Quadric-Based Polygonal Surface Simplification

Michael Garland

May 9, 1999
CMU-CS-99-105

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Paul Heckbert, Chair
Andrew Witkin
Martial Hebert
Jarek Rossignac, Georgia Institute of Technology

This research was supported in part by the National Science Foundation (grants CCR-9357763, CCR-9619853, & CCR-9505472) and by the Schlumberger Foundation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of these organizations.

Keywords: surface simplification, multiresolution modeling, level of detail, edge contraction, quadric error metric

Abstract

Many applications in computer graphics and related fields can benefit from automatic simplification of complex polygonal surface models. Applications are often confronted with either very densely over-sampled surfaces or models too complex for the limited available hardware capacity. An effective algorithm for rapidly producing high-quality approximations of the original model is a valuable tool for managing data complexity.

In this dissertation, I present my simplification algorithm, based on iterative vertex pair contraction. This technique provides an effective compromise between the fastest algorithms, which often produce poor quality results, and the highest-quality algorithms, which are generally very slow. For example, a 1000 face approximation of a 100,000 face model can be produced in about 10 seconds on a PentiumPro 200. The algorithm can simplify both the geometry and topology of manifold as well as non-manifold surfaces. In addition to producing single approximations, my algorithm can also be used to generate multiresolution representations such as progressive meshes and vertex hierarchies for view-dependent refinement.

The foundation of my simplification algorithm, is the quadric error metric which I have developed. It provides a useful and economical characterization of local surface shape, and I have proven a direct mathematical connection between the quadric metric and surface curvature. A generalized form of this metric can accommodate surfaces with material properties, such as RGB color or texture coordinates.

I have also developed a closely related technique for constructing a hierarchy of well-defined surface regions composed of disjoint sets of faces. This algorithm involves applying a dual form of my simplification algorithm to the dual graph of the input surface. The resulting structure is a hierarchy of face clusters which is an effective multiresolution representation for applications such as radiosity.

Acknowledgements

To begin at the beginning, I am deeply indebted to my parents, Jeff and Carol Garland. Where I am today is in no small part due to their encouragement and support.

I also owe a lot to my advisor, Paul Heckbert. His assistance throughout my years of study here has been invaluable. He was willingly subjected to several early, very rough, versions of this dissertation, and his many comments and suggestions have undoubtedly improved things. My thanks as well to the rest of my thesis committee, Jarek Rossignac, Martial Hebert, and Andy Witkin, for all their help.

I would also like to thank all those who have contributed to this work by providing both data and results. Tim Vadnais of Iris Development provided the dental model. The scanned face model is from Jon Webb of Visual Interface. Both the Buddha statue and the bunny model are publicly distributed by the Stanford Graphics Lab. The dragon model is distributed by Viewpoint DataLabs. The turbine blade and heat exchanger are from the Visualization Toolkit (VTK) distributed by GE and KitWare. Peter Lindstrom provided a copy of the turbine blade in the appropriate file format. Joel Welling arranged access to a machine with enough memory to simplify the very largest sample models. Roberto Scopigno gave his permission to reuse the performance data which forms the basis for Figure 6.16. Andrew Willmott developed the face cluster radiosity algorithm. He provided the performance results in Chapter 8 and the dragon radiosity solution.

Finally, I must express my appreciation to my wife, Michelle. She has been a source of constant support throughout the many long days spent working on this dissertation. For this, and many other things, I am profoundly grateful.

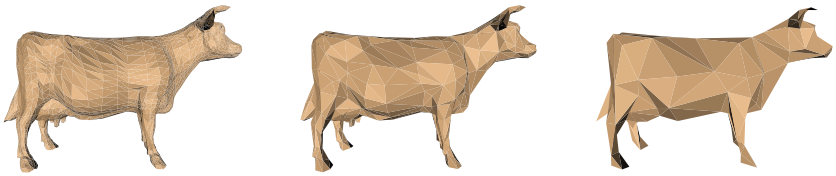
Contents

Chapter 1

Introduction

Numerous applications in computer graphics and related fields rely on polygonal surface models for both simulation and display. Traditionally, such models have been fixed sets of polygons, providing a single level of detail. However, this single level of detail may often be ill-suited for the diverse contexts in which a model of this type will be used.

The central focus of this work is the *automatic simplification* of highly detailed polygonal surface models into faithful approximations containing fewer polygons (see Figure 1.1). I will describe the simplification algorithm which I



(a) Original (b) 83% fewer faces (c) 95% fewer faces

Figure 1.1: Polygonal model with automatically produced approximations.

have developed, and I will examine the hierarchical structure that is induced on the surface as a result of simplification. This resulting hierarchy can be used as a *multiresolution model* — a surface representation which supports the reconstruction of a wide range of approximations to the original surface model. Some authors use the term "multiresolution" to refer specifically to wavelet-based representations. I use the term in a broader sense; wavelet representations are only one particular kind of multiresolution model.

1.1 Motivation

Advances in technology have provided vast databases of polygonal surface models, one of the most common surface representations used in practice. But these models are often very complex; surfaces containing millions of polygons are not uncommon. Laser range scanners, computer vision systems, and medical imaging devices can produce models of intricate physical objects. Many companies now design products using computer-aided design (CAD) systems, resulting in very complex, highly detailed surfaces. Models produced by surface reconstruction and isosurface extraction methods can often be very densely sampled meshes with a uniform distribution of points on the surface. Applications in areas ranging from distributed virtual environments to finite element methods to movie special effects rely on polygonal surface models generated by these kinds of systems.

In all these applications, a tradeoff exists between the accuracy with which a surface is modeled and the amount of time required to process it. To achieve acceptable running times, we must often substitute simpler approximations of the original model. A model which captures very fine surface detail may in fact be desirable when creating archival datasets; it helps ensure that applications which later process the model have sufficient and accurate data. However, many applications will require far less detail than is present in the full dataset. Surface simplification is a valuable tool for tailoring large datasets to the needs of individual applications and for producing more economical surface models. Consider the model shown in Figure 1.2. The original surface, containing a little

(a) 1 million faces (b) 20,000 faces (c) 1000 faces

Figure 1.2: A scanned Buddha figurine with two approximations.

over one million triangular faces, is very densely over-sampled. By comparison, approximation (b) contains 98% fewer triangles, but retains all the basic fea-

tures of the model. For many applications, including interactive rendering, this approximation would be a suitable replacement. Approximation (c) contains a mere 1000 faces. While most of the fine detail of the surface is gone, the overall structure remains. An application trying to measure some gross property of the surface, say volume, could arrive at a reasonable estimate from this very simple model.

Level of detail control is also very important in real-time rendering systems. For any given system, available hardware capacity — such as frame buffer fill rates, transformation and lighting throughput, and network bandwidth — is essentially fixed. But the complexity of the scene to render may vary considerably. In order to maintain a constant frame rate, of say 30 Hz, we need to keep the level of detail in the scene from exceeding the available hardware capacity. This need arises at the low end, where computer games and distributed virtual environments must often operate on systems where available resources are highly constrained. But it arises at the high end as well, where realistic simulation and scientific visualization systems typically have object databases that far exceed the capacity of even the most powerful graphics workstations.

In order to manage the level of detail of an object, we need multiresolution model representations which allow the surface to adapt at run time. To be effective, these multiresolution representations must support the reconstruction of a wide range of levels of detail to accommodate a wide range of viewing contexts. As an example, consider a surface model such as the one shown in

(a) Close-up of fangs (b) Normal view (c) From far away

Figure 1.3: Three very different views of a dragon model.

Figure 1.3b, containing about 100,000 triangular faces. Suppose the viewer is closely examining the surface as in Figure 1.3a; the screen is filled by a small portion of the total surface. Under these conditions, the area being examined may well have too few triangles while the rest of the model, which falls beyond the field of view, can be ignored. Now consider a view like that in Figure 1.3c; the model appears as a few small dots. In this case, the model has far too many polygons for the number of pixels being rendered. Not only must a multiresolution model allow us to extract approximations suitable for these three diverse circumstances, but it must also allow us to change the level of

detail without excessive overhead. If the time necessary to switch to and render a lower level of detail exceeds the time necessary to simply render a higher level of detail, we would gain no advantage from the multiresolution model.

1.2 Context of Prior Work

The general idea of multiresolution modeling is not new; suggestions for such a framework arose twenty years ago. Multiresolution models for large-scale terrain surfaces appeared in flight simulators at around the same time, and new adaptive terrain techniques continue to be developed. In contrast, multiresolution representations for general polygonal surfaces have only appeared fairly recently. Early methods were based on discrete levels of detail; they maintained a fixed set of static approximations. In the last couple of years, more adaptive methods have been developed which can reconstruct a wider range of approximations. Closely related work has explored surface representations which allow for efficient compression and progressive network transmission of surface models. In all three cases, polygonal surface simplification is the underlying mechanism used to construct multiresolution and progressive representations.

Effective algorithms for simplifying curves and height fields date back two decades. Indeed, curves are sufficiently simple objects that feasible algorithms exist for constructing optimal approximations. In the last eight years, polygonal surface simplification has received much greater attention. As we will see in the next chapter, a fairly broad range of algorithms have been developed. At one end of the spectrum are very fast algorithms which can often produce rather poor results. At the other end are very high quality algorithms which are also very slow. In between are several algorithms of varying efficiency and quality.

My goal has been to produce a fast simplification algorithm that produces high quality results.

1.3 Contributions

The primary contributions of my work as described in this dissertation are:

- **Quadric Error Metric.** I have developed an error metric which provides a useful characterization of local surface shape. It requires only modest storage space and computation time. Through a simple extension, it can be used on surfaces for which each vertex has an associated set of material properties, such as RGB color and texture coordinates. I have also proven a direct connection between the quadric error metric and surface curvature.

- **Surface Simplification Algorithm.** By combining the quadric error metric with iterative vertex pair contraction, I have developed a fast algorithm for producing high-quality approximations of polygonal surfaces. This algorithm can simplify both manifold and non-manifold models. It

is also capable of joining unconnected regions of the model together, thus ultimately simplifying the surface topology by aggregating separate components. In addition to producing single approximations, my algorithm can also be used to generate multiresolution representations such as progressive meshes and vertex hierarchies for view-dependent refinement.

- **Face-Hierarchy Construction.** Finally, I have developed a technique for constructing a hierarchy of well-defined surface regions composed of disjoint sets of faces. This algorithm involves applying a dual form of my simplification algorithm to the dual graph of the input surface. The resulting structure is a hierarchy of face clusters which is an effective multiresolution representation for certain applications, including radiosity.

