robotics and Computer Assisted Surgery, MRCAS '95*, 1995.

[21] B. Hamann. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*, 11:197–214, 1994.

[22] T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.

[23] James Helman. Graphics techniques for walkthrough applications. In *Interactive Walkthrough of Large Geometric Databases, Course Notes 32, SIGGRAPH '95*, pages B1–B25, 1995.

[24] P. Hinker and C. Hansen. Geometric optimization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings Visualization '93*, pages 189–195, October 1993.

[25] H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 99 – 108. ACM SIGGRAPH, ACM Press, August 1996.

[26] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proceedings of SIGGRAPH 93 (Anaheim, California, August 1–6, 1993)*, Computer Graphics Proceedings, Annual Conference Series, pages 19–26. ACM SIGGRAPH, August 1993.

[27] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings, 1995 Symposium on Interactive 3D Graphics*, pages 105 – 106, 1995.

[28] P. W. C. Maciel and P. Shirley. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 Symposium on Interactive 3D Computer Graphics*, pages 95–102, 1995.

[29] L. McMillan and G. Bishop. Plenoptic modeling: An image-based rendering system. In *Computer Graphics Annual Conference Series (SIGGRAPH '95)*, pages 39–46. ACM, 1995.

[30] J. Rohlf and J. Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, July 1994.

[31] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering. In *Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, June–July 1993.

[32] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *Computer Graphics: Proceedings SIGGRAPH '92*, volume 26, No. 2, pages 65–70. ACM SIGGRAPH, 1992.

[33] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996)*, Computer Graphics Proceedings, Annual Conference Series, pages 75–82. ACM SIGGRAPH, ACM Press, August 1996.

[34] S. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics: Proceedings of SIGGRAPH'91*, 25, No. 4:61–69, 1991.

[35] G. Turk. Re-tiling polygonal surfaces. In *Computer Graphics: Proceedings SIGGRAPH '92*, volume 26, No. 2, pages 55–64. ACM SIGGRAPH, 1992.

[36] A. Varshney. Hierarchical geometric approximations. Ph.D. Thesis TR-050-1994, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994.

A Hierarchy of Techniques for Simplifying Polygonal Datasets

Amitabh Varshney
Department of Computer Science
State University of New York at Stony Brook

Levels of Simplification

- Lossless
- Genus-preserving
- Genus-reducing

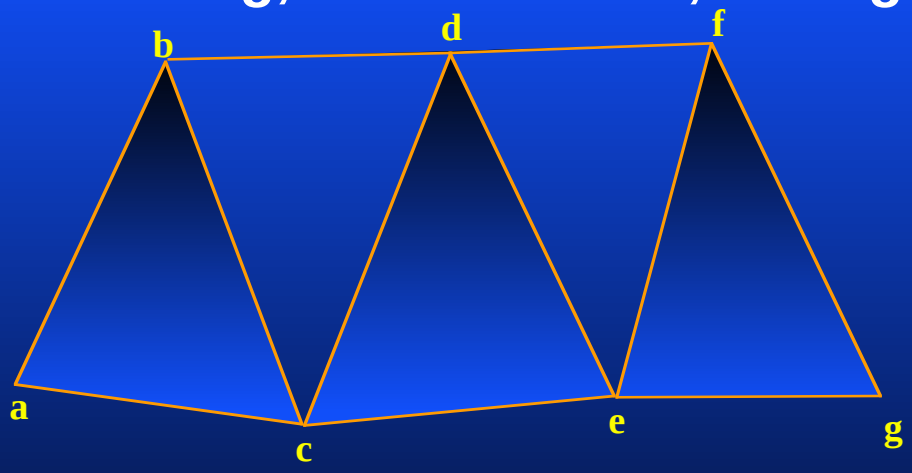
Switching Criteria

- Coarse-grained
  - Screen projected object area
  - Distance to the eye
- Fine-grained
  - Screen projected edge length
  - Local illumination
  - Silhouettes
  - Visibility-based culling

