APPEARANCE-PRESERVING SIMPLIFICATION OF POLYGONAL MODELS

by

Jonathan David Cohen

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

Chapel Hill

1999

Approved by:

Advisor: Professor Dinesh Manocha

Reader: Professor Anselmo Lastra

Reader: Professor Gregory Turk

Professor Frederick Brooks, Jr.

Professor Ming Lin

ABSTRACT

Jonathan David Cohen: Appearance-Preserving Simplification of Polygonal Models

(Under the direction of Dinesh Manocha)

Over the last six years, the automatic simplification of polygonal models has become an important tool for achieving interactive frame rates in the visualization of complex virtual environments. Initial methods employed clever heuristics, but no real quality metrics; then measures of quality for these simplified output models began to develop. This dissertation focuses on the use of error metrics to provide guaranteed error bounds for the simplifications we create. We guarantee bounds on the total appearance of our simplifications with respect to the three major appearance attributes: surface position, surface curvature (normal), and material color.

We present the simplification envelopes and successive mappings algorithms for geometric simplification, two unique approaches that bound the maximum surface-to-surface deviation between the input surface and the simplified output surfaces. We also present the first bound on maximum texture deviation. Using these tools, we develop the first appearance-preserving simplification algorithm. The geometric simplification provides the filtering of the surface position attribute, bounding the error by means of the surface deviation bound. The color and curvature information are stored in texture and normal map structures, which are filtered at run time on a per-pixel basis. The geometry and maps are tied together by the texture deviation metric, guaranteeing not only that the attributes are sampled properly, but that they cover the correct pixels on the screen, all to within a user- or application-supplied tolerance in pixels of deviation.

We have tested these algorithms on polygonal environments composed of thousands of objects and up to a few million polygons, including the auxiliary machine room of a notional submarine model, a lion sculpture from the Yuan Ming garden model, a Ford Bronco model,

a detailed “armadillo” model, and more. The algorithms have proven to be efficient and effective. We have seen improvements of up to an order of magnitude in the frame rate of interactive graphics applications, with little or no degradation in image quality.

ACKNOWLEDGEMENTS

Good models for testing 3D geometric algorithms are hard to come by, and they generally are not created without some painstaking effort. With this in mind, we would like to express our gratitude to those who have graciously provided us with their models: Greg Angelini, Jim Boudreaux, and Ken Fast at the Electric Boat division of General Dynamics for the submarine model; Rich Riesenfeld and Elaine Cohen of the Alpha_1 group at the University of Utah for the rotor model; the Stanford Computer Graphics Laboratory for the bunny and telephone models; Stefan Gottschalk for the wrinkled and bumpy torus models (or, rather, the software tool used for created them); Lifeng Wang and Xing Xing Computer for the lion model from the Yuan Ming Garden; Division and Viewpoint for the Ford Bronco model; and Venkat Krishnamurthy and Marc Levoy at the Stanford Computer Graphics Laboratory and Peter Schröder for the "armadillo" model. These excellent models were essential for the development and presentation of our research.

We would also like to thank those who have provided thoughtful research discussion, as well as the necessary hardware and software to carry out our work: our co-authors in the papers related to this research — Marc Olano, Amitabh Varshney, Greg Turk, Hans Weber, Pankaj Agarwal, Fred Brooks, and Bill Wright; Michael Hohmeyer for his linear programming library; Hansong Zhang for his scene graph library; Greg Turk for his PLY polygon format library; Jeff Erikson for some invaluable discussion on geometric optimization; the now-defunct UNC Simplification Group for providing a great forum for discussing ideas; the UNC Walkthrough Project for inspiring us with increasingly difficult visualization scenarios and allowing us to deal with them in our own ways; the former UNC PixelFlow Project and HP Visualize PxFl team for the years of devotion it took to make the PixelFlow hardware and software a reality; and finally, all the graphics groups at UNC for providing a stellar graphics

laboratory and research environment, where immersion has as much to do with a turbulent sea of ideas as it does with head-mounted displays.

We have received financial support from a variety of sources, both directly and indirectly, for which we are grateful: Alfred P. Sloan Foundation Fellowship, ARO Contract P-34982-MA, ARO Contract DAAH04-96-1-0257, ARO MURI grant DAAH04-96-1-0013, ARPA Contract DABT63-93-C-0048, DARPA Contract DABT63-93-C-0048, Honda, Intel, Link Foundation Fellowship, NIH Grant RR02170, NIH/National Center for Research Resources Award 2P41RR02170-13 on Interactive Graphics for Molecular Studies and Microscopy, NSF/ARPA Center for Computer Graphics and Scientific Visualization, NSF Grant CCR-9319957, NSF Grant CCR-9301259, NSF Grant CCR-9625217, NSF Career Award CCR-9502239, NYI award with matching funds from Xerox Corporation, ONR Contract N00014-94-1-0738, ONR Young Investigator Award, UNC On-Campus Dissertation Fellowship, and U.S.-Israeli Binational Science Foundation grant.

Finally, I would like to personally thank a few of the people who have helped me to reach the completion of this dissertation: Dan Aliaga, Jyoti Chaudry, Bill Mark, Michael North, Bill Yakowenko, and the many other friends who have made my time at UNC enjoyable; Julie Tsao for her exuberant friendship and who's life will always remind me what is important; Marc Olano for his skill and patience at helping me solve my most pesky research problems; Carl Mueller for being the ultimate office-mate — providing friendship, motivation, and assistance at just the right times; my dissertation committee members — Fred Brooks, Anselmo Lastra, Ming Lin, and Greg Turk — for supporting my research, even when the direction was vague, and for testing the quality of this dissertation; my advisor, Dinesh Manocha, for demonstrating the joy and the art of research and for always helping to make the mediocre excellent; Suzy Maska for being with me through this whole process, from the search for a dissertation topic through the completion of the dissertation and the start of a teaching career, for uplifting me at my lowest points, and for motivating me to graduate; and my parents, Dr. Arthur and Sandra Cohen, for being with me even longer, steering me through a happy childhood, advising me, and sharing in my adult life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivation

In 3D computer graphics, polygonal models are often used to represent individual objects and entire environments. Planar polygons, especially triangles, are used primarily because they are easy and efficient to render. Their simple geometry has enabled the development of custom graphics hardware, currently capable of rendering millions or even tens of millions of triangles per second. In recent years, such hardware has become available even for personal computers. Due to the availability of such rendering hardware and of software to generate polygonal models, polygons will continue to play an important role in 3D computer graphics for many years to come.

However, the simplicity of the triangle is not only its main advantage, but its main disadvantage as well. It takes many triangles to represent a smooth surface, and environments of tens or hundreds of millions of triangles or more are becoming quite common in the fields of industrial design and scientific visualization. For instance, in 1994, the UNC Department of Computer Science received a model of a notional submarine from the Electric Boat division

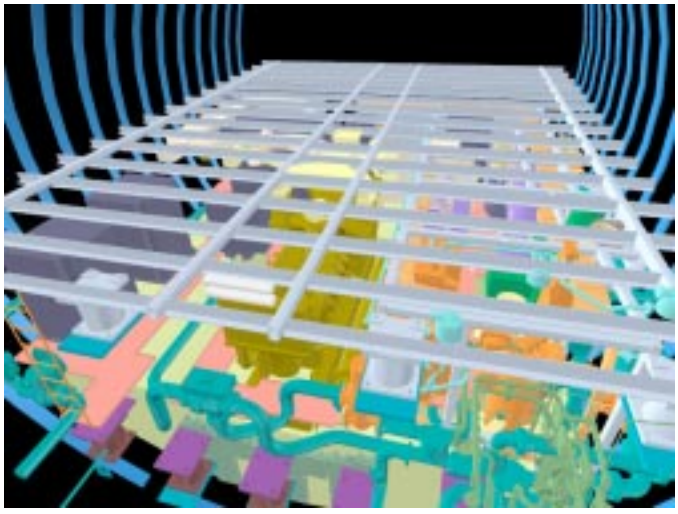


Figure 1: The auxiliary machine room of a notional submarine model: 250,000 triangles

of General Dynamics, including an auxiliary machine room composed of 250,000 triangles (see Figure 1) and a torpedo room composed of 800,000 triangles. In 1997, we received from ABB Engineering a coarsely-tessellated model of an entire coal-fired power plant, composed of over 13,000,000 triangles. It seems that the remarkable performance increases of 3D graphics hardware systems cannot yet match the desire and ability to generate detailed and realistic 3D polygonal models.

1.2 Polygonal Simplification

This imbalance of 3D rendering performance to 3D model size makes it difficult for graphics applications to achieve *interactive* frame rates (10-20 frames per second or more). Interactivity is an important property for applications such as architectural walkthrough, industrial design, scientific visualization, and virtual reality. To achieve this interactivity in spite of the enormity of data, it is often necessary to trade fidelity for speed.

We can enable this speed/fidelity tradeoff by creating a *multi-resolution* representation of our models. Given such a representation, we can render smaller or less important objects in the scene at a lower resolution (i.e. using fewer triangles) than the larger or more important objects, and thus we render fewer triangles overall. Figure 2 shows a widely-used test model: the Stanford bunny. This model was acquired using a laser range-scanning device; it contains over 69,000 triangles. When the 2D image of this model has a fairly large area, this may be a reasonable number of triangles to use for rendering the image. However, if the image is smaller, like Figure 3 or Figure 4, this number of triangles is probably too large. The right-most image in each of these figures shows a bunny with fewer triangles. These complexities are often more appropriate for image of these sizes. Each of these images is typically some small piece of a much larger image of a complex scene.

For CAD models, such representations could be created as part of the process of building the original model. Unfortunately, the robust modeling of 3D objects and environments is already a difficult task, so we would like to explore solutions that do not add extra burdens to the original modeling process. Also, we would like to create such representations for models acquired by other means (e.g. laser scanning), models that already exist, and models in the process of being built.

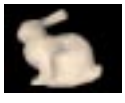
Figure 2: The Stanford bunny model: 69,451 triangles

69,451 triangles

2,204 triangles

Figure 3: Medium-sized bunnies.

69,451 triangles



575 triangles

Figure 4: Small-sized bunnies.

Simplification is the process of automatically reducing the complexity of a given model. By creating one or more simpler representations of the input model (generally called *levels of detail*), we convert it to a multi-resolution form. This problem of automatic simplification is rich enough to provide many interesting and useful avenues of research. There are many issues related to how we represent these multi-resolution models, how we create them, and how we manage them within an interactive graphics application. This dissertation is concerned primarily with the issues of level-of-detail quality and rendering performance. In particular, we explore the question of how to preserve the appearance of the input models to within an intuitive, user-specified tolerance and still achieve a significant increase in rendering performance.

1.3 Thesis Statement

By applying 3D Euclidean distance metrics to the process of geometric simplification and by representing appearance attribute fields in a decoupled form, we can preserve the appearance of polygonal models to within an intuitive, user-specified tolerance while achieving significant increases in rendering performance.

1.4 Design Criteria

In this dissertation, we present techniques for automatically generating and employing simplifications of polygonal models. We have developed these techniques with two major design criteria in mind.

- Provide guaranteed, measured quality in the output models.
- Pre-compute as much as possible, keeping the real-time graphics application as fast as possible.

The first criterion is the one that really defines our work. With guaranteed error bounds available for all our levels of detail, it is possible for a graphics application to automatically choose which level of detail to render for each object without any user-intervention on a per-object basis. This is crucial for complex environments containing thousands of objects or more.

The second criterion is based on empirical observation. Several excellent simplification systems, like those of [Hoppe 1997] and [Luebke and Erikson 1997], now exist that dynamically update the simplification of an object or scene while an interactive graphics application is running. This becomes more feasible as more processing power becomes available on the host machine. However, we have chosen to emphasize run-time efficiency. In our view, those extra CPU cycles available on the host may be required for other application-related purposes, rather than for assisting our level-of-detail management. Our research here explores the limits of what level of quality preservation is possible using only statically-computed levels of detail, allowing us to maximize the processing resources of our graphics computer during interactive applications. This trade-off is discussed further in Section 2.4.3.

1.5 Input Domain

The algorithms we develop operate on *manifold* triangle meshes, including those with *borders*. In the continuous domain, a manifold surface is one that is everywhere homeomorphic to an open disc. In the discrete domain of triangle meshes, such a surface has two topological properties. First, every vertex is adjacent to a set of triangles that form a single, complete cycle around the vertex. Second, each edge is adjacent to exactly two triangles. For a manifold mesh with borders, these restrictions are slightly relaxed. A vertex may be surrounded by a single, incomplete cycle (i.e. the beginning need not meet the end). Also, an edge may be adjacent to either one or two triangles.

A mesh that does not have these properties is said to be *non-manifold*. Such meshes may occur in practice by accident or by design. Accidents are possible, for example, during either the creation of the mesh or during conversions between representation, such as the conversion from a solid to a boundary representation. The correction of such accidents is a subject of much interest [Barequet and Kumar 1997, Murali and Funkhouser 1997]. They may occur by design because such a mesh may require fewer triangles to render than a visually-comparable manifold mesh or because such a mesh may be easier to create in some situations.

Although our algorithms are designed to operate on strictly manifold meshes, and the current implementations reflect this, they may be modified to deal well with meshes that are

"mostly manifold". The simplest such modification might just leave the non-manifold portions of the mesh unchanged from the input surface. A more sophisticated modification may break the non-manifold mesh into a set of manifold meshes with borders, noting the adjacency of these meshes. Such a modification fits well into the patch-based approach described in Chapter 5. We acknowledge, however, that such modifications do not totally solve the problem of non-manifold meshes. If a significant portion of the input mesh is non-manifold, such algorithms may be of limited use for reducing complexity.

1.6 Research Summary

We began this research project in the summer of 1995. At that time, there were relatively few publications on the subject of general polygonal mesh simplification; much of the earlier work focused on the simplification of convex polyhedra [Das and Joseph 1990] and polyhedral terrains [Agarwal and Suri 1994]. There were several publications on the more general problem, though. The most well-known of these were [Rossignac and Borrel 1992], [Schroeder et al. 1992], [Turk 1992], and [Hoppe et al. 1993]. During the course of this research, the field of automatic simplification has become much more active, and quite a few interesting techniques have been developed by other researchers. We discuss only the previous work here; the most relevant of the concurrent work is discussed in Chapter 2.

1.6.1 Previous Work

[Rossignac and Borrel 1992] present a simple, but powerful, scheme based on the clustering of nearby vertices and the removal of any resulting degenerate geometry. This approach is remarkably flexible, and guarantees that the distance of the resulting geometry from the original is no greater than the maximum vertex displacement. Unfortunately, this error bound is quite loose. Also, the resulting geometry is often poorly shaped because no attention is given to the local topology (connectivity) or curvature of the original geometry.

The approaches of [Schroeder et al. 1992] and [Turk 1992], on the other hand, pay close attention to these details, preserving the local topology of the original geometry, and allowing more simplification in regions of lower curvature than in those of higher curvature. These approaches, which produce fairly nice-looking simplifications, provide no error bounds to describe the quality of the final output.

[Hoppe et al. 1993] pose the simplification problem in an optimization framework. Taking a topology-preserving approach, the algorithm performs a sequence of mesh-simplifying operations according to the guidance of an energy function. This function includes terms for distance error, mesh complexity, and an extra spring force. This optimization process provides some confidence that the competing concerns of quality and complexity are balanced in a reasonable fashion. However, the actual metric used for measuring distance is not rigorous; a number of sample points are recorded on the original surface, and their minimum distances from the new surface are measured as the simplification progresses. This metric does not provide a guarantee on the maximum distance, and it is one-sided, with the potential for points on the simplified surface to be quite far from the original surface.

We also knew of another interesting algorithm, which appeared in [Varshney 1994]. The algorithm measures error by building a pair of geometric constraint surfaces around the input surface to some distance tolerance, using these constraints to guarantee a desired error bound. This approach preserves local topology, and provides a guaranteed error bound on the distance from the original surface to the simplified surface. Thus, it achieves the quality appearance of [Schroeder et al. 1992], [Turk 1992], and [Hoppe et al. 1993], as well as the guaranteed error bounds of [Rossignac and Borrel 1992]. However, the running time of the algorithm grows at least quadratically with the size of the input mesh. We took this algorithm as an appealing starting point for our endeavor.

1.6.2 Our Approach

The main goal of our research is to automatically generate simplifications that preserve the appearance of the original models. We define *appearance preservation* as the proper sampling of all the appearance attributes that determine the final, shaded colors of a rendered surface. For current real-time image generation systems, the most common appearance attributes that vary across a surface are position, curvature, and color. Our implementation supports these three attributes. Current off-line rendering systems, such as those supporting the RenderMan shading language [Upstill 1989, Hanrahan and Lawson 1990], allow a myriad of other appearance attributes to vary across a surface. In the limit, such attributes describe a bidirectional reflectance distribution function [Foley et al. 1990] over the surface domain.

Our approach seems general enough to handle a wide variety of such additional attributes as the need arises.

Figure 5 depicts the components of our system. In the left side of the diagram, we convert the original mesh representation to a decoupled form; the position attribute is stored at the polygon vertices, as usual, whereas the color and curvature information is stored in auxiliary texture and normal map structures. These maps are linked to the polygon mesh using a parameterization of the mesh, stored as 2D texture coordinates at the polygon vertices.

We then apply the actual simplification process, shown in the right side of the diagram. We generate the simplified meshes using a surface approximation algorithm that provides guaranteed bounds on the surface deviation (we develop two such surface approximation algorithms: the *simplification envelopes* algorithm and the *successive mapping* algorithm). We augment the surface approximation algorithm with a new texture deviation metric, which guarantees bounds on the texture deviation. Our simplified meshes are thus equipped with both surface and texture deviation bounds, and are supplemented by texture and normal maps, which preserve the other attribute data.

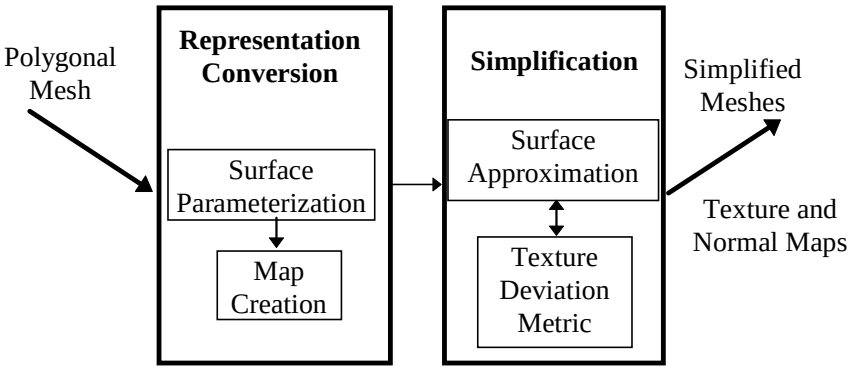


Figure 5: Components of an appearance-preserving simplification system.

When we render the model, we choose an appropriate level of detail to use as the geometry, then perform a mip-mapped look-up to find the appropriate values for the color and normal vector at each pixel covered by the geometry, shading that pixel according to these attribute values. If we consider this data reduction as a filtering process, the simplification pre-processing has taken care of the filtering of the surface position attribute, whereas the run-time mip-mapping properly filters the curvature and color attributes on a per-pixel basis.

The error bounds we compute during the simplification process are essential for guaranteeing the quality of the resulting images. Both the surface and texture deviations are measured as the maximum 3D distances between corresponding points on the original and simplified surfaces. When we apply the current viewing parameters to project these 3D distances into 2D we get an error bound in terms of pixels of deviation. For instance, if the surface and texture deviations project to 2 pixels of deviation, the shaded pixels in an image of the simplified model will be no more than 2 pixels from their correct positions in an image of the original model. This intuitive error bound allows the user or application to automatically control the levels of detail of all the objects in a complex environment, guaranteeing a uniform quality, if desired, and accelerating the frame rate accordingly.

1.6.3 Results

We have applied our simplification algorithms to polygonal environments composed of thousands of objects and up to a few million polygons, including the auxiliary machine room of a notional submarine model, a lion sculpture from the Yuan Ming garden model, a Ford Bronco model, a detailed “armadillo” model, and more. The algorithms have proven to be efficient and effective. We have seen improvements of up to an order of magnitude in the frame rate of interactive graphics applications, with little or no degradation in image quality.

For example, look at the bunnies in Figure 3 and Figure 4. Although the positions of the surfaces are preserved quite well, as evidenced by the similarity of the silhouettes of the bunnies, the shading makes it quite easy to tell which bunnies have been simplified and which have not (i.e. the appearance has not been totally preserved). Figure 6 shows a view of a complex “armadillo” model. We have applied our appearance-preserving algorithm to this model to generate the simplified versions of Figure 7 and Figure 8, in which it is nearly impossible to distinguish the simplifications from the original. This demonstrates that our definition of appearance preservation may match our more intuitive notion of what it means to preserve appearance.

Figure 6: Armadillo model: 249,924 triangles

249,924 triangles

7,809 triangles

Figure 7: Medium-sized armadillos

249,924 triangles

975 triangles

Figure 8: Small-sized armadillos

1.7 Contributions

This dissertation presents three major algorithms: the *simplification envelopes* and *successive mapping* algorithms for geometric simplification, and an overall *appearance-preserving simplification* algorithm. These techniques make a number of contributions to the field of polygonal mesh simplification:

1. Increased robustness and scalability of the simplification envelopes algorithm
2. Local error metric for surface-to-surface deviation between original and simplified surfaces
3. Bijective (one-to-one and onto) mappings between original and simplified surfaces for the edge collapse operation
4. Local error metric for texture deviation, with bijective mappings between original and simplified surfaces
5. Appearance-preserving simplification algorithm
6. Intuitive, screen-space error metric for surface and texture deviations

We now summarize each of these contributions.

1.7.1 Increased Robustness and Scalability of the Simplification Envelopes Algorithm

The simplification envelopes algorithm, which first appeared in [Varshney 1994], has several useful properties: it provides a global error metric for surface-to-surface deviation between original and simplified surfaces, using highly-detailed offset-like surfaces to provide tight error bounds, it preserves global topology, preventing self-intersections that may result in large screen-space artifacts for models built with close geometric tolerances, and it scales well to environments composed of large numbers of objects, automating not only the process of simplification, but the selection of appropriate viewing distances for simplified objects.

We present new techniques for both the creation of envelope surfaces and the simplification of complex meshes using these envelopes. Our new approaches perform only conservative intersection tests between linear elements (line segments and triangles). Intersections are reported conservatively (an intersection is reported if the elements come within some small

tolerance of each other), and actual intersection points are never computed. This approach is more geometrically robust than the original algorithm of [Varshney 1994]. In addition, we have improved the asymptotic running time of the overall simplification algorithm to $O(n\log n)$ for typical input models of n triangles, so it scales well to input meshes of very high complexity.

1.7.2 Local Error Metric for Surface-to-Surface Deviation, with Bijective Mappings between Original and Simplified Surfaces

The successive mapping algorithm provides a local error metric for measuring surface deviation as the simplification process progresses. Although a number of simplification algorithms now provide local error metrics, most provide bounds on the distance from the original vertices to points on the simplified surface, rather than the true surface-to-surface deviation. The tolerance volume simplification algorithm by Guéziec [Guéziec 1995] is a noteworthy exception. The two-sided Hausdorff metric algorithm by Klein [Klein et al. 1996] is also an exception, though the one-sided metric he advocates does not provide such an error bound. However, neither of these other algorithms provides a bijective (one-to-one and onto) mapping between the original and simplified surfaces, nor do they extend gracefully to deal with other important attribute fields.

1.7.3 Bijective Mappings between Original and Simplified Surfaces for the Edge Collapse Operation

Our successive mapping algorithm is the first edge-collapse-based simplification algorithm to provide bijective mappings among the levels of detail. The wavelet-based algorithms [DeRose et al. 1993] provide bijective mappings for the unsubdivide operation and the mapping algorithm of [Bajaj and Schikore 1996] provides a bijective mapping for the vertex remove operation (though it does not provide correct error bounds for this mapping). We use this mapping in our edge-collapse-based simplification algorithm both to bound the surface deviation error and to maintain a texture coordinate parameterization of the input surface through the simplification process. Such a mapping has many potential uses, including measuring and localizing the deviations of other attribute fields.

1.7.4 Local Error Metric for Texture Deviation between Original and Simplified Surfaces

The successive mapping simplification algorithm may be used not only to maintain a texture coordinate parameterization of the input surface, but also to bound the maximum texture deviation between the original and simplified surfaces, measuring it as a 3D distance between corresponding points of the 2D texture domain. Such an error bound is important for models rendered with generic, re-usable texture maps as well as for models with per-vertex attributes stored in texture, normal, or other attribute maps. These models have been common in the off-line rendering community for some time, and models with texture maps have become quite common in the real-time graphics community as well. As the graphics acceleration hardware becomes capable of performing more complex shading operations, models with these sorts of various maps will likely become increasingly common.

1.7.5 Appearance-preserving Simplification Algorithm

Our appearance-preserving simplification algorithm is the first to preserve the shaded appearance of the original model in the simplified levels of detail it creates. It guarantees proper sampling of the surface position, surface curvature, and material color attributes of the original model, which fully determine the its final appearance in rendered images. To accomplish this, the algorithm first decouples the sampling rates of these attributes by storing the model's per-vertex color and normal vectors in texture and normal maps, respectively. The simplification algorithm then filters the model's surface position attribute, guaranteeing bounds on both the surface and texture deviations. The color and normal data is filtered on a per-pixel basis at the time of rendering. The algorithm thus guarantees that the attributes are sampled properly across the surface and that they are mapped to the correct pixels on the screen, within a user- or application-specified error tolerance.

1.7.6 Intuitive, Screen-space Error Metric Incorporating both Surface and Texture Deviations

The approach we take to appearance preservation provides an intuitive, screen-space error metric. The simplification algorithm measures both the surface deviation and texture deviation as the maximum lengths of displacement vectors in 3D. During an interactive visualization application, we use the current viewing parameters to convert these 3D lengths to bounds

on 2D, screen-space displacements. These 2D bounds are measured in pixels of displacement. They tell the user and the application the maximum pixel length of any apparent shift in the rendered image due to replacing the original model with a particular level of detail. In a way, this metric views the results of our simplification algorithm as some form of image warp, with bounds on the maximum pixel displacement of the warp. This metric has an intuitive feel, avoiding complicated issues such as determining the effects of further reduction of the color and normal data on the perceived appearance of the rendered image.

1.8 Organization

The remainder of the dissertation is organized as follows. Chapter 2 provides a summary of the techniques used by many surface approximation algorithms, including known bounds on complexity and optimality, local simplification operations, multi-resolution representations, methods of measuring error, and representation and preservation of appearance attributes. It is organized by topic, rather than by researcher or research project. Although it does not cover the body of simplification literature in its entirety, it covers the topics that are most relevant to this dissertation and our sub-topics of interest in this field.

Chapter 3 presents the idea of *simplification envelopes* for generating a hierarchy of level-of-detail approximations for a given polygonal model. These envelopes are geometric constructions that use global information to bound the error of the simplification process. The 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 simplification algorithms, one local, the other global. The local algorithm uses the vertex remove operation to provide 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 the approach are: it is a general technique providing guaranteed error bounds for genus-preserving simplification,

it provides automation of both the simplification process and the selection of appropriate viewing distances, it prevents self-intersections in the output models, it preserves sharp features, and it allows variation of approximation distance across different portions of a model.

In Chapter 4, we present a more incremental, local approach to guaranteeing error bounds, using mapping functions. We develop a piece-wise linear mapping function for each simplification operation and use this function to measure deviation of the new surface from both the previous level of detail and from the original surface. In addition, we use the mapping function to compute appropriate texture coordinates if the original map has texture coordinates at its vertices. Our overall algorithm uses edge collapse operations. We present rigorous procedures for the generation of local planar projections as well as for the selection of a new vertex position for the edge collapse operation. Our algorithm is able to compute tight error bounds on surface deviation and produce an entire continuum of levels of detail with mappings between them. We demonstrate the effectiveness of our algorithm on several models: a Ford Bronco consisting of over 300 parts and 70,000 triangles, a textured lion model consisting of 49 parts and 86,000 triangles, and a textured, wrinkled torus consisting of 79,000 triangles.

Chapter 5 builds on the techniques of Chapter 4 to develop a new algorithm for appearance-preserving simplification. Not only does it generate a low-polygon-count approximation of a model, but it also preserves the appearance. This is accomplished for a particular display resolution in the sense that we properly sample the surface position, curvature, and color attributes of the input surface. We convert the input surface to a representation that decouples the sampling of these three attributes, storing the colors and normals in texture and normal maps, respectively. Our simplification algorithm employs a new *texture deviation metric*, which guarantees that these maps shift by no more than a user-specified number of pixels on the screen. The simplification process filters the surface position, and the run-time system filters the colors and normals on a per-pixel basis. We have applied our simplification technique to several large models achieving significant amounts of simplification with little or no loss in rendering quality.

Finally, Chapter 6 closes the dissertation, discussing future work and putting our simplification techniques into the context of the general rendering acceleration problem.

2. BACKGROUND

This chapter reviews some fundamental concepts necessary to understand algorithms for simplification of polygonal models at a high level. These concepts include optimal/near-optimal solutions for the simplification problem, the use of local simplification operations, topology preservation, level-of-detail representations for polygonal models, error measures for surface deviation, and the preservation of appearance attributes. It should be noted that the research of this dissertation was performed over the period from 1995 through 1998, so some of the related work discussed here should be considered previous work, whereas some of it is more properly classified as contemporary to this dissertation. This is not a complete survey of the field of polygonal model simplification, which has grown to be quite large. For more information, several survey papers are available [Erikson 1996, Heckbert and Garland 1997].