1.4 Overview of Material

I will begin with a more detailed discussion of surface simplification and a review of the prior work in the field (Chapter 2). Having established this background information, I present the details of the quadric error metric and the surface simplification algorithm built around it in Chapter 3. The quadric error metric itself has several useful interpretations, particularly in connection with surface curvature, and I present these in Chapter 4. The algorithm developed in Chapter 3 considers surface geometry alone. In Chapter 5, I discuss the extension of the quadric error metric to encompass surfaces with material properties (e.g., color and texture). Chapter 6 contains a performance analysis of my simplification algorithm, and Chapter 7 is an overview of possible applications of the simplification algorithm developed in earlier chapters. Finally, I present my hierarchical face clustering algorithm, the dual of my surface simplification algorithm, in Chapter 8. To highlight certain design choices and techniques, I have included Appendix A, which contains details on my implementation of my simplification algorithm.

Chapter 2

Background & Related Work

This chapter provides an overview of background material used throughout the rest of this dissertation. Having provided a definition of polygonal models, I will examine the need for simplification techniques in managing the complexity of surface models. I will then describe the criteria upon which we can judge the quality of an approximation, and review the methods which others have developed for producing approximations.

By convention, all vectors in this text are assumed to be column vectors and are set in lowercase bold type. Therefore, $\mathbf{u}^\mathsf{T}\mathbf{v} = \mathbf{u}\cdot\mathbf{v}$ denotes the inner product of two column vectors $\mathbf{u}$ and $\mathbf{v}$. Matrices are set in uppercase bold type; thus $\mathbf{A} = \mathbf{u}\mathbf{v}^\mathsf{T}$ denotes the outer product matrix $a_{ij} = u_i v_j$.

2.1 Surface Representation

In the most general sense, a polygonal surface model is simply a set of planar polygons in the three-dimensional Euclidean space $\mathbf{R}^3$. Without loss of generality, we can assume that the model consists entirely of triangular faces[1], since any non-triangular polygons may be triangulated in a pre-processing phase [174, 139]. A model might conceivably contain isolated vertices and edges which are not part of any triangle. For best results in practice, we should maintain them during simplification and render them at run-time [128, 144, 161, 167]. However, to streamline the discussion, I will assume that models consist of triangles only. For most algorithms, my own included, the only effect of isolated vertices and edges is to complicate the implementation; the underlying algorithms remain the same. Finally, I will also assume that the connectivity of the model is *consistent* with its geometry — if the corners of two triangles coincide in space then those triangles share a common vertex.

[1] Only a very few simplification methods relax this assumption [75, 154].

Given these assumptions, I make the following definition: a polygonal surface model $M = (V, F)$ is a pair containing a list of vertices V and a list of triangular faces F. The vertex list $V = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_r)$ is an ordered sequence; each vertex may be identified by a unique integer i. The face list $F = (f_1, f_2, \ldots, f_n)$ is also ordered, assigning a unique integer to each face. Every vertex $\mathbf{v}_i = [x_i\ y_i\ z_i]^\mathsf{T}$ is a column vector in the Euclidean space $\mathbf{R}^3$. Each triangle $f_i = (j, k, l)$ is an ordered list of three indices identifying the corners $(\mathbf{v}_j, \mathbf{v}_k, \mathbf{v}_l)$ of f_i.

By design, this definition of a polygonal model corresponds to a form of simplicial complex [3]. For our purposes here, a *simplex* σ is either a vertex (or 0-simplex), a line segment (1-simplex), or a triangle (2-simplex). In general, a k-simplex σ^k is the smallest closed convex set[2] defined by $k+1$ linearly independent points $\sigma^k = a_0 a_1 \ldots a_k$ which are called its vertices. We can express any point p within this set as a convex combination of the vertices $p = \sum_i t_i a_i$ where $\sum_i t_i = 1$ and $t_i \in [0, 1]$. Any simplex defined by a subset of the points $a_0 a_1 \ldots a_k$ is a *subsimplex* of the simplex σ^k. A two-dimensional *simplicial complex* K is a collection of vertices, edges, and triangles satisfying the conditions:

1. If $\sigma_i, \sigma_j \in K$, then they are either disjoint or intersect only at a common subsimplex. Specifically, two edges can only intersect at a common vertex, and two faces can only intersect at a shared edge or vertex.

2. If $\sigma_i \in K$, then all of its subsimplices are in K. For instance, if a triangle f is in K, then its vertices and edges must also be in K.

The surface defined by this complex is the union of the point sets defined by its constituent simplices. While any set of vertices, edges, and faces satisfying these conditions can be considered a two-dimensional complex, our definition of a polygonal model is slightly different. It is only explicitly a collection of vertices and faces. The only allowable edges are those which are implied by the intersection of neighboring faces. The additional assumption that the model does not contain any isolated vertices or edges implies that the model is a *pure* complex.

2.1.1 Manifold & Non-Manifold Surfaces

Surfaces, in the mathematical sense, are often assumed to be manifolds. As Henle [87] aptly points out, the intuitive concept of a manifold surface is that people living on it, their perception limited to a small surrounding area, are "unable to distinguish their situation from that of people actually living on a plane." In other words, it fits our perception of the surface of the Earth. More formally, a *manifold* is a surface, all of whose points have a neighborhood which is topologically equivalent to a disk. A *manifold with boundary* is a surface all of whose points have a neighborhood which is topologically equivalent to either a disk or a half-disk. A polygonal surface is a manifold (with boundary) if every edge has exactly two incident[3] faces (except edges on the boundary which must

[2] In other words, the convex hull [147].

[3] The incident faces of an edge e are all the 2-simplices f_i of which e is a subsimplex.

have exactly one), and the neighborhood of every vertex consists of a closed loop of faces (or a single fan of faces on the boundary). Figure 2.1 illustrates four kinds of vertex neighborhoods in a polygonal model.

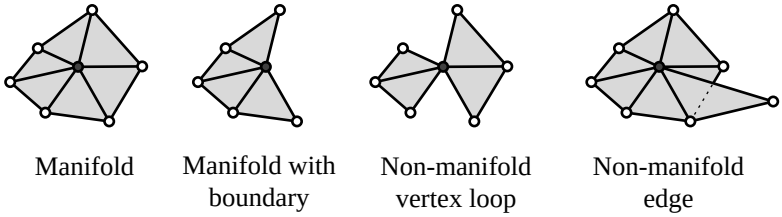


Figure 2.1: Neighborhoods of a given vertex (in black). On the left, two manifold neighborhoods. On the right, two non-manifold neighborhoods.

Many surfaces encountered in practice tend to be manifolds, and many surface-based algorithms require manifold input. It is possible to apply such algorithms to non-manifold surfaces by cutting the surface into manifold components and subsequently stitching them back together [78]. However, it can be advantageous for simplification algorithms to explicitly allow non-manifold surfaces. Not only does this broaden the class of permissible input models, but it provides more flexibility during simplification. Many simplification algorithms proceed by repeatedly making local simplifications to the model. These local transformations can easily result in non-manifold regions. Consider the example shown in Figure 2.2. The same local simplification, namely edge con-

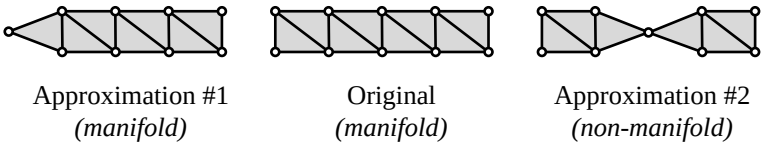


Figure 2.2: Two approximations of the same surface, both constructed by contracting a single edge.

traction, is applied in two different ways. Depending on the choice of edge, contraction may result in either a manifold or non-manifold result. By allowing non-manifold surfaces, we allow the simplification algorithm to select the better choice based on criteria such as geometric fidelity rather than artificially limiting it to only apply operations which produce manifold surfaces. This issue is of particular relevance in algorithms which seek to simplify the topology of the model. Imagine a model of a metal plate with many small holes drilled in it. The common contraction-based approach for removing a hole from this model would begin by collapsing one end of the hold into a single point, resulting in a non-manifold vertex neighborhood. While it is possible to explicitly cut

and re-stitch the surface during simplification [171], this can add substantial complexity to the algorithm.

2.1.2 Non-Polygonal Representations

The aim of polygonal surface simplification is to provide a mechanism for controlling the complexity of polygonal surface models, but these are not the only available surface representation. Various alternatives exist, and they each provide certain benefits and drawbacks as compared with polygonal models. However, none of these alternatives provide a solution which would obviate the need for simplification. In fact, they suffer from some of the same problems addressed by polygonal surface simplification.

The most important reason to focus on polygonal models is purely pragmatic: polygonal models are both flexible and ubiquitous. They are supported by the vast majority of rendering and modeling packages, and polygonal surface data is widely available. Hardware acceleration of polygon rendering is also becoming much more widely available; affordable yet reasonably powerful accelerator cards are now available in consumer-level computers. Currently, no other single type of model enjoys the same level of support. In fact, it is common practice in various situations to convert other model types into polygonal surfaces prior to processing.

Parametric spline surfaces are probably the most widely used alternative to polygonal models. For the most part, we can view them as a generalization of polygonal models. They are composed of piecewise-polynomial, rather than piecewise-linear, patches. By using higher-order polynomials, we can approximate smooth surfaces much more accurately and compactly than with planar polygons. Just as with polygonal models, a model composed of piecewise-polynomial elements discretizes the model into a fixed set of patches. Consequently, such models share the problem of static polygonal models: we are presented with a fixed set of surface elements which may or may not be appropriate for the task at hand. This has led to the development of more adaptive spline methods such as subdivision surfaces [18, 47, 125, 170] and hierarchical splines [60]. There has also been some work done on adaptively fitting [169] spline patch models and decimating [72] models composed of triangular Bézier patches [55].

Volumetric models, and voxel grids in particular, are a common representation in scientific visualization. This representation is quite effective if the model data is acquired as a volume and will be used directly as a volume. The resolution of voxel grids can easily be controlled via traditional image processing techniques [160, 164]. However, if the model is not being used as a volume but as a surface, it must be repolygonized. This is likely to produce an overly complex model because most isosurface extraction algorithms generate uniformly sampled meshes. In fact, the complexity of volume polygonizations was a motivating factor behind several proposed surface simplification algorithms. There has also been some work on simplifying tetrahedral volume meshes [24, 180, 186] that closely parallels work on polygonal surface simplification.

For rendering applications, image-based models are an attractive option. Representations such as light fields [73, 121] and panoramas [184] are quite effective at reproducing real-world scenes. Image-based modeling is also a powerful technique when used in tandem with traditional geometric models. In particular, cached images of rendered geometry can be used to decrease load in real-time rendering systems [177, 175, 6, 151, 131, 168]. Some image-based techniques use textured depth images and apply polygonal surface simplification to achieve compact representations of the depth map [153].