Talk Outline

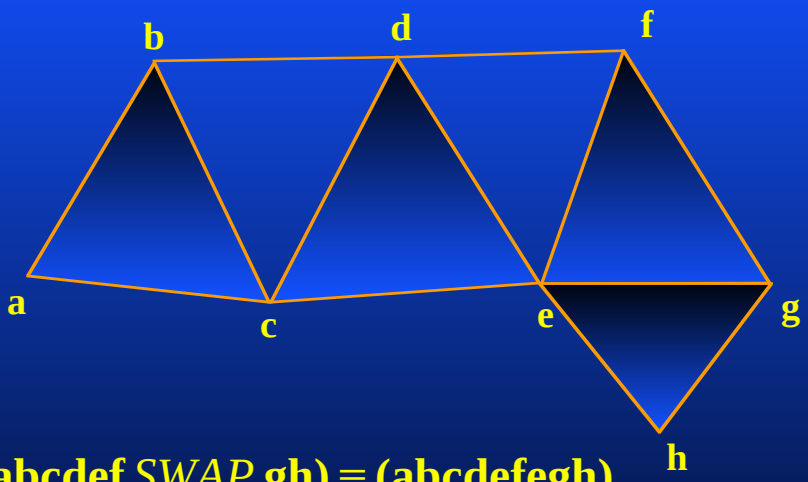
- Lossless compression
  Triangle strips
- Genus-preserving simplification
  Simplification Envelopes
- Genus-reducing simplification
  Volumetric approach
- View-Dependent Rendering
  Merge Trees

Triangle Strips for Efficient Rendering, Transmission, Storage



(abc, bcd, cde, def, efg) → (abcdefg)

Generalized Triangle Strips



(abcdef *SWAP* gh) ≡ (abcdefegh)

Triangle Strip Generation Complexity

- Finding Hamiltonian Triangulations (generalized triangle strips with swaps) is NP-complete [Arkin et al 94]

- Finding Sequential Triangulations (simple triangle strips without swaps) is NP-complete [Evans, Skiena,Varshney 97]

Experiments with Triangle Strips

Tried 20 approaches for triangle strips

- Local versus Global

- Static vs Dynamic / 1D vs 2D patches

- Various tie-breaking heuristics

Francine Evans, Steve Skiena, Amitabh Varshney
IEEE Visualization 1996

STRIPE

- Tested on over 100 polygonal models
- Best heuristic: Global row/column strips
- Compared to previous best (on an average):
  - 50% less strips
  - 30% faster to render
  - Two times slower in pre-processing
- Available free for non-commercial use from
  http://www.cs.sunysb.edu/~evans/stripe.html