2.1 Optimality

There are two common formulations of the simplification problem, described in [Varshney 1994], to which we may seek optimal solutions:

- **Min-# Problem**: Given some error bound, ε, and an input model, *I*, compute the minimum complexity approximation, *A*, such that no point of *A* is farther than ε distance away from *I* and vice versa (the complexity of *A* is measured in terms of number of vertices or faces).
- **Min-ε Problem**: Given some target complexity, *n*, and an input model, *I*, compute the approximation, *A*, with the minimum error, ε, described above.

In computational geometry, it has been shown that computing the min-# problem is NP-hard for both convex polytopes [Das and Joseph 1990] and polyhedral terrains [Agarwal and Suri 1994]. Thus, algorithms to solve these problems have evolved around finding polynomial-time approximations that are *close* to the optimal.

Let k_0 be the size of a min-# approximation. An algorithm has been given in [Mitchell and Suri 1992] for computing an ε-approximation of size $O(k_0 \log n)$ for convex polytopes of initial complexity n. This has been improved by Clarkson in [Clarkson 1993]; he proposes a randomized algorithm for computing an approximation of size $O(k_0 \log k_0)$ in expected time $O(k_0 n^{1+\delta})$ for any $\delta > 0$ (the constant of proportionality depends on δ, and tends to $+\infty$ as δ tends to 0). In [Brönnimann and Goodrich 1994] 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, [Agarwal and Suri 1994] present a polynomial-time algorithm that computes an ε-approximation of size $O(k_0 \log k_0)$ to a polyhedral terrain.

Because the surfaces requiring simplification may be quite complex (tens of thousands to millions of triangles), the simplification algorithms used in practice must be $o(n^2)$ (typically $O(n \log n)$) for the running time to be reasonable. Due to the difficulty of computing near-optimal solutions for general polygonal meshes and the required efficiency, most of the algorithms described in the computer graphics literature employ local, greedy heuristics to achieve what appear to be reasonably good simplifications with no guarantees with respect to the optimal solution.

2.2 Local Simplification Operations

Simplification is often achieved by performing a series of local operations. Each such operation serves to coarsen the polygonal model by some small amount. A simplification algorithm generally chooses one of these operation types and applies it repeatedly to its input surface until the desired complexity is achieved for the output surface.

2.2.1 Vertex Remove

The vertex remove operation involves removing from the surface mesh a single vertex and all the triangles touching it. This removal process creates a hole that we then fill with a new set of triangles. Given a vertex with n adjacent triangles, the removal process creates a hole with n sides. The hole filling problem involves a discrete choice from among a finite number of possible retriangulations for the hole. The n triangles around the vertex are replaced by this new triangulation with n-2 triangles. The Catalan sequence,

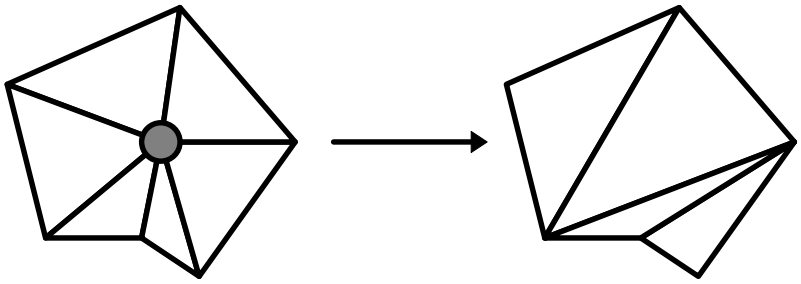


Figure 9: Vertex remove operation

$$C(i) = \frac{1}{i+1} * \binom{2i}{i} = \frac{1}{i+1} * \frac{(2i)!}{i!(2i-i)!} = \frac{1}{i+1} * \frac{(2i)!}{i!i!} = \frac{(2i)!}{(i+1)!i!}, \qquad (1)$$

describes the number of unique ways to triangulate a convex, planar polygon with i+2 sides [Dörrie 1965, Plouffe and Sloan 1995]. This provides an upper bound on the number of non-self-intersecting triangulations of a hole in 3D. For example, holes with 3 sides have only 1 triangulation, and holes with 4, 5, 6, 7, 8, and 9 sides have up to 2, 5, 14, 42, 132, and 429 triangulations, respectively.

Both [Turk 1992] and [Schroeder et al. 1992] apply the vertex remove approach as part of their simplification algorithms. Turk uses point repulsion (weighted according to curvature) to distribute some number of new vertices across the original surface, then applies vertex remove operations to remove most of the original vertices. Holes are retriangulated using a planar projection approach. Schroeder also uses vertex remove operations to reduce mesh complexity, employing a recursive loop splitting algorithm to fill the necessary holes.

2.2.2 Edge Collapse

The edge collapse operation has become popular in the graphics community in the last several years. The two vertices of an edge are merged into a single vertex. This process distorts all the neighboring triangles. The triangles that contain both of the vertices (i.e. those that touch the entire edge) degenerate into 1-dimensional edges and are removed from the mesh. This typically reduces the mesh complexity by 2 triangles.

Whereas the vertex remove operation amounts to making a discrete choice of triangulations, the edge collapse operation requires us to choose the coordinates of the new vertex from a continuous domain. Common choices for these new coordinates include the coordi-

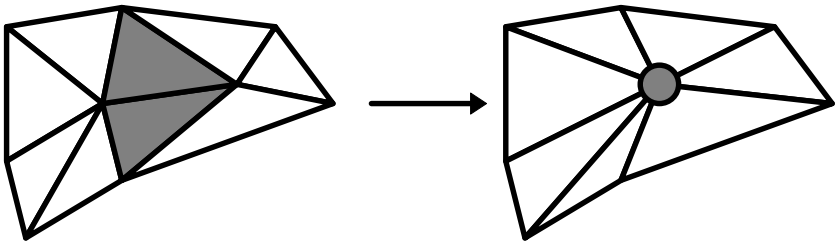


Figure 10: Edge collapse operation

nates of one of the two original vertices, the midpoint of the collapsed edge, arbitrary points along the collapsed edge, or arbitrary points in the neighborhood of the collapsed edge.

Not only is the choice of new vertex coordinates for the edge collapse a continuous problem, but the actual edge collapse operation may be performed continuously in time. We can linearly interpolate the two vertices from their original positions to the final position of the new vertex. This allows us to create smooth transitions as we change the mesh complexity. As described in [Hoppe 1996], we can even perform *geomorphs*, which smoothly transition between versions of the model with widely varying complexity by performing many of these interpolations simultaneously.

In terms of the ability to create identical simplifications, the vertex removal and edge collapse operations are not equivalent. If we collapse an edge to one of its original vertices, we can create n of the triangulations possible with the vertex remove, but there are still $C(n+2)-n$ triangulations that the edge collapse cannot create. Of course, if we allow the edge collapse to choose arbitrary coordinates for its new vertex, it can create infinitely many simplifications that the vertex remove operation cannot create. For a given input model and desired output complexity, it is not clear which type of operation can achieve a closer approximation to the input model.

The edge collapse was used by [Hoppe et al. 1993] as part of a mesh optimization process that employed the vertex remove and edge swap operations as well (the edge swap is a discrete operation that takes two triangles sharing an edge and swaps which pair of opposite vertices are connected by the edge). In [Hoppe 1996], the vertex remove and edge swaps are discarded, and the edge collapse alone is chosen as the simplification operation, allowing a simpler system that can take advantage of the features of the edge collapse. Although systems employing multiple simplification operations might possibly result in better simplifications,

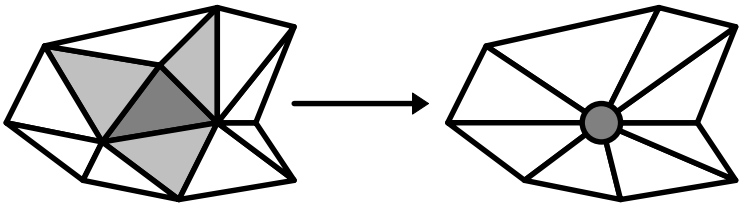


Figure 11: Face collapse operation

they are generally more complex and cannot typically take advantage of the inherent features of any one operation.

2.2.3 Face Collapse

The face collapse operation is similar to the edge collapse operation, except that it is more coarse-grained. All three vertices of a triangular face are merged into a single vertex. This causes the original face to degenerate into a point and three adjacent faces to degenerate into line segments, removing a total of four triangles from the model. The coarser granularity of this operation may allow the simplification process to proceed more quickly, at the expense of the fine-grained local control of the edge collapse operation. Thus, the error is likely to accumulate more quickly for a comparable reduction in complexity. [Hamann 1994, Gieng et al. 1997] use the face collapse operation in their simplification systems. The new vertex coordinates are chosen to lie on a local quadratic approximation to the mesh. Naturally, it is possibly to further generalize these collapse operations to collapse even larger connected portions of the input model. It may even be possible to reduce storage requirements by grouping nearby collapse operations with similar error bounds into larger collapse operations. Thus, the fine-grained control may be traded for reduced storage and other overhead requirements in certain regions of the model.

2.2.4 Vertex Cluster

Unlike the preceding simplification operations, the vertex cluster operation relies solely on the geometry of the input (i.e. the vertex coordinates) rather than the topology (i.e. the adjacency information) to reduce the complexity. Like the edge and face collapses, several vertices are merged into a single vertex. However, rather than merging a set of topologically adjacent vertices, a set of “nearby” vertices are merged [Rossignac and Borrel 1992]. For instance, one possibility is to merge all vertices that lie within a particular 3D axis-aligned

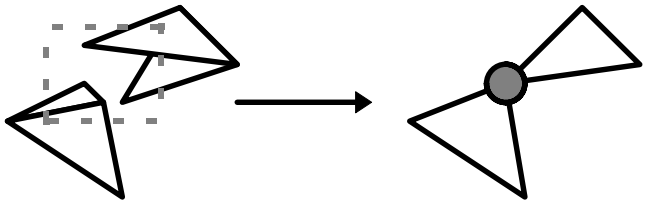


Figure 12: Vertex Cluster operation

box. The new, merged vertex may be one of the original vertices that "best represents" the entire set, or it may be placed arbitrarily to minimize some error bound. An important property of this operation is that it can be robustly applied to arbitrary sets of triangles, whereas all the preceding operations assume that the triangles form a connected, manifold mesh.

The effects of this vertex cluster are similar to those of the collapse operations. Some triangles are distorted, whereas others degenerate to a line segment or a point. In addition, there may be coincident triangles, line segments, and points originating from non-coincident geometry. One may choose to render the degenerate triangles as line segments and points, or one may simply not render them at all. Depending on the particular graphics engine, rendering a line or a point may not be much faster than rendering a triangle. This is an important consideration, because achieving a speed-up is one of the primary motivations for simplification.

There is no point in rendering several coincident primitives, so multiple copies are filtered down to a single copy. However, the question of how to render coincident geometry is complicated by the existence of other surface attributes, such as normals and colors. For instance, suppose two triangles of wildly different colors become coincident. No matter what color we render the triangle, it may be noticeably incorrect.

[Rossignac and Borrel 1992] use the vertex clustering operation in their simplification system to perform very fast simplification on arbitrary polygonal models. They partition the model space with a uniform grid, and vertices are collapsed within each grid cell. [Luebke and Erikson 1997] build an octree hierarchy rather than a grid at a single resolution. They dynamically collapse and split the vertices within an octree cell depending on the current size of the cell in screen space as well as silhouette criteria.

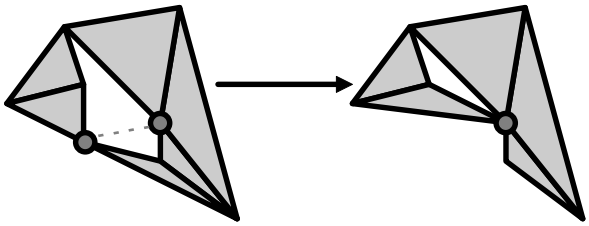


Figure 13: Generalized edge collapse operation

2.2.5 Generalized Edge Collapse

The generalized edge collapse operation combines the fine-grained control of the edge collapse operation with the generality of the vertex cluster operation. Like the edge collapse operation, it involves the merging of two vertices and the removal of degenerate triangles. However, like the vertex cluster operation, it does not require that the merged vertices be topologically connected (by a topological edge), nor does it require that topological edges be manifold.

[Garland and Heckbert 1997] apply the generalized edge collapse in conjunction with error quadrics to achieve simplification that gives preference to the collapse of topological edges, but also allows the collapse of virtual edges (arbitrary pairs of vertices). These virtual edges are chosen somewhat heuristically, based on proximity relationships in the original mesh.

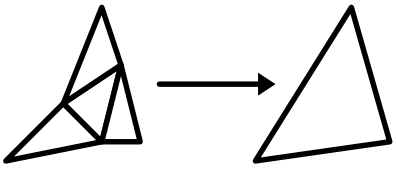


Figure 14: Unsubdivide operation

2.2.6 Unsubdivide

Subdivision surface representations have also been proposed as a solution to the multiresolution problem. In the context of simplification operations, we can think of the "unsubdivide" operation (the inverse of a subdivision refinement) as our simplification operation. A common form of subdivision refinement is to split one triangle into four triangles. Thus the unsubdivide operation merges four triangles of a particular configuration into a single triangle, reducing the triangle count by three triangles.

[DeRose et al. 1993] shows how to represent a subdivision surface at some finite resolution as a sequence of wavelet coefficients. The sequence of coefficients is ordered from lower to higher frequency content, so truncating the sequence at a particular point determines a particular mesh resolution. [Eck et al. 1995] presents an algorithm to turn an arbitrary topology mesh into one with the necessary subdivision connectivity. They construct a base mesh of minimal resolution and guide its refinement to come within some tolerance of the original mesh. This new refined subdivision mesh is used in place of the original mesh, and its resolution is controlled according to the wavelet formulation.

2.3 Topology Preservation

The topological structure of a polygonal surface typically refers to features such as its *genus* (number of topological holes, e.g. 0 for a sphere, 1 for a torus or coffee mug) and the number and arrangement of its *borders* (chains of edges that are adjacent to a single face rather than a pair of faces). These features are fully determined by the adjacency graph of the vertices, edges, and faces of a polygonal mesh. For manifold meshes with no borders (i.e. closed surfaces), the Euler equation holds:

$$F - E + V = 2 - G, \tag{2}$$

where F is the number of faces, E is the number of edges, V is the number of vertices, and G is the genus.

In addition to this combinatorial description of the topological structure, the embedding of the surface in 3-space impacts its perceived topology in 3D renderings. Generally, we expect the faces of a surface to intersect only at their shared edges and vertices.

Most of the simplification operations described in section 2.2 (all except the vertex cluster and the generalized edge collapse) preserve the connectivity structure of the mesh. If a simplification algorithm uses such an operation and also prevents local self-intersections (intersections within the adjacent neighborhood of the operation), we say the algorithm *preserves local topology*. If the algorithm prevents any self-intersections in the entire mesh, we say it *preserves global topology*.

If the simplified surface is to be used for purposes other than rendering (e.g. finite element computations), topology preservation may be essential. For rendering applications, however, it is not always necessary. In fact, it is often possible to construct simplifications with fewer polygons for a given error bound if topological modifications are allowed.

However, some types of topological modifications may have a dramatic impact on the appearance of the surface. For instance, many meshes are the surfaces of solid objects. For example, consider the surface of a thin, hollow cylinder. When the surface is modified by more than the thickness of the cylinder wall, the interior surface will intersect the outer surface. This can cause artifacts that cover a large area on the screen. Problems also occur when polygons with different color attributes become coincident.

Certain types of topological changes are clearly beneficial in reducing complexity, and have a smaller impact on the rendered image. These include the removal of topological holes and thin features (such as the antenna of a car). Topological modifications are encouraged in [Rossignac and Borrel 1992], [Luebke and Erikson 1997], [Garland and Heckbert 1997] and [Erikson and Manocha 1998] and controlled modifications are performed in [He et al. 1996] and [El-Sana and Varshney 1997].

2.4 Level-of-Detail Representations

We can classify the possible representations for level-of-detail models into two broad categories: *static* and *dynamic*. Static levels of details are computed totally off-line. They are fully determined as a pre-process to the visualization program. Dynamic levels of detail are typically computed partially off-line and partially on-line within the visualization program. We now discuss these representations in more detail.

2.4.1 Static Levels of Detail

The most straightforward level-of-detail representation for an object is a set of independent meshes, where each mesh has a different number of triangles. A common heuristic for the generation of these meshes is that the complexity of each mesh should be reduced by a factor of two from the previous mesh. Such a heuristic generates a reasonable range of complexities, and requires only twice as much total memory as the original representation.

It is common to organize the objects in a virtual environment into a hierarchical *scene graph* [van Dam 1988, Rohlf and Helman 1994]. Such a scene graph may have a special type of node for representing an object with levels of detail. When the graph is traversed, this level-of-detail node is evaluated to determine which child branch to traverse (each branch represents one of the levels of detail). In most static level-of-detail schemes, the children of the level-of-detail nodes are the leaves of the graph. [Erikson and Manocha 1998] presents a scheme for generating *hierarchical levels of detail*. This scheme generates level-of-detail nodes throughout the hierarchy rather than just at the leaves. Each such interior level-of-detail node involves the merging of objects to generate even simpler geometric representations. This overcomes one of the previous limitations of static levels of detail — the necessity for choosing a single scale at which objects are identified and simplified.

The transitions between these levels of detail are typically handled in one of three ways: discrete, blended, or morphed. The discrete transitions are instantaneous switches; one level of detail is rendered during one frame, and a different level of detail is rendered during the following frame. The frame at which this transition occurs is typically determined based on the distance from the object to the viewpoint. This technique is the most efficient of the three transition types, but also results in the most noticeable artifacts.

Blended transitions employ alpha-blending to fade between the two levels of detail in question. For several frames, both levels of detail are rendered (increasing the rendering cost during these frames), and their colors are blended. The blending coefficients change gradually to fade from one level of detail to the other. It is possible to blend over a fixed number of frames when the object reaches a particular distance from the viewpoint, or to fade over a fixed range of distances [Rohlf and Helman 1994]. If the footprints of the objects on the screen are not identical, blending artifacts may still occur at the silhouettes.

Morphed transitions involve gradually changing the shape of the surface as the transition occurs. This requires the use of some correspondence between the two levels of detail. Only one representation must be rendered for each frame of the transition, but the vertices require some interpolation each frame. For instance, [Hoppe 1996] describes the *geomorph* transition for levels of detail created by a sequence of edge collapses. The simpler level of detail was

originally generated by collapsing some number of vertices, and we can create a transition by simultaneously interpolating these vertices from their positions on one level of detail to their positions on the other level of detail. Thus the number of triangles we render during the transition is equal to the maximum of the numbers of triangles in the two levels of detail. It is also possible to morph using a mutual tessellation of the two levels of detail, as in [Turk 1992], but this requires the rendering of more triangles during the transition frames.

2.4.2 Dynamic Levels of Detail

Dynamic levels of detail provide representations that are more carefully tuned to the viewing parameters of each particular rendered frame. Due to the sheer number of distinct representations this requires, each representation cannot simply created and stored independently. The common information among these representations is used to create a single representation for each simplified object. From this unified representation, a geometric representation that is tuned to the current viewing parameters is extracted. The coherence of the viewing parameters enables incremental modifications to the geometry rendered in the previous frame; this makes the extraction process feasible at interactive frame rates.

[Hoppe 1996] presents a representation called the *progressive mesh*. This representation is simply the original object plus an ordered list of the simplification operations performed on the object. It is generally more convenient to reverse the order of this intuitive representation, representing the simplest *base mesh* plus the inverse of each of the simplification operations. Applying all of these inverse operations to the base mesh will result in the original object representation. A particular level of detail of this progressive mesh is generated by performing some number of these operations.

In [Hoppe 1997], the progressive mesh is reorganized into a vertex hierarchy. This hierarchy is a tree that captures the dependency of each simplification operation on certain previous operations. Similar representations include the *merge tree* of [Xia et al. 1997], the *multiresolution model* of [Klein and Krämer 1997], the *vertex tree* of [Luebke and Erikson 1997], and the *multi-triangulation* of [DeFloriani et al. 1997]. Such hierarchies allow selective refinement of the geometry based on various metrics for screen-space deviation, normal deviation, color deviation, and other important features such as silhouettes and specular

highlights. A particular level of detail may be expressed as a *cut* through these graphs, or a *front* of vertex nodes. Each frame, the nodes on the current front are examined, and may cause the graph to be refined at some of these nodes.

[DeFloriani et al. 1997] discuss the properties of such hierarchies in terms of graph characteristics. Examples of these properties include compression ratio, linear growth, logarithmic height, and bounded width. They discuss several different methods of constructing such hierarchies and test these methods on several benchmarks. For example, one common heuristic for building these hierarchies is to choose simplification operations in a greedy fashion according to an error metric. Another method is to choose a set of operations with disjoint areas of influence on the surface and apply this entire set before choosing the next set. The former method does not guarantee logarithmic height, whereas the latter does. Such height guarantees can have practical implications in terms of the length of the chain of dependent operations that must be performed in order to achieve some particular desired refinement.

[DeRose et al. 1993] present a wavelet-based representation for surfaces constructed with subdivision connectivity. [Eck et al. 1995] make this formulation applicable to arbitrary triangular meshes by providing a remeshing algorithm to approximate an arbitrary mesh by one with the necessary subdivision connectivity. Both the remeshing and the filtering/reconstruction of the wavelet representation provide bounded error on the surfaces generated. [Lee et al. 1998] provide an alternate remeshing algorithm based on a smooth, global parameterization of the input mesh. Their approach also allows the user to constrain the parameterization at vertices or along edges of the original mesh to better preserve important features of the input.

2.4.3 Comparison

Static levels of detail allow us to perform simplification entirely as a pre-process. The real-time visualization system performs only minimal work to select which level of detail to render at any given time. Because the geometry does not change, it may be rendered in retained mode (i.e. from cached, optimized *display lists*). Retained-mode rendering should always be at least as fast as immediate mode rendering, and is much faster on most current high-end hardware. Perhaps the biggest shortcoming of using static levels of detail is that

they require that we partition the model into independent “objects” for the purpose of simplification. If an object is large with respect to the user or the environment, especially if the viewpoint is often contained inside the object, little or no simplification may be possible. This may require that such objects be subdivided into smaller objects, but switching the levels of detail of these objects independently causes visible cracks, which are non-trivial to deal with.

Dynamic levels of detail perform some of simplification as a pre-process, but defer some of the work to be computed by the real-time visualization system at run time. This allows us to provide more fine-tuning of the exact tessellation to be used, and allows us to incorporate more view-dependent criteria into the determination of this tessellation. The shortcoming of such dynamic representations is that they require more computation in the visualization system as well as the use of immediate mode rendering. Also, the memory requirements for such representations are often somewhat larger than for the static levels of detail.

2.5 Surface Deviation Error Bounds

Measuring the deviation of a polygonal surface as a result of simplification is an important component of the simplification process. This surface deviation error gives us an idea of the quality of a particular simplification. It helps guide the simplification process to produce levels of detail with low error, determine when it is appropriate to show a particular level of detail of a given surface, and optimize the levels of detail for an entire scene to achieve a high overall image quality for the complexity of the models actually rendered.

2.5.1 Distance Metrics

Before discussing the precise metrics and methods used by several researchers for measuring surface deviation, we consider two formulations of the distance between two surfaces. These are the Hausdorff distance and the mapping distance. The Hausdorff distance is a well-known concept from topology, used in image processing as well as surface modeling, and the mapping distance is a commonly used metric for parametric surfaces.

2.5.1.1 Hausdorff Distance

The Hausdorff distance is a distance metric between point sets. Given two sets of points, *A* and *B*, the Hausdorff distance is defined as

$$H(A,B) = \max(h(A,B), h(B,A)), \tag{3}$$

where

$$h(A,B) = \max_{a \in A} \min_{b \in B} \| a - b \|. \tag{4}$$

Thus the Hausdorff distance measures the farthest distance from a point in one point set to its closest point in the other point set (notice that $h(A,B) \neq h(B,A)$). Because a surface is a particular type of continuous point set, the Hausdorff distance provides a useful measure of the distance between two surfaces.

2.5.1.2 Mapping Distance

The biggest shortcoming of the Hausdorff distance metric for measuring the distance between surfaces is that it makes no use of the point neighborhood information inherent in the surfaces. The function $h(A,B)$ implicitly assigns to each point of surface *A* the closest point of surface *B*. However, this mapping may have discontinuities. If points *i* and *j* are neighboring points on surface *A* (i.e. there is a path on the surface of length no greater than ε that connects them), their corresponding points, *i´* and *j´*, on surface *B* may not be neighboring points. In addition, the mapping implied by $h(A,B)$ is not identical to the mapping implied by $h(B,A)$.

For the purpose of simplification, we would like to establish a continuous mapping between the surface's levels of detail. Ideally, the correspondences described by this mapping should coincide with a viewer's perception of which points are "the same" on the surfaces. Given such a continuous mapping

$$F: A \rightarrow B$$

the mapping distance is defined as

$$D(F) = \max_{a \in A} \| a - F(a) \|. \tag{5}$$

Because there are many such mappings, there are many possible mapping distances. The minimum mapping distance is simply

$$D_{min} = \min_{F \in M} D(F), \qquad (6)$$

where *M* is the set of all such continuous mapping functions. Note that although D_{min} and its associated mapping function may be difficult to compute, all continuous mapping functions provide an upper bound on D_{min}.

2.5.2 Surface Deviation Algorithms

We now classify several simplification algorithms according to how they measure the surface deviation error of their levels of detail.

2.5.2.1 Mesh Optimization

[Hoppe et al. 1993] pose the simplification problem in terms of optimizing an energy function. This function has terms corresponding to number of triangles, surface deviation error, and a heuristic spring energy. To quantify surface deviation error, they maintain a set of point samples from the original surface and their closest distance to the simplified surface. The sum of squares of these distances is used as the surface deviation component of the energy function. The spring energy term is required because the surface deviation error is only measured in one direction: it approximates the closest distance from the original surface to the simplified surface, but not vice versa. Without this term, small portions of the simplified surface can deviate quite far from the original surface, as long as all the point samples are near to some portion of the simplified surface.

2.5.2.2 Vertex Clustering

[Rossignac and Borrel 1993] present a simple and general algorithm for simplification using vertex clustering. The vertices of each object are clustered using several different sizes of uniform grid. The surface deviation in this case is a Hausdorff distance and must be less than or equal to the size of grid cell used in determining the vertex clusters. This is a very conservative bound, however. A slightly less conservative bound is the maximum distance from a vertex in the original cluster to the single representative vertex after the cluster is collapsed. Even this bound is quite conservative in many cases; the actual maximum devia-

tion from the original surface to the simplified surface may be considerably smaller than the distance the original vertices travel during the cluster operation.

[Luebke and Erikson 1997] take a similar approach, but their system uses an octree instead of a single-resolution uniform grid. This allows them to take a more dynamic approach, folding and unfolding octree cells at run-time and freely merging nearby objects. The measure of surface deviation remains the same, but they allow a more flexible choice of error tolerances in their run-time system. In particular, they use different tolerances for silhouette and non-silhouette clusters.

2.5.2.3 Superfaces

[Kalvin and Taylor 1996] present an efficient simplification algorithm based on merging adjacent triangles to form polygonal patches, simplifying the boundaries of these patches, and finally retriangulating the patches themselves. This algorithm guarantees a maximum deviation from vertices of the original surface to the simplified surface and from vertices of the simplified surface to the original surface. Unfortunately, even this bidirectional bound does not guarantee a maximum deviation between points on the simplified surface and points on the original surface. For instance, suppose we have two adjacent triangles that share an edge, forming a non-planar quadrilateral. If we retriangulate this quadrilateral by performing an edge swap operation, the maximum deviation between these two surfaces is non-zero, even though their four vertices are unchanged (thus the distance measured from vertex to surface is zero).

2.5.2.4 Error Tolerance Volumes

[Guéziec 1995] presents a simplification system that measures surface deviation using error volumes built around the simplified surface. These volumes are defined by spheres, specified by their radii, centered at each of the simplified surface's vertices. We can associate with any point in a triangle a sphere whose radius is a weighted average of the spheres of the triangle's vertices. The error volume of an entire triangle is the union of the spheres of all the points on the triangle, and the error volume of a simplified surface is the union of the error volumes of its triangles. As edge collapses are performed, not only are the coordinates of the new vertex computed, but new sphere radii are computed such that the new error volume

contains the previous error volume. The maximum sphere radius is a bound on the Hausdorff distance of the simplified surface from the original, and thus provides a bound for surface deviation in both 3D and 2D (after perspective projection).

2.5.2.5 Error Quadrics

[Ronfard and Rossignac 1996] describe a fast method for approximating surface deviation. They represent surface deviation error for each vertex as a sum of squared distances to a set of planes. The initial set of planes for each vertex are the planes of its adjacent faces. As vertices are merged, the sets of planes are unioned. This metric provides a useful and efficient heuristic for choosing an ordering of edge collapse operations, but it does not provide any guarantees about the maximum or average deviation of the simplified surface from the original.

[Garland and Heckbert 1997] present some improvements over [Ronfard and Rossignac 1996]. The error metric is essentially the same, but they show how to approximate a vertex's set of planes by a quadric form (represented by a single 4x4 matrix). These matrices are simply added to propagate the error as vertices are merged. Using this metric, it is possible to choose an optimal vertex placement that minimizes the error. In addition, they allow the merging of vertices that are not joined by an edge, allowing increased topological modification. [Erikson and Manocha 1998] further improve this technique by automating the process of choosing which non-edge vertices to collapse and by encouraging such merging to preserve the local surface area.