2.2 Simplification and Multiresolution Models

Traditional polygonal models are composed of a fixed set of vertices and a fixed set of faces (§2.1). Therefore, they provide a single fixed resolution representation of an object. But this single resolution may not be appropriate for all the contexts in which the model will be used. Consider the three views shown in Figure 2.4. A polygonal model of a dragon, containing 108,588 faces, is shown flat-shaded in each view. A detailed view of the surface mesh is shown in Figure 2.3. This model contains a reasonable level of detail for view (b), although a slightly simpler model might do just as well. In view (a), greater detail around the fangs might be desirable, while the rest of the surface can be safely ignored. Finally, in view (c), where the model is at a substantial distance from the viewer, we could probably achieve identical results with fewer than 100 triangles.

Suppose we have a polygonal model M and we would like an approximation M'. While this approximation will have fewer polygons than the original, it should also be as similar as possible to M (see Section 2.3 for details on measuring similarity). The goal of polygonal *surface simplification* is to automatically produce such approximations. User supervision is generally not feasible. Simplification is naturally targeted towards large and complex datasets which would be very cumbersome to manipulate manually.

A common application of simplification is reducing the complexity of very densely over-sampled models. Models generated by scanning devices and isosurfaces extracted by algorithms such as marching cubes [126] often benefit from simplification. Such models are often uniformly tessellated (see Figure 2.3) — an artifact of the nature of most reconstruction algorithms. Triangle density is the same in both flat and highly curved regions. It is usually preferable to be more economical with triangle coverage; local triangle density should adapt to local curvature. The number of triangles can often be reduced by 50 percent or more, and the result will be nearly identical to the original.

More generally, we may want to produce an approximation which is tailored for a specific use. For instance, we might want to produce an approximation of the dragon model in Figure 2.4 suitable for viewing conditions such as depicted in Figure 2.4c. Consider an analogous situation in image processing. Suppose we have a raster image scanned at a resolution of 300 dots per inch, but we plan to display it on a monitor with a resolution of 72 dots per inch. We can resample the original image at the lower resolution and produce a much smaller

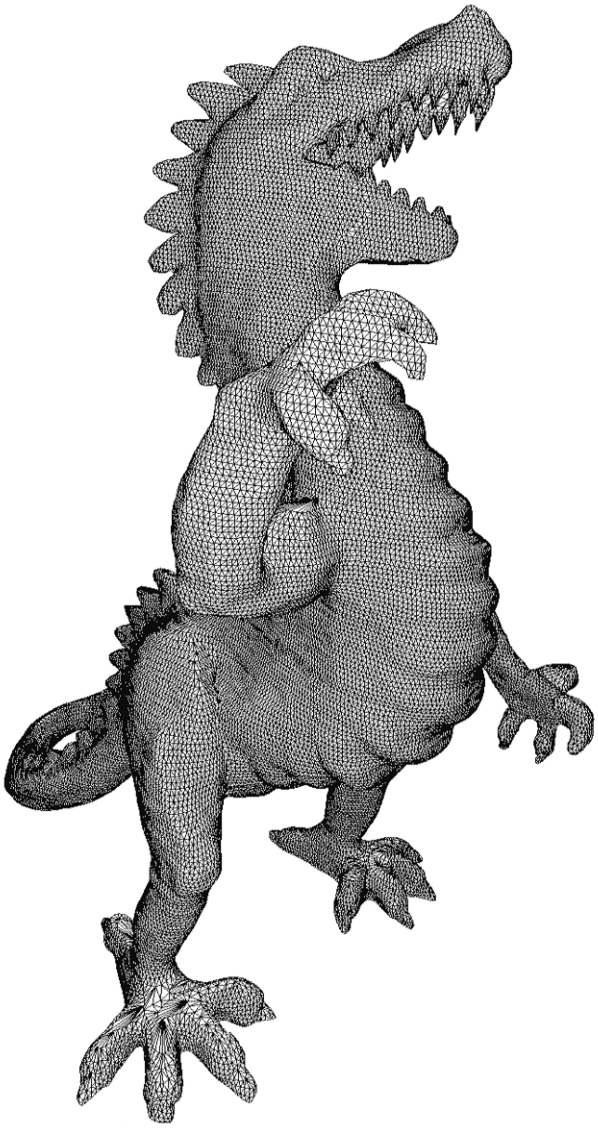
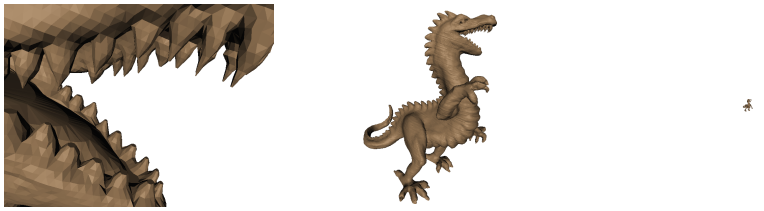


Figure 2.3: Original polygonal surface model of a dragon. This unsimplified model contains 54,296 vertices and 108,588 triangles. Except for parts of the feet, the surface is tessellated with uniformly sized triangles.



(a) Close inspection (b) Normal viewing (c) Far in distance

Figure 2.4: The same model used in widely differing contexts.

representation of the image. If we resample the image well, the reduced image should be nearly indistinguishable from the original at the output resolution of 72 dots per inch.

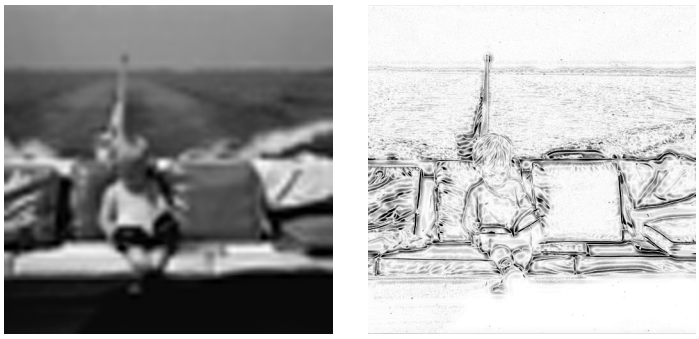
In other cases, we would like something more flexible than a single fixed approximation. Suppose that, during an interactive session, a user was viewing the model shown in Figure 2.4 in the diverse contexts shown. There is no single set of polygons which is appropriate for all of these different viewing conditions. Instead, we would like to have multiple different approximations available, selecting the best one for the current viewing conditions. Rather than a fixed resolution model, we would like a multiresolution model.

A *multiresolution model* is a model representation which captures a wide range of approximations of an object and which can be used to reconstruct any one of these on demand. The cost of reconstructing approximations should be low because we will often need to use many different approximations at run time. It is also important that a multiresolution representation have roughly the same size as the most detailed approximation alone, although a small constant factor increase in size is acceptable For the moment, I will focus on the use of multiresolution models to control running times in real-time rendering systems. Thus, the appropriate surface approximation for a particular model will depend upon current viewing conditions (e.g., distance to the viewer). As we will see, the appropriate level of detail may also vary considerably over the surface.

Image pyramids provide an important motivating example. They are a successful and fairly easy to use multiresolution representation of raster images [160] and are widely used as an optimization technique in texture mapping [194]. The key idea of an image pyramid is to store, along with the original image, a series of downsampled images as well. Figure 2.5 shows a simple example. In addition to the original image, of 512×512 pixels, we store versions reduced by increasing powers of 2. For the price of a $1/3$ increase in size over the original image, we can efficiently produce resampled versions at a wide range of resolutions.

An alternative type of pyramid, the Laplacian pyramid [17], actually has more multiresolution characteristics. This type of pyramid stores a simple base

Figure 2.5: Four level image pyramid ranging from 512×512 to 64×64 pixels.



(a) 64×64 image (b) Difference with 512×512

Figure 2.6: Downsampled 64×64 image and difference image.

image plus a sequence of successively larger difference images — a method quite similar to encoding the image using a Haar wavelet basis [182]. As an example, Figure 2.6a shows the 64×64 pixel level of the pyramid expanded to 512×512 pixels. Figure 2.6b shows the difference between this and the original 512×512 base image. As the difference image shows, the significant differences between these two levels of the pyramid are quite sparse. By encoding the image as a simple base image plus a set of difference images, we can achieve a more compact representation [17]. This representation also lends itself very nicely to applications such as multiresolution image editing [142].

I will formalize a multiresolution model as a parameterized family of models $M : \mathcal{C} \rightarrow \mathcal{M}$ where the domain $\mathcal{C}$ is the space of viewing contexts and $\mathcal{M}$ is the space of all models. Thus, $M(\xi) \in \mathcal{M}$ is the model, or level of detail, which is appropriate for the viewing context $\xi \in \mathcal{C}$. In the case of raster images, ξ might be the desired image dimensions (w, h) and $M(w, h)$ would be a suitably filtered image of $w \times h$ pixels.

2.2.1 Discrete Multiresolution

The simplest method for creating multiresolution surface models is to generate a set of increasingly simpler models. For any given frame, a renderer could select which model to use and render that model in the current frame [64]. In this case, we would be using a series of *discrete levels of detail;* our multiresolution model would consist of the set of levels — such as in Figure 2.7 — and the threshold parameters to control the switching between them. This would be analogous to

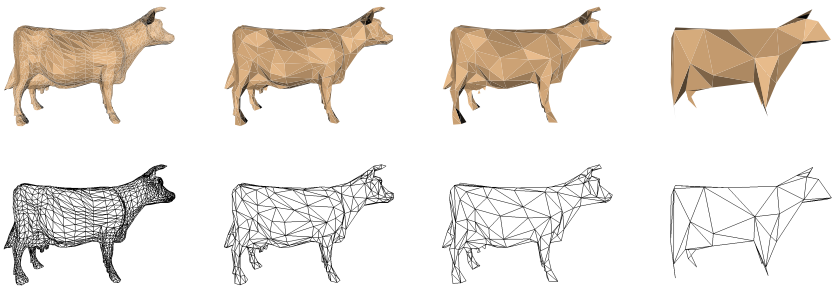


Figure 2.7: Fixed set of levels of detail for a cow model.

building an image pyramid, as in Figure 2.5 and, at any one time, selecting which of the levels to use. The simplicity of the discrete multiresolution approach is its primary attraction. If we can produce good surface approximations, we can produce discrete multiresolution models.