Visual Comparison of Results

SGI

STRIPE

Dataset Courtesy: Viewpoint Datalabs

Visual Comparison of Results

SGI **STRIPE**

Dataset Courtesy: Viewpoint Datalabs

Stanford Bunny Model

69,451 triangles

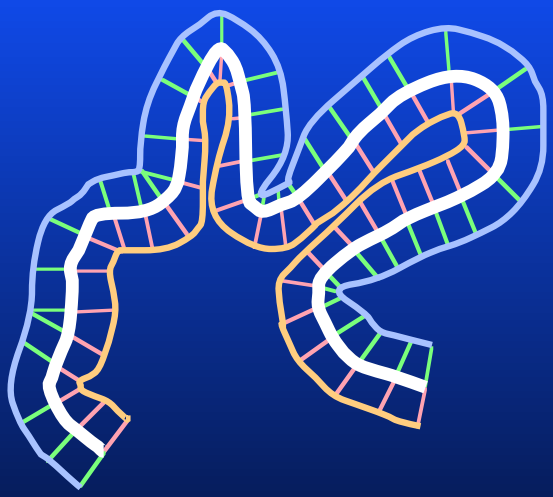
2D Simplification Envelopes

Original Surface

Outer Envelope

Inner Envelope

Analytically-computed Envelope



Original Surface

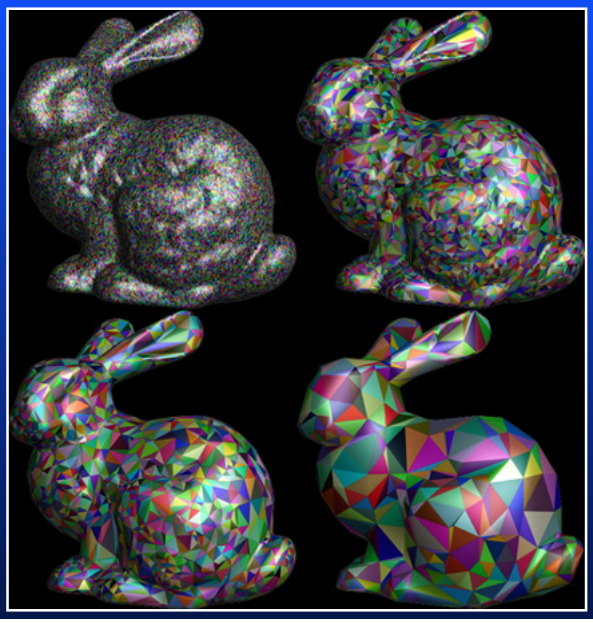
Outer Envelope

Inner Envelope

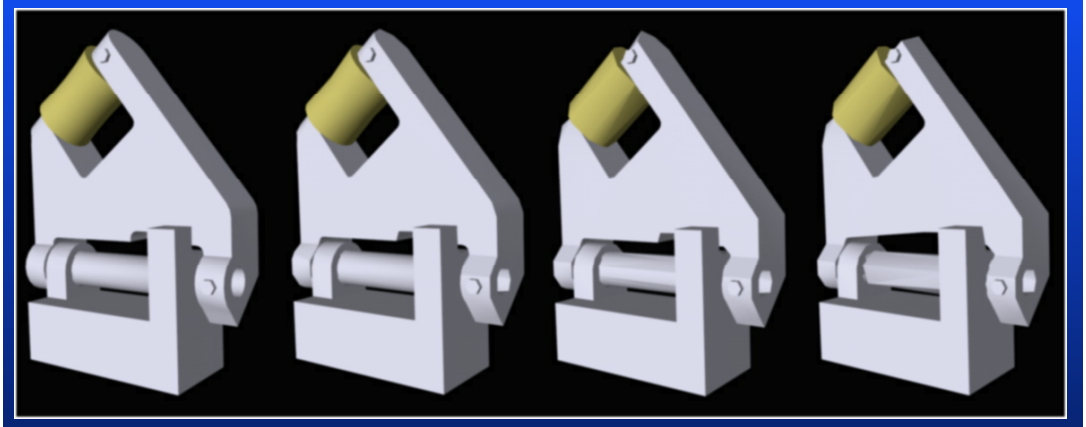
ε^{+}

ε^{-}

Results



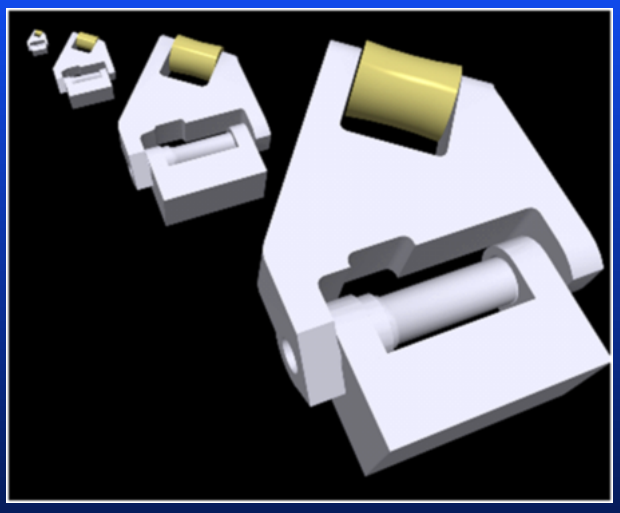
Results



Cohen, Varshney, Manocha, Turk,
Weber,Agarwal, Brooks, Wright
ACM Siggraph 96

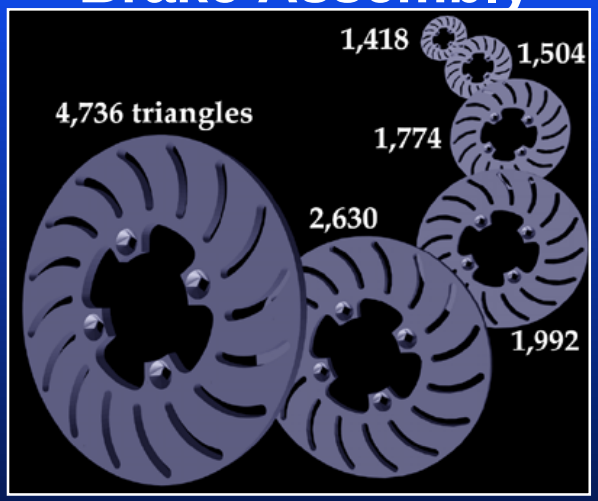
Dataset Courtesy:
Electric Boat Division