2.5.2.6 Mapping Error

[Bajaj and Schikore 1996] perform simplification using the vertex remove operation, and measure surface deviation using local, bijective (one-to-one and onto) mappings in the plane between points on the surface just before and just after the simplification operation. This approach provides a fairly tight bound on the maximum deviation over all points on the surface, not just the vertices (as does [Guéziec 1995]) and provides pointwise mappings between the original and simplified surfaces. Our successive mapping algorithm, presented in Chapter 4, is based on this approach, and makes several significant improvements over their original algorithm.

2.5.2.7 Hausdorff Error

[Klein et al. 1996] measure a one-sided Hausdorff distance (with appropriate locality restrictions) between the original surface and the simplified surface. By definition, this approach produces the smallest possible bound on maximum one-sided surface deviation, but the one-sided formulation does not guarantee a true bound on overall maximum deviation. At each step of the simplification process, the Hausdorff distance must be measured for each of the original triangles mapping to the modified portion of the surface. The computation time for each simplification operation grows as the simplified triangles cover more and more of the mesh, but of course, there are also fewer and fewer triangles to simplify. [Klein and Krämer 1997] present an efficient implementation of this algorithm.

2.6 Appearance Attribute Preservation

We now classify several algorithms according to how they preserve the appearance attributes of their input models.

2.6.1 Scalar Field Deviation

The mapping algorithm presented in [Bajaj and Schikore 1996] allows the preservation of arbitrary scalar fields across a surface. Such scalar fields are specified at the mesh vertices and linearly interpolated across the triangles. Their approach computes a bound on the maximum deviation of the scalar field values between corresponding points on the original surface and the simplified surface.

2.6.2 Color Preservation

[Hughes et al. 1996] describes a technique for simplifying colored meshes resulting from global illumination algorithms. They use a logarithmic function to transform the vertex colors into a more perceptually linear space before applying simplification. They also experiment with producing mesh elements that are quadratically- or cubically-shaded in addition to the usual linearly-shaded elements.

[Hoppe 1996] extends the error metric of [Hoppe et al. 1993] to include error terms for scalar attributes and discontinuities as well as surface deviation. Like the surface deviation, the scalar attribute deviation is measured as a sum of squared Euclidean distances in the

attribute space (e.g. the RGB color cube). The distances are again measured between sampled points on the original surface and their closest points on the simplified surface. This metric is useful for prioritizing simplification operations in order of increasing error. However, it does not provide much information about the true impact of attribute error on the final appearance of the simplified object on the screen. A better metric should incorporate some degree of area weighting to indicate how the overall illuminance of the final pixels may be affected.

[Erikson and Manocha 1998] present a method for measuring the maximum attribute deviation in Euclidean attribute spaces. Associated with each vertex is an attribute volume for each attribute being measured. The volume is a disc of the appropriate dimension (i.e. an interval in 1D, a circle in 2D, a sphere in 3D, etc.). Each attribute volumes is initially a point in the attribute space (an *n*-disk with radius zero). As vertex pairs are merged, the volumes grow to contain the volumes of both vertices.

[Certain et al. 1996] present a method for preserving vertex colors in conjunction with the wavelet representation for subdivision surfaces [DeRose et al. 1993]. The geometry and color information are stored as two separate lists of wavelet coefficients. Coefficients may be added or deleted from either of these lists to adjust the complexity of the surface and its geometric and color errors. They also use the surface parameterization induced by the subdivision to store colors in texture maps to render as textured triangles for machines that support texture mapping in hardware.

[Bastos et al. 1997] use texture maps with bicubic filtering to render the complex solutions to radiosity illumination computations. The radiosity computation often dramatically increases the number of polygons in the input mesh in order to create enough vertices to store the resulting colors. Storing the colors instead in texture maps removes unnecessary geometry, reducing storing requirements and rasterization overhead. Our appearance preservation approach, presented in Chapter 5, is in some sense a generalization of this “radiosity as textures” work. Whereas [Bastos et al. 1997] reduces geometry complexity to that of the pre-radiositized mesh, our approach simplifies complex geometry much farther, quantifying the distortions caused by the simplification of non-planar, textured surfaces.

2.6.3 Normal Vector Preservation

[Xia et al. 1997] associate a cone of normal vectors with each vertex during their simplification preprocess. These cones initially have an angle of zero, and grow to contain the cones of the two vertices merged in an edge collapse. Their run-time, dynamic simplification scheme uses this range of normals and the light direction to compute a range of reflectance vectors. When this range includes the viewing direction, the mesh is refined, adapting the simplification to the specular highlights. The results of this approach are visually quite compelling, though they do not allow increased simplification of the highlight area as it gets smaller on the screen (i.e. as the object gets farther from the viewpoint).

[Klein 1998] maintains similar information about the cone of normal deviation associated with each vertex. The refinement criterion takes into account the spread of reflected normals (i.e. the specular exponent, or shininess) in addition to the reflectance vectors themselves. Also, refinement is performed in the neighborhood of silhouettes with respect to the light sources as well as specular highlights. Again, this normal deviation metric does not allow increased simplification in the neighborhood of the highlights and light silhouettes as the object gets smaller on the screen.

3. A GLOBAL ERROR METRIC FOR SURFACE DEVIATION

We now present the first of the three major simplification algorithms of this dissertation — a simplification technique that provides a global error bound on surface deviation. The technique is named for the geometric construct it employs: *simplification envelopes*. Using this technique, we create a small number of levels of detail for each input object (typically $\log_2 n$ levels of detail for an object with an initial complexity of n triangles). For each level of detail we create, we know a 3D error bound, ε, which is an upper bound on the minimum Euclidean distance from every point on the original surface to the level of detail and vice versa. The error bound is global in the sense that we know only a single bound on the minimum distance for the entire object; we have no tighter bound for each particular place on the object's surface. In contrast, Chapter 4 presents a local metric, which provides a more localized bound for the points on each triangle of the levels of detail.

This work was performed in collaboration with Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Fred Brooks, and Bill Wright. The majority of this chapter has appeared in the *Proceedings of SIGGRAPH 96* [Cohen et al. 1996], and some of it has appeared in the dissertation of Amitabh Varshney [Varshney 1994]. In particular, the concept of simplification envelopes, the fundamental prism and edge half-spaces (Section 3.2), the analytical method for computing simplification envelopes (Section 3.3.1), and the global algorithm for generating a level of detail (Section 3.4.3) were part of Amitabh Varshney's dissertation and are thus not new to this dissertation; they are included here for the sake of completeness.

3.1 Overview

Simplification envelopes is a technique 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, though not efficient for large models, 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.

Simplification envelopes are a generalization of *offset surfaces* [Hoffmann 1989]. Given a polygonal representation of an object, they allow the generation of simpler approximations that are guaranteed not to deviate from the original by more than a user-specifiable amount as well as preserving global topology. We surround the original polygonal surface with two envelopes, then perform simplification within this volume (our implementation employs vertex remove operations). A sample level-of-detail hierarchy generated by the algorithms we describe can be seen in Figure 15.

Figure 15: 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 dependent 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 property of *scalability* (i.e. automatically handling a large number of objects) is important for any simplification algorithm. One of our main goals is to create a method for simplification that is not only automatic for large data sets, but tends to preserve the shapes of the original models.

The rest of the chapter is organized in the following manner: we explain our assumptions and terminology in Section 3.2, describe the envelope and approximation computations in Sections 3.3 and 3.4, present some useful extensions to and properties of the approximation algorithms in Section 3.5, develop the use of 3D error bounds to compute screen-space error bounds in Section 3.6, and explain our implementation and results in Section 3.7.

3.2 Terminology and Assumptions

Let us assume that I is a three-dimensional compact and orientable object whose polygonal representation P has been given to us. Our objective is to compute a piecewise-linear approximation A of P. Given two piecewise linear objects P and Q, we say that P and Q are *ε-approximations* of each other iff every point on P is within a distance ε of some point of Q and every point on Q is within a distance ε of some point of P. Our goal is to outline a method to generate two envelope surfaces surrounding P and demonstrate how the envelopes can be used to solve the following polygonal approximation problem. Given a polygonal representation P of an object and an approximation parameter ε, generate a genus-preserving ε-approximation A with fewer polygons such that the vertices of A are a subset of the vertices of P.

We assume that all polygons in P are triangles and that 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 P has a single *normal vector*, which must lie within 90^o of the normal of each of its surrounding triangles. For the purpose of generating envelope surfaces, we compute a this single vertex normal as a combination of the normals of the surrounding triangles. If this normal does not lie within 90^o of the normal of each of its surrounding triangles, the envelope generation will create local self-intersections. Section 3.7.2.2 describes two methods we have tested for computing these vertex normals. Note that for the purpose of rendering, each vertex may have either a single normal for smooth shading or multiple normals in the presence of a visually sharp edge or cusp.

The three-dimensional ε-offset surface for a parametric surface

$$\mathbf{f}(s,t) = (\mathrm{f}_1(s,t), \mathrm{f}_2(s,t), \mathrm{f}_3(s,t)), \tag{7}$$

whose unit normal to **f** is

$$\mathbf{n}(s,t) = (\mathrm{n}_1(s,t), \mathrm{n}_2(s,t), \mathrm{n}_3(s,t)), \tag{8}$$

is defined as

$$\mathbf{f}^{\varepsilon}(s,t) = (\mathrm{f}_1^{\varepsilon}(s,t), \mathrm{f}_2^{\varepsilon}(s,t), \mathrm{f}_3^{\varepsilon}(s,t)), \tag{9}$$

where

$$\mathrm{f}_i^{\varepsilon}(s,t) = \mathrm{f}_i(s,t) + \varepsilon \mathrm{n}_i(s,t). \tag{10}$$

Note that offset surfaces for a polygonal object can self-intersect and may contain non-linear elements (quadric elements, in particular). We define a simplification envelope $P(+\varepsilon)$ (respectively $P(-\varepsilon)$) for an object I to be a *polygonal* surface that lies *within* a distance of ε from every point p on I in the same (respectively opposite) direction as the normal to I at p. Thus, the simplification envelopes can be thought of as an approximation to offset surfaces. Henceforth we shall often refer to "simplification envelopes" by simply "envelopes."

Let us refer to the triangles of the given polygonal representation P as the *fundamental triangles*. Let $e=(v_1,v_2)$ be an edge of P. If the normals $\mathbf{n}_1$, $\mathbf{n}_2$ to I at both v_1 and v_2, respec-

tively, 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 Figure 16(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 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 Figure 16(b).

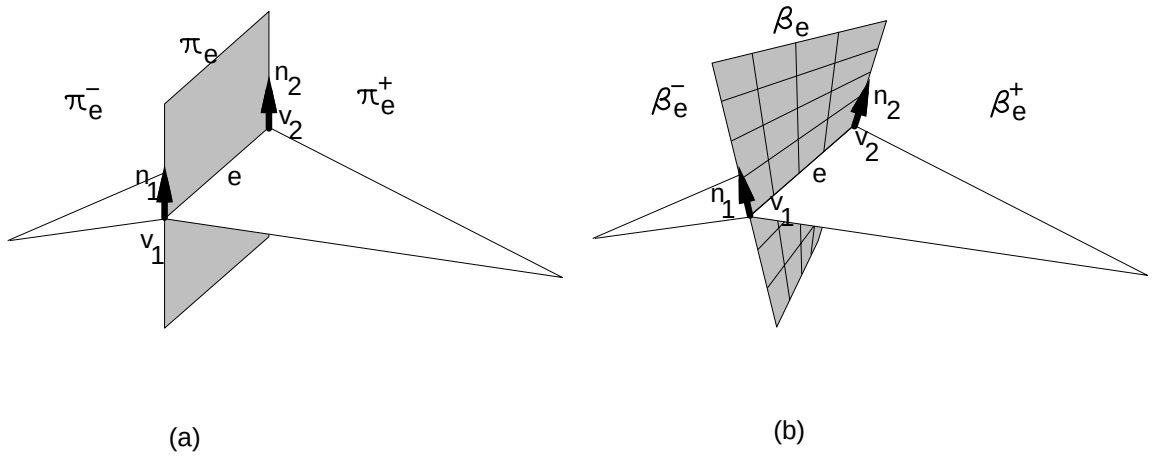


Figure 16: Edge Half-spaces

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 (+ε)-offset vertex v_i^+ for a vertex v_i are: $\mathbf{c}(v_i^+) = \mathbf{c}(v_i) + \varepsilon\, \mathbf{n}(v_i)$, and $\mathbf{n}(v_i^+) = \mathbf{n}(v_i)$. The (-ε) -offset vertex can be similarly defined in the opposite direction. These offset vertices for a fundamental triangle are shown in Figure 17.

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 17) and we will henceforth refer to it as the *fundamental prism*.

3.3 Simplification Envelope Computation

In order to preserve the input topology of P, we desire that the simplification envelopes do not self-intersect. To meet this criterion we reduce our level of approximation at certain

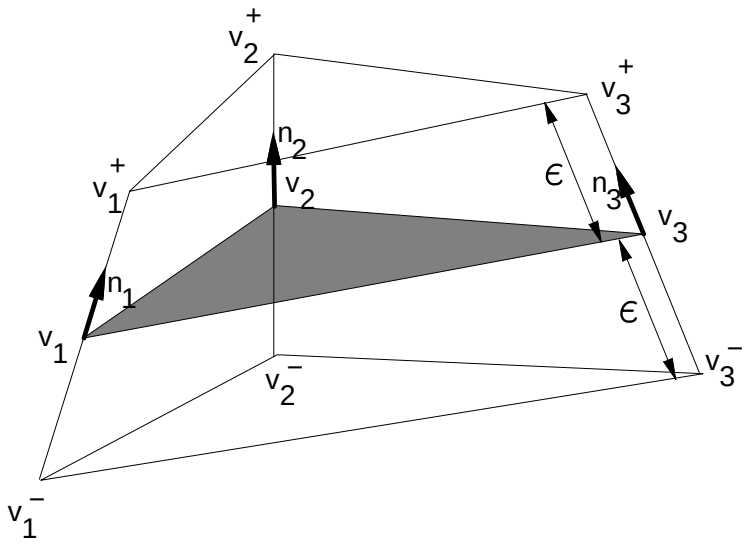


Figure 17: The Fundamental Prism

places. In other words, to guarantee that no intersections amongst the envelopes occur, we have to be content at certain places with the distance between *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.

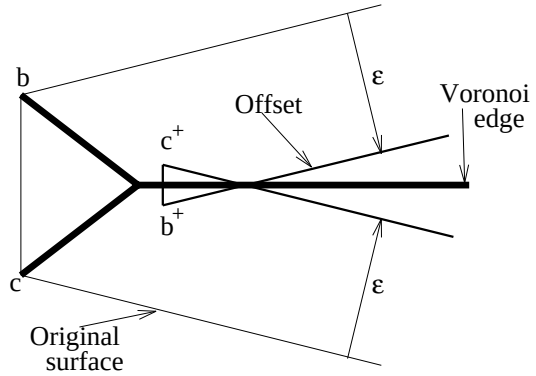


Figure 18: Offset Surface

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, $P(+\varepsilon)$ and $P(-\varepsilon)$, 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 18 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.

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.

Although 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.

3.3.1 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 discuss this approach for the case of computing the positive ε for each vertex. Computation of negative ε follows similarly.

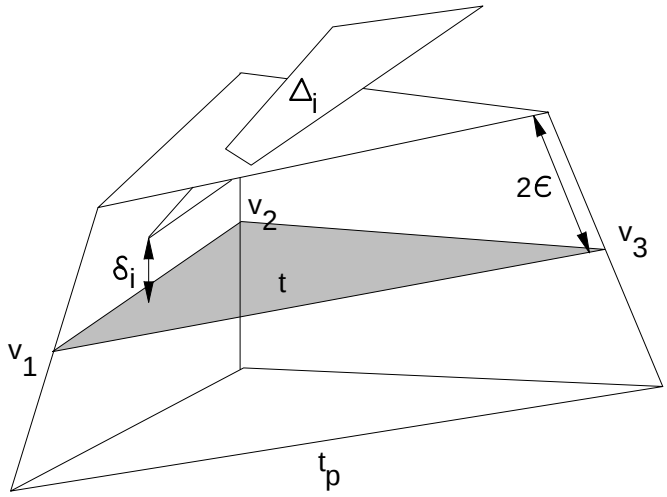


Figure 19: Computation of Δ_i

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 19.

Once we have done this for all Δ_i, we compute the new value of the positive ε for the triangle t as $\varepsilon_{new} = \frac{1}{2} \min_i \delta_i$. If the vertices for this triangle t have this value of positive ε, their positive envelope surface will not self-intersect. Once the $\varepsilon_{new}(t)$ values for all the triangles t have been computed, the $\varepsilon_{new}(v)$ for each vertex v is set to be the minimum of the $\varepsilon_{new}(t)$ 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.

3.3.2 Numerical ε Computation

As an alternative to the analytical approach, we may compute an envelope surface numerically, taking an iterative approach. Our envelope surface is initially identical to the input model surface. At 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 that 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, the algorithm 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) is a lower bound of sorts — each vertex will move at least K/(K+1)% of its potential. 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); this accounts for local geometry that we may intersect if we do not step carefully. 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, and fair to all the vertices (discouraging some vertices from making large displacements at the expense of other 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 20 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.

3.4 Generation of Approximation

Generating a surface approximation typically involves starting with the input surface and iteratively making modifications to ultimately reduce its complexity. We use a vertex remove operation (as described in Section 2.2.1), but other simplification operations are also possible in the simplification envelopes framework. The vertex remove operation 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.

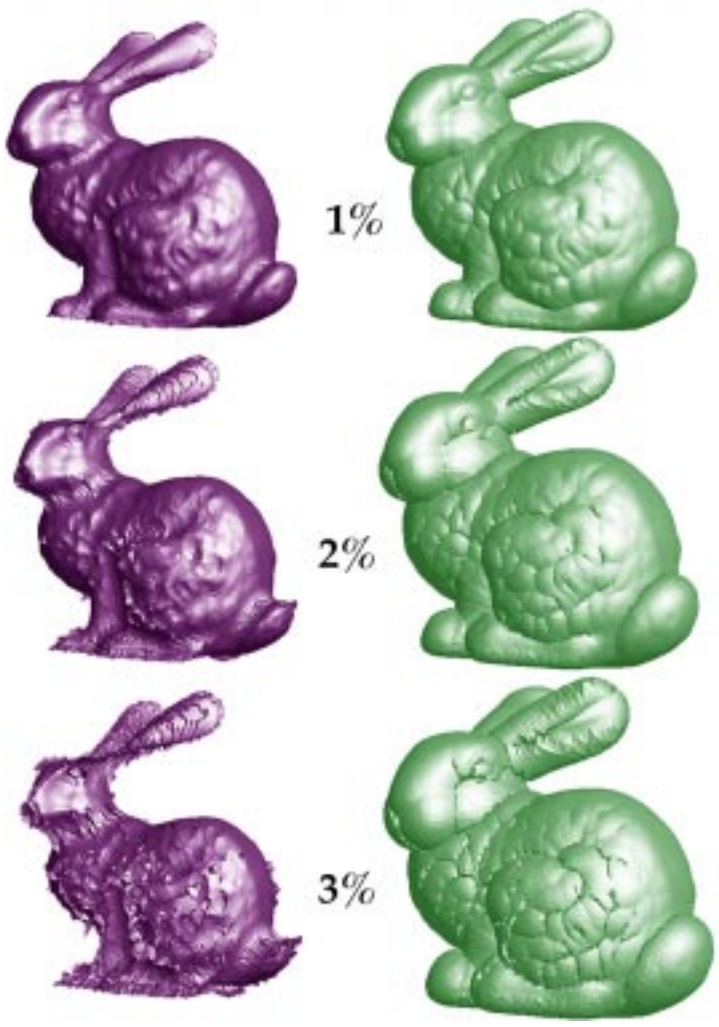


Figure 20: Simplification envelopes for various ε, measured as a percentage of bounding box diagonal

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, whereas 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.

3.4.1 Validity Test

A *candidate triangle* is one that 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 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.

3.4.2 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 [Schroeder et al. 1992], [Turk 1992], and [Hoppe et al. 1993]. 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.

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 they have been previously removed). If we cannot fill the hole, 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. The number of triangles is cubic with respect to the number of triangles on the boundary ($n(n\text{-}1)(n\text{-}2)$), but n is typically quite small. 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 21). 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.

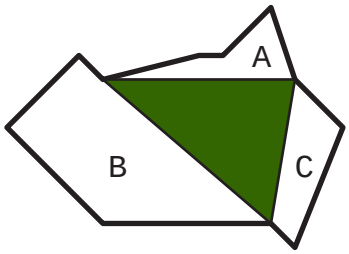


Figure 21: Adding a triangle into a hole creates up the three smaller holes.

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 or more (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 for large models. As the simplification reduces the model to very small number of triangles, we may wish to incorporate such an exhaustive search to get every reduction possible.

3.4.3 Global Algorithm

This algorithm extends the algorithm presented in [Clarkson 1993] 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 [Varshney 1994] 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 that do not intersect either of the envelopes. We may reduce this number somewhat by constructing these triangles from a set of candidate edges, which are all 2-tuples of the input vertices that do not intersect the enve-

lopes. 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 that *overlap* this candidate triangle. Now we begin the recursive hole-filling process by placing this candidate triangle into the hole and filling all the sub-holes 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.

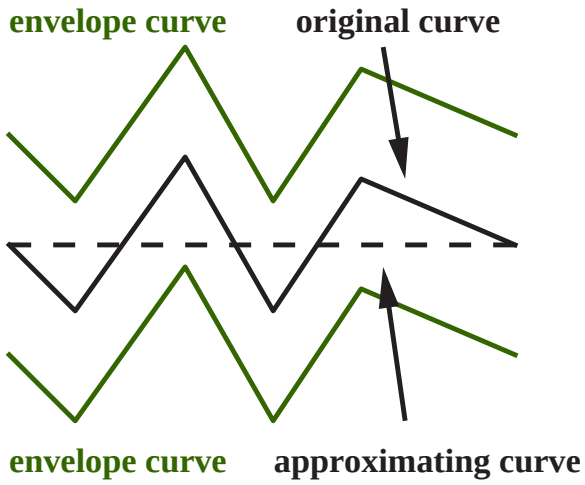


Figure 22: Curve at local minimum of approximation.

3.4.4 Algorithm Comparison

The local simplification algorithm is fast and robust enough to be applied to large models. The global strategy is theoretically elegant. Although the global algorithm works well for small models, its complexity rises at least quadratically, 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, whereas an ideal global algorithm would avoid all such minima.

Figure 22 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. Removing any one of the four interior vertices would cause a new edge to penetrate an envelope curve. But if we remove all four of the interior vertices simultaneously, 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.

3.5 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.

3.5.1 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.

Across certain edges, some models may contain artificial normal discontinuities that are not inherent to the local geometry. Both the global and local simplification algorithms may be modified to preserve such discontinuities by adding constraints to maintain these chains of edges throughout the simplification process. This type of constraint has been demonstrated effectively in [Hoppe 1996].

3.5.2 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 Section 3.4 are sufficient for closed triangle meshes, but they will not guarantee our global error bound for meshes with borders. Although 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.

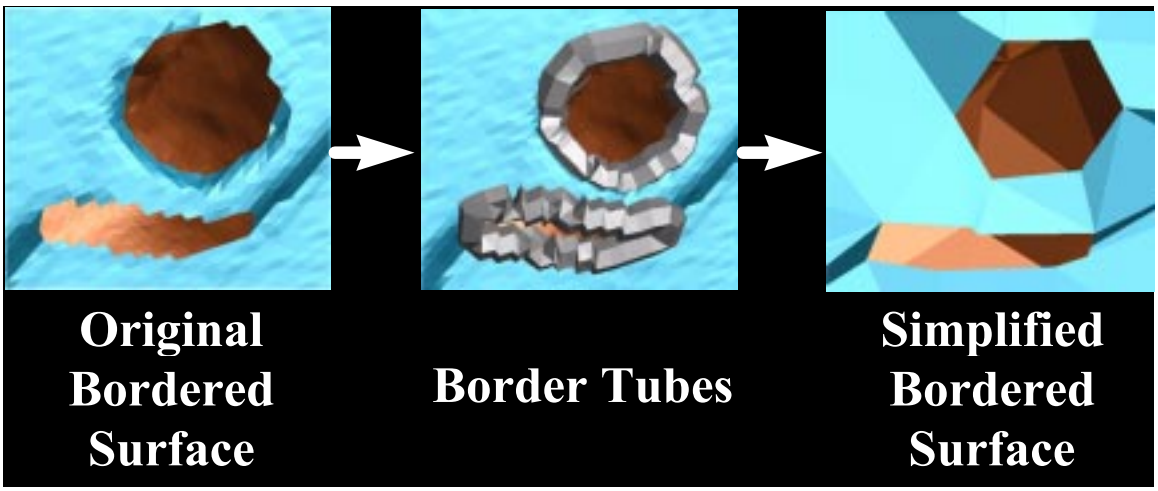


Figure 23: Simplifying a bordered surface using border tubes.

We construct a set of border tubes, as shown in Figure 23, 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 (in practice, we use 6 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 that encourages smooth, non-self-intersecting tubes. The most obvious approach might be to use the idea of *Frenet frames* [O'Neill 1966, Koenderink 1989] from differential geometry to define a set of coordinate systems for the polyline vertices. However, Frenet frames are defined for locally 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.

3.5.3 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 24 illustrates such an approximation.

Figure 24: An adaptive simplification of the bunny model that favors the face, while simplifying its hind quarters.

3.6 Computing Screen-space Deviation

Given a level of detail, created with a bound on maximum 3D surface deviation, ε, measured in absolute coordinates, and a set of viewing parameters, we now describe how to compute a bound on the maximum screen space deviation, *p*, measured in units of pixels. Figure 25 depicts a particular view of a level of detail within a rendering application. In this figure, θ is the total field of view, *d* is distance in the viewing direction from the eye point to the level of detail (or its bounding volume), and *r* is the resolution of the screen in pixels. *w* is simply the width of the viewing frustum at distance *d*. Notice that we generate a conservative bound by placing an error vector of the maximum size as close to the viewer as possible, and aligning this error vector perpendicular to the viewing vector. We see from the diagram, using the properties of similar triangles, that

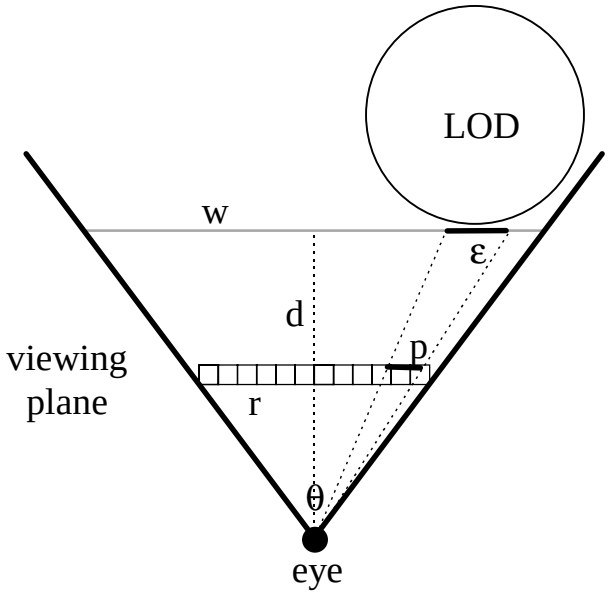


Figure 25: Viewing a level of detail.

$$\frac{\varepsilon}{w} = \frac{p}{r}, \tag{11}$$

which we then solve for p:

$$p = \frac{\varepsilon \ r}{w} = \frac{\varepsilon \ r}{2d \tan\theta} \tag{12}$$

Given this formulation, it is easy to compute this bound, p, on the screen-space deviation of a given level of detail at a given viewing distance. It is also possible to quickly choose which level of detail (from a hierarchy) to render to minimize the number of triangles while meeting a tolerance bound, $t \geq p$, on the maximum allowable screen-space error. Solving Equation (12) for ε yields:

$$\varepsilon \ = p\frac{2d \tan\theta}{r} \leq t\frac{2d \tan\theta}{r} \tag{13}$$

The application chooses to render the level of detail whose ε is as large as possible, while still small enough to satisfy the inequality; this is the level of detail with the smallest number of triangles for the error tolerance.

3.7 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.

3.7.1 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 [Barequet and Kumar 1997, Murali and Funkhouser 1997].

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.

3.7.2 Results

We have simplified a total of 2,636 objects from the auxiliary machine room (AMR) of a submarine data set, pictured in Figure 26, 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. These simplifications were performed on a Hewlett-Packard 735/125 with 80 MB of main memory. Images of these simplifications appear in Figure 27 and Figure 28. For the sake of comparison with timings in the following chapters and with other work, we have also timed a few of the simplifications on an SGI MIPS R10000 processor (see Table 2). It is interesting to compare the results on the bunny and phone models with those of [DeRose et al. 1993, Eck et al. 1995]. For the same error bound, we are able to obtain much improved solutions.

Bunny			Phone			Rotor			AMR		
ε %	# Tris	Time	ε %	# Tris	Time	ε %	# Tris	Time	ε %	# Tris	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	195,466	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 as a percentage of bounding box diagonal and run times in minutes on HP 735/125 MHz.

Model	Number of Triangles	ε %	Time (Mins:Secs)
Bunny	69,451	1/8	5:38
Phone	165,936	1/8	15:40
Rotor	4,735	1/8	2:04

Table 2: A few simplification timings run on a SGI MIPS R10000 processor.

We have automated the process that 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, Figure 27, and Figure 28 by using these normalized tolerances. 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 distances 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, as described in Section 3.6.