Level of Detail Blending

Simply switching levels of detail between frames by substituting one whole discrete model will often incur negligible overhead at display time. Many systems are designed to transmit all the geometry of the world to the graphics subsystem at each frame. Thus, ignoring external factors such as paging the relevant geometry into main memory, switching levels of detail simply involves transmitting different geometry for the current frame. If the graphics subsystem supports caching several levels of detail in pre-compiled display lists, we might not even have to transmit any new geometry at all. However, it can potentially cause significant visual artifacts. In most cases, the number of polygons in the two models will differ significantly, and this will cause their appearances in the output image to be significantly different as well. Making such a substantial change in appearance between two consecutive frames can lead to "popping" artifacts. This effect can be mitigated by extending the level of detail transition over several frames and using alpha blending to perform a smooth cross-dissolve between the images of the two models [65]. Visual artifacts are still evident, but are much less objectionable. Unfortunately, this technique causes the overall rendering cost to increase during transitions since the system must render two levels of the model at the same time.

Geomorphing

Another alternative is to smoothly interpolate between the geometries of two consecutive levels over several frames. This *geomorphing* technique has been used in line-based [134] and terrain-specific systems for some time [35, 57]. Provided that we have a correspondence between the vertices of successive levels of detail, we can also apply geomorphs to general polygonal surfaces [90]. Suppose that we are transitioning between a model M and a simpler model M' and that every vertex $\mathbf{v} \in V$ corresponds[4] to a vertex $\phi(\mathbf{v}) \in V'$. Iterative contraction algorithms generate exactly this sort of correspondence (§3.2). During the transition, the model will have the same mesh connectivity as the more complex model M, but its geometry will vary continuously between that of M and M'. For each vertex $\mathbf{v}$ in M, we substitute an interpolated position $t\mathbf{v} + (1-t)\phi(\mathbf{v})$. At $t = 0$, the model will have exactly the same shape as M, and at $t = 1$, the model will have the shape of M'. By moving t between 0 and 1 over several successive frames, we can smoothly transition between the two models. Unlike the alpha blending approach, the geometric complexity of the object being rendered is the same as M. While this is less than ideal, because we have determined that the required level of complexity is only that of M', it is certainly lower than the combined size of both levels. However, there is the additional overhead of interpolating the vertex positions for each frame. Whether this is less expensive than blending the images of M and M' may depend on the hardware architecture.

[4] Since V' is smaller than V, this mapping is not one-to-one. There will be at least one pair of vertices $\mathbf{v}_i, \mathbf{v}_j$ such that $\phi(\mathbf{v}_i) = \phi(\mathbf{v}_j)$.

The principal drawback of discrete multiresolution models is that the levels of detail available at run time are rather limited. A renderer would be forced to pick one of our pre-generated models, even if it needed an intermediate level. Thus, the renderer would either have to pick a model without sufficient detail (and sacrifice image quality) or choose a model with excess detail (and waste time). Unless the model is divided into interchangeable blocks, the renderer would also be unable to vary the level of detail over different parts of the model. Suppose, for example, that we are standing near the corner of a building looking down one side. At the corner nearest the viewpoint, the renderer needs a high level of detail to maintain image quality. However, as the walls recede into the distance, the renderer could potentially use less and less detail. If the renderer is forced to use the same level of detail over the whole model, it must again choose to use an insufficient level and sacrifice quality or use an excessive level and waste time.

Despite this limitation, discrete multiresolution models can be quite useful in certain situations. If an object is displayed such that the entire surface is at roughly the same scale, then discrete multiresolution models are an effective means of controlling level of detail. For instance, the discrete method seems to have been effective in the walkthrough system described by Funkhouser and Séquin [64]. Support for discrete levels of detail has also been included in a number of commercial rendering systems, including RenderMan [188], Open Inventor [192], and IRIS Performer [157]. The RenderMan interface provides for "mixing" successive levels of detail together, but leaves the exact mechanism undefined. Performer provides explicit support for both alpha blending and geomorphing. Discrete levels of detail have also been used for accelerating the computation of radiosity solutions [163].

2.2.2 Continuous Multiresolution

As we have just seen, discrete multiresolution models are sufficient in some circumstances, but there are other cases in which they are inadequate. A large surface, such as a terrain, being viewed at close range from an oblique angle is particularly problematic. Consider the example shown in Figure 2.8. The

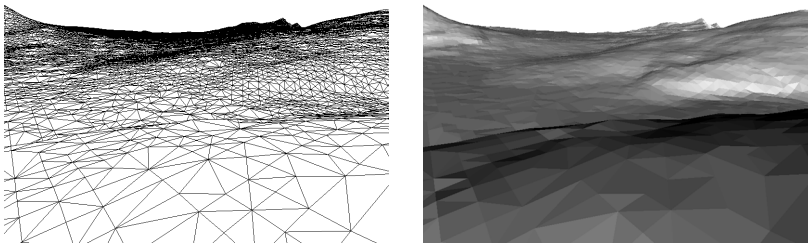


Figure 2.8: Large terrain viewed from near the surface.

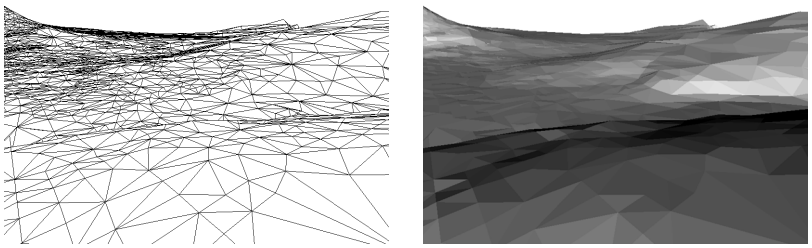


Figure 2.9: Identical view with adaptive tessellation.

viewpoint is positioned just above the surface, looking out towards the horizon. Notice how the screen-space density of the triangulation increases as the surface recedes into the distance. An approximation with a constant level of detail would either be too dense in the distance (as in Figure 2.8) or too sparse near the viewpoint. We would prefer an approximation where the level of detail is allowed to vary *continuously* over the surface. In particular, we would like the level of detail of a particular neighborhood to be *view dependent*. Figure 2.9 demonstrates the results. While the approximation shown in Figure 2.8 contains many distant triangles whose projected screen size is minute, the approximation shown in Figure 2.9 uses a much lower level of detail for distant surface regions. The result is an approximation which is specifically tailored to the current viewpoint. Thus, we are looking for a multiresolution representation that continuously adapts the surface at run time based on viewing conditions. Run-time adaptation can be combined with the geomorphing technique described earlier to produce smooth transitions.

The need for adaptive level of detail control is particularly pronounced in the case of terrains, and continuous multiresolution models have been in use by flight simulator systems for twenty years [35]. Several effective adaptive terrain techniques are available [49, 123]. Most are based on a regular subdivision (e.g., quadtrees) of the terrain surface. Using regular subdivisions helps to minimize the run-time overhead incurred by maintaining an adaptive level of detail.

There has been comparatively less work on continuous multiresolution representations for general triangulated surfaces. Multi-triangulations [42, 41, 149] provide a fairly general framework that can describe most commonly used multiresolution representations. Vertex hierarchies [91, 93, 130, 198] are a particular multiresolution representation that have received considerable attention. An important feature of vertex hierarchies is that they can be constructed as a by-product of standard surface simplification. While the associated overhead is often acceptable, it is certainly higher than that of discrete multiresolution models. For instance, Hoppe [91] reports that model adaptation consumed 14% of the frame time for his implementation. I will discuss vertex hierarchies and their construction in greater detail in Chapter 7.

Depending on the application at hand, discrete or continuous multiresolution

models may be more appropriate. Discrete methods are simpler and require less overhead. Continuous methods are more flexible but have higher overhead. This flexibility is important for models such as terrains, but may not be necessary for objects such as chairs in a room. Indeed, the best solution is to support different multiresolution representations which are tailored to different classes of model. Despite their differences, both types of multiresolution model share one significant characteristic: they can be constructed using surface simplification methods.

2.3 Evaluating Surface Approximations

As stated earlier, the primary aim of simplification is to produce a surface approximation which is as similar as possible to the original. In order to assess the quality of an approximation, we need some means of quantifying the notion of similarity. Suppose that we are given a polygonal model M and an approximation M'. We would like an error metric $E : \mathcal{M} \times \mathcal{M} \rightarrow \mathbf{R}$ for which the value $E(M, M')$ measures the *approximation error* of M'. The lower the error value assigned by E to M', the greater its similarity to the original model M.

While the preferred criteria are application-dependent, similarity of appearance is the natural choice for rendering applications. However, in almost all cases, researchers in the field of simplification have chosen to use similarity of shape as the primary criterion for evaluating approximation quality. Not only do shape-based metrics appear to be more computationally convenient, but they are also more appropriate in non-rendering applications such as finite element analysis.

2.3.1 Similarity of Appearance

Rendering systems are one of the primary application areas of interest in this work. For such systems, similarity of appearance should be the ultimate criterion for evaluating the quality of an approximation [83]. As mentioned earlier, most simplification methods are based on purely geometric criteria. However, since similarity of appearance is often what we would like to achieve, it is important to consider how we might define it.

The appearance of a model M under viewing conditions ξ is determined by the raster image I^{ξ} which a renderer would produce. We may say that two models M_1 and M_2 appear identical in view ξ if their corresponding images I_1^{ξ} and I_2^{ξ} are identical. If I_1 and I_2 are both $m \times m$ RGB raster images, we can define the difference between them as the average sum of squared differences between all corresponding pixels

$$\|I_1 - I_2\|_{img} = \frac{1}{m^2} \sum_u \sum_v \|I_1(u, v) - I_2(u, v)\|^2 \tag{2.1}$$

where $\|I_1(u, v) - I_2(u, v)\|$ is the Euclidean length of the difference of the two RGB vectors $I_1(u, v)$ and $I_2(u, v)$. While there are many more elaborate metrics

for comparing images [162], this very simple definition appears suitable for the simplification domain. If M_2 is a good approximation of M_1 for the given view ξ, then $\|I_1 - I_2\|_{img}$ should be small. Given this image metric, we can characterize the total difference in appearance between two models by integrating $\|I_1^\xi - I_2^\xi\|_{img}$ over all possible views ξ. Naturally, we would expect in practice to merely sample these per-image differences over some finite set of viewpoints.

A simplification algorithm guided by an appearance-based metric of this type has several interesting characteristics. Its primary advantage is that it directly measures similarity of appearance, which is precisely what we are interested in preserving in rendering systems. It also allows us to discard occluded details. Suppose that we have some probability distribution on the possible viewpoints that will occur at run time. Any features which are occluded in all possible views can be immediately removed. For example, if we have a complex model of a submarine and we know that the viewpoint will always be outside the hull, we can remove all polygons on the interior without introducing any error into the approximation.