of General Dynamics

Results



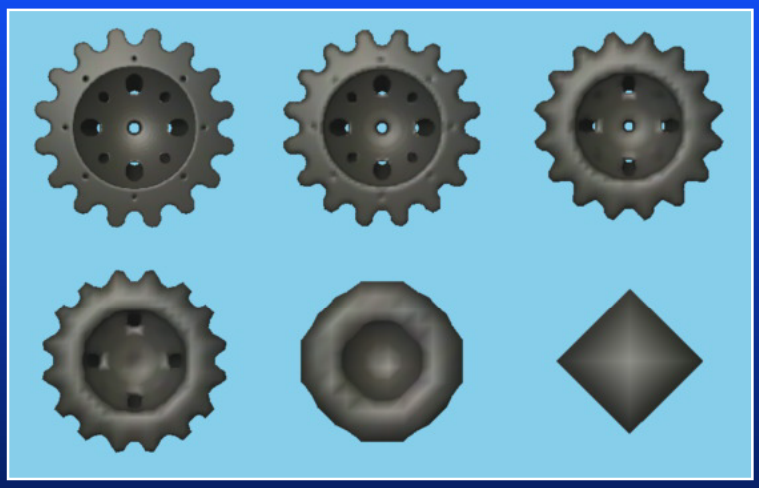
Dataset Courtesy: Electric Boat Division of General Dynamics

Simplifications of the Brake Assembly



Dataset Courtesy: University of Utah, Alpha-1 Project

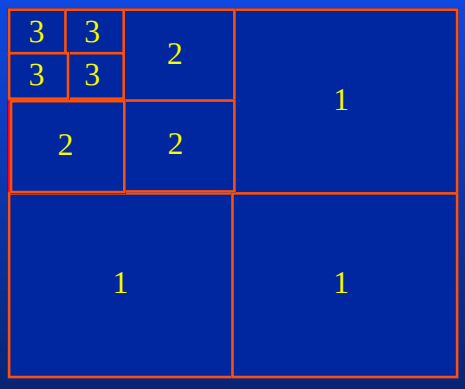
Genus-Reducing Simplifications



Taosong He, Lichan Hong, Amitabh Varshney, Sidney Wang
IEEE Trans. Vis. & Comp. Graphics 1996

Requirements for View-Dependent Simplifications

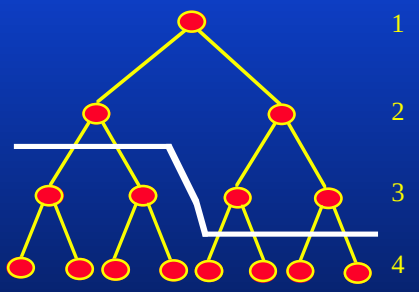
- Varying levels of detail across different regions of object
- Seamless merging
- Real-time determination



Julie C. Xia, Jihad El-Sana, Amitabh Varshney
IEEE Visualization 96,
IEEE Trans on Vis & Comp Graphics, June 97