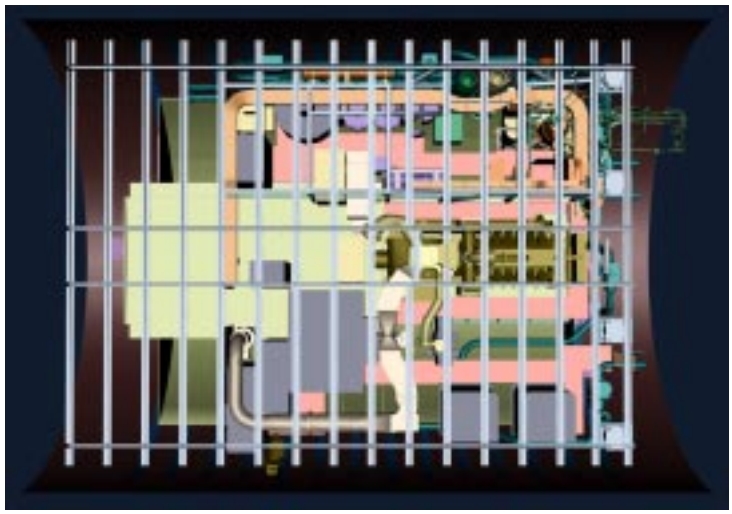


Figure 26: Looking down into the auxiliary machine room (AMR) of a submarine model. This model contains nearly 3,000 objects, for a total of over half a million triangles. We have simplified over 2,600 of these objects, for a total of over 430,000 triangles.

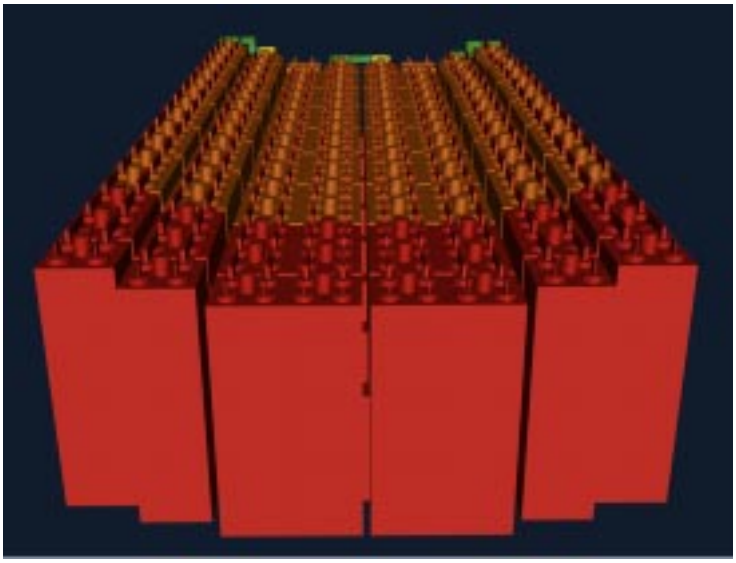
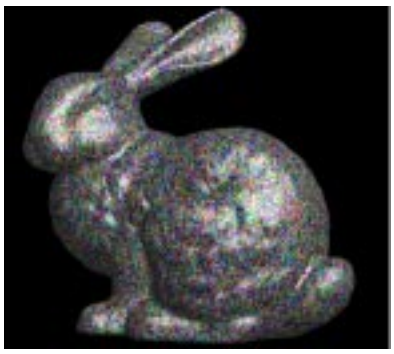
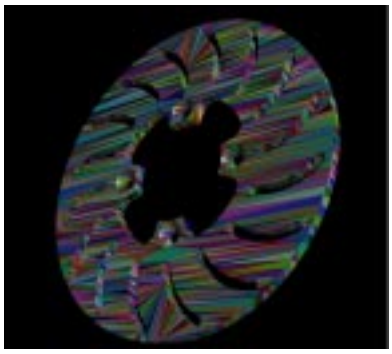


Figure 27: A battery from the AMR. All parts but the red are simplified representations. At full resolution, this array requires 87,000 triangles. At this distance, allowing 4 pixels of error in screen space, we have reduced it to 45,000 triangles.



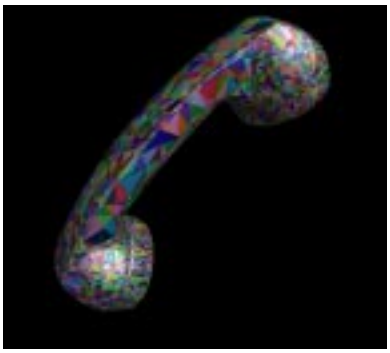
(a) bunny model: 69,451 triangles

(e) phone model: 165,936 triangle

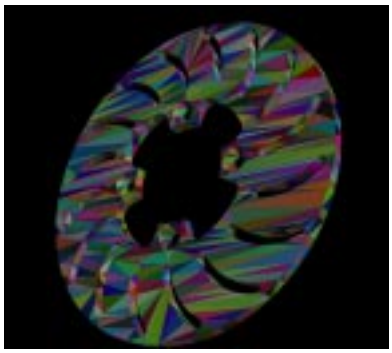


(i) rotor model: 4,736 triangles

(b) e = 1/16 %; 10,793 triangles



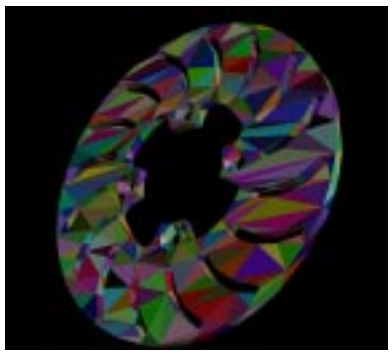
(f) e = 1/32 %; 12,364 triangles



(j) e = 1/8 %; 2,146 triangles

(c) e = 1/4 %; 2,204 triangles

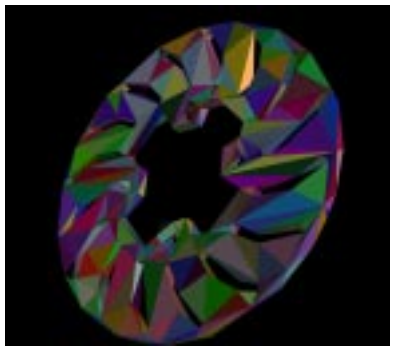
(g) e = 1/16 %; 4,891 triangles



(k) e = 3/4 %; 1,266 triangles

(d) e = 1 %; 575 triangles

(h) e = 1 %; 412 triangles



(l) e = 3 3/4 %; 716 triangles

Figure 28: Level-of-detail hierarchies for three models. The approximation distance, ε, is taken as a percentage of the bounding box diagonal.

3.7.2.1 Cascaded vs. Non-cascaded Simplification

For the rotor and AMR models in the above results, we sometimes performed *cascaded* simplifications: the i^{th} level of detail was obtained by simplifying the i-1^{th} level of detail. This causes the total ε to be the sum of all previous ε's, so choosing ε's of 1, 2, 4, and 8 percent results in total ε*'s* of 1, 3, 7, and 15 percent for the AMR. 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.

For any given model, it may be difficult to decide whether to perform these cascaded simplifications, where the starting point for each new simplification is the previously-computed level of detail. Table 3 demonstrates this dilemma for the rotor model. The non-cascaded column presents the number of triangles in the levels of detail resulting from a non-cascaded approach, where each simplification is performed on the original model. The later columns perform some number of non-cascaded simplifications, then begin cascading each new simplification from the previous one. Once the cascading begins, the incremental ε tolerances must be added to compute the total tolerance, as described above. The column labeled "Cascaded After 1/4 %" was used to report the simplifications of the rotor in Table 1.

ε %	Non-Cascaded	Cascaded After 1/8 %	Cascaded After 1/4 %	Cascaded After 1/2 %	Cascaded After 1 %	Cascaded After 2 %	Cascaded After 4 %
0	4,735						
1/8	2,146						
1/4	1,514	1,766					
1/2	1,266	1,252	1,266				
1	1,084	892	850	866			
2	1,026	760	716	772	874		
4	1,006	720	688	726	810	848	
8	1,006	706	674	722	794	808	846

Table 3: Experimenting with cascading on the rotor model and the resulting number of triangles.

One possible avenue of research we have not explored is allowing the paths of the vertices to change during the envelope construction, as the vertex normals change. Currently, each vertex is offset along the straight path of its original normal. Allowing the path to change as

the local envelope geometry changes may avoid some intersections and allow a greater volume within the envelopes. This might provide an interesting compromise between non-cascaded and cascaded simplifications.

3.7.2.2 Methods of Normal Vector Computation

The computation of normal vectors for vertices is a common task in computer graphics. Vertex normals are used for performing lighting calculations that smoothly shade adjacent triangles. These normals are typically computed using a average of the normals of the surrounding triangles. It may be an unweighted average, or it may be weighted by triangle area or angular span about the vertex. Regardless of the weighting heuristic, there is no guarantee that the computed vertex normal will be within 90^o of the normals of the adjacent faces. When this criterion is not met, shading artifacts may be apparent.

In the case of this use of normals for lighting calculations, several workarounds are possible, such as using multiple normals at a vertex and allowing a shading discontinuity. However, the use of normals for envelope construction does not allow such a workaround. Instead, we use a linear programming technique described in Section 4.3.1. This technique guarantees that we find a valid vertex normal if one exists. This improvement in our normal vector computation is useful, because the vertices with invalid normals cannot be offset during the envelope construction, limiting the simplification process.

Table 4 demonstrates the difference between using an averaging heuristic for vertex normal computation and using linear programming to find valid normals for as many vertices as possible. The bunny model has first been simplified from the original using a tolerance of $\varepsilon = 0.5$ %, and then simplifications are cascaded from that point on. The linear programming technique allows us to achieve a coarse model with half as many triangles as the averaging technique.

3.8 Conclusions

In this chapter, we have presented 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

ε	Average Normal Vector		Linear Programming Normal Vector	
	Invalid Normals	Triangles	Invalid Normals	Triangles
0.5		1,004		1,004
1	41	536	3	489
2	45	364	5	275
4	57	297	6	185
8	57	279	4	148
16	59	267	4	136
32	58	264	5	130

Table 4: Comparison of simplification using average normal vectors for offset computation vs. using linear programming to achieve fewer invalid normals. The bunny model is simplified using cascaded simplifications after ε=1/2 %.

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.

4. A LOCAL ERROR METRIC FOR SURFACE DEVIATION

The simplification envelopes error metric for surface deviation presented in Chapter 3 is global in the sense that we compute only a single error measure for each level of detail we create. The simplification envelopes algorithm may seen as a “pay up front” method, because we devote considerable effort to constructing a pair of envelope surfaces, then perform our simplification operations rather quickly, verifying only that the resulting surface is still contained between the envelopes.

In this chapter, we present a more local approach, called the *successive mapping* algorithm, employing more of a “pay as you go” paradigm. We will still incur some initialization overhead as we prioritize a set of edge collapse operations on our surface, but then we will perform the majority of the work as we simplify the surface. This work consists of measuring the local error in the neighborhood of each edge collapse.

The algorithm computes a piece-wise linear mapping between the original surface and the simplified surface. It uses the edge collapse operation due to its simplicity, local control, and suitability for generating smooth transitions between levels of detail. We also present rigorous and complete algorithms for collapsing an edge to a vertex such that there are no local self-intersections and a *bijective* (one-to-one and onto) mapping is guaranteed. The algorithm keeps track of both incremental surface deviation from the current level of detail and total deviation from the original surface.

The output of the algorithm a sequence of edge collapse operations, each with its own error bound describing the error in the neighborhood of the operation. This progressive mesh [Hoppe 1996], as described in Section 2.4.2, may be used as part of a dynamic simplification system, though our current system only renders from static levels detail (see Sections 2.4.1 and 4.6.2).

This research was performed in collaboration with Dinesh Manocha and Marc Olano. Much of this work appeared in the *Proceedings of IEEE Visualization '97* [Cohen et al. 1997]. The initial algorithm concept is based on work by Schikore and Bajaj [Bajaj and Schikore 1996]. The key differences between our work and theirs is discussed in Section 4.8.

The rest of the chapter is organized as follows. We give an overview of our algorithm in Section 4.1. In Section 4.2 we provide the details of the mathematical underpinnings of our projection-based mapping algorithms. Section 4.2 discusses the creation of local mappings for the purpose of collapsing edges. Using these mappings, we bound and minimize surface deviation error in Section 4.4. Section 4.5 describes how to compute new texture coordinates for the new mesh vertices. The implementation is discussed in Section 4.6 and its performance in Section 4.7. In Section 4.8 we compare our approach to some other algorithms, and we conclude the chapter with Section 4.9.

4.1 Overview

Our simplification approach may be seen as a high-level algorithm that controls the simplification process with a lower-level cost function based on local mappings. Next we describe this high-level control algorithm and the idea of using local mappings for cost evaluation.

4.1.1 High-level Algorithm

At a broad level, our simplification algorithm is a generic greedy algorithm. Our simplification operation is the edge collapse. We initialize the algorithm by measuring the cost of all possible edge collapses, then we perform the edge collapses in order of increasing cost. The cost function represents local error bounds on surface deviation and other attributes. After performing each edge collapse, we locally re-compute the cost functions of all edges whose neighborhoods were affected by the collapse. This process continues until none of the remaining edges can be collapsed.

The output of our algorithm is the original model plus an ordered list of edge collapses and their associated cost functions. This *progressive mesh* [Hoppe 1996] represents an entire

continuum of levels of detail for the surface. Section 4.6.2 discusses how we use these levels of detail to render the model with the desired quality or speed-up.

4.1.2 Local Mappings

The edge collapse operation we perform to simplify the surface contracts an edge (the *collapsed edge*, *e*) to a single, new vertex (the *generated vertex*, v_{gen}). Most of the earlier algorithms position the generated vertex to one of the end vertices or mid-point of the collapse edge. These choices for the generated vertex position are reasonable heuristics, and may reduce storage overhead. However, these choices may not minimize the surface deviation or other attribute error bound and can result in a local self-intersection. We choose a vertex position in two dimensions to avoid self-intersections and optimize in the third dimension to minimize error. This optimization of the generated vertex position and measurement of the error are the keys to simplifying the surface without introducing significant error.

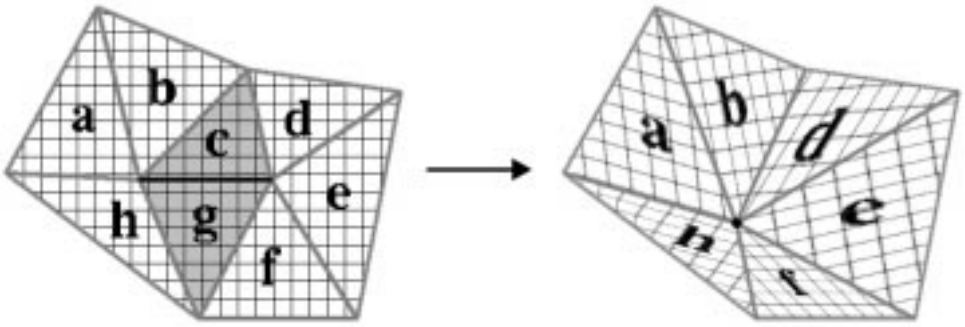


Figure 29: The natural mapping primarily maps triangles to triangles. The two grey triangles map to edges, and the collapsed edge maps to the generated vertex

For each edge collapse, we consider only the neighborhood of the surface that is modified by the operation (i.e. those faces, edges and vertices adjacent to the collapsed edge). There is a *natural mapping* between the neighborhood of the collapsed edge and the neighborhood of the generated vertex (see Figure 29). Most of the triangles incident to the collapsed edge are stretched into corresponding triangles incident to the generated vertex. However, the two triangles that share the collapsed edge are themselves collapsed to edges. These natural correspondences are one form of mapping.

This natural mapping has two weaknesses.

1. The degeneracy of the triangles mapping to edges prevents us from mapping points of the simplified surface back to unique points on the original surface. This also implies

that if we have any sort of attribute field across the surface, a portion of it disappears as a result of the operation.

2. The error implied by this mapping may be larger than necessary.

We measure the surface deviation error of the edge collapse operation as the distances between corresponding points of our mapping. Using the natural mapping, the maximum distance between any pair of corresponding points is defined as:

$$E = \max(\text{distance}(v_1, v_{gen}), \text{distance}(v_2, v_{gen}), \quad (14)$$

where v_1 and v_2 are the vertices of e.

If we place the generated vertex at the midpoint of the collapsed edge, this distance error will be half the length of the edge. If we place the vertex at any other location, the error will be even greater.

We can create mappings that are free of degeneracies and often imply less error than the natural mapping. For simplicity, and to guarantee no self-intersections, we perform our mappings using orthogonal projections of our local neighborhood to the plane. We refer to them as *successive mappings*.

4.2 Projection Theorems

The simplification algorithm we will present depends on our ability to efficiently compute orthogonal projections that provide bijective mappings between small portions of triangle meshes. With this in mind, we present the mathematical properties of the mapping used in designing the projection algorithm.

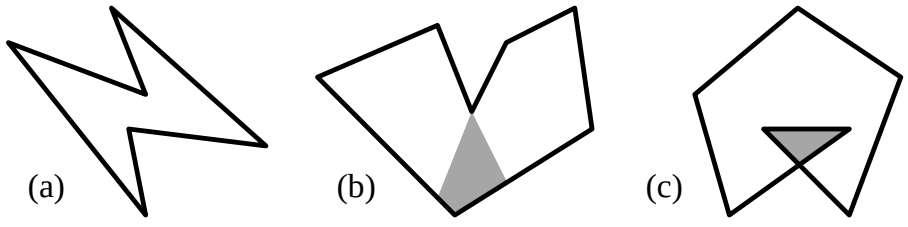


Figure 30: Polygons in the plane. (a) A simple polygon (with an empty kernel). (b) A star-shaped polygon with its kernel shaded. (c) A non-simple polygon with its kernel shaded.

Definition 1 *A* simple *polygon is a planar polygon in which edges only intersect at their two endpoints (vertices) and each vertex is adjacent to exactly two edges (see Figure 30(a)).*

Definition 2 *The* kernel *of a simple polygon is the intersection of the inward-facing half-spaces bounded by its edges (see Figure 30(b)). For a non-simple polygon (see Figure 30(c)), the kernel is the intersection of a consistently-oriented set of half-spaces bounded by its edges (i.e. if we traverse the edges in a topological order, the half-spaces must be either all to our right or all to our left).*

Definition 3 *A* star-shaped *polygon is a simple polygon with a non-empty kernel (see Figure 30(b)).*

By construction, any point in the kernel of a star-shaped polygon has an unobstructed line of sight to the polygon's entire boundary.

Definition 4 *A* complete vertex neighborhood, N_v, *is a set of triangles that forms a complete cycle around a vertex, v.*

The triangles of N_v are ordered: Δ_0, Δ_1, ..., Δ_{n-1}, Δ_0. Each pair of consecutive triangles in this ordering, (Δ_i, Δ_{i+1}), is adjacent, sharing a single edge, e_i; one of the vertices of e_i is v.

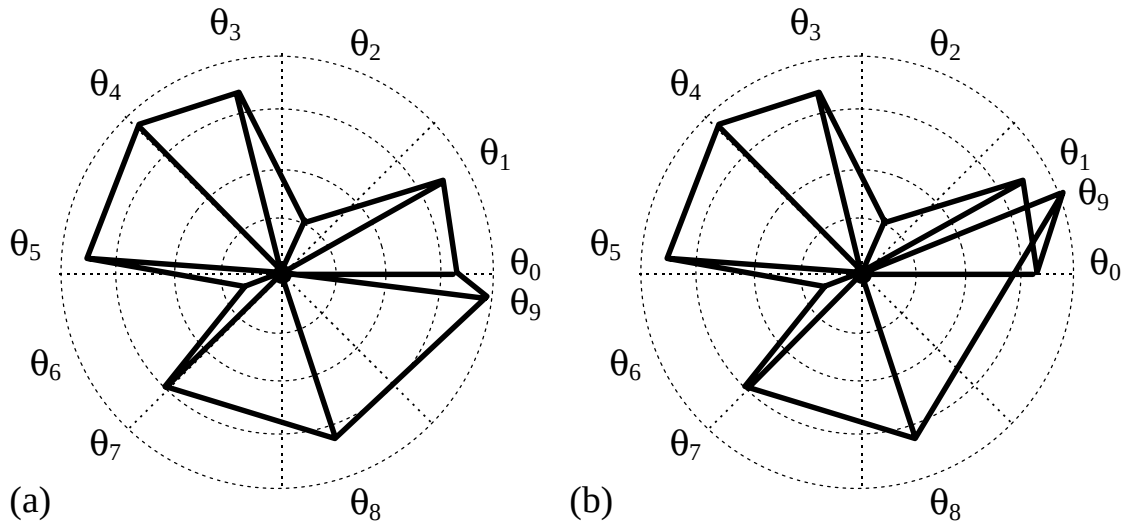


Figure 31: Projections of a vertex neighborhood, visualized in polar coordinates. (a) No angular intervals overlap, so the boundary is star-shaped, and the projection is a bijection. (b) Several angular intervals overlap, so the boundary is not star-shaped, and the projection is not a bijection.

Definition 5 *The* angle space *of an orthogonal projection of a complete vertex neighborhood, N_v, is the θ-coordinate space, $[0,2\pi]$, constructed by converting the projected neighborhood to polar coordinates, (r,θ), with v at the origin (see Figure 31(a))).*

Definition 6 *The* angular interval *covered by the orthogonal projection of triangle, Δ_i, from a complete vertex neighborhood, N_v, is the interval $[\theta_i,\theta_{(i+1) \bmod n}]$, where θ_i is the θ-coordinate of edge e_i.*

Definition 7 *The angle space of an orthogonal projection of a complete vertex neighborhood is* multiply-covered *if each angle, $\theta \in [0,2\pi]$, is covered by the projections of at least two triangles from N_v. It is k-*covered *if each angle is covered the projections of exactly k such triangles. A k-covered angle space is* exactly multiply-covered *if $k > 1$.*

Lemma 1 *The orthogonal projection of a complete vertex neighborhood, N_v, onto the plane, $\mathcal{P}$, provides a bijective mapping between N_v and a polygonal subset of $\mathcal{P}$ iff the angular intervals of the projected triangles of N_v do not overlap.*

Proof. Consider the projection of N_v in polar coordinates, with v at the origin, and e_0 at $\theta=0$ (see Figure 31). Each triangle, Δ_i, spans an angular interval in θ, bounded by e_i on one side and $e_{(i+1) \bmod n}$ on the other. If the intervals of the triangles do not overlap , then the triangles cannot overlap, and the projection must be a bijection. If the intervals do overlap, the triangles themselves must overlap (near the origin, which they both contain), and the projection cannot be a bijection (see Figure 31(b)). □

Corollary 1 *The orthogonal projection of a complete vertex neighborhood, N_v, onto the plane, $\mathcal{P}$, provides a bijective mapping between N_v and a polygonal subset of $\mathcal{P}$ iff the angle space of the projection of N_v is 1-covered.*

Proof. Lemma 1 shows that for a bijective mapping, the angle space cannot be multiply-covered. Because the triangles of N_v form a complete cycle around v, the angle space must be fully covered. Thus, each angle must be covered exactly once. □

Lemma 2 *The orthogonal projection of N_v onto the plane, $\mathcal{P}$, provides a bijective mapping between N_v and a polygonal subset of $\mathcal{P}$ iff the projection of N_v's boundary forms a star-shaped polygon in $\mathcal{P}$, with v in its kernel.*

Proof. If the projection provides a bijective mapping, the angular intervals of the triangles do not overlap, and the boundary forms a simple polygon, with the origin in the interior. The entire boundary of the polygon is visible from the origin. This is by definition a star-

shaped polygon, with the origin, *v*, in its kernel. In the case where one or more interval pairs overlap, portions of the boundary are typically occluded from the origin's point of view. Thus *v* cannot be in the kernel of a star-shaped polygon. Note that if the angle space is exactly multiply-covered, and the boundaries of these coverings are totally coincident, the entire boundary also seems to be visible from the origin. However, such a polygon is not technically simple, thus the projection of N_v is not technically star-shaped. □

Definition 8 *A* fold *in an orthogonal projection of a triangle mesh is an edge with two adjacent triangle whose projections lie to the same side of the projected edge. A* degenerate fold *is an edge with at least one triangle with a degenerate projection, lying entirely on the projected edge.*

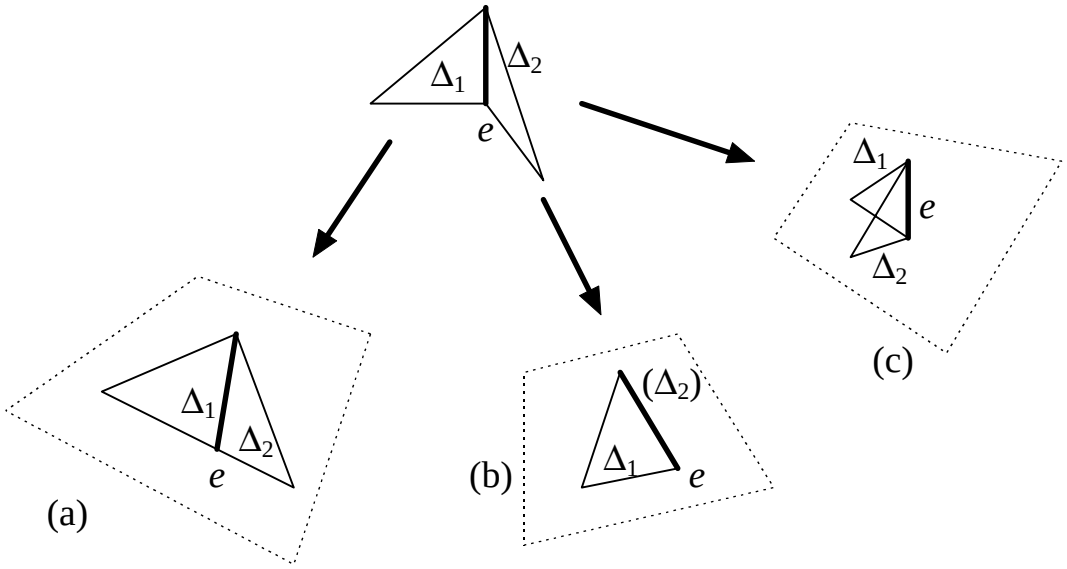


Figure 32: Three projections of a pair of edge-adjacent triangles. (a) The projected edge is not a fold, because the normals of both triangles are within 90° of the direction of projection. (b) The projected edge is a degenerate fold, because the normal of Δ_2 is perpendicular to the direction of projection. (c) The projected edge is a fold because the normal of Δ_2 is more than 90° from the direction of projection.

Lemma 3 *An orthogonal projection of a consistently-oriented triangle mesh is* fold-free *iff the triangle normals either all lie less than 90° or all lie greater than 90° from a vector in the direction of projection.*

Proof. We are given that the triangle mesh is orientable, with consistently oriented triangles and consistent normal vectors. The orientation of a projected triangle depends only on the relationship of its normal vector to the direction of projection (see Figure 32). When these two vectors are less than 90° apart, the projected triangle will have one orientation, whereas if

they are greater than 90^o apart, the projected triangle will have the opposite orientation. At exactly 90^o, the projected triangle degenerates into a line segment.

At a fold, the two triangles adjacent to the folded edge have opposite orientations in the plane, whereas at a non-folded edge, they have the same orientation. If all the triangle normals lie within the same hemisphere, either less than or greater than 90^o from the direction of projection, all the projected triangles will be consistently oriented, implying that none of the edges are folded.

If the normals do not all lie in one of these two hemispheres, the projected triangles may be divided into three groups according to their orientations in the plane (one group is for degenerate projections). Because the triangle mesh is fully connected, there must exist some edge that is adjacent to two triangles from different groups; this edge is a fold (or degenerate fold). □

Lemma 4 *The orthogonal projection of N_v onto $\mathcal{P}$ provides a bijective mapping iff the projection is fold-free and its angle space is not exactly multiply-covered.*

Proof. Again, consider the projection of N_v in polar coordinates. When a fold occurs, the angular intervals of these triangles overlap. Thus a projection with a fold does not provide a bijective mapping. On the other hand, if the projection is fold-free, every edge around v has its triangles laid out to either side. Because the final triangle of N_v connects to the initial triangle, this fold-free projection provides a k-covering of the angle space. If k=1, the projection provides a bijective mapping (from Corollary 1). If k>1, the projection is exactly multiply-covered, implying that angular intervals overlap, and the projection does not provide a bijective mapping. □

Lemma 5 *The orthogonal projection of N_v onto $\mathcal{P}$ provides a bijective mapping iff the projected triangles are consistently oriented and the angle-space of the projection is not exactly multiply-covered.*

Proof. We must show that the consistent orientation criterion is equivalent to the fold-free criterion of Lemma 4. The projection of each of the edges, $e_0 ... e_n$, is either a fold or not a fold. The two triangles adjacent to each non-folded edge are consistently oriented, whereas

those adjacent to each folded edge are inconsistently oriented (or degenerate). If none of the edges are folded, all adjacent pairs of triangles are consistently oriented, implying that all of N_v is consistently oriented. If any of the edges are folded, N_v is not consistently oriented. □

Theorem 1 *The following statements about the orthogonal projection of a complete vertex neighborhood, N_v, onto the plane, $\mathcal{P}$, are equivalent:*

- *The projection provides a bijective mapping between N_v and a polygonal subset of P.*
- *The angular intervals of the projected triangles of N_v do not overlap.*
- *The angle space of the projection of N_v is 1-covered.*
- *The projection of N_v's boundary forms a star-shaped polygon in P, with the vertex, v, in its kernel.*
- *The normals of the triangles of N_v all lie within the same hemisphere about the line of projection and the angle space of the projection is not exactly multiply-covered.*
- *The projection of N_v is fold-free and its angle space is not exactly multiply-covered.*
- *The projected triangles of N_v are consistently oriented in P and the angle space of the projection is not exactly multiply-covered.*

Proof. This equivalence list is a direct consequence of Lemmas 1, 2, 3, 4, and 5 and Corollary 1. □

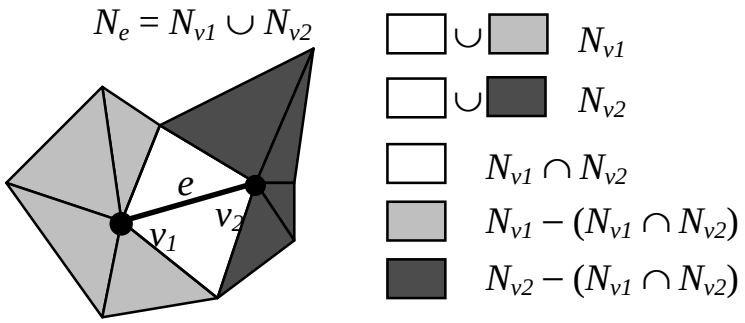


Figure 33: The edge neighborhood is the union of two vertex neighborhoods. If we remove the two triangles of their intersection, we get two independent polygons in the plane.