While appearance-based metrics have some appealing benefits, they also raise some difficult issues. In particular, the foremost problem is the need to adequately sample the possible viewpoints. If we neglect some important part of the viewpoint space, we may very well remove perceptually significant features. And since each sample may involve an expensive rendering step, we cannot make many samples. Indeed, rendering the models for comparison is likely to be quite expensive; simplification is generally performed on models which are prohibitively expensive to render in the first place.

2.3.2 Geometric Approximation Error

While similarity of appearance is the foremost goal for approximations used in rendering systems, it is generally easier to consider geometric measures of error instead. We can use geometric similarity as a proxy for visual similarity. By striving to produce geometrically faithful results, we can also produce approximations that will be useful in application domains other than rendering.

Function Approximation

Before considering the full problem of measuring approximation error for polygonal models, let us examine a much simpler case: function approximation. This area of study has a long history in the mathematics literature, and it will provide us with some intuition which will carry over into the polygonal domain.

The two most commonly used error metrics are the L_∞ and L_2 norms [146]. Suppose a real-valued function $f(t)$, an approximation $g(t)$, and an interval of interest $[a, b]$ are given. The L_∞ norm, which measures the maximum deviation between the original and the approximation is defined by

$$\|f - g\|_\infty = \max_{a \le t \le b} |f(t) - g(t)| \tag{2.2}$$

The L_2 norm[5] defined by

$$\|f - g\|_2 = \sqrt{\int_a^b (f(t) - g(t))^2 \, dt} \tag{2.3}$$

provides a measure of the average deviation between the two functions. A piecewise-linear approximation $g(t)$ composed of n segments is called *optimal* if there is no other n-segment approximation having a smaller error. The L_∞ norm is generally regarded as a stronger measure of error in the function approximation literature [146]. Because it provides a global absolute bound on the distance between the original and the approximation, it is often easier to prove quality guarantees. However, the L_2 norm is somewhat more general. Certain functions, such as $f(t) = t^{-1/3}$ on the interval $[0, 1]$, have a well-defined L_2 norm but no L_∞ norm.

The L_∞ norm is most useful because it provides absolute distance bounds which are a useful error guarantee. However, it can be overly sensitive to any noise that might be present in the original model. Furthermore, there might also be times when we are willing to allow a few sections of the model to deviate more than some threshold distance from the original to gain a better overall fit. In contrast, the L_2 norm better reflects overall fit, but may discount large, but highly localized, deviations. For example, consider the curves shown in

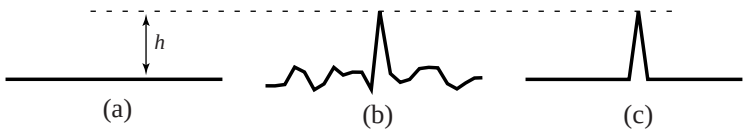


Figure 2.10: Two approximations to the same base curve.

Figure 2.10. The two approximations (b) and (c) have the same L_∞ error, namely the distance h. However, curve (c) certainly seems to be a better overall approximation. The L_2 norm would assign a higher error to curve (b) than curve (c). Now consider the curves shown in Figure 2.11. The base curve (a)

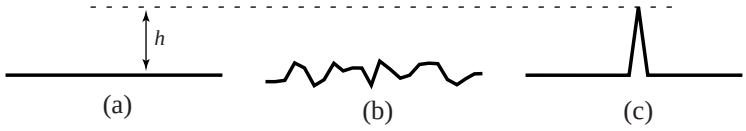


Figure 2.11: Alternative approximations to the same base curve.

and the approximation (c) are the same as before. We can choose the size of the tent in approximation (c) such that both (b) and (c) have the same L_2 error.

[5] When divided by $b - a$, this is also called the "root mean square" or RMS error.

However, there are certainly cases in which (b) is a preferable approximation given that (c) deviates significantly further from the base curve. Also, suppose that we allow the width ϵ of the tent in (c) to approach 0. The L_2 error of (c) will also approach 0, while its L_∞ error will remain h.

Neither of these error metrics is completely ideal. The L_∞ norm provides strong error bounds, but can be overly influenced by noise and local deviations. On the other hand, the L_2 norm provides a better estimate of the overall fit and is more tolerant of noise, but it may discount local deviations. A combination of these two metrics is preferable: we would like approximations with small L_2 error for which the L_∞ error is bounded by some known threshold.

Surface Approximation

We can formulate surface-based analogs of both the L_2 and L_∞ function approximation norms. First, we need to generalize the notion of deviation between the original and the approximation. In the functional case outlined in the previous section, we measured deviation as the vertical distance $|f(x)-g(x)|$. When comparing general surfaces, there is no single distinguished direction along which to measure distances. Instead, we will measure distances between closest pairs of points. If we denote the set of all points on the surface of a model[6] M by $P(M)$, the distance from a point $\mathbf{v}$ to the model M is defined to be the distance to the closest point on the model:

$$d_{\mathbf{v}}(M) = \min_{\mathbf{w} \in P(M)} \|\mathbf{v} - \mathbf{w}\| \tag{2.4}$$

where $\|\cdot\|$ is the usual Euclidean vector length operator.

One commonly used geometric error measure is the Hausdorff distance [147], which corresponds closely to the L_∞ metric. Based on the point-wise distance measure (2.4), we can define the Hausdorff error metric $\hat{E}_{max}(M_1, M_2)$ as

$$\hat{E}_{max}(M_1, M_2) = \max\left(\max_{\mathbf{v} \in P(M_1)} d_{\mathbf{v}}(M_2), \max_{\mathbf{v} \in P(M_2)} d_{\mathbf{v}}(M_1)\right) \tag{2.5}$$

The Hausdorff error measures the maximum deviation between the two models. If $\hat{E}_{max}(M, M')$ is bounded by ϵ, then we know that every point of the approximation is within ϵ of the original surface and that every point of the original is within ϵ of the approximation. We can also define an analog of the L_2 metric, a measure of the average squared distance between the two models.

$$\hat{E}_{avg}(M_1, M_2) = \frac{1}{w_1 + w_2}\left(\int_{P(M_1)} d^2_{\mathbf{v}}(M_2)\, d\mathbf{v} + \int_{P(M_2)} d^2_{\mathbf{v}}(M_1)\, d\mathbf{v}\right) \tag{2.6}$$

where w_1, w_2 are the surface areas of M_1, M_2. I have chosen to normalize the sum of both integrals by the combined surface area; one could also consider

[6] This set, encompassing *all* points on the surface, is infinite and should not be confused with the set of vertices V.

normalizing each individual integral by its own corresponding area. Note the symmetric construction of both $\hat{E}_{max}$ and $\hat{E}_{avg}$. It is not sufficient to simply consider every point on M_1 and find the closest corresponding point on M_2. We must also do the same for every point on M_2.

In practice, these error metrics can be prohibitively expensive to compute exactly. It is common to formulate approximations of these ideal metrics based on sampling the distance $d_{\mathbf{v}}$ at a discrete set of points. Given $P(M_1)$ and $P(M_2)$, we can select two sets $X_1 \subset P(M_1)$ and $X_2 \subset P(M_2)$ containing k_1 and k_2 sample points, respectively. These sets should, at a minimum, contain all the vertices of their respective models. The sampled error metrics, approximations of (2.5) and (2.6), are

$$E_{max}(M_1, M_2) = \max\left(\max_{\mathbf{v}\in X_1} d_{\mathbf{v}}(M_2), \max_{\mathbf{v}\in X_2} d_{\mathbf{v}}(M_1)\right) \tag{2.7}$$

and

$$E_{avg}(M_1, M_2) = \frac{1}{k_1 + k_2}\left(\sum_{\mathbf{v}\in X_1} d^2_{\mathbf{v}}(M_2) + \sum_{\mathbf{v}\in X_2} d^2_{\mathbf{v}}(M_1)\right) \tag{2.8}$$

Note that this definition of E_{avg} is very similar to the E_{dist} energy term used by Hoppe *et al.* [95, 90]. It differs only in using the $k_1 + k_2$ averaging term and the symmetric construction by which it measures closest distances in both directions between M_1 and M_2.

To further reduce the cost of evaluating these error metrics, others [179, 106, 22, 108] define *localized* versions of the underlying distance function $d_{\mathbf{v}}$. As defined above, $d_{\mathbf{v}}(M)$ finds the distance of $\mathbf{v}$ to the closest point on M. However, we can restrict our search to a small region R of M and evaluate the localized distance $d_{\mathbf{v}}(R)$. Many surface simplification algorithms, including contraction-based algorithms such as my own, produce a correspondence between vertex neighborhoods on the approximation and regions on the original surface. Thus, we can quite naturally define a localized error metric based on measuring distances to these corresponding regions.

I will use the sampled form of E_{avg} (2.8) as the basis for objective evaluation of approximation error. Thus, the quality of an approximation M' to a surface M is characterized by the magnitude of the error $E_{avg}(M, M')$. All empirical measurements of approximation quality, such as those in Chapter 6, will be computed using this metric. For certain applications such as collision detection and part-removal path planning in CAD systems, the Hausdorff distance E_{max} might be preferable. However, as in the function approximation case, E_{avg} generally gives a better measurement of overall fit than E_{max} and is less sensitive to noise, even though it may over-discount localized deviations.

2.3.3 Mesh Quality

In addition to the geometric fidelity of the surface defined by a particular approximation M', we may also be interested in the quality of the mesh used to tessellate the surface.

Perhaps the most widely studied aspect of mesh quality is triangle shape. This has received particular attention in the study of the finite element method. Of common concern are sliver triangles — triangles containing a small interior angle close to 0. When error is measured using a Hilbert norm, which penalizes slope as well as position errors, sliver triangles can result in poor approximations [9]. For certain problem domains, slivers can also lead to high condition numbers during finite element analysis [146]; this leads to numerical instability during the solution process. The Delaunay triangulation [147, 79], which maximizes the minimum angle over all triangulations of the vertices [116], is a popular method for producing well-shaped meshes in planar domains. Generalized Delaunay-like methods can also be formulated for manifold surfaces [21]. However, if we are solely concerned with minimizing an L_2 or L_∞ error norm, sliver-shaped triangles can produce better results [156, 138, 37].

For the problem domains of interest here (e.g., rendering) approximation error is indeed more important than triangle shape. Consequently, mesh quality will be only a minor concern when evaluating approximations generated by surface simplification.