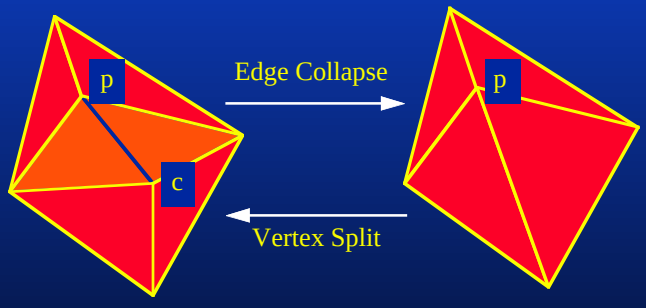
Merge Tree

- Levels in Merge Tree represent levels of detail
- Off-line construction of Merge Tree
- Real-time Retriangulation



Merge Tree Construction

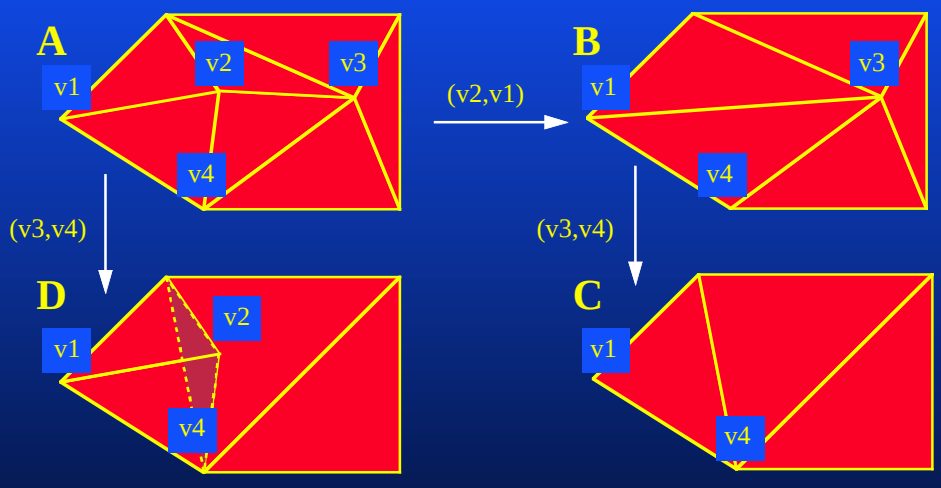
- Construct from high to low detail level
- Determine parent-child relationships
- Order for edge collapses: shortest edge first



Real-Time Triangulation

- Display Vertices and Display Triangles
- Determination of Display Vertices
- Determination of Display Triangles
- Utilize frame-to-frame temporal coherence

Mesh Folding Problem



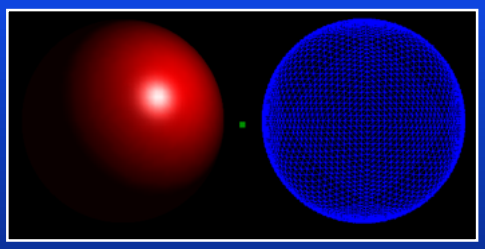
Merge Tree Dependencies

- Sliver triangles
- Edge collapse and vertex split dependencies
- Run-time dependency checking versus floating-point checking

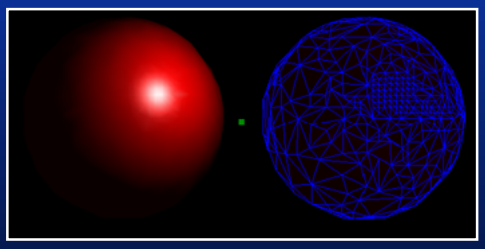
Simplification Factors

- Screen-Space Projection
- Local Illumination
- Visibility Culling
- Silhouettes

Results



8192 triangles



537 triangles

Results

Original (293 K tris, 4.6 secs) Simplified (68 K tris, 1.2 secs)

Dataset Courtesy: Stanford Computer Graphics Lab

Watch out for

- Data degeneracies
- Dataset complexity distribution
- Geometric debugging

Conclusion

Select right level in hierarchy of simplification techniques based on target application

- Triangle strips – almost always
- Genus-preserving – realism then speed, screen querying
- Genus-reducing – speed over topological fidelity
- View-dependent – sci viz, high complexity objects