Definition 9 *A* complete edge neighborhood, *N_e, is a set of triangles that forms a complete cycle around an edge, e (see Figure 33).*

If v_1 and v_2 are the vertices of e, we can also write:

$$N_e = N_{v_1} \cup N_{v_2} \tag{15}$$

Lemma 6 *Given an edge, e, and its vertices, v_1 and v_2, the orthogonal projection of N_e onto the plane, $\mathcal{P}$, is fold-free iff the projections of N_{v_1} and N_{v_2} onto $\mathcal{P}$ are fold-free.*

Proof. The set of triangle edges in N_e is the union of the edges from N_{v_1} and N_{v_2}. If neither N_{v_1} nor N_{v_2} contains a folded edge, then N_e cannot contain a folded edge. Similarly, if either N_{v_1} or N_{v_2} contains a folded edge, N_e will contain that folded edge as well. Also note that the projections of N_{v_1} and N_{v_2} must have the same orientation, because they have two triangles and one interior edge (e) in common. □

Lemma 7 *The orthogonal projection of N_e onto $\mathcal{P}$ provides a bijective mapping between N_e and a polygonal subset of $\mathcal{P}$ iff the projections of its vertices, v_1 and v_2, provide bijective mappings between their neighborhoods and the plane, and the projection of the boundary of N_e is a simple polygon in $\mathcal{P}$.*

Proof. The projection provides a bijective mapping between N_{v_1} and a star-shaped subset of $\mathcal{P}$, and between N_{v_2} and a star-shaped subset of $\mathcal{P}$. The only way for N_e to not have a bijective mapping with a polygon in the plane is if the projections of N_{v_1} and N_{v_2} overlap, covering some points in the plane more than once.

Let N'_{v_1} and N'_{v_2} be the neighborhoods N_{v_1} and N_{v_2} with the two common triangles removed, as shown in Figure 33:

$$N'_{v_1} = N_{v_1} - (N_{v_1} \cap N_{v_2}); \quad N'_{v_2} = N_{v_2} - (N_{v_1} \cap N_{v_2}); \tag{16}$$

The projections of N'_{v_1} and N'_{v_2} are two polygons in $\mathcal{P}$. If the projections of N_{v_1} and N_{v_2} are each bijections, and these two polygons do not overlap, then the projection of N_e is a bijection. If the two polygons do overlap, the projection is not a bijection, because multiple points on N_e are projecting to the same point in $\mathcal{P}$. Given that the projections of N_{v_1} and N_{v_2}

are fold-free, the only way for the two polygons to overlap is for their boundaries to intersect. This intersection implies that the projection of N_e is a non-simple polygon.

So we have shown that if the projections of N_{v_1} and N_{v_2} provide bijective mappings with polygons in $\mathcal{P}$ and the projection of N_e's boundary is a simple polygon in $\mathcal{P}$, then the projection provides a bijective mapping between N_e and this simple polygon in $\mathcal{P}$. Also, if the projection covers a non-simple polygon, there can be no bijective mapping. □

Theorem 2 *The orthogonal projection of N_e onto $\mathcal{P}$ provides a bijective mapping between N_e and a polygonal subset of $\mathcal{P}$ iff the projection of N_e is fold-free, the projections of the neighborhoods of its vertices, v_1 and v_2, are not exactly multiply covered, and the projection of its boundary is a simple polygon in $\mathcal{P}$.*

Proof. Given Lemma 7, we only need to show that the projections of N_{v_1} and N_{v_2} provide bijective mappings iff the projection of N_e is fold-free, and the projections of N_{v_1} and N_{v_2} are not exactly multiply-covered. This is a direct consequence of Lemmas 4 and 6. □

Definition 10 *An* edge collapse *operation applied to edge e, with vertices v_1 and v_2, merges v_1 and v_2 into a single, generated vertex, v_{gen}. In the process, any triangles adjacent to e become degenerate and are deleted.*

Lemma 8 *Given an edge, e, which is collapsed to a vertex, v_{gen}, an orthogonal projection of N_e is a simple polygon iff the same orthogonal projection of $N_{v_{gen}}$ is a simple polygon.*

Proof. The collapse of *e* to v_{gen} does not move affect the vertices on the boundary of N_e, so N_e and $N_{v_{gen}}$ have the same boundary. Thus the projection of the boundary of N_e is simple iff the projection of the boundary of $N_{v_{gen}}$ is simple. □

Lemma 9 *A planar polygon with a non-empty kernel is simple iff it is star-shaped.*

Proof. A star-shaped polygon is defined as a simple polygon with a non-empty kernel. Thus if a polygon with a non-empty kernel is simple, it is star-shaped by definition. If a polygon with a non-empty kernel is not simple, it cannot be star-shaped. □

Lemma 10 *Given an edge, e, which is collapsed to a vertex, v_{gen}, inside the kernel of e, an orthogonal projection of N_e is simple iff the same projection of $N_{v_{gen}}$ is star-shaped.*

Proof. Recall from Lemma 8 that N_e and $N_{v_{gen}}$ have the same projected boundary. We have been given that this projected boundary is a planar polygon with a non-empty kernel. From Lemma 9, we know that this polygon is simple iff it is star-shaped. Thus the projection of the boundary of N_e is a simple polygon iff the projection of the boundary of $N_{v_{gen}}$ is a star-shaped polygon. □

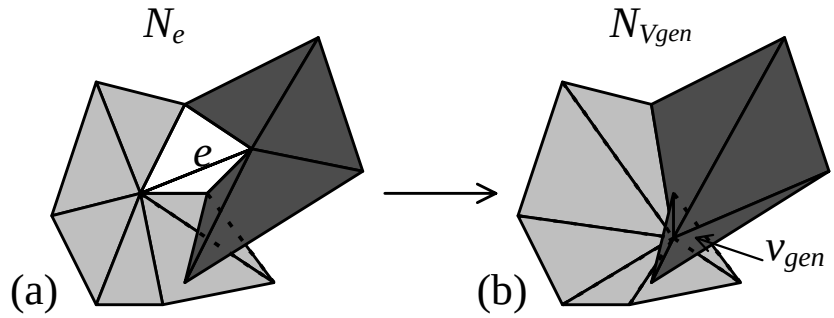


Figure 34: A fold-free projection of an edge neighborhood, N_e, that is not a bijection. (a) The projection of N_e has a non-empty kernel. (b) The projection of $N_{v_{gen}}$ has a 2-covered angle space. This can be detected by noting that the sum of the angular intervals of the triangles of $N_{v_{gen}}$ sum to 4π.

Theorem 3 *Given an edge, e, which is collapsed to a vertex, v_{gen} in the kernel of e, an orthogonal projection of N_e onto $\mathcal{P}$ provides a bijective mapping between N_e and a polygonal subset of $\mathcal{P}$ iff the projection of N_e is fold-free and the projected triangles of $N_{v_{gen}}$ are consistently oriented and do not multiply-cover the angle space.*

Proof. Theorem 2 shows that the projection of N_e is a bijection iff it is fold-free and simple. Lemma 10 shows that it is simple iff the projection of $N_{v_{gen}}$ is star-shaped. Theorem 1 shows that the projection of $N_{v_{gen}}$ is star-shaped iff its projected triangles are consistently oriented and do not multiply-cover the angle space. Figure 34 depicts an example of a fold-free edge projection that is not a bijection and collapses to a multiply-covered vertex neighborhood. □

Edge Collapse in the Plane

Theorems 1 and 3 lead us to an efficient algorithm for performing an edge collapse operation in the plane.

First, we find a fold-free projection for the edge, *e*. We can use linear programming with the normals of N_e as constraints to find a direction that guarantees such a fold-free projection, if one exists for this edge. We do not yet know if the projection is a bijection, but rather than check to see if the projection forms a simple polygon, we defer this test until later.

Second, we find a point inside the kernel of the projection of N_e. Again, we can use linear programming to find such a point, if one exists for this projection.

Third, we collapse *e*'s vertices, v_1 and v_2 to this point, v_{gen}, in the kernel. We do not know yet if the overall polygon is star-shaped, because even a non-simple polygon may have a non-empty kernel. If the polygon is not simple, neither the projection of N_e nor the projection of $N_{v_{gen}}$ provides a bijective mapping with a polygon in $\mathcal{P}$.

Finally, we verify that projections of N_{v_1}, N_{v_2}, and $N_{v_{gen}}$ are all bijections. For N_{v_1} and N_{v_2}, we verify that they are not exactly multiply-covered by adding up the spans of the angular intervals of their triangles. These spans should sum to 2π (within some floating point tolerance). For $N_{v_{gen}}$, we check not only the sum of the angular spans, but also the orientations of the projected triangles. If the spans sum to 2π and the orientations are consistent, $N_{v_{gen}}$ has a bijective mapping, and its boundary is star-shaped. These are all simple, $O(n)$ tests, with small constant factors. They guarantee that we have a bijective mapping between N_e and the plane, and also between $N_{v_{gen}}$ and the plane; this also provides a bijective mapping between N_e and $N_{v_{gen}}$.

All the steps of the preceding algorithm run in $O(n)$ time (though we will later need to find $O(n^2)$ edge-edge intersections, which we will use in the error calculation and 3D vertex placement). This algorithm for performing an edge-collapse in the plane is described in more detail in Section 4.3.

4.3 Successive Mapping

In this section we present an algorithm to compute the mappings we use to compute error bounds and to guide the simplification process. We present efficient and complete algorithms for computing a planar projection, finding a generated vertex in the plane, and creating a mapping in the plane. These algorithms employ well-known techniques from computational geometry and are efficient in practice. The basis for these algorithms are proven in Section 4.2.

4.3.1 Computing a Planar Projection

Given a set of triangles in 3D, we present an efficient algorithm to compute a planar projection that is *fold-free*. Such a fold-free projection contains no pair of edge-adjacent triangles that overlap in the plane. This fold-free characteristic is a necessary, but not sufficient, condition for a projection to provide a *bijective mapping* between the set of triangles and a portion of the plane. In practice, most fold-free projections provide such a bijective mapping. We later perform an additional test to verify that our fold-free projection is indeed a bijection (see Section 4.3.3).

The projection we seek should be a bijection to guarantee that the operations we perform in the plane are meaningful. For example, suppose we project a connected set of triangles onto a plane and then re-triangulate the polygon described by their boundary. The resulting set of triangles will contain no self-intersections, as long as the projection is a bijection. Many other simplification algorithms, such as those by Turk [Turk 1992], also use such projections for vertex removal. However, they simply choose a likely direction, such as the average of the normal vectors of the triangles of interest. To test the validity of the resulting projection, these earlier algorithms project all the triangles onto the plane and check for self-intersections. This process can be relatively expensive and does not provide a robust method for finding a bijective projecting plane.

We improve on earlier brute-force approaches in two ways. First, we present a simple, linear-time algorithm for testing the validity of a given direction, ensuring that it produces a fold-free projection. Second, we present a slightly more complex, but still expected linear-time, algorithm that will find a valid direction if one exists, or report that no such direction

exists for the given set of triangles. We defer until Section 4.3.3 a final, linear-time test to guarantee that our fold-free projection provides a bijective mapping.

4.3.1.1 Validity Test for Planar Projection

In this section, we briefly describe the algorithm that checks whether a given set of triangles has a fold-free planar projection. Assume that we can calculate a consistent set of normal vectors for the set of triangles in question (if we cannot, the local surface is non-orientable and cannot be mapped onto a plane in a bijective fashion). If the angle between a given direction of projection and the normal vector of each of the triangles is less than 90^o, then the direction of projection is valid, and defines a fold-free mapping from the 3D triangles to a set of triangles in the plane of projection (any plane perpendicular to the direction of projection). Note that for a given direction of projection and a given set of triangles, this test involves only a single dot product and a sign test for each triangle in the set. The correctness of this test is demonstrated in Section 4.2.

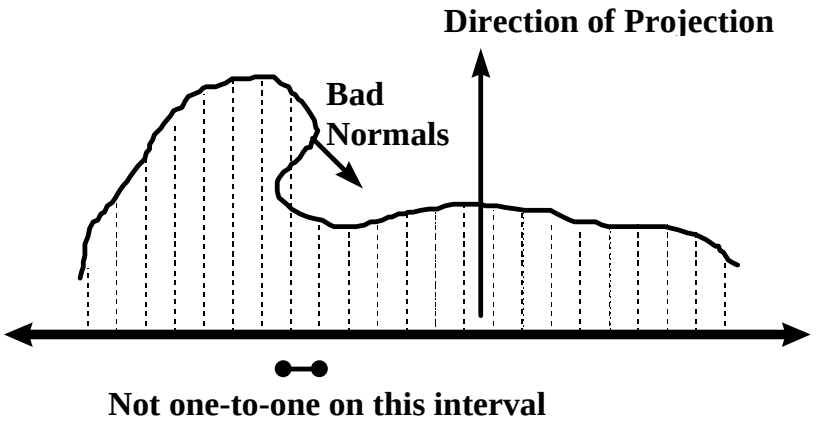


Figure 35: A 2D example of an invalid projection due to folding.

To develop some intuition, we examine a 2D version of our problem, shown in Figure 35. We would like to determine if the projection of the curve onto the line is fold-free. Without loss of generality, assume the direction of projection is the y-axis. Each point on the curve projects to its x-coordinate on the line. If we traverse the curve from its left-most endpoint, we will project onto a previously projected location if and only if we reverse our direction along the x-axis. This can only occur when the y-component of the curve's normal vector goes from a positive value to a negative value. This is equivalent to our statement that the invalid normal will be more than 90^o from the direction of projection.

4.3.1.2 Finding a Valid Direction

The validity test in the previous section provides a quick method of testing the validity of a likely direction as a fold-free projection. Unfortunately, the wider the spread of the normal vectors of our set of triangles, the less likely we are to find a valid direction by using any sort of heuristic. It is possible, in fact, to compute the set of all valid directions of projection for a given set of triangles. However, to achieve greater efficiency and to reduce the complexity of the software system, we choose to find only a single valid direction, which is typically all we require.

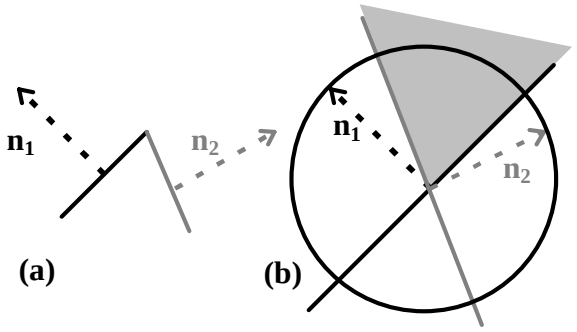


Figure 36: A 2D example of the valid projection space. (a) Two line segments and their normals. (b) The 2D Gaussian circle, the planes corresponding to each segment, and the space of valid projection directions (shaded in grey).

The *Gaussian sphere* [Carmo 1976] is the unit sphere on which each point corresponds to a unit normal vector with the same coordinates. Given a triangle, we define a plane through the origin with the same normal as the triangle. For a direction of projection to be valid with respect to this triangle, its point on the Gaussian sphere must lie on the correct side of this plane (i.e. within the correct hemisphere). If we consider two triangles simultaneously (shown in 2D in Figure 36) the direction of projection must lie on the correct side of each of the two planes determined by the normal vectors of the triangles. This is equivalent to saying that the valid directions lie within the intersection of half-spaces defined by these two planes. Thus, the valid directions of projection for a set of N triangles lie within the intersection of N half-spaces.

This intersection of half-spaces forms a convex polyhedron. This polyhedron is a cone, with its apex at the origin and an unbounded base (shown as a shaded, triangular region in Figure 36). We can force this polyhedron to be bounded by adding more half-spaces (we use the six faces of a cube containing the origin). By finding a point on the interior of this cone

and normalizing its coordinates, we shall construct a unit vector in a valid direction of projection.

Rather than explicitly calculating the boundary of the cone, we simply find a few corners (vertices) and average them to find a point that is strictly inside. By construction, the origin is definitely such a corner, so we just need to find three more *unique* corners to calculate an interior point. We can find each of these corners by solving a 3D *linear programming* problem. Linear programming allows us to find a point that maximizes a linear objective function subject to a collection of linear constraints [Kolman and Beck 1980]. The equations of the half-spaces serve as our linear constraints. We maximize in the direction of a vector to find the corner of our cone that lies the farthest in that direction.

As stated above, the origin is our first corner. To find the second corner, we try maximizing in the positive-x direction. If the resulting point is the origin, we instead maximize in the negative-x direction. To find the third corner, we maximize in a direction orthogonal to the line containing the first two corners. If the resulting point is one of the first two corners, we maximize in the opposite direction. Finally, we maximize in a direction orthogonal to the plane containing the first three corners. Once again, we may need to maximize in the opposite direction instead. Note that it is possible to reduce the worst-case number of optimizations from six to four by using the triangle normals to guide the selection of optimization vectors.

We used Seidel's linear time randomized algorithm [Seidel 1990] to solve each linear programming problem. A public domain implementation of this algorithm by Hohmeyer is available. It is very fast in practice.

4.3.2 Placing the Vertex in the Plane

In the previous section, we presented an algorithm to compute a valid plane. The edge collapse, which we use as our simplification operation, merges the two vertices of a particular edge into a single vertex. The topology of the resulting mesh is completely determined, but we are free to choose the position of the vertex, which will determine the geometry of the resulting mesh.

When we project the triangles neighboring the given edge onto a valid plane of projection, we get a triangulated polygon with two interior vertices, as shown in Figure 37. The

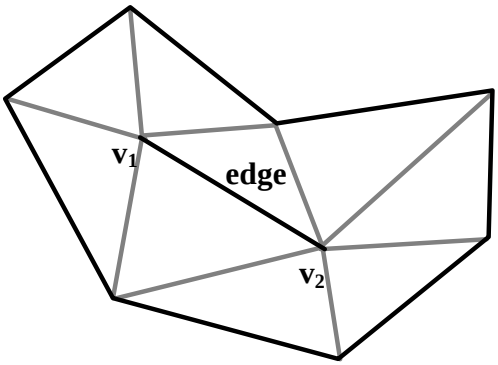


Figure 37: The neighborhood of an edge as projected into 2D

edge collapse will reduce this edge to a single vertex. There will be edges connecting this generated vertex to each of the vertices of the polygon. We would like the set of triangles around the generated vertex to have a bijective mapping with our chosen plane of projection, and thus to have a one-to- one mapping with the original edge neighborhood as well.

In this section, we present linear time algorithms both to test a candidate vertex position for validity, and to find a valid vertex position, if one exists.

4.3.2.1 Validity Test for Vertex Position

The edge collapse operation leaves the boundary of the polygon in the plane unchanged. For the neighborhood of the generated vertex to have a bijective mapping with the plane, its edges must lie entirely within the polygon, ensuring that no edge crossings occur.

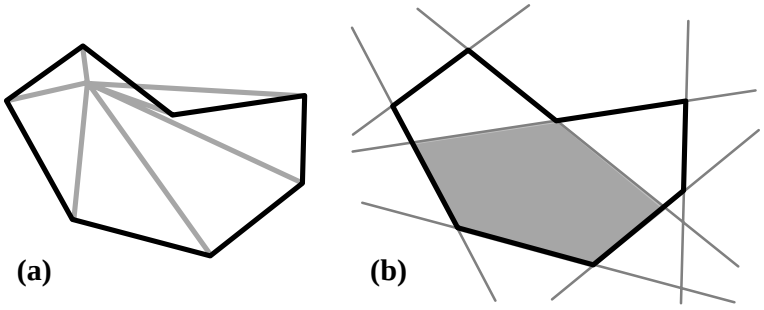


Figure 38: (a) An invalid 2D vertex position. (b) The kernel of a polygon is the set of valid positions for a single, interior vertex to be placed. It is the intersection of a set of inward half-spaces.

This 2D visibility problem has been well-studied in the computational geometry literature [O'Rourke 1994]. The generated vertex must have an unobstructed line of sight to each of the surrounding polygon vertices (unlike the vertex shown in Figure 38(a)). This condition holds if and only if the generated vertex lies within the polygon's *kernel*, shown in Figure 38(b). This kernel is the intersection of inward-facing half-planes defined by the polygon's edges.

Given a candidate position for the generated vertex in 2D, we test its validity by plugging it into the implicit-form equation of each of the lines containing the polygon's edges. If the position is on the interior with respect to each line, the position is valid; otherwise it is invalid.

4.3.2.2 Finding a Valid Position

The validity test described above is useful if we wish to test out a likely candidate for the generated vertex position, such as the midpoint of the edge being collapsed. If such a heuristic choice succeeds, we can avoid the work necessary to compute a valid position directly.

Given the kernel definition for valid points, it is straightforward to find a valid vertex position using 2D linear programming. Each of the lines provides one of the constraints for the linear programming problem. Using the same methods as in Section 4.3.1.2, we can find a point in the kernel with no more than four calls to the linear programming routine. The first and second corners are found by maximizing in the positive- and negative-x directions. The final corner is found using a vector orthogonal to the first two corners.

4.3.3 Guaranteeing a Bijective Projection

Although rare in practice, it is possible in theory for us to find both a fold-free projection and a vertex position within the planar polygon's kernel, yet still have a projection that is not a bijection. Figure 34 shows an example of such a projection.

As proved in Section 4.2, we can verify that both the neighborhoods of the generated vertex and the collapsed edge have bijective projections with the plane with a simple, linear-time test. Given our edge, e, its two vertices, v_1 and v_2, and the generated vertex, v_{gen}, these projections are bijections if and only if the orientations of the triangles surrounding v_{gen} are consistent and the triangles surrounding v_1, v_2, and v_{gen} each cover angular ranges in the plane that sum to 2π.

We can verify the orientations of v_{gen}'s triangles by performing a single cross product for each triangle. If the signed areas of all the triangles have the same sign, they are consistently oriented, and the projections are bijections. We verify the angular sums of triangles surrounding v_1, v_2, and v_{gen} using a vector normalization, dot product, and arccosine operation

for each triangle to compute its angular range. Each floating point sum will be within some small tolerance of an integer multiple of 2π, with 1 being the valid multiplier.

4.3.4 Creating a Mapping in the Plane

After mapping the edge neighborhood to a valid plane and choosing a valid position for the generated vertex, we define a mapping between the edge neighborhood and the generated vertex neighborhood. We shall map to each other the pairs of 3D points that project to identical points on the plane. These correspondences are shown in Figure 39(a) by superimposing these two sets of triangles in the plane.

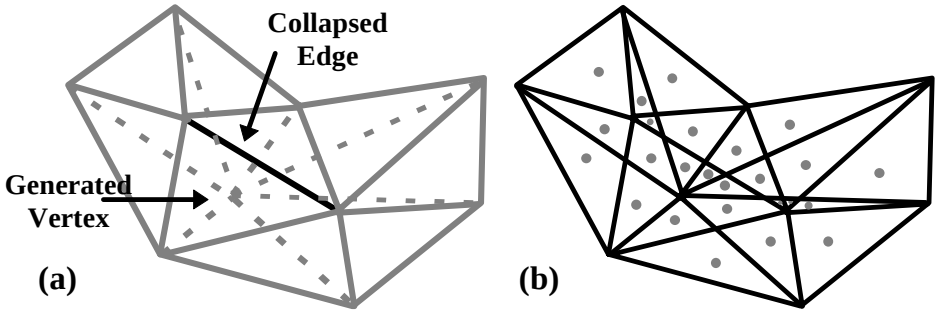


Figure 39: (a) Edge neighborhood and generated vertex neighborhood superimposed. (b) A mapping in the plane, composed of 25 polygonal cells (each cell contains a dot). Each cell maps between a pair of planar elements in 3D.

We can represent the mapping by a set of map cells, shown in Figure 39(b). Each cell is a convex polygon in the plane and maps a piece of a triangle from the edge neighborhood to a similar piece of a triangle from the generated vertex neighborhood. The mapping represented by each cell is linear.

The vertices of the polygonal cells fall into four categories: vertices of the overall polygon in the plane, vertices of the collapsed edge, the generated vertex itself, and edge-edge intersection points. We already know the locations of the first three categories of cell vertices, but we must calculate the edge-edge intersection points explicitly. Each such point is the intersection of an edge adjacent to the collapsed edge with an edge adjacent to the generated vertex. The number of such points can be quadratic (in the worst case) in the number of neighborhood edges. If we choose to construct the actual cells, we may do so by sorting the intersection points along each neighborhood edge and then walking the boundary of each cell. However, this is not necessary for computing the surface deviation.

4.4 Measuring Surface Deviation Error

Up to this point, we have projected the collapsed edge neighborhood onto a plane, collapsed the edge to the generated vertex in this plane, and computed a mapping in the plane between these two local meshes. The generated vertex has not yet been placed in 3D. We will choose its 3D position to minimize the error in surface deviation.

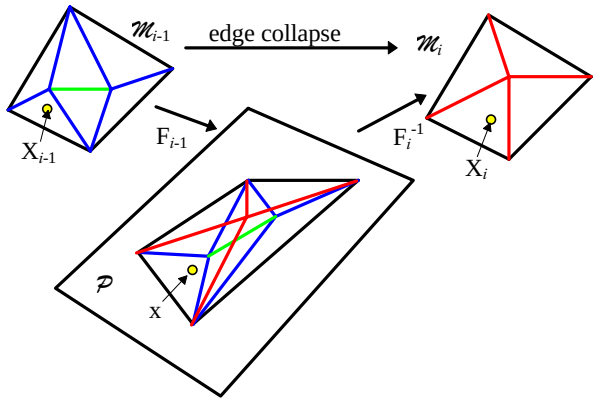


Figure 40: Each point, *x*, in the plane of projection corresponds to two 3D points, X_{i-1} and X_i on meshes $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$, respectively.

Given the overlay in the plane of the collapsed edge neighborhood, $\mathcal{M}_{i-1}$, and the generated vertex neighborhood, $\mathcal{M}_i$, we define the *incremental surface deviation* between them:

$$E_{i,i-1}(x) = \left\| F_i^{-1}(x) - F_{i-1}^{-1}(x) \right\| \tag{17}$$

The function, $F_i(X):\mathcal{M}_i \rightarrow \mathcal{P}$, maps points on the 3D mesh, $\mathcal{M}_i$, to points, x, in the plane. $F_{i-1}(X):\mathcal{M}_{i-1} \rightarrow \mathcal{P}$ acts similarly for the points on $\mathcal{M}_{i-1}$. $E_{i,i-1}$ measures the distance between the pair of 3D points corresponding to each point, x, in the planar overlay (shown in Figure 40).

Within each of our polygonal mapping cells in the plane, the incremental deviation function is linear, so the maximum incremental deviation for each cell occurs at one of its boundary points. Thus, we bound the incremental deviation using only the deviation at the cell vertices, V:

$$E_{i,i-1}(\mathcal{P}) = \max_{x \in \mathcal{P}} E_{i,i-1}(x) = \max_{v_k \in V} E_{i,i-1}(v_k) \tag{18}$$

4.4.1 Distance Functions of the Cell Vertices

To preserve our bijective mapping, it is necessary that all the points of the generated vertex neighborhood, including the generated vertex itself, project back into 3D along the direction of projection (the normal to the plane of projection). This restricts the 3D position of the generated vertex to the line parallel to the direction of projection and passing through the generated vertex's 2D position in the plane. We choose the vertex's position along this line such that it minimizes the incremental surface deviation.

We parameterize the position of the generated vertex along its line of projection by a single parameter, *t*. As *t* varies, the distance between the corresponding cell vertices in 3D varies linearly. Notice that these distances will always be along the direction of projection, because the distance between corresponding cell vertices is zero in the other two dimensions (those of the plane of projection). The distance function for each cell vertex, v_k, has the form (see Figure 41):

$$E_{i,i-1}(v_k) = \left| m_k t + b_k \right|, \tag{19}$$

where m_k and b_k are the slope and y-intercept of the signed distance function for v_k as *t* varies.

4.4.2 Minimizing the Incremental Surface Deviation

Given the distance function, we would like to choose the parameter *t* that minimizes the maximum distance between any pair of mapped points. This point is the minimum of the so-called *upper envelope*, shown in Figure 41. For a set of *k* functions, we define the upper envelope function as follows:

$$U(t) = \left\{ f_i(t) \mid f_i(t) > f_j(t) \quad \forall i, j \quad 1 \le i, j \le k; \quad i \ne j \right\} \tag{20}$$

For linear functions with no boundary conditions, this function is convex. We convert the distance function for each cell vertex to two linear functions, then use linear programming to find the *t* value at which the minimum occurs. We use this value of *t* to calculate the 3D position for the generated vertex that minimizes the maximum incremental surface deviation.