2.3.4 Topological Validity

We must also be concerned about topological[7] degeneracies in the mesh. One of the most common forms of degeneracy that arises in practice is mesh fold-over. Figure 2.12 shows two different tessellations with identical geometry. The

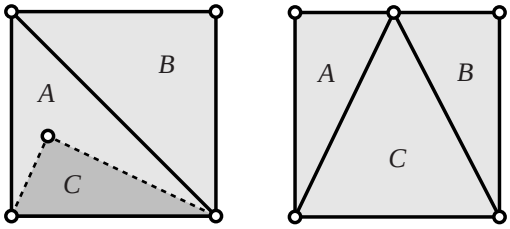


Figure 2.12: Two planar triangulations of the same region. On the left, triangle C is folded over onto triangle A.

geometric error measures E_{max} and E_{avg} would regard these models as identical. But the mesh on the left is clearly problematic; triangle C is folded over onto triangle A. This mesh would not even be considered a proper triangulation under several common definitions. For instance, it is not a simplicial complex (§2.1) because triangles A and C intersect in places other than in a shared vertex or edge. For algorithms based on local operations (e.g., edge contraction) this sort of local degeneracy can be avoided through the use of fairly simple consistency

[7] The use of the term "topology" here refers to surface topology [3]. The same term is occasionally used to refer to the structure of the mesh, but I reserve the term "connectivity" for this purpose.

checks. See Section 3.7 for details on detecting and preventing fold-over for 3-D surfaces.

A more difficult problem involves avoiding non-local self-intersection. Consider a model of the bones forming a human knee joint. The individual bones are in close proximity, yet they are separate. As the surface is modified during simplification, it is quite possible that the surfaces will be adjusted so that they interpenetrate. In general, preventing this kind of behavior can involve rather elaborate techniques [31]. Given its high cost, guaranteeing that self-intersection does not occur is not feasible in fast simplification systems. We might also have to contend with input models which may themselves contain self-intersections.

2.4 Survey of Polygonal Simplification Methods

The problems of surface simplification and multiresolution modeling have received increasing attention in recent years. Simplification has much in common with function approximation, which has been an area of mathematical research for well over a century. The underlying concept of multiresolution surface models is not particularly new; Clark [28] discussed the general idea twenty-five years ago. However, with the exception of work done on simpler objects such as curves and height fields, most of the results in the field are fairly recent.

In this section, I will survey some of the results which are most relevant to my own work described in later chapters. Paul Heckbert and I have written a more complete survey [84] of prior work in the field. Data on the relative performance of various simplification algorithms can be found in the survey of Cignoni *et al.* [26] and elsewhere [124, 22].

The two most common methodologies in surface simplification are *refinement* and *decimation*. A refinement algorithm is an iterative algorithm which begins with an initial coarse approximation and adds elements at each step. Essentially the opposite of refinement, a decimation algorithm begins with the original surface and iteratively removes elements at each step. Both refinement and decimation share a very important characteristic: they seek to derive an approximation through a *transformation* of some initial surface.

An important distinction between algorithms is whether they perform topological simplification on the surface. Most methods fall into one of three categories. Some specifically *prohibit* any topological alteration [31]. The majority of algorithms simplify the topology *implicitly*. In other words, they make choices based on geometric criteria, but they may simplify the topology as a side-effect. Finally, some algorithms *explicitly* consider the simplification of surface topology [82, 53] along with geometric simplification.

Before considering general surface simplification, let us briefly examine two lower-dimensional problem domains — the simplification of curves and height fields.

2.4.1 Curves and Functions

Not surprisingly, the simplification of functions and curves has the longest history. The literature on the approximation of curves satisfying an equation $y = f(x)$ is vast. Plane curves, usually defined parametrically as $\mathbf{v} = f(t) = [x(t)\ y(t)]^\mathsf{T}$, have also received substantial attention. The work on the simplification of piecewise-linear curves is most closely related to the problem of simplifying polygonal surfaces. It has developed in, among other areas, cartography[8], computer vision, and computer graphics.

Suppose that we have a piecewise-linear curve with n vertices, and we would like an approximation with $m < n$ vertices. For these simple geometric objects, we can actually construct optimal approximations. Algorithms have been developed for constructing L_∞–optimal[9] approximations of functions [98], plane curves [98], and 3-D space curves [97]. However, finding these optimal solutions quickly becomes expensive. While the algorithm for finding optimal approximations of functions has a time complexity of $O(n)$, the algorithms for plane curves and space curves have much higher complexities of $O(n^2 \log n)$ and $O(n^3 \log m)$, respectively. This makes them rather impractical for very large datasets.

Regular subsampling, or the nth-point algorithm, is a particularly simple algorithm. It simply keeps every nth point of the original vertex set. This is both very fast and trivial to implement; however, the resulting approximations can be quite poor [136].

Perhaps the most widely used algorithm for curve simplification is a simple refinement algorithm, commonly referred to as the Douglas–Peucker algorithm. This algorithm begins with with some minimal approximation, normally a single line segment from the first to last vertex. This segment is split at the point on the original curve which is furthest from the approximation. Each of the two new subsegments can be recursively split until the approximation meets some termination criteria. This is evidently a rather natural algorithm for curve approximation, since it was independently invented by a number of people [152, 50, 48, 12, 187, 141, 10].

Decimation algorithms, which in essence are the Douglas–Peucker refinement algorithm in reverse, have also been developed [16, 120]. The quality of their results is probably at least as good, if not somewhat better, than the refinement algorithm. However, they are almost certain to be moderately less efficient. Broadly speaking, the time and memory requirements of these iterative algorithms depend on the size of the current approximations being tracked through successive iterations. The refinement approach begins with a minimal approximation and gradually refines it, rather than starting with the full model and gradually simplifying it. Therefore, the intermediate approximations which it constructs tend to be fairly small, particularly if the target approximation is only say 10% or less the size of the original.

[8] Simplification is more commonly referred to as "generalization" in cartography.

[9] An L_∞–optimal approximation uses the minimum number of vertices necessary to achieve a given L_∞ error ϵ (see (2.2) for L_∞ definition).

2.4.2 Height Fields

Height fields are among the simplest types of surface. The surface is defined as the set of points satisfying an equation of the form $z = f(x, y)$ where x and y range over a subset of the Cartesian plane.

In contrast to curve simplification, it is not feasible to construct optimal approximations of height fields. Agarwal and Suri [2] have shown that computing an L_∞–optimal approximation of a height field is NP-Hard [34]. In other words, an optimal approximation cannot be computed in less than exponential time. Polynomial time approximation algorithms have been developed [2, 1]; they can generate approximations with some L_∞ error ϵ using $O(k \log k)$ triangles, where there are k triangles in the optimal approximation. However, their running time is at best $O(n^2)$ for a height field with n input points — too high for practical use on large datasets.

Refinement is the most popular approach for terrain approximation, as it was in the case of curves. One particularly common refinement algorithm is an analog of the Douglas–Peucker curve refinement algorithm. It begins with a minimal approximation and iteratively inserts the point where the approximation and the original are farthest apart. This *greedy insertion* technique [67] has received significant attention [58, 61, 62, 39, 40, 38, 86, 143, 150, 155] and has been independently rediscovered repeatedly. Incremental Delaunay triangulation [79] is often used to triangulate the selected vertices, but other data-dependent triangulations can produce approximations with lower error [155, 67].

Decimation algorithms for simplifying height fields have also been proposed [119, 166, 96]. However, as was the case with curves, they do not seem to be as widely used as refinement methods. Depending on the exact algorithms chosen, decimation may produce higher quality results than refinement. But the greater speed and smaller memory requirements of refinement seem to have made it the more common choice.

2.4.3 Surfaces

Successful algorithms for simplifying curves and height fields were developed twenty years ago [48, 61], but the work on more general surface simplification is much more recent. Since these algorithms are of the greatest relevance to my work, I will discuss them in more detail. Note that, since height fields are a special case of general surfaces, optimally approximating a surface is NP-Hard.

Manual Preparation

The traditional approach to multiresolution surface models has been manual preparation. A human designer must construct various levels of detail by hand. Manual techniques have been in use in the flight simulator field for decades [35], and similar techniques are in use today by game developers [181]. While this process may be aided by a specially designed surface editor [63], it can still be a time-consuming and difficult task. The general goal of the work done on surface simplification has been to automate this task.

Polyhedral Refinement

Only a small number of algorithms for progressively refining polygonal surfaces have been proposed [56, 43, 44, 45]. While refinement has traditionally been the method of choice for approximating curves and height fields, decimation has been much more widely used for simplifying more general surfaces. Perhaps the primary difficulty with refinement in this case involves actually constructing the base approximation. If we limit ourselves to refining via simple subdivision rules, then the initial approximation must necessarily have the same topology as the original model. However, not only does this prevent us from simplifying the topology, but it is not always easy to discover the topology of the input surface.

Vertex Clustering

Vertex clustering methods [161, 128, 167, 129] spatially partition the vertex set into a set of clusters and unify all vertices within the same cluster. They are generally very fast and work on arbitrary collections of triangles. Not only do they generally support non-manifold objects, but they do not require complete connectivity information. Unfortunately, they can often produce relatively poor quality approximations.

The simplest clustering method is the uniform vertex clustering algorithm described by Rossignac and Borrel [161]. A simple example of uniform clustering is shown in Figure 2.13. This algorithm was designed for automatically

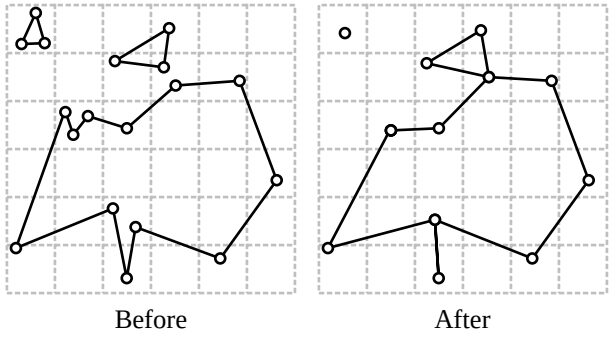


Figure 2.13: Uniform clustering in two dimensions.

constructing multiresolution models from CAD data, hence the necessity to allow arbitrary polygon input. The vertex set is partitioned by subdividing a bounding box on a regular grid, and the new representative vertex for each cell is computed using cheap heuristics based on criteria such as edge length. This process can be implemented quite efficiently. The algorithm also tends to make substantial alterations to the topology of the original model. Looking at Figure 2.13, we can see that the triangle in the upper left corner is reduced to a point,

and two separate components along the top are joined together. Note that, much like uniform subsampling of images, the results of this algorithm can be quite sensitive to the actual placement of the grid cells. It is also incapable of simplifying features larger than the cell size. In particular, a planar rectangle consisting of many triangles all larger than the cell size will not be simplified at all, even though it can be approximated using two triangles without error.

The most natural way to extend uniform clustering is to use more elaborate spatial partitioning schemes. Luebke [129, 130] examined the generalization of this algorithm to use a more adaptive octree partition of space. Low and Tan [128] proposed a more flexible partitioning scheme. Cells may be any simple shape, such as a cube or sphere, and cells are centered around the vertex of highest importance. Vertices that fall within multiple cells are assigned to the cell with the closest center.

Clustering methods tend to work well if the original model is highly oversampled and the required degree of simplification is not too great. They also tend to perform better when the surface triangles are smaller than the cell size. Since no vertex moves further than the diameter of its cell, clustering algorithms provide guaranteed bounds on the Hausdorff approximation error (§2.3.2) sampled at the vertices of M and M'. However, to achieve substantial simplification, the required cell size increases quite rapidly, making the error bound rather weak. In particular, at more aggressive simplification levels, the quality of the resulting approximations can degrade rapidly. For an example, see Figure 6.30 in Section 6.4.

Region Merging

A handful of simplification algorithms [102, 103, 89, 75, 154] operate by merging surface regions together. For example, the "superfaces" algorithm of Kalvin and Taylor [102, 103] partitions the surface into disjoint connected regions based on a planarity criterion. Each region is replaced by a polygonal patch whose boundary is simplified, and the resulting region is retriangulated. These algorithms are generally restricted to manifold surfaces, and do not alter the topology of the model. The algorithms of Hinker & Hanson [89] and Gourdon [75] appear to be best suited for smooth surfaces that are not highly curved. However, Kalvin and Taylor's algorithm seems to produce good quality results, and it provides bounds on the approximation error.

Region merging techniques do not seem to have become widespread. This may well be because they are somewhat more complicated to implement in comparison to other algorithms without offering superior approximations. And in contrast to iterative edge contraction, they do not produce a natural multiresolution representation.

Wavelet Decomposition

Wavelet methods [182] provide a fairly clean mathematical framework for the decomposition of a surface into a base shape plus a sequence of successively

finer surface details. Approximations can be generated by discarding the least significant details. Wavelet decomposition has been used quite successfully for producing multiresolution representations of signals and images [132, 182].

Lounsbery *et al.* [127] developed a method for generating a wavelet decomposition of surfaces with subdivision connectivity[10]. Consequently, the resulting approximations may be relatively far from optimal because they may use a large number of triangles simply to preserve subdivision connectivity. Wavelet decompositions are also generally unable to resolve creases on the surface unless they fall along edges in the base mesh; Hoppe [90] provides a good illustration of this effect. Eck *et al.* [51] developed a procedure for producing a subdivision mesh from a surface with arbitrary connectivity. However, this pre-process introduces some level of error into the base shape, although this error can be limited by a specified tolerance value. Like other subdivision-based schemes, wavelet methods cannot construct approximations with a topology different from the original surface.

Vertex Decimation

One of the more widely used algorithms is vertex decimation, an iterative simplification algorithm originally proposed by Schroeder *et al.* [172]. In each step of

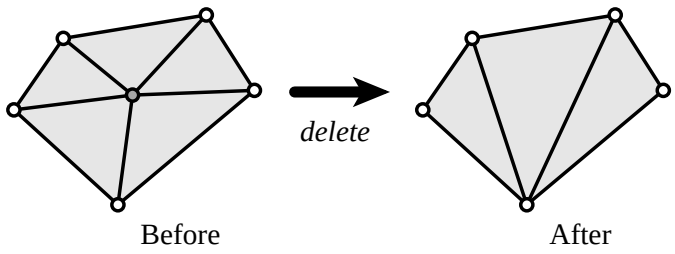


Figure 2.14: A vertex is removed and the resulting hole is retriangulated.

the decimation process, a vertex is selected for removal, all the faces adjacent to that vertex are removed from the model, and the resulting hole is retriangulated (see Figure 2.14). Since this retriangulation requires a projection of the local surface onto a plane, these algorithms are generally limited to manifold surfaces. The fundamental operation of vertex deletion is also incapable of simplifying the topology of the model. Schroeder [171] was able to lift these restrictions by incorporating cutting and stitching operations into the simplification process. The original vertex decimation algorithm [172] used a fairly conservative estimate of approximation error. More recent methods [179, 106, 22, 108] use more accurate error metrics, like the localized Hausdorff error (§2.3.2). They maintain links between points on the original surface and the corresponding neighborhood on

[10] Starting with a base mesh, such as an octahedron, triangles are repeatedly subdivided into four subtriangles by splitting each edge and connecting these new vertices.

the approximation, and the distances between these points and the associated faces define the approximation error.

The algorithm of Schroeder *et al.* is reasonably efficient, both in time and space, but it seems to have some difficulty preserving smooth surfaces [171, Fig. 9]. The body of the turbine blade (see §6.2 Figure 6.13) is initially smooth, but becomes quite rough during simplification. While the other vertex decimation algorithms produce higher quality results, they are substantially slower and consume more space.

This methodology of vertex decimation is in fact closely related to iterative contraction (discussed in the next section). In particular, note that the vertex removal pictured in Figure 2.14 can just as easily be accomplished by contracting the bottom edge. Removing a vertex by edge contraction [171] is generally more robust than projecting the neighborhood onto a plane a retriangulating [172]. In this case, we do not need to worry about finding a plane onto which the neighborhood can be projected without overlap.

Iterative Contraction

The final major class of algorithms is based on the iterative contraction of edges[11] [90, 144, 93, 158, 76, 77, 68, 70, 5, 124, 137, 114]. My own algorithm [68, 69], discussed in much greater detail in Chapter 3, falls into this category.

When an edge is contracted, its end points are replaced by a single point and triangles which degenerate to edges are removed (see Figure 2.15). Unless

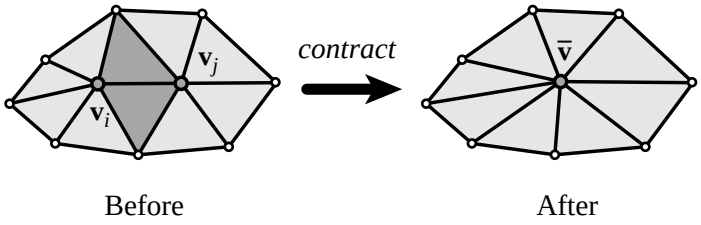


Figure 2.15: Edge $(\mathbf{v}_i, \mathbf{v}_j)$ is contracted; two faces and one vertex are removed.

the topology is explicitly preserved, edge contraction algorithms may implicitly alter the topology by closing holes in the surface. Hoppe *et al.* [95] appear to have been the first to use edge contraction as the underlying mechanism for accomplishing surface simplification. I [68] and others [144] have proposed a generalization of edge contraction to permit the contraction of arbitrary vertex pairs rather than just edges. This allows unconnected regions to be joined together which also results in topological simplification. Also note that the fundamental operation of contraction does not require the immediate neighborhood to be manifold. In fact, contraction can be applied to any simplicial

[11] The algorithm of Gieng *et al.* [70] is actually formulated using face contraction, but since a face can be contracted by contracting two of its edges, the distinction is minor.

complex. Thus contraction-based algorithms can more conveniently deal with non-manifold surfaces than vertex decimation algorithms.

Hoppe's algorithm [90] for constructing progressive meshes is based on minimization of an energy function. One of its primary components is a geometric error term very much like E_{avg} (2.8). The algorithm maintains a set of sample points on both the original surface and the approximation. The distances between these points and the closest point on the opposing surface determine the geometric error. This algorithm produces some of the highest quality results among currently available methods. However, the price of this precision is a very long running time. Hoppe reports [91] running times of around an hour for a model of about 70,000 faces. The original algorithm could simplify topology by closing holes in the surface, and the extension by Popović and Hoppe [144] can join unconnected regions.

The algorithm developed by Guéziec [76, 77] maintains a tolerance volume around the approximation such that the original surface is guaranteed to lie within that volume. The volume itself is defined by spheres located at each vertex of the approximation. The convex combination of these spheres over the faces of the model creates so called "fat triangles" which comprise the tolerance volume. Vertices of the approximation are positioned to preserve the volume of the object. While this algorithm appears somewhat slow, it is faster than Hoppe's algorithm, and it appears to generate good quality results.

Ronfard and Rossignac [158] have developed a fairly efficient simplification algorithm. Each vertex in the approximation has an associated set of planes, and the error at that vertex is defined by the maximum of squared distances to the planes in this set. These sets are merged when vertices are contracted together. Because measuring distances to planes is much cheaper than measuring distances to triangles, this error metric is much cheaper than Hoppe's, for example. While it is necessarily less precise, Ronfard and Rossignac [159] show that localized Hausdorff error bounds can be derived from their metric. The resulting approximations appear to have generally good quality, and the algorithm is fairly efficient compared to the more exact algorithms. Indeed, this error metric is closely related to my own (see Section 3.3.1 for more details).

The "memoryless" algorithm recently developed by Lindstrom and Turk [124] is interesting in that, unlike most algorithms, it makes decisions based purely on the current approximation alone. No information about the original shape is retained. They use linear constraints, based primarily on conservation of volume, in order to select an edge for contraction and the position at which the remaining vertex will be located. In fact, this error metric is in part a simple variation of my own (see Section 4.1.3). The reported results suggest that it can generate good quality results, and that it is fairly efficient, particularly in memory consumption.

The mesh optimization algorithm of Hoppe *et al.* [95] is an earlier form of Hoppe's progressive mesh construction algorithm [90]. It performs explicit search rather than simple greedy contraction. Consequently, it exhibits even longer running times, but may produce the highest quality results. Edge contraction, edge split, and edge flip are the operators used to explore the space of

possible approximations. Johnson and Hebert [100] used iterative application of edge contraction and edge split to produce approximations that are both geometrically faithful and that have roughly uniform edge length.

One of the major benefits of iterative contraction is the hierarchical structure that it creates (see Chapter 7 for a more detailed discussion). This quite naturally leads to a useful multiresolution surface representation [90, 91, 93, 198]. Note that similar hierarchies can be built using vertex decimation algorithms [171, 41].

2.4.4 Material Properties

Much of the work done on simplifying surfaces has focused exclusively on the geometry of the surface. But in practice, models may often have various material properties. For example, models intended for use in rendering systems might often have color and texture attached to the surface.

For restricted surface classes, such as height fields, very simple methods can work reasonably well. My own version of the greedy insertion terrain approximation algorithm [67] can approximate terrains with color samples at each vertex. However, more general surfaces require more advanced techniques, and some available simplification methods do address this need.