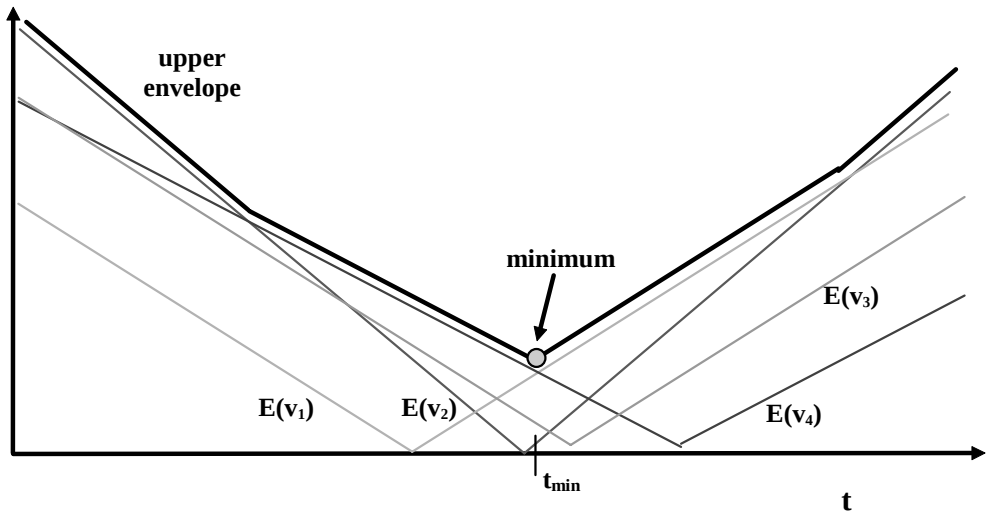


Figure 41: The minimum of the upper envelope corresponds to the vertex position that minimizes the incremental surface deviation.

4.4.3 Bounding Total Surface Deviation

Although it is straightforward to measure the incremental surface deviation and choose the position of the generated vertex to minimize it, this is not the error we eventually store with the edge collapse. To know how much error the simplification process has created, we need to measure the *total surface deviation* of the mesh $\mathcal{M}_i$:

$$S_i(X) = E_{i,0}(F_i(X)) = \left\| X - F_0^{-1}(F_i(X)) \right\| \tag{21}$$

Unfortunately, our projection formulation of the mapping functions provides only $F_{i\text{-}1}^{-1}$ and F_i^{-1} when we are performing edge collapse i; it is more difficult to construct F_0^{-1}, and the complexity of this mapping to the original surface will retain the complexity of the original surface.

We approximate $E_{i,0}$ by using a set of axis-aligned boxes (other possible choices for these approximation volumes include triangle-aligned prisms and spheres). This provides a convenient representation of a bound on $S_i(X)$ that we can update from one simplified mesh to the next without having to refer to the original mesh. Each triangle, Δ_k, in $\mathcal{M}_i$ has its own axis-aligned box, $b_{i,k}$ such that at every point on the triangle, the Minkowski sum of the 3D point with the box gives a region that contains the corresponding point on the original surface.

$$\forall X \in \Delta_k,\ F_0^{-1}(F_i(X)) \in X \oplus b_{i,k} \tag{22}$$

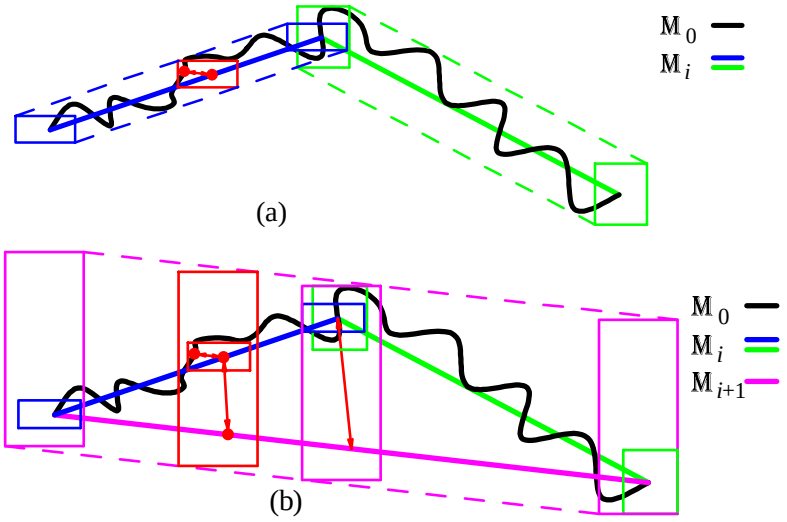


Figure 42: 2D illustration of the box approximation to total surface deviation. (a) A curve has been simplified to two segments, each with an associated box to bound the deviation. (b) As we simplify one more step, the approximation is propagated to the newly created segment.

Figure 42(a) shows an original surface (curve) and a simplification of it, consisting of two thick lines. Each line has an associated box. As the box slides over the line it is applied to each point along the way; the corresponding point on the original mesh is contained within the translated box. One such correspondence is shown halfway along the left line.

From (21) and (22), we produce $\tilde{S}_i(X)$, a bound on the total surface deviation, $S_i(X)$. This is the surface deviation error reported with each edge collapse.

$$\tilde{S}_i(X) = \max_{X' \in X \oplus b_{i,k}} \|X - X'\| \geq S_i(X) \qquad (23)$$

$\tilde{S}_i(X)$ is the distance from X to the farthest corner of the box at X. This will always bound the distance from X to $F_0^{-1}(F_i(X))$. The maximum deviation over an edge collapse neighborhood is the maximum $\tilde{S}_i(X)$ for any cell vertex.

The boxes, $b_{i,k}$, are the only information we keep about the position of the original mesh as we simplify. We create a new set of boxes, $b_{i+1,k}$, for mesh $\mathcal{M}_{i+1}$ using an incremental computation (described in Figure 43). Figure 42(b) shows the propagation from $\mathcal{M}_i$ to $\mathcal{M}_{i+1}$. The two lines from Figure 42(a) have now been simplified to a single line. The new box, $b_{i+1,0}$, is constant as it slides across the new line. The size and offset is chosen so that, at every point of application, $b_{i+1,0}$ contains the box $b_{i,0}$ or $b_{i,1}$, as appropriate.

```
PropagateError():
foreach cell vertex, v
  foreach triangle, Δi-1, in Mi-1 touching v
    foreach triangle, Δi, in Mi touching v
      PropagateBox(v, Δi-1, Δi)
PropagateBox(v, Δi-1, Δi)
Xi-1 = Fi-1^-1(v), Xi = Fi^-1(v)
Expand Δi's box so that when applied at Xi, it contains
  Δi-1's box applied at Xi-1
```

Figure 43: Pseudo-code to propagate the total deviation from mesh $\mathcal{M}_{i-1}$ to $\mathcal{M}_i$.

If X is a point on $\mathcal{M}_i$ in triangle k, and Y is the corresponding point on $\mathcal{M}_{i+1}$, the containment property of (22) holds:

$$F_0^{-1}(F_{i+1}(Y)) \in X \oplus b_{i,k} \subseteq Y \oplus b_{i+1,k'} \qquad (24)$$

For example, all three dots in Figure 42(b) correspond to each other. The dot on original surface, $\mathcal{M}_0$ is contained in a small box, $X \oplus b_{i,0}$, which is contained in the larger box, $Y \oplus b_{i+1,0}$.

Because each mapping cell in the overlay between $\mathcal{M}_i$ and $\mathcal{M}_{i+1}$ is linear, we compute the sizes of the boxes, $b_{i+1,k'}$, by considering only the box correspondences at cell vertices. In Figure 42(b), there are three places we must consider. If $b_{i+1,0}$ contains $b_{i,0}$ and/or $b_{i,1}$ at all three places, it will contain them everywhere.

Together, the propagation rules, which are simple to implement, and the box-based approximation to the total surface deviation, provide the tools we need to efficiently provide a surface deviation for the simplification process.

4.4.4 Accommodating Bordered Surfaces

Bordered surface are those containing edges adjacent to only a single triangle, as opposed to two triangles. Such surfaces are quite common in practice. Borders create some complications for the creation of a mapping in the plane. The problem is that the total shape of the neighborhood projected into the plane changes as a result of the edge collapse.

Bajaj and Schikore [Bajaj and Schikore 1996], who employ a vertex-removal approach, deal with this problem by mapping the removed vertex to a length-parameterized position

along the border. This solution can be employed for the edge-collapse operation as well. In their case, a single vertex maps to a point on an edge. In ours, three vertices map to points on a chain of edges.

4.5 Computing Texture Coordinates

The use of texture maps has become common over the last several years, as the hardware support for texture mapping has increased. Texture maps provide visual richness to computer-rendered models without adding more polygons to the scene.

Texture mapping requires 2D *texture coordinates* at every vertex of the model. These coordinates provide a parameterization of the texture map over the surface.

As we collapse an edge, we must compute texture coordinates for the generated vertex. These coordinates should reflect the original parameterization of the texture over the surface. We use linear interpolation to find texture coordinates for the corresponding point on the old surface, and assign these coordinates to the generated vertex.

This approach works well in many cases, as demonstrated in Section 4.7. However, there can still be some sliding of the texture across the surface. We have extended our mapping approach to also measure and bound the deviation of the texture coordinates (see Chapter 5). In this approach, the texture coordinates produce a new set of pointwise correspondences between simplifications, and the deviation measured using these correspondences measures the deviation of the texture. This extension allows us to make guarantees about the complete appearance of the simplified meshes we create and render.

As we add more error measures to our system, it becomes necessary to decide how to weight these errors to determine the overall cost of an edge collapse. Each type of error at an edge mandates a particular viewing distance based on a user-specified screen-space tolerance (e.g. number of allowable pixels of surface or texture deviation). We conservatively choose the farthest of these. At run-time, the user can still adjust an overall screen-space tolerance, but the relationships between the types of error are fixed at the time of the simplification preprocess.

4.6 System Implementation

We divide our software system into two major components: the simplification pre-process, which performs the automatic simplification described previously in this chapter, and the interactive visualization application, which employs the resulting levels of detail to perform high-speed, high-quality rendering.

4.6.1 Simplification Pre-Process

All the algorithms described in this chapter have been implemented and applied to various models. Although the simplification process itself is only a pre-process with respect to the graphics application, we would still like it to be as efficient as possible. The most time-consuming part of our implementation is the re-computation of edge costs as the surface is simplified, as described in Section 4.1.1. To reduce this computation time, we allow our approach to be slightly less greedy by performing a *lazy evaluation* of edge costs as the simplification proceeds.

Rather than recompute all the local edge costs after a collapse, we simply set a *dirty flag* for these edges. When we pick the next edge to collapse off the priority queue, which check to see if the edge is dirty. If so, we re-compute it's cost, place it back in the queue, and pick again. We repeat this until the lowest cost edge in the queue is clean. This clean edge has a lower cost than the known costs of all the other edges, be they clean or dirty. If the recent edge collapses cause an edge's cost to increase significantly, we will find out about it before actually choosing to collapse it. The potentially negative effect is that if the cost of dirty edge has decreased, we may not find out about it immediately, so we will not collapse the edge until later in the simplification process.

This lazy evaluation of edge costs significantly speeds up the algorithm without much effect on the error growth of the progressive mesh. Table 5 shows the number of edge cost evaluations and running times for simplifications of the bunny and torus models with the complete and lazy evaluation schemes. Figure 44 shows the effect of lazy evaluation on error growth for these models. The lazy evaluation has a minimal effect on error. In fact in some cases, the error of the simplification using the lazy evaluation is actually smaller. This is not

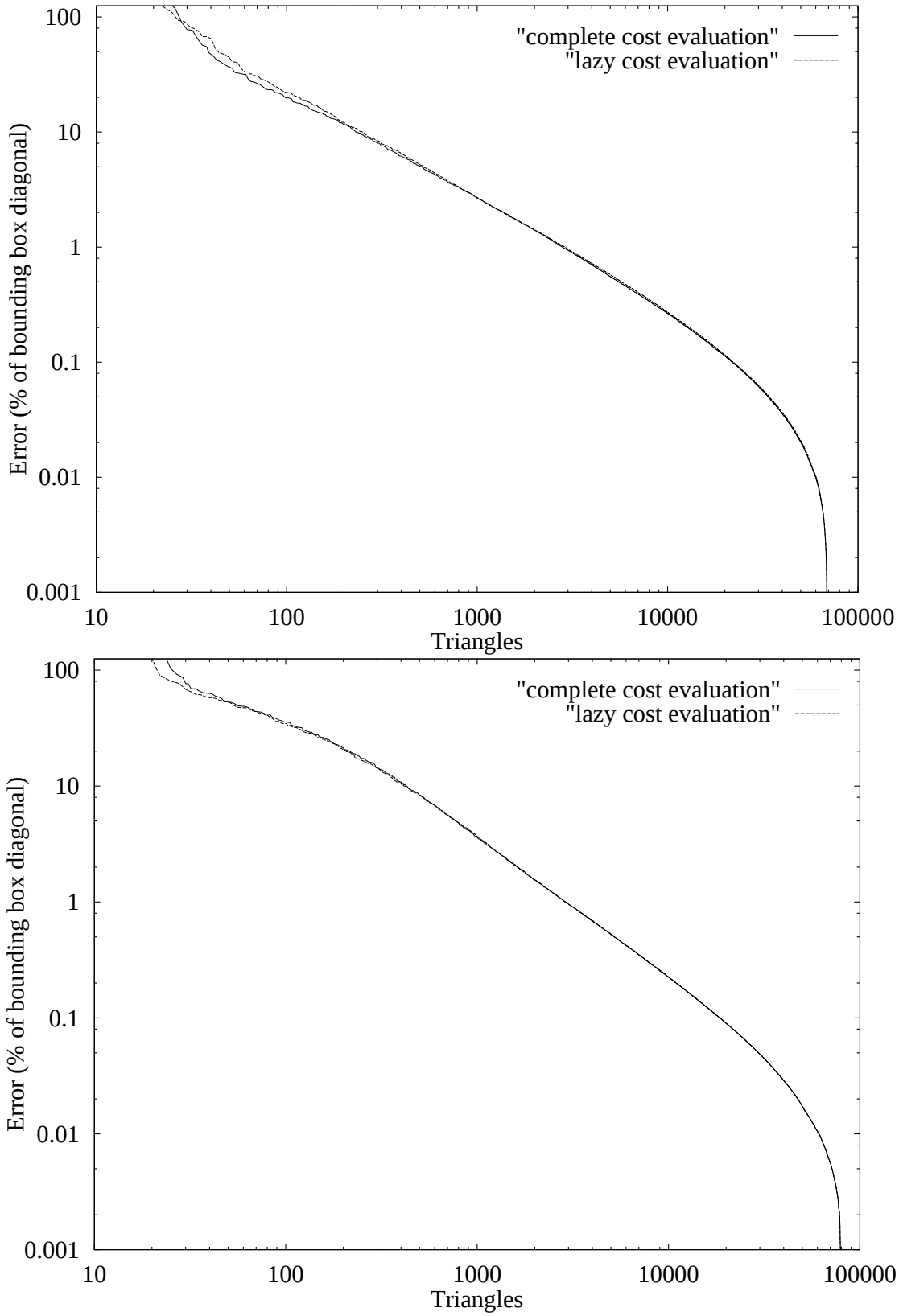


Figure 44: Error growth for simplification of two models: (top) bunny model (bottom) wrinkled torus model. The nearly coincident curves indicate that the error for the lazy cost evaluation method grows no faster than error for the complete cost evaluation method over the course of a complete simplification.

surprising, because a strictly greedy choice of edge collapses does not guarantee optimal error growth.

Given that the lazy evaluation is so successful at speeding up the simplification process with little impact on the error growth, we still have room to be more aggressive in speeding up the process. For instance, it may be possible to include a *cost estimation* method in our prioritization scheme. If we have a way to quickly estimate the cost of an edge collapse, we can use these estimates in our prioritization. Of course, we must still record the guaranteed error bound when we finally perform a collapse operation. If our guaranteed bound is too far off from our initial estimate, we may choose to put the edge back on the queue, prioritized by its true cost.

Model	Method	# Evaluations	# Collapses	#E / #C	CPU Time	% Speedup
Bunny	complete	1,372,122	34,819	39.4	5:01	N/A
	lazy	436,817	34,819	12.5	1:56	61.5
Torus	compete	1,494,625	39,982	37.4	5:27	N/A
	lazy	589,839	39,987	14.8	2:44	49.8

Table 5: Effect of lazy cost evaluation on simplification speed. The lazy method reduces the number of edge cost evaluations performed per edge collapse operation performed, speeding up the simplification process. Time is in minutes and seconds on a 195 MHz MIPS R10000 processor.

4.6.2 Interactive Visualization Application

More important than the speed of the simplification itself is the speed at which our graphics application runs. The simplification algorithm outputs a list of edge collapses and associated error bounds. Although it is possible to use this output to create view-dependent simplifications on the fly in the visualization application (as described by Hoppe [Hoppe 1997]), such a system is fairly complex, requiring computational resources to adapt the simplifications and *immediate-mode* rendering of the final triangles.

Our application is written to be simple and efficient. We first sample the progressive mesh to generate a static set of levels of detail. These are chosen to have triangle counts that decrease by a factor of two from level to level. This limits the total memory usage to twice the size of the input model.

We next load these levels of detail into our visualization application, which stores them as display lists (often referred to as *retained mode*). On machines with high-performance

graphics acceleration, such display lists are retained in a cache on the accelerator and do not need to be sent by the CPU to the accelerator every frame. On an SGI Onyx with InfiniteReality graphics, we have seen a speedup of 2-3 times, just due to the use of display lists.

Our interactive application is written on top of SGI's Iris Performer library [Rohlf and Helman 1994], which provides a software pipeline designed to achieve high graphics performance. The geometry of our model, which may be composed of many individual objects at several levels of detail, is stored in a scene graph. One of the scene graph structures, the LODNode, is used to store the levels of detail of an object. This LODNode also stores a list of switching distances, which indicate at what viewing distance each level of detail should be used (the viewing distance is the 3D distance from the eye point to the center of the object's bounding sphere). We compute these switching distances based on the 3D surface deviation error we have measured for each level of detail.

The rendering of the levels of detail in this system involves minimal overhead. When a frame is rendered, the viewing distance for each object is computed and this distance is compared to the list of switching distances to determine which level of detail to render.

The application allows the user to set a 2D error tolerance, which is used to scale the switching distances. When the error tolerance is set to 1.0, the 3D error for the rendered levels of detail will project to no more than a single pixel on the screen. Setting it to 2.0 allows two pixels of error, etc. This screen-space surface deviation amounts to the number of pixels the objects' silhouettes may be off from a rendering of the original level of detail.

4.7 Results

We have applied our simplification algorithm to four distinct objects: a bunny rabbit, a wrinkled torus, a lion, and a Ford Bronco, with a total of 390 parts. Table 6 shows the total input complexity of each of these objects as well as the time needed to generate a progressive mesh representation. All simplifications were performed on an SGI MIPS R10000 processor.

Figure 45 graphs the complexity of each object vs. the number of pixels of screen-space error for a particular viewpoint. Each set of data was measured with the object centered in the foreground of a 1000x1000-pixel viewport, with a 45^o field-of-view, like the Bronco in Plates 2 and 3. This was the easiest way for us to measure the screen-space error, because the lion

Model	Parts	Original Triangles	CPU Time (Min:Sec)
Bunny	1	69,451	1:56
Torus	1	79,202	2:44
Lion	49	86,844	1:56
Bronco	339	74,308	1:29

Table 6: Simplifications performed. CPU time indicates time to generate a progressive mesh of edge collapses until no more simplification is possible.

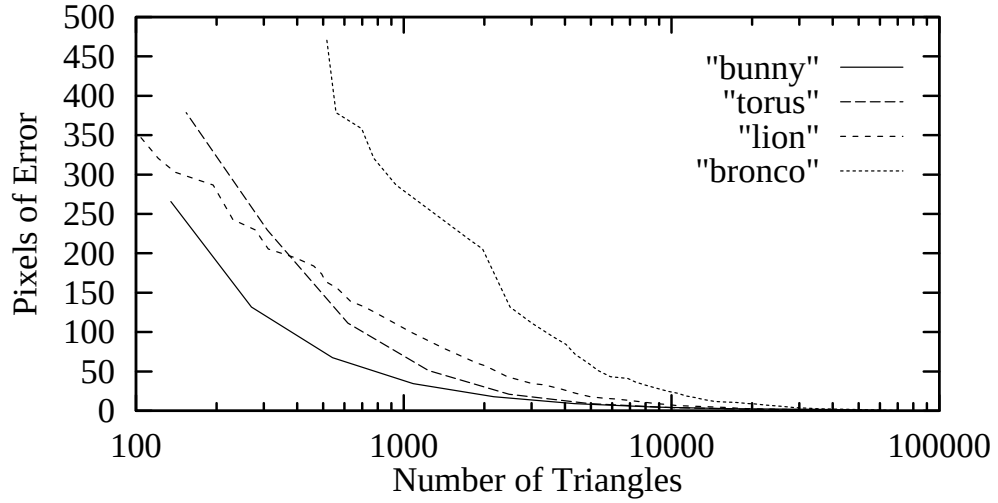


Figure 45: Complexity vs. screen-space error for several simplified models.

and bronco models each have multiple parts that independently switch levels of detail. Conveniently, this function of complexity vs. error at a fixed distance is proportional to the function of complexity vs. viewing distance with a fixed error. The latter is typically the function of interest.

Figure 46 shows the typical way of viewing levels of detail — with a fixed error bound and levels of detail changing as a function of distance. Figure 47 is a snapshot from our “circling Broncos” video. We achieve a speedup nearly proportional to the reduction in triangles. Figure 48 shows close-ups of the Bronco model at full and reduced resolutions.

Figure 49 and Figure 50 show the application of our algorithm to the texture-mapped lion and wrinkled torus models. If you know how to free-fuse stereo image pairs, you can fuse the tori or any of the adjacent pairs of textured lion. Because the tori are rendered at an appropriate distance for switching between the two levels of detail, the images are nearly indistinguishable, and fuse to a sharp, clear image. The lions, however, are not rendered at their appropriate viewing distances, so certain discrepancies will appear as fuzzy areas. Each of the lion's 49 parts is individually colored in the wire-frame rendering to indicate which of its levels of detail is currently being rendered.

Triangle counts: 41,855 27,970 20,922 12,939 8,385 4,766

Figure 46: The Ford Bronco model at 6 levels of detail, all at 2 pixels of screen-space error (0.17mm)

(a) Full resolution: 594,000 triangles

(b) 4 pixels (0.34 mm) of screen-space error: 94,000 triangles

Figure 47: 8 circling Bronco models

(a) Full Resolution: 74,000 triangles

(b) 2 pixels (0.17 mm) of error: 42,000 triangles

(c) 6 pixels (0.51 mm) of error: 29,000 triangles

(d) 26 pixels (2.2 mm) of error: 9,000 triangles

Figure 48: Close-ups of the Ford Bronco model at several resolutions.

39,600 triangles 19,800 triangles

19,800 triangles 9.900 triangles

Figure 49: Two transitional distances for the wrinkled torus model at 1 pixel (0.085 mm) of error.

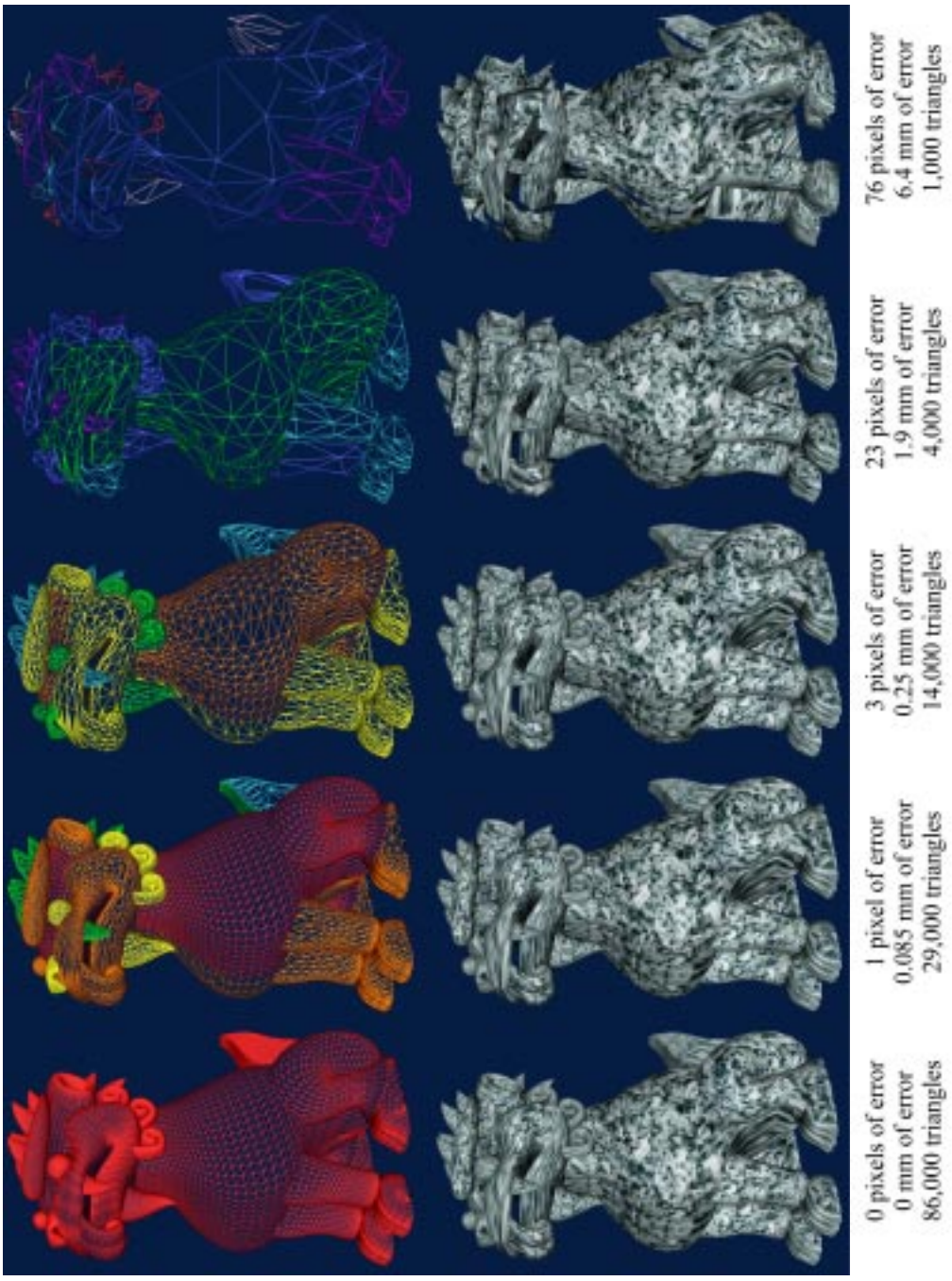


Figure 50: 6 levels of detail for the lion (colors indicate levels of detail of individual parts)

4.7.1 Applications of the Projection Algorithm

We have also applied the technique of finding a bijective planar projection to the simplification envelopes algorithm (Chapter 3). The simplification envelopes method requires the calculation of a vertex normal at each vertex that may be used as a direction to offset the vertex. The criterion for being able to move a vertex without creating a local self-intersection is the same as the criterion for being able to project to a plane. The original algorithm used a heuristic based on averaging the face normals.

By applying the projection algorithm based on linear programming (presented in Section 4.3.1) to the computation of the offset directions, we were able to perform more drastic simplifications. Because we found valid offset directions where previous heuristics failed, the envelopes were displaced more, allowing more room for simplification between the envelopes. These results are detailed in Section 3.7.2.2.

4.8 Comparison to Previous Work

Although concrete comparisons are difficult to make without careful implementations of all the related approaches readily available, we compare some of the features of our algorithm with those of a few others. The efficient and complete algorithms for computing the planar projection and placing the generated vertex after edge collapse should improve the performance of many earlier algorithms that use vertex removals or edge collapses.

We have directly compared our implementation with that of the simplification envelopes approach (Chapter 3). We generated levels of detail of the Stanford bunny model (70,000 triangles) using the simplification envelopes method, then generated levels of detail with the same number of triangles using the successive mapping approach. Visually, the models were comparable. The error bounds for the simplification envelopes method were smaller by about a factor of two for a given number of triangles, because the error bounds for the two methods measure different things. Simplification envelopes only bounds the surface deviation in the direction normal to the original surface, whereas the mapping approach prevents the surface from sliding around as well. Also, simplification envelopes created local creases in the bunnies, resulting in some shading artifacts. The successive mapping approach discourages such creases by its use of planar projections. At the same time, the performance of the

simplification envelopes approach (in terms of complexity vs. error) has been improved by our new projection algorithm.

Hoppe's progressive mesh [Hoppe 1996] implementation is more complete than ours in its handling of colors, textures, and discontinuities. However, this technique provides no guaranteed error bounds, so there is no simple way to automatically choose switching distances that guarantee some visual quality.

The multi-resolution analysis approach to simplification [DeRose et al. 1993, Eck et al. 1995, Lee et al. 1998] does, in fact, provide strict error bounds as well as a mapping between surfaces. However, the requirements of its subdivision topology and the coarse granularity of its simplification operation do not provide the local control of the edge collapse. In particular, it does not deal well with sharp edges. Hoppe [Hoppe 1996] had previously compared his progressive meshes with the multi-resolution analysis meshes. For a given number of triangles, his progressive meshes provide much higher visual quality. However, recent advances [Lee et al. 1998] have improved the quality of the multi-resolution analysis meshes by allowing the specification of constraints (e.g. along sharp edges).

Guéziec's tolerance volume approach [Guéziec 1995] also uses edge collapses with local error bounds. Unlike the boxes used by the successive mapping approach, Guéziec's error volume can grow as the simplified surface fluctuates closer to and farther away from the original surface. This is due to the fact that it uses spheres that always remain centered at the vertices, and the newer spheres must always contain the older spheres. The boxes used by our successive mapping approach are not centered on the surface and do not grow as a result of such fluctuations. Also, the tolerance volume approach does not generate mappings between the surfaces for use with other attributes. However, the tolerance volume approach allows the spheres for each triangle's vertices to be different sizes. It is unclear how this greater flexibility compares with our single box size per triangle or which approach can achieve tighter bounds over the course of an entire simplification.

We have made several significant improvements over the simplification algorithm presented by Bajaj and Schikore [Schikore and Bajaj 1995, Bajaj and Schikore 1996]. First, we have replaced their projection heuristic with a robust algorithm for finding a valid direction

of projection. Second, we have generalized their approach to handle operations with a more complex projected footprint, such as the edge collapse, which includes two interior vertices rather than a single interior vertex. Finally, we have presented an error propagation algorithm that correctly bounds the error in the surface deviation. Their approach represented error as infinite slabs surrounding each triangle. Because there is no information about the extent of these slabs, it is impossible to correctly propagate the error from a slab with one orientation to a new slab with a different orientation.

4.9 Conclusions

In this chapter, we have developed a new simplification algorithm that provides a local error metric for each edge collapse operation, generating a progressive mesh with guaranteed bounds at each increment. The main features of our approach are:

1. **Successive Mapping:** This mapping between the levels of detail is a useful tool. We use the mapping here in several ways: to measure the distance between the levels of detail before an edge collapse, to choose a location for the generated vertex that minimizes this distance, to accumulate an upper bound on the distance between the new level of detail and the original surface, and to map surface attributes to the simplified surface.

2. **Tight Error Bounds:** Our approach can measure and minimize the error for surface deviation and is extendible to other attributes. These error bounds give guarantees on the shape of the simplified object and screen-space deviation.

3. **Generality:** The algorithm for collapsing an edge into a vertex is rather general and does not restrict the vertex to lie on the original edge. Furthermore, portions of our approach can be easily combined with other algorithms, such as the simplification envelopes algorithm of Chapter 3.

4. **Surface Attributes:** Given an original surface with texture coordinates, our algorithm uses the successive mapping to compute appropriate texture coordinates for the simplified mesh. We can even provide guarantees on the final shaded appearance of the simplified mesh by maintaining colors and normals in texture and normal maps and

bounding the deviation of texture coordinates (see Chapter 5). Our approach can also be used to bound the error of any associated scalar fields [Schikore and Bajaj 1995].

5. **Continuum of Levels of Details:** The algorithm incrementally produces an entire spectrum of levels-of-details as opposed to a few discrete levels; the algorithm incrementally stores the error bounds for each level. Thus, the simplified model can be stored as a progressive mesh [Hoppe 1996] if desired.

The algorithm has been successfully applied to a number of models. These models consist of hundreds of parts and tens of thousands of polygons, including a Ford Bronco with 300 parts, a textured lion model and a textured wrinkled torus.

This new algorithm has several advantages over the simplification envelopes algorithm of Chapter 3:

- It generates an entire progressive mesh with error bounds, rather than a small number of levels of detail.
- It provides bijective mappings between levels of detail, useful for supporting textures and other attributes that vary across the surface.
- It allows geometric surface features of various scales to be simplified away without the limiting constraints of envelope surfaces.
- It tends to discourage unsightly creases in the levels of detail it creates.

The simplification envelopes algorithm also has a few advantages over this successive mapping algorithm:

- It produces smaller error bounds; if maximum surface deviation is all you want to measure, without any regard for pointwise correspondences, the simplification envelopes bounds may be advantageous for fast rendering with smaller guaranteed bounds.
- It does not depend on the existence of planar projections for the simplification operations it performs.

- It prevents self-intersections (however, the octree-based algorithm for preventing self-intersections could also be built into the successive mapping implementation, though it currently has not been).

In Chapter 5, we will build on the successive mapping approach developed in this chapter to guarantee screen-space bounds on the deviation of texture coordinates. This new capability provides the final tool we need make guarantees about the shaded appearance of our levels of detail when they are rendered by our graphics application.

5. PRESERVATION OF APPEARANCE ATTRIBUTES

In this chapter we present our third major algorithm: the appearance-preserving simplification algorithm. Our main goal for simplification is to generate a low-polygon-count approximation that maintains the high fidelity of the original model. This involves preserving the model's main features and overall appearance. Typically, there are three *appearance attributes* that contribute to the overall, *shaded appearance* of a polygonal surface:

1. **Surface position**, represented by the coordinates of the polygon vertices.
2. **Surface curvature**, represented by a field of normal vectors across the polygons.
3. **Surface color**, also represented as a field across the polygons.

The number of samples necessary to represent a surface accurately depends on the nature of the model and its area in screen pixels (which is related to its distance from the viewpoint). For a simplification algorithm to preserve the appearance of the input surface, it must guarantee adequate sampling of these three attributes. If it does, we say that it has preserved the appearance with respect to the display resolution.

The majority of work in the field of simplification has focused on *surface approximation* algorithms, such as those presented in Chapters 3 and 4. These algorithms bound the error in surface position only. Such bounds can be used to guarantee a maximum deviation of the object's silhouette in units of pixels on the screen. Although this guarantees that the object will cover the correct pixels on the screen, it says nothing about the final colors of these pixels.

Of the few simplification algorithms that deal with the remaining two attributes, most provide some threshold on a maximum or average deviation of these attribute values across the model. Although such measures do guarantee adequate sampling of all three attributes, they do *not* generally allow increased simplification as the object becomes smaller on the

screen. These threshold metrics do not incorporate information about the object's distance from the viewpoint or its area on the screen. As a result of these metrics and of the way we typically represent these appearance attributes, simplification algorithms have been quite restricted in their ability to simplify a surface while preserving its appearance.

In this chapter, we present a new algorithm for appearance-preserving simplification. We convert our input surface to a *decoupled representation*. Surface position is represented in the typical way, by a set of triangles with 3D coordinates stored at the vertices. Surface colors and normals are stored in texture and normal maps, respectively. These colors and normals are mapped to the surface with the aid of a surface parameterization, represented as 2D texture coordinates at the triangle vertices.

The surface position is filtered using a standard surface approximation algorithm that makes local, complexity-reducing simplification operations (e.g. edge collapse, vertex removal, etc.). The color and normal attributes are filtered by the run-time system at the pixel level, using standard mip-mapping techniques [Williams 1983].

Because the colors and normals are now decoupled from the surface position, we employ a new *texture deviation metric,* which effectively bounds the deviation of a mapped attribute value's position from its correct position on the original surface. We thus guarantee that each attribute is appropriately sampled and mapped to screen-space. The deviation metric necessarily constrains the simplification algorithm somewhat, but it is much less restrictive than retaining sufficient tessellation to accurately represent colors and normals in a standard, per-vertex representation. The preservation of colors using texture maps is possible on all current graphics systems that supports real-time texture maps. The preservation of normals using normal maps is possible on prototype machines today, and there are indications that hardware support for real-time normal maps will become more widespread in the next several years.

One of the nice properties of this approach is that the user-specified error tolerance, ε, is both simple and intuitive; it is a screen-space deviation in pixel units. A particular point on the surface, with some color and some normal, may appear to shift by at most ε pixels on the screen.

We have applied our algorithm to several large models. Figure 51 clearly shows the improved quality of our appearance-preserving simplification technique over a standard surface approximation algorithm with per-vertex normals. By merely controlling the switching distances properly, we can discretely switch between a few statically-generated levels of detail (sampled from a progressive mesh representation) with no perceptible artifacts. Overall, we are able to achieve a significant speedup in rendering large models with little or no loss in rendering quality. This research was performed in collaboration with Marc Olano and Dinesh Manocha, and has been published in the *Proceedings of SIGGRAPH 98* [Cohen et al. 1998].

Figure 51: Bumpy Torus Model. *Left*: 44,252 triangles full resolution mesh. *Middle and Right*: 5,531 triangles, 0.25 mm maximum image deviation. *Middle*: per-vertex normals. *Right*: normal maps

The rest of the chapter is organized as follows. In Section 5.1, we review some background material in the area of map-based representations. Section 5.2 presents an overview of our appearance-preserving simplification algorithm. Sections 5.3 through 5.5 describe the components of this algorithm, followed by a discussion of our particular implementation and results in Section 5.6. Finally, we conclude in Section 5.7.

5.1 Background on Map-based Representations

Texture mapping is a common technique for defining color on a surface. It is just one instance of mapping, a general technique for defining attributes on a surface. Other forms of mapping use the same texture coordinate parameterization, but with maps that contain something other than surface color. *Displacement maps* [Cook 1984] contain perturbations of the surface position. They are typically used to add surface detail to a simple model. *Bump maps* [Blinn 1978] are similar, but instead give perturbations of the surface normal. They can make a smooth surface appear bumpy, but will not change the surface's silhouette. *Normal maps* [Fournier 1992] can also make a smooth surface appear bumpy, but contain the actual normal instead of just a perturbation of the normal.

Texture mapping is available in most current graphics systems, including workstations and PCs. We expect to see bump mapping and similar surface shading techniques on graphics systems in the near future [Peercy et al. 1997]. In fact, many of these mapping techniques are already possible using the procedural shading capabilities of PixelFlow [Olano and Lastra 1998].

Several researchers have explored the possibility of replacing geometric information with texture. Kajiya first introduced the "hierarchy of scale" of geometric models, mapping, and lighting [Kajiya 1985]. Cabral et. al. [Cabral et al. 1987] addressed the transition between bump mapping and lighting effects. Westin et. al. [Westin et al. 1992] generated BRDFs from a Monte-Carlo ray tracing of an idealized piece of surface. Becker and Max [Becker and Max 1993] handle transitions from geometric detail in the form of displacement maps to shading in the form of bump maps. Fournier [Fournier 1992] generates maps with normal and shading information directly from surface geometry. Krishnamurthy and Levoy [Krishnamurthy and Levoy 1996] fit complex, scanned surfaces with a set of smooth B-spline patches, then store some of the lost geometric information in a displacement map or bump map. Many algorithms first capture the geometric complexity of a scene in an image-based representation by rendering several different views and then render the scene using texture maps [Maciel and Shirley 1995, Aliaga 1996, Shade et al. 1996, Darsa et al. 1997].

5.2 Overview

We now present an overview of our appearance-preserving simplification algorithm. Figure 52 presents a breakdown of the algorithm into its components. The input to the algorithm is the polygonal surface, $\mathcal{M}_0$, to be simplified. The surface may come from one of a wide variety of sources, and thus may have a variety of characteristics. The types of possible input models include:

- **CAD models**, with per-vertex normals and a single color
- **Radiositized models**, with per-vertex colors and no normals
- **Scientific visualization models**, with per-vertex normals and per-vertex colors
- **Textured models**, with texture-mapped colors, with or without per-vertex normals

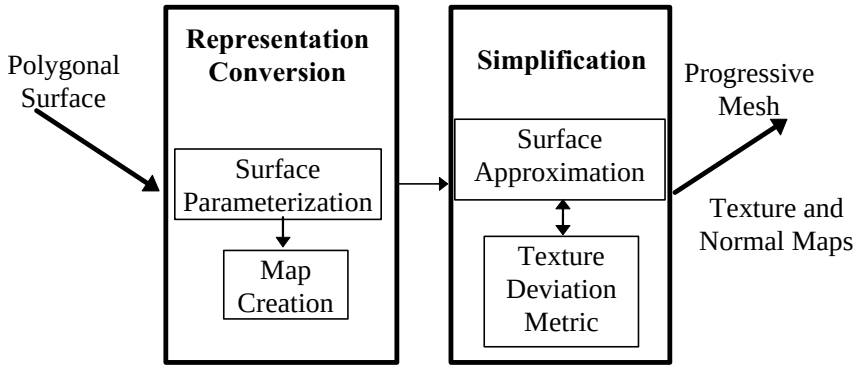


Figure 52: Components of an appearance-preserving simplification system.

To store the colors and normals in maps, we need a parameterization of the surface, $F_0(X)$: $\mathcal{M}_0 \rightarrow \mathcal{P}$, where $\mathcal{P}$ is a 2D texture domain (*texture plane*), as shown in Figure 53. If the input model is already textured, such a parameterization comes with the model. Otherwise, we create one and store it in the form of per-vertex texture coordinates. Using this parameterization, per-vertex colors and normals are then stored in texture and normal maps.

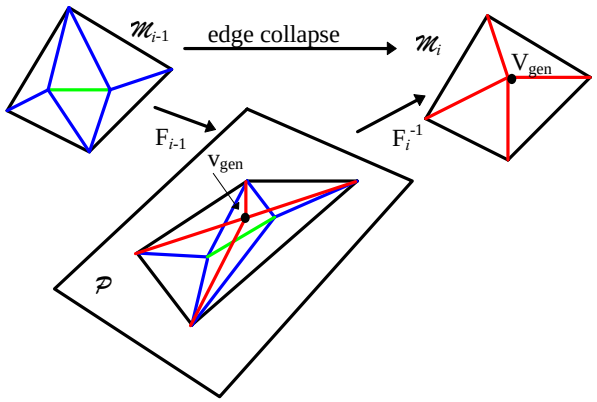


Figure 53: A look at the *i*th edge collapse. Computing V_{gen} determines the shape of the new mesh, $\mathcal{M}_i$. Computing v_{gen} determines the new mapping F_i, to the texture plane, $\mathcal{P}$.

The original surface and its texture coordinates are then fed to the surface simplification algorithm. This algorithm is responsible for choosing which simplification operations to perform and in what order. It calls our texture deviation component to measure the deviation of the texture coordinates caused by each proposed operation. It uses the resulting error bound to help make its choices of operations, and stores the bound with each operation in its progressive mesh output.

We can use the resulting progressive mesh with error bounds to create a static set of levels of detail with error bounds, or we can use the progressive mesh directly with a view-

dependent simplification system at run-time. Either way, the error bound allows the run-time system to choose or adjust the tessellation of the models to meet a user-specified tolerance. It is also possible for the user to choose a desired polygon count and have the run-time system increase or decrease the error bound to meet that target.

5.3 Representation Conversion

Before we apply the actual simplification component of our algorithm, we perform a representation conversion (as shown in Figure 52). The representation we choose for our surface has a significant impact on the amount of simplification we can perform for a given level of visual fidelity. To convert to a form that decouples the sampling rates of the colors and normals from the sampling rate of the surface, we first parameterize the surface, then store the color and normal information in separate maps.

5.3.1 Surface Parameterization

To store a surface's color or normal attributes in a map, the surface must first have a 2D parameterization. This function, $F_0(X)$: $\mathcal{M}_0 \rightarrow \mathcal{P}$, maps points, X, on the input surface, $\mathcal{M}_0$, to points, x,* on the texture plane, $\mathcal{P}$ (see Figure 53). The surface is typically decomposed into several *polygonal patches*, each with its own parameterization. The creation of such parameterizations has been an active area of research and is fundamental for shape transformation, multi-resolution analysis, approximation of meshes by NURBS, and texture mapping. Though we do not present a new algorithm for such parameterization here, it is useful to consider the desirable properties of such a parameterization for our algorithm. They are:

1. **Number of patches**: The parameterization should use as few patches as possible. The triangles of the simplified surface must each lie in a single patch, so the number of patches places a bound on the minimum mesh complexity.
2. **Vertex distribution**: The vertices should be as evenly distributed in the texture plane as possible. If the parameterization causes too much area compression, we will re-

* Capital letters (e.g. X**)** refer to points in 3D, while lowercase letters (e.g. x) refer to points in 2D.

quire a greater map resolution to capture all of our original per-vertex data.

3. **Bijective mapping**: The mapping from the surface to the texture plane should be a bijection. If the surface has folds in the texture plane, parts of the texture will be incorrectly stored and mapped back to the surface

Our particular application of the parameterization makes us somewhat less concerned with preserving aspect ratios than some other applications are. For instance, many applications apply $F^{-1}(x)$ to map a pre-synthesized texture map to an arbitrary surface. In that case, distortions in the parameterization cause the texture to look distorted when applied to the surface. However, in our application, the color or normal data originates on the surface itself. Any distortion created by applying $F(X)$ to map this data onto $\mathcal{P}$ is reversed when we apply $F^{-1}(x)$ to map it back to $\mathcal{M}$.

Algorithms for computing such parameterizations have been studied in the computer graphics and graph drawing literature.

Computer Graphics: In the recent computer graphics literature, [Kent et al. 1992, Maillot et al. 1993, Eck et al. 1995] use a spring system with various energy terms to distribute the vertices of a polygonal patch in the plane. [Maillot et al. 1993, Eck et al. 1995, Krishnamurthy and Levoy 1996, Pedersen 1996] provide methods for subdividing surfaces into separate patches based on automatic criteria or user-guidance. This body of research addresses the above properties one and two, but unfortunately, parameterizations based on spring-system algorithms do not generally guarantee a bijective mapping.

Graph Drawing: The field of graph drawing addresses the issue of bijective mappings more rigorously. Relevant topics include straight-line drawings on a grid [Fraysseix et al. 1990] and convex straight-line drawings [Chiba et al. 1985]. Di Battista et al. [Di Battista et al. 1994] present a survey of the field. These techniques produce guaranteed bijective mappings, but the necessary grids for a graph with V vertices are worst case (and typically) $O(V)$ width and height, and the vertices are generally unevenly spaced.

To break a surface into polygonal patches, we currently apply an automatic subdivision algorithm like that presented in [Eck et al. 1995]. Their application requires a patch network

with more constraints than ours. We can generally subdivide the surface into fewer patches. During this process, which grows Voronoi-like patches, we simply require that each patch not expand far enough to touch itself. To produce the parameterization for each patch, we employ a spring system with uniform weights. A side-by-side comparison of various choices of weights in [Eck et al. 1995] shows that uniform weights produce more evenly-distributed vertices than some other choices. For parameterizations used only with one particular map, it is also possible to allow more area compression where data values are similar. Although this technique will generally create reasonable parameterizations, it would be better if there were a way to *also* guarantee that $F(X)$ is a bijection, as in the graph drawing literature.

5.3.2 Creating Texture and Normal Maps

Given a polygonal surface patch, $\mathcal{M}_0$, and its 2D parameterization, F, it is straightforward to store per-vertex colors and normals into the appropriate maps using standard rendering software. To create a map, scan convert each triangle of $\mathcal{M}_0$, replacing each of its vertex coordinates, V_j, with $F(V_j)$, the texture coordinates of the vertex. For a texture map, apply the Gouraud method for linearly interpolating the colors across the triangles. For a normal map, interpolate the per-vertex normals across the triangles instead (as shown in Figure 54).

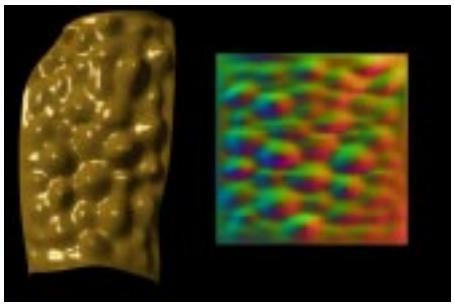


Figure 54: A patch from the leg of an armadillo model and its associated normal map.

The most important question in creating these maps is what the maximum resolution of the map images should be. To capture all the information from the original mesh, each vertex's data should be stored in a unique texel. We can guarantee this conservatively by choosing $1/d$ x $1/d$ for our map resolution, where d is the minimum distance between vertex texture coordinates:

$$d = \min_{\mathbf{V}_i, \mathbf{V}_j \in \mathcal{M}_0, i \neq j} \left\| F(V_i) - F(V_j) \right\| \qquad (25)$$

If the vertices of the polygonal surface patch happen to be a uniform sampling of the texture space (e.g. if the polygonal surface patch was generated from a parametric curved surface patch), then the issues of scan conversion and resolution are simplified considerably. Each vertex color (or normal) is simply stored in an element of a 2D array of the appropriate dimensions, and the array itself is the map image.

It is possible to trade off accuracy of the map data for run-time texturing resources by scaling down the initial maps to a lower resolution. In fact, it may be possible to tune the parameterization to work with the particular attribute values of a given model. For instance, the parameterization should ideally place topologically adjacent vertices with similar attribute values closer together in the texture space than adjacent vertices with more dissimilar values. Using such a scheme, we can minimize the filtering error that occurs at each level of the mip-map, and thus also reduce the negative effects of discarding some of the higher-resolution map levels when desired.

5.4 Simplification Algorithm

Once we have decomposed the surface into one or more parameterized polygonal patches with associated maps, we begin the actual simplification process. Many simplification algorithms perform a series of edge collapses or other local simplification operations to gradually reduce the complexity of the input surface. The order in which these operations are performed has a large impact on the quality of the resulting surface, so simplification algorithms typically choose the operations in order of increasing error according to some metric. This metric may be local or global in nature, and for surface approximation algorithms, it provides some bound or estimate on the error in surface position. The operations to be performed are typically maintained in a priority queue, which is continually updated as the simplification progresses. This basic design is applied by many of the current simplification algorithms, including [Guéziec 1995, Hoppe 1996, Cohen et al. 1997, Garland and Heckbert 1997] and the successive mapping algorithm of Chapter 4.

To incorporate our appearance-preservation approach into such an algorithm, the original algorithm is modified to use our texture deviation metric in addition to its usual error metric. When an edge is collapsed, the error metric of the particular surface approximation algorithm

is used to compute a value for V_{gen}, the surface position of the new vertex (see Figure 53). Our texture deviation metric is then applied to compute a value for v_{gen}, the texture coordinates of the new vertex.

For the purpose of computing an edge's priority, there are several ways to combine the error metrics of surface approximation along with the texture deviation metric, and the appropriate choice depends on the algorithm in question. Several possibilities for such a *total error metric* include a weighted combination of the two error metrics, the maximum or minimum of the error metrics, or one of the two error metrics taken alone. For instance, when integrating with Garland and Heckbert's algorithm [Garland and Heckbert 1997], it would be desirable to take a weighted combination in order to retain the precedence their system accords the topology-preserving collapses over the topology-modifying collapses. Similarly, a weighted combination may be desirable for an integration with Hoppe's system [Hoppe 1996], which already optimizes error terms corresponding to various mesh attributes.

To integrate with the successive mapping simplification algorithm of Chapter 4, we use the error calculated by the orthogonal projection mapping to compute V_{gen}, compute a heuristic guess for v_{gen} based on this projection mapping, then compute the final value for v_{gen} and measure the texture deviation. As we discuss in Section 5.6.2, the resulting texture deviation error may be taken alone as the reported error for the edge collapse.

The interactive display system later uses the error reported by any or all of the metrics to determine appropriate distances from the viewpoint either for switching between static levels of detail or for collapsing/splitting the edges dynamically to produce adaptive, view-dependent tessellations. If the system intends to guarantee that certain tolerances are met, the maximum of the error metrics is often an appropriate choice.

5.5 Texture Deviation Metric

A key element of our approach to appearance-preservation is the measurement of the *texture coordinate deviation* caused by the simplification process. We provide a bound on this deviation, to be used by the simplification algorithm to prioritize the potential edge collapses and by the run-time visualization system to choose appropriate levels of detail based on the current viewpoint.

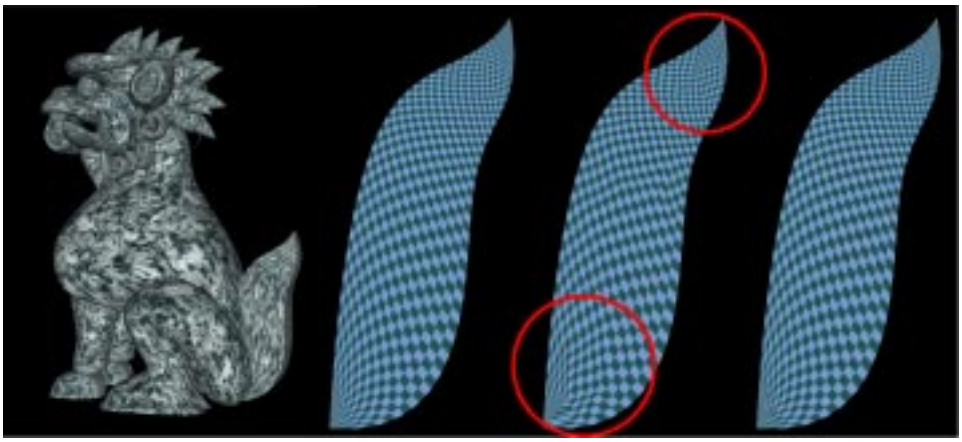


Figure 55: Lion model.

Figure 56: Texture coordinate deviation and correction on the lion s tail. *Left*: 1,740 triangles full resolution. *Middle and Right*: 0.25 mm maximum image deviation. *Middle*: 108 triangles, no texture deviation metric. *Right*: 434 triangles with texture metric.

The lion's tail in Figure 56 (one part of the lion model of Figure 50 and Figure 55) demonstrates the need to measure texture coordinate deviation. The center figure is simplified by a surface approximation algorithm without using a texture deviation metric. The distortions are visible in the areas marked by red circles. The right tail is simplified using our texture deviation metric and does not have visible distortions. The image-space deviation bound now applies to the texture as well as to the surface.

For a given point, **X**, on simplified mesh $\mathcal{M}_i$, this deviation is the distance in 3D from **X** to the point on the input surface with the same texture coordinates:

$$T_i(X) = \left\| X - F_0^{-1}(F_i(X)) \right\| \tag{26}$$

This texture deviation function looks remarkably similar to the total surface deviation function, $S_i(X)$ defined by (21) on page 84. In fact, they are just the same; they are both measures of 3D distances between pairs of corresponding points, with one point on $\mathcal{M}_i$ and the other point on $\mathcal{M}_0$. The only difference is the choice of the mapping function, F. In Chapter 4, we used orthogonal projections of each edge neighborhood to define F. Here, F is the texture-coordinate parameterization, represented by the texture coordinates at each vertex.

One implication of this similarity is that the texture deviation metric can actually serve as a surface deviation metric as well. It does measure how far the surface has moved according to some mapping. Another implication is that the algorithms for computing a bound on the

total texture deviation will be the same as those for computing the total surface deviation. Replacing the orthogonal projection mapping with the texture coordinate mapping implies that all our 2D computations will be performed in the texture plane rather than the plane of projection.

We define the texture coordinate deviation of a whole triangle to be the maximum deviation of all the points in the triangle, and similarly for the whole surface:

$$T_i(\Delta) = \max_{X \in \Delta} T_i(X); \quad T_i(\mathcal{M}_i) = \max_{X \in \mathcal{M}_i} T_i(X) \tag{27}$$

To compute the texture coordinate deviation incurred by an edge collapse operation, our algorithm takes as input the set of triangles before the edge collapse and V_{gen}, the 3D coordinates of the new vertex generated by the collapse operation. The algorithm outputs v_{gen}, the 2D texture coordinates for this generated vertex, and a bound on $T_i(\Delta)$ for each of the triangles after the collapse.

5.5.1 Computing New Texture Coordinates

Computing the new texture coordinates, v_{gen}, is equivalent to the process of placing the vertex in the plane of projection, described in Section 4.3.2. For v_{gen} to be valid, it must lie in the kernel of our polygon in the texture plane (see Figure 38 on page 79). Meeting this criterion ensures that the set of triangles after the edge collapse covers exactly the same portion of the texture plane as the set of triangles before the collapse.

Given a candidate point in the texture plane, we efficiently test the kernel criterion with a series of dot products to see if it lies on the inward side of each polygon edge. We first test some heuristic choices for the texture coordinates – the midpoint of the original edge in the texture plane or one of the edge vertices. In our case, we use the corresponding point provided by the projection mapping. If the heuristic choices fail we compute a point inside the kernel by averaging three corners, found using the same linear programming techniques presented in Section 4.3.2.

5.5.2 Patch Borders and Continuity

Unlike an interior edge collapse, an edge collapse on a patch border can change the cov-

erage in the texture plane, either by cutting off some of texture space or by extending into a portion of texture space for which we have no map data. Since neither of these is acceptable, we add additional constraints on the choice of v_{gen} at patch borders.

We assume that the area of texture space for which we have map data is rectangular (though the method works for any map that covers a polygonal area in texture space), and that the edges of the patch are also the edges of the map. If the entire edge to be collapsed lies on a border of the map, we restrict v_{gen} to lie on the edge. If one of the vertices of the edge lies on a corner of the map, we further restrict v_{gen} to lie at that vertex. If only one vertex is on the border, we restrict v_{gen} to lie at that vertex. If one vertex of the edge lies on one border of the map and the other vertex lies on a different border, we do not allow the edge collapse.

The surface parameterization component typically breaks the input model into several connected patches. To preserve geometric and texture continuity across the boundary between them, we add further restrictions on the simplifications that are performed along the border. The shared border edges must be simplified on both patches, with matching choices of V_{gen} and v_{gen}.

5.5.3 Measuring Texture Deviation

Texture deviation is a measure of the parametric distortion caused by the simplification process. We measure this deviation using the same method we use to measure surface deviation in Chapter 4, except we now measure the deviation using our mapping in the texture plane, rather than in the plane of an orthogonal projection.

We first complete our mapping (as in Section 4.3.4) by computing the vertices of the mapping cells (see Figure 39 on page 81). The maximum *incremental texture deviation* for each mapping cell must occur at one of its vertices. We then propagate this incremental texture deviation (as in Section 0) using a set of axis-aligned boxes, one for each triangle. These boxes accumulate our bound on the total texture deviation (see Figure 42 on page 85).

Because our current system ultimately uses only this bound on total texture deviation as its measure of error for an edge collapse, it is not necessary in the context of this appearance-preserving system to perform error propagation for the *surface* deviation — only for the *texture* deviation. In this system, the projection mapping algorithm is used strictly to choose

the optimized 3D position for V_{gen} and to provide the heuristic choose for the new texture coordinates, v_{gen}. The optimization of V_{gen} requires only the incremental surface deviation at the cell vertices.

5.6 Implementation and Results

In this section we present some details of our implementation of the various components of our appearance-preserving simplification algorithm. These include methods for representation conversion, simplification and, finally, interactive display.

5.6.1 Representation Conversion

We have applied our technique to several large models, including those listed in Table 7. The bumpy torus model (Figure 51) was created from a parametric equation to demonstrate the need for greater sampling of the normals than of the surface position. The lion model (Figure 55) was designed from NURBS patches as part of a much larger garden environment, and we chose to decorate it with a marble texture (and a checkerboard texture to make texture deviation more apparent in static images). Neither of these models required the computation of a parameterization. The "armadillo" (Figure 58) was constructed by merging several laser-scanned meshes into a single, dense polygon mesh. It was decomposed into polygonal patches and parameterized using the algorithm presented in [Krishnamurthy and Levoy 1996], which eventually converts the patches into a NURBS representation with associated displacement maps.

Because all these models were not only parameterized, but available in piecewise-rational parametric representations, we generated polygonal patches by uniformly sampling these representations in the parameter space. We chose the original tessellation of the models to be high enough to capture all the detail available in their smooth representations. Due to the uniform sampling, we were able to use the simpler method of map creation (described in Section 5.3.2), avoiding the need for a scan-conversion process.

5.6.2 Simplification

We integrated our texture deviation metric with the successive mapping algorithm for surface approximation (Chapter 4). The error metric for that algorithm is a 3D surface

deviation. We used this deviation only in the computation of V_{gen}. Our total error metric for prioritizing edges and choosing switching distances is just the texture deviation. This is sensible because the texture deviation metric is also a measure of surface deviation, whose particular mapping is the parameterization. Thus, if the successive mapping metric is less than the texture deviation metric, we must apply the texture deviation metric, because it is the minimum bound we know that guarantees the bound on our texture deviation. On the other hand, if the successive mapping metric is greater than the texture deviation metric, the texture deviation bound is still sufficient to guarantee a bound on both the surface deviation and the texture.

To achieve a simple and efficient run-time system, we apply a post-process to convert the progressive mesh output to a static set of levels of detail, reducing the mesh complexity by a factor of two at each level.

Our implementation can either treat each patch as an independent object or treat a connected set of patches as one object. If we simplify the patches independently, we have the freedom to switch their levels of detail independently, but we will see cracks between the patches when they are rendered at a sufficiently large error tolerance. Simplifying the patches together allows us to prevent cracks by switching the levels of detail simultaneously.

Table 7 gives the computation time to simplify several models, as well as the resolution of each map image. Figure 57 and Figure 58 show results on the “armadillo” model. It should be noted that the latter figure is not intended to imply equal computational costs for rendering models with per-vertex normals and normal maps. Simplification using the normal map representation provides measurable quality and reduced triangle overhead, with an additional overhead dependent on the screen resolution.

5.6.3 Interactive Display System

We have implemented two interactive display systems: one on top of SGI’s IRIS Performer library, and one on top of a custom library running on a PixelFlow system. The SGI system supports color preservation using texture maps, and the PixelFlow system supports color and normal preservation using texture and normal maps, respectively. Both systems apply a bound on the distance from the viewpoint to the object to convert the texture devia-

tion error in 3D to a number of pixels on the screen (as described in Section 3.6), and allow the user to specify a tolerance for the number of pixels of deviation. The tolerance is ultimately used to choose the primitives to render from among the statically generated set of levels of detail.

Our custom shading function on the PixelFlow implementation performs a mip-mapped look-up of the normal and applies a Phong lighting model to compute the output color of each pixel. The current implementation looks up normals with 8 bits per component, which seems sufficient in practice (using the reformulation of the Phong lighting equation described in [Lyon 1993]).

Model	Patches	Input Triangles	Time	Map Resolution
Torus	1	44,252	4.4	512x128
Lion	49	86,844	7.4	N/A
"Armadillo"	102	2,040,000	190	128x128

Table 7: Several models used to test appearance-preserving simplification. Simplification time is in minutes on a MIPS R10000 processor.

5.7 Conclusions

In this chapter we have demonstrated an appearance-preserving simplification system, which provides proper sampling of appearance attributes. Our current implementation demonstrates the feasibility and desirability of our approach to appearance-preserving simplification. It produces high-fidelity images using a small number of high-quality triangles. In addition, it provides an intuitive error metric based on the notion of screen-space displacement of essentially correctly-shaded pixels. This approach allows us to maintain the rendering quality we desire while managing the geometry required to achieve this quality.

Figure 57: Levels of detail of the armadillo model shown with 1.0 mm maximum image deviation. Triangle counts are: 7,809, 3,905, 1,951, 975, 488

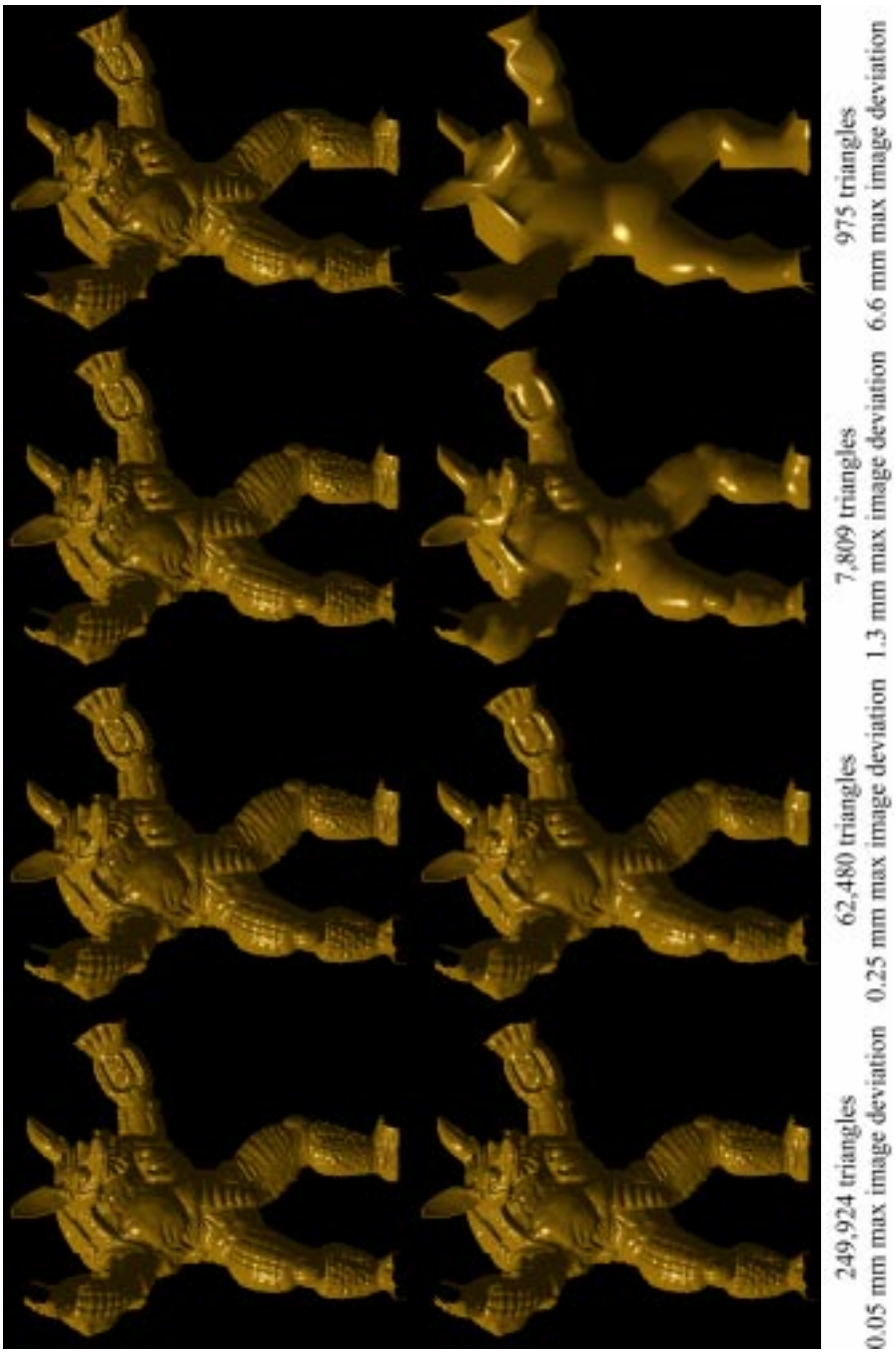


Figure 58: Close-up of several levels of detail of the armadillo model. *Top*: normal maps *Bottom*: per-vertex normals

6. CONCLUSION

6.1 Contributions

This dissertation presents three major simplification algorithms:

1. Simplification Envelopes
2. Successive Mappings
3. Appearance-Preserving Simplification

Together, they make a number of contributions to the field of polygonal mesh simplification (see Section 1.7 for more detail):

- Increased robustness and scalability of the simplification envelopes algorithm
- Local error metric for surface-to-surface deviation between original and simplified surfaces
- Bijective mappings between original and simplified surfaces for the edge collapse operation
- Local error metric for texture deviation, with bijective mappings between original and simplified surfaces
- Appearance-preserving simplification algorithm
- Intuitive, screen-space error metric for surface and texture deviations

6.2 Future Extensions

We now describe some possible extensions to this dissertation research. These extensions are organized by category: minimizing error, bijective mappings, non-manifold meshes, parameterization, and appearance preservation.

6.2.1 Minimizing Error

It seems clear that there is much more we could do algorithmically to reduce the error in our simplified models. This can be accomplished by reducing the actual error, tightening the bounds on our measurement of the actual error, or both. One of the shortcomings of greedy simplification algorithms with no bound on their output size compared to the optimal solution is that we have little idea of how much better we could be doing. Reducing our reported error may very well increase the time it takes to perform the simplification. If we are approaching the point of diminishing returns, this may be of purely academic interest. On the other hand, there may still be significant room for improvement, and we may find the error reduction to be of great practical use.

The minimization of surface deviation for the successive mapping algorithm currently uses only one degree of freedom for its optimization of the 3D vertex position. Moving in the other two dimensions can change the mapping, making the optimization process more complex. In addition, we currently minimize the incremental surface deviation, not the total surface deviation. Finally, we accumulate our total error from the original surface using axis-aligned boxes. If we could measure and minimize the total error directly from the original surface, we could avoid some unnecessary error accumulation.

For the appearance-preserving simplification algorithm, the minimization of texture deviation is even more heuristic. The orthogonal projection mapping is used to compute the corresponding texture coordinates for the generated vertex. These texture coordinates are not currently optimized to minimize the texture deviation. Such an optimization would be similar to optimizing the remaining two dimensions of the 3D vertex in the successive mapping algorithm; it would change the mapping in the texture plane, making the optimization process more complex.

For the simplification of complex models, the heuristic nature of the current minimization is offset somewhat by the priority queue mechanism. There are many edge collapse operations to choose from to determine the next simplification step, so it may not matter too much if each operation is not optimized as well as possible. However, as the model becomes more coarse, the minimization of error becomes more important, and simplification with small

error growth becomes more difficult. The need for better error minimization depends, to some extent, on what level of simplification the graphics applications require.

6.2.2 Bijective Mappings

The mappings we generate using the successive mappings approach are not the best mappings possible. Given any choice 3D position for the generated vertex, it should be possible to optimize the mapping so it minimizes the maximum deviation. Adjusting the mapping affects the distance function by changing the correspondences. In fact, we could even construct our mappings directly on the 3D surface, rather than relying on planar projections at all. Such a mapping might be initialized to be the natural mapping, then allow an optimization process to minimize the error by adding, removing, or adjusting the piece-wise linear mapping cells on the two surfaces. A similar process might even be used to generate mappings between levels of detail create by the simplification envelopes approach.

6.2.3 Non-manifold Meshes

These simplification algorithms, like most topology-preserving simplification algorithms, apply most readily to manifold meshes. However, non-manifold meshes are of great importance for practical applications. Many meshes are not manifold, and we can even convert meshes with all sorts of arbitrary polygon intersections into non-manifold meshes, with all the intersections identified.

Our appearance-preserving simplification algorithm has the makings of an approach to simplifying non-manifold meshes. The non-manifold meshes may be decomposed into several manifold meshes and their adjacency information, much like the meshes in the appearance-preserving algorithm are decomposed into adjacent patches with their individual parameterizations. This sort of approach may work well for meshes that are “mostly manifold,” with a few non-manifold regions. If the mesh is “mostly non-manifold,” there will be too many patches, and we must develop new ways to achieve some degree of coherence across the surface.

There is considerable overlap of the handling of non-manifold meshes with the handling of topological modifications. In this area, there is a great need to explore ways of modifying

topology while preserving the surface appearance. This may include a characterization of which types of topological modifications can have the most impact on the appearance of a surface.

6.2.4 Parameterization

Most algorithms for polygonal surface parameterization today optimize the parameterization to minimize distortions between the parameter space and the surface. This is important for applying prefabricated, general-purpose textures to a variety of objects. Our application of the parameterization is different, however, because the data originates at the vertices of the polygonal surface; any distortion in mapping the data to the texture domain is reversed when the data is reapplied to the surface during rendering.

In our case, it is more important to optimize the parameterization for the minimization of filtering error and storage consumption. Using the actual data to be stored in a map, we should be able to optimize the parameterization to further these goals. Topologically adjacent data values with similar values should be stored more closely together in the parameter space than those with dissimilar values. This will help minimize the error that occurs at each level of the mip-map pyramid structure. For applications willing to tolerate some additional error, it may be possible to discard one or more of the highest resolution mip-map levels, reducing storage and possibly rendering bandwidth.

6.2.5 Appearance Preservation

Our appearance-preserving algorithm takes a clear and consistent approach to preserving appearance attributes that vary across a given polygonal surface. However, the algorithm depends on the commercial availability of graphics acceleration hardware that provides:

a) Sufficient bandwidth to render all the necessary appearance attribute maps at the desired screen resolution

b) Computing resources and flexibility to light and shade polygonal primitives according to attribute map values (in either single- or multi-pass fashion)

Even for those platforms with these capabilities, it is important for us to manage the bandwidth required for complex scenes by ensuring that only the smallest necessary mip-map levels are transmitted within the graphics engine.

For less capable graphics platforms, it will be necessary to develop other algorithms for preserving appearance, using the more traditionally accelerated rendering techniques. Though most graphics accelerators today have support for texture mapping, they may not have the necessary bandwidth to transmit enough unique texture data to cover the screen at the desired resolution (many gaming applications cover most of the screen area with generic, repeatable texture patterns applied to the polygons). Thus, we may need to rely only on the capability of handling per-vertex or per-polygon colors and normals. The metrics we use to measure the effects of simplification on these colors and normals must take into account not only the changes in these values, but the value changes times the area on the screen, and the resulting changes in final, lit colors times the area on the screen. Such metrics will allow us to make some guarantees as to the quality of the appearance of the resulting images.

6.3 Simplification in Context

Polygonal simplification is one of many techniques used to accelerate the rendering of complex scenes. Other techniques include back-face culling, view-frustum culling, occlusion culling, cell-and-portal culling, and image replacement. The various culling techniques attempt to remove hidden geometry, whereas polygonal simplification and image replacement (replacing some amount of complex geometry with an image) attempt to reduce the complexity of the visible geometry.

An ambitious goal for an interactive 3D graphics algorithm is to have a rendering complexity that is output sensitive. That is to say, the rendering algorithm should ideally take an amount of time that is proportional only to the screen resolution, but not to the scene complexity, to generate an image. Also, the rendered image should have as high a quality as that produced with a reasonable non-output-sensitive algorithm. Polygonal simplification is not likely to become a powerful enough tool to achieve this goal on its own. Even if we incorporate topological changes and object merging into our simplification scheme, as in [Luebke and Erikson 1997], it will be difficult to achieve both our complexity and quality goals.

The combination of simplification techniques and image replacement techniques seems a promising approach to achieving our high-quality, interactive rendering goals. A prototype system that takes just such an approach is demonstrated in [Aliaga et al. 1998]. Pre-computed images are used to replace far geometry, whereas simplification is used to reduce the complexity of nearby geometry. Each technique is thus applied to its area of greatest strength — geometric primitives up close and images far away.

Among its other contributions, this dissertation presents an algorithm for appearance-preserving simplification, using rendering primitives that provide a high ratio of quality to rendering complexity (especially reducing transformation complexity). The representation is a hybrid of sorts, itself employing both geometry and mapped images. We hope that our approaches to appearance preservation and simplification in general, which provide guaranteed error bounds with the simplified models, will be useful components of the interactive rendering algorithms of the future.

7. REFERENCES

Agarwal, Pankaj K. and Subhash Suri. Surface Approximation and Geometric Partitions. *Proceedings of 5th ACM-SIAM Symposium on Discrete Algorithms.* 1994. pp. 24-33.

Aliaga, Daniel, Jonathan Cohen, Andrew Wilson, Hansong Zhang, Carl Erikson, Kenneth Hoff, Thomas Hudson, Eric Baker, Rui Bastos, Mary Whitton, Frederick Brooks, and Dinesh Manocha. A Framework for the Real-Time Walkthrough of Massive Models. Technical Report TR #98-013. Department of Computer Science, University of North Carolina at Chapel Hill. March. 1998.

Aliaga, Daniel G. Visualization of Complex Models using Dynamic Texture-Based Simplification. *Proceedings of IEEE Visualization'96.* pp. 101-106.

Bajaj, Chandrajit and Daniel Schikore. Error-bounded Reduction of Triangle Meshes with Multivariate Data. *SPIE.* vol. 2656. 1996. pp. 34-45.

Barequet, Gill and Subodh Kumar. Repairing CAD Models. *Proceedings of IEEE Visualization '97.* October 19-24. pp. 363-370, 561.

Bastos, Rui, Mike Goslin, and Hansong Zhang. Efficient Rendering of Radiosity using Texture and Bicubic Interpolation. *Proceedings of 1997 ACM Symposium on Interactive 3D Graphics.*

Becker, Barry G. and Nelson L. Max. Smooth Transitions between Bump Rendering Algorithms. *Proceedings of SIGGRAPH 93.* pp. 183-190.

Blinn, Jim. Simulation of Wrinkled Surfaces. *Proceedings of SIGGRAPH 78.* pp. 286-292.

Brönnimann, H. and Michael T. Goodrich. Almost Optimal Set Covers in Finite VC-Dimension. *Proceedings of 10th Annual ACM Symposium on Computational Geometry.* 1994. pp. 293-302.

Cabral, Brian, Nelson Max, and R. Springmeyer. Bidirectional Reflection Functions from Surface Bump Maps. *Proceedings of SIGGRAPH 87.* pp. 273-281.

Carmo, M. do. *Differential Geometry of Curves and Surfaces.* Prentice Hall 1976.

Certain, Andrew, Jovan Popovic, Tony DeRose, Tom Duchamp, David Salesin, and Werner Stuetzle. Interactive Multiresolution Surface Viewing. *Proceedings of SIGGRAPH 96.* pp. 91-98.

Chiba, N., T. Nishizeki, S. Abe, and T. Ozawa. A Linear Algorithm for Embedding Planar Graphs using PQ-Trees. *Journal of Computer and System Sciences.* vol. 30(1). 1985. pp. 54-76.

Clarkson, Kenneth L. Algorithms for Polytope Covering and Approximation. *Proceedings of 3rd Workshop on Algorithms and Data Structures.* 1993. pp. 246-252.

Cohen, Jonathan, Dinesh Manocha, and Marc Olano. Simplifying Polygonal Models using Successive Mappings. *Proceedings of IEEE Visualization '97.* pp. 395-402.

Cohen, Jonathan, Marc Olano, and Dinesh Manocha. Appearance-Preserving Simplification. *Proceedings of ACM SIGGRAPH 98.* pp. 115-122.

Cohen, Jonathan, Amitabh Varshney, Dinesh Manocha, Gregory Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification Envelopes. *Proceedings of SIGGRAPH 96.* pp. 119-128.

Cook, Robert L. Shade Trees. *Proceedings of SIGGRAPH 84.* pp. 223-231.

Darsa, Lucia, Bruno Costa Silva, and Amitabh Varshney. Navigating Static Environments using Image-Space Simplification and Morphing. *Proceedings of 1997 Symposium on Interactive 3D Graphics.* pp. 25-34.

Das, G. and D. Joseph. The Complexity of Minimum Convex Nested Polyhedra. *Proceedings of 2nd Canadian Conference on Computational Geometry.* 1990. pp. 296-301.

DeFloriani, Leila, Paola Magillo, and Enrico Puppo. Building and Traversing a Surface at Variable Resolution. *Proceedings of IEEE Visualization '97.* pp. 103-110.

DeRose, Tony, Michael Lounsbery, and J. Warren. Multiresolution Analysis for Surfaces of Arbitrary Topology Type. Technical Report TR 93-10-05. Department of Computer Science, University of Washington. 1993.

Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for Drawing Graphs: An Annotated Bibliography. *Computational Geometry Theory and Applications.* vol. 4. 1994. pp. 235-282.

Dörrie, H. Euler's Problem of Polygon Division. *100 Great Problems of Elementary Mathematics: Their History and Solutions.* Dover, New York.1965. pp. 21-27.

Eck, Matthias, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution Analysis of Arbitrary Meshes. *Proceedings of SIGGRAPH 95.* pp. 173-182.

El-Sana, Jihad and Amitabh Varshney. Controlled Simplification of Genus for Polygonal Models. *Proceedings of IEEE Visualization'97.* pp. 403-410.

Erikson, Carl. Polygonal Simplification: An Overview. Technical Report TR96-016. Department of Computer Science, University of North Carolina at Chapel Hill. 1996.

Erikson, Carl and Dinesh Manocha. Simplification Culling of Static and Dynamic Scene Graphs. Technical Report TR98-009. Department of Computer Science, University of North Carolina at Chapel Hill. 1998.

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice. The Systems Programming Series.* 2nd edition. Addison-Wesley, Reading, MA. 1990. 1175 pages.

Fournier, Alain. Normal Distribution Functions and Multiple Surfaces. *Proceedings of Graphics Interface '92 Workshop on Local Illumination.* pp. 45-52.

Fraysseix, H. de, J. Pach, and R. Pollack. How to Draw a Planar Graph on a Grid. *Combinatorica.* vol. 10. 1990. pp. 41-51.

Garland, Michael and Paul Heckbert. Surface Simplification using Quadric Error Bounds. *Proceedings of SIGGRAPH 97.* pp. 209-216.

Gieng, Tran S., Bernd Hamann, Kenneth I. Joy, Gregory L. Schlussmann, and Isaac J. Trotts. Smooth Hierarchical Surface Triangulations. *Proceedings of IEEE Visualization '97.* pp. 379-386.

Guéziec, André. Surface Simplification with Variable Tolerance. *Proceedings of Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery (MRCAS '95).* pp. 132-139.

Hamann, Bernd. A Data Reduction Scheme for Triangulated Surfaces. *Computer Aided Geometric Design.* vol. 11. 1994. pp. 197-214.

Hanrahan, Pat and Jim Lawson. A Language for Shading and Lighting Calculations. *Proceedings of SIGGRAPH 90.* pp. 289--298.

He, Taosong, Lichan Hong, Amitabh Varshney, and Sidney Wang. Controlled Topology Simplification. *IEEE Transactions on Visualization and Computer Graphics.* vol. 2(2). 1996. pp. 171-814.

Heckbert, Paul and Michael Garland. Survey of Polygonal Simplification Algorithms. *SIGGRAPH 97 Course Notes.*1997.

Hoffmann, Christoff M. *Geometric and Solid Modeling.* in B. A. Barsky, ed. *The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling.* Morgan Kaufmann, San Mateo, California. 1989. 338 pages.

Hoppe, Hugues. Progressive Meshes. *Proceedings of SIGGRAPH 96.* pp. 99-108.

Hoppe, Hugues. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH 97.* pp. 189-198.

Hoppe, Hugues, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. *Proceedings of SIGGRAPH 93.* pp. 19-26.

Hughes, Merlin., Anselmo Lastra, and Eddie Saxe. Simplification of Global-Illumination Meshes. *Proceedings of Eurographics '96, Computer Graphics Forum.* pp. 339-345.

Kajiya, Jim. Anisotropic Reflection Models. *Proceedings of SIGGRAPH 85.* pp. 15-21.

Kalvin, Alan D. and Russell H. Taylor. Superfaces: Polygonal Mesh Simplification with Bounded Error. *IEEE Computer Graphics and Applications.* vol. 16(3). 1996. pp. 64-77.

Kent, James R., Wayne E. Carlson, and Richard E. Parent. Shape Transformation for Polyhedral Objects. *Proceedings of SIGGRAPH 92.* pp. 47-54.

Klein, Reinhard. Multiresolution Representations for Surface Meshes Based on the Vertex Decimation Method. *Computers and Graphics.* vol. 22(1). 1998. pp. 13-26.

Klein, Reinhard and J. Krämer. Multiresolution Representations for Surface Meshes. *Proceedings of Spring Conference on Computer Graphics 1997.* June 5-8. pp. 57-66.

Klein, Reinhard, Gunther Liebich, and Wolfgang Straßer. Mesh Reduction with Error Control. *Proceedings of IEEE Visualization '96.*

Koenderink, Jan J. *Solid Shape.* in J. M. Brady, D. G. Bobrow and R. Davis, eds. *Series in Artificial Intelligence.* MIT Press, Cambridge, MA. 1989. 699 pages.

Kolman, B. and R. Beck. *Elementary Linear Programming with Applications.* Academic Press, New York. 1980.

Krishnamurthy, Venkat and Marc Levoy. Fitting Smooth Surfaces to Dense Polygon Meshes. *Proceedings of SIGGRAPH 96.* pp. 313-324.

Lee, Aaron W. F., Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution Adaptive Parameterization of Surfaces. *Proceedings of SIGGRAPH 98.* pp. 95-104.

Luebke, David and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of SIGGRAPH 97.* pp. 199-208.

Lyon, Richard F. Phong Shading Reformulation for Hardware Renderer Simplification. Technical Report #43. Apple Computer. 1993.

Maciel, Paulo W. C. and Peter Shirley. Visual Navigation of Large Environments using Textured Clusters. *Proceedings of 1995 Symposium on Interactive 3D Graphics.* pp. 95-102.

Maillot, Jérôme, Hussein Yahia, and Anne Veroust. Interactive Texture Mapping. *Proceedings of SIGGRAPH 93.* pp. 27-34.

Mitchell, Joseph S. B. and Subhash Suri. Separation and Approximation of Polyhedral Surfaces. *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms.* 1992. pp. 296-306.

Murali, T. M. and Thomas A. Funkhouser. Consistent Solid and Boundary Representations from Arbitrary Polygonal Data. *Proceedings of 1997 Symposium on Interactive 3D Graphics.* April 27-30. pp. 155-162, 196.

Olano, Marc and Anselmo Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. *Proceedings of SIGGRAPH 98.* July 19-24. pp. 159-168.

O'Neill, Barrett. *Elementary Differential Geometry.* Academic Press, New York, NY. 1966. 411 pages.

O'Rourke, Joseph. *Computational Geometry in C.* Cambridge University Press 1994. 357 pages.

Pedersen, Hans. A Framework for Interactive Texturing Operations on Curved Surfaces. *Proceedings of SIGGRAPH 96.* pp. 295-302.

Peercy, Mark, John Airey, and Brian Cabral. Efficient Bump Mapping Hardware. *Proceedings of SIGGRAPH 97.* pp. 303-306.

Plouffe, Simon and Neil James Alexander Sloan. The Encyclopedia of Integer Sequences. Academic Press 1995. pp. 587.

Rohlf, John and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real--Time 3D Graphics. *Proceedings of SIGGRAPH 94.* July 24-29. pp. 381-395.

Ronfard, Remi and Jarek Rossignac. Full-range Approximation of Triangulated Polyhedra. *Computer Graphics Forum.* vol. 15(3). 1996. pp. 67-76 and 462.

Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering. *Modeling in Computer Graphics.* Springer-Verlag1993. pp. 455-465.

Rossignac, Jarek and Paul Borrel. Multi-Resolution 3D Approximations for Rendering Complex Scenes. Technical Report RC 17687-77951. IBM Research Division, T. J. Watson Research Center. Yorktown Heights, NY 10958. 1992.

Schikore, Daniel and Chandrajit Bajaj. Decimation of 2D Scalar Data with Error Control. Technical Report CSD-TR-95-004. Department of Computer Science, Purdue University. 1995.

Schroeder, William J., Jonathan A. Zarge, and William E. Lorensen. Decimation of Triangle Meshes. *Proceedings of SIGGRAPH 92.* pp. 65-70.

Seidel, R. Linear Programming and Convex Hulls Made Easy. *Proceedings of 6th Annual ACM Symposium on Computational Geometry.* 1990. pp. 211-215.

Shade, Jonathan, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. *Proceedings of SIGGRAPH 96.* pp. 75-82.

Turk, Greg. Re-tiling Polygonal Surfaces. *Proceedings of SIGGRAPH 92.* pp. 55-64.

Upstill, Steve. *The RenderMan Companion.* Addison-Wesley 1989.

van Dam, Andries. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics.* vol. 22(3). 1988. pp. 125-218.

Varshney, Amitabh. Hierarchical Geometric Approximations. Ph.D. Thesis. Department of Computer Science. University of North Carolina at Chapel Hill. 1994.

Westin, Steven H., James R. Arvo, and Kenneth E. Torrance. Predicting Reflectance Functions from Complex Surfaces. *Proceedings of SIGGRAPH 92.* pp. 255-264.

Williams, Lance. Pyramidal Parametrics. *Proceedings of SIGGRAPH 83.* pp. 1-11.

Xia, Julie C., Jihad El-Sana, and Amitabh Varshney. Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models. *IEEE Transactions on Visualization and Computer Graphics.* vol. 3(2). 1997. pp. 171-183.a