Hoppe [90] explicitly included attributes in his error metric which supports both per-vertex (or scalar) attributes and per-face (discrete) attributes. As was the case with geometric fidelity, this algorithm seems to produce high quality results at the cost of rather high running times. Certain *et al.* [19] outlined a technique for adding surface color to a wavelet surface decomposition. Hughes *et al.* [96] investigated the simplification of Gouraud-shaded meshes produced by radiosity simulations; it does not appear that their method would generalize well to other domains. My own algorithm can also be extended to consider material properties as part of the error metric [69] (see Chapter 5).

An alternative approach is to focus more on attributes as maps on the surface. In this case, we would focus on reparameterizing the maps rather than preserving actual attribute values. Cohen *et al.* [30, 29] developed an algorithm capable of reparameterizing both texture and normal maps as a surface is simplified. Others — including Maruya [133], Soucy *et al.* [178], and Cignoni *et al.* [25] — decouple attributes and geometry even further. They first compute a simplified surface, without regard for the surface attributes. Given the final approximation, they resample the attributes of the original into a texture map on the approximation. However, this complete decoupling has certain disadvantages. Most significantly, the final texture map is highly fragmented and this technique would not work well for generating progressive output (see Chapter 5 for more details).

2.4.5 Summary of Prior Work

In the last few years, significant progress has been made in the field of polygonal surface simplification. A range of fairly effective algorithms is now available.

Simplification facilities have appeared in a number of commercial packages. Effective multiresolution surface representations, produced via simplification, have begun to appear [91, 130, 198, 41].

Hoppe's simplification algorithm [90] produces very good results, but is quite slow. Vertex clustering (e.g., the Rossignac–Borrel algorithm [161]) is very fast and broadly applicable to arbitrary collections of triangles. But the resulting approximations can be quite poor. The vertex decimation algorithm of Schroeder *et al.* [172, 171] seems to be reasonably fast, but also appears to have some difficulty preserving smooth surfaces. Related vertex decimation algorithms [179, 106, 22, 108] produce higher quality results, but also appear to be substantially slower [26].

Between the very high quality, very slow algorithm proposed by Hoppe and the very fast, but low quality algorithms such as Rossignac & Borrel's, there is a need for a fast simplification algorithm that produces high quality results. This is the role that my simplification algorithm attempts to fill. It provides a practical mix of efficiency and quality.

Chapter 3

Basic Simplification Algorithm

In the previous chapter, I have reviewed the various algorithms which have been devised for automatically simplifying polygonal surface models. Now, I will present the basic simplification algorithm which I have developed. It is founded on two fundamental components: iterative vertex pair contraction and the quadric error metric. In this chapter, I will focus on the description of the algorithm itself. I will analyze the quadric error metric in much greater detail in Chapter 4 and examine my algorithm's performance in Chapter 6.

For the moment, I will consider the surface geometry alone. I will delay the issue of material properties such as color until Chapter 5. At that point, I will present an extended form of the quadric metric which can account for the presence of such properties.

3.1 Design Goals

A fast algorithm capable of producing high-quality approximations has been the primary goal of my work. For surface simplification, like most problems where optimal algorithms are infeasible, a tradeoff exists between quality and efficiency. As I stated in Section 2.4.5, there has been a gap between the very fastest algorithms, which can produce poor approximations, and those algorithms which produce very high quality approximations, but which can be rather slow. Therefore, I have opted to compromise quality in some respects in order to create a more efficient algorithm. For instance, my algorithm does not provide guaranteed error bounds for the approximations which it produces.

I have also assumed that approximations need not maintain the topology of the original surface. Clearly, there are applications in which this is not true. In medical imaging, for instance, topology may provide critical information; depending on the application, preserving a hole in the heart wall may be much more important than preserving the exact shape of the surface. However, in

most rendering applications, not only can we safely simplify the topology of models, it is often desirable to do so. Consider a model of a sponge. When examined closely, the intricate structure of holes in the sponge is a visually important feature. However, when viewed from a distance, these holes are imperceptible. The entire sponge can be adequately approximated by a simple block, particularly if we can apply an appropriate texture image to the block that simulates the texture of the original.

As I discussed in Section 2.1.1, it is also helpful for simplification algorithms to allow non-manifold surfaces. Not only does this broaden the class of models which can be input to the algorithm, but it gives the algorithm more flexibility in simplifying the surface. The process of simplification itself may well transform manifold inputs into non-manifold approximations, and I allow this sort of topological modification.

3.2 Iterative Vertex Contraction

The simplification algorithm which I have developed is a decimation algorithm. It begins with the original surface and iteratively removes vertices and faces from the model. Each iteration involves the application of a single atomic operation: vertex pair contraction.

A vertex pair contraction, which I denote $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$, modifies the surface in three steps:

1. Move the vertices $\mathbf{v}_i$ and $\mathbf{v}_j$ to the position $\bar{\mathbf{v}}$;
2. Replace all occurrences of $\mathbf{v}_j$ with $\mathbf{v}_i$;
3. Remove $\mathbf{v}_j$ and all faces which become degenerate — that no longer have three distinct vertices.

The first step modifies the geometry of the surface, and the second step modifies the connectivity of its mesh. Depending on the structure of the mesh, this may also alter the topology of the surface. The final step simply removes elements of the surface which are no longer needed. Pair contraction is a generalization of edge contraction; the vertices $\mathbf{v}_i, \mathbf{v}_j$ need not be connected by an edge. I proposed this generalization as a means of aggregating separate topological components during simplification [68]. Popović and Hoppe [144] independently proposed the same generalization, which they referred to as vertex unification. Figure 3.1 illustrates a single edge contraction in a manifold neighborhood. Note that the effect of a contraction is small and highly localized. For the example shown in Figure 3.1, one vertex and two faces are removed from the model. As we will see shortly, this locality is very important for the efficiency of the algorithm.

The effect of an edge contraction such as the one pictured in Figure 3.1 is to remove one vertex and one or more faces from the model. In contrast, contracting a non-edge pair will remove one vertex and join previously unconnected regions of the surface (see Figure 3.2). By allowing the contraction of non-edge

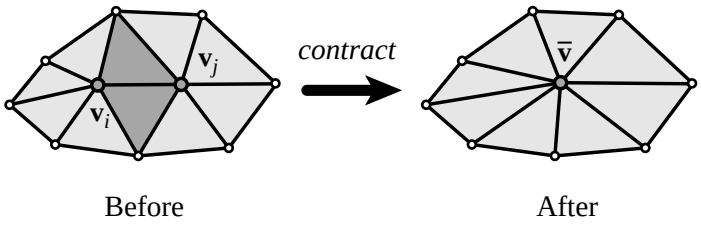


Figure 3.1: Edge $(\mathbf{v}_i, \mathbf{v}_j)$ is contracted. The darker triangles become degenerate and are removed.

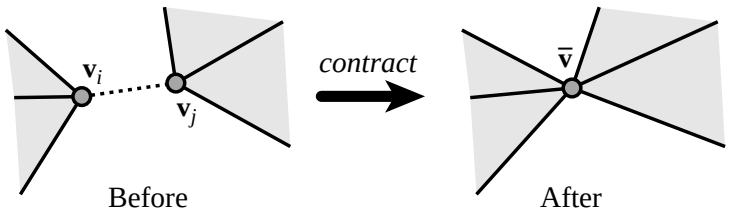


Figure 3.2: Non-edge pair $(\mathbf{v}_i, \mathbf{v}_j)$ is contracted, joining previously unconnected areas. No triangles are removed.

pairs, we are allowing the algorithm to topologically simplify the model by joining separate components. This is also one reason why it is advantageous to support non-manifold surfaces. At the instant when two separate components are joined together, a non-manifold region will almost certainly be created. For instance, the resulting neighborhood pictured in Figure 3.2 is non-manifold because the faces surrounding $\bar{\mathbf{v}}$ form two separate fans.

The algorithm which I have developed, like most related methods, is a simple greedy procedure. It produces large-scale simplification by applying a sequence of vertex pair contractions. This sequence is selected in a purely greedy fashion; once a contraction is selected, it is never reconsidered. At a high level, the outline of the algorithm is as follows:

1. Select a set of candidate vertex pairs (§3.2.1).
2. Assign a cost of contraction to each candidate (§3.3.1–§3.4).
3. Place all candidates in a heap keyed on cost with the minimum cost pair at the top.
4. Repeat until the desired approximation is reached:
   (a) Remove the pair $(\mathbf{v}_i, \mathbf{v}_j)$ of least cost from the heap.
   (b) Contract this pair.
   (c) Update costs of all candidate pairs involving $\mathbf{v}_i$.

Most of the various algorithms based on iterative contraction have this basic structure. Each pair being considered for contraction is assigned a "cost". The way in which this cost is determined is the primary differentiating factor between algorithms of this type. Generally, this cost of contraction is meant to reflect the amount of error introduced into the approximation by the contraction of the pair in question. At each iteration, the lowest cost pair is contracted. In Sections 3.3.1 and 3.4 I will discuss how my algorithm calculates contraction costs. For now, let us simply assume that some available procedure, when given a pair $(\mathbf{v}_i, \mathbf{v}_j)$, returns a real number $E(\mathbf{v}_i, \mathbf{v}_j)$ which is the cost of contracting that pair.

In general, we can not expect this simple greedy algorithm to produce optimal approximations. However, finding optimal approximations of surfaces is an NP-Hard [2, 34] problem, and no methods currently exist which generate provably good approximations. A more thorough search algorithm might conceivably produce much better approximations than greedy simplification, but it's results would remain sub-optimal.

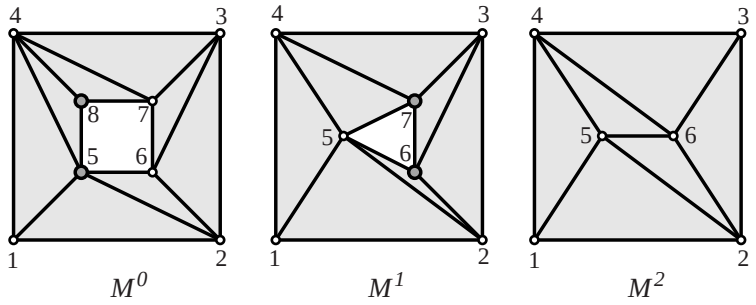


Figure 3.3: Simplification of a simple planar object.

Figure 3.3 illustrates the simplification of a very simple model by a series of edge contractions. The original model M^0 is a square with a hole cut through it; the total model consists of eight triangular faces. By contracting the highlighted pair of vertices $(\mathbf{v}_5, \mathbf{v}_8)$, we produce the approximation M^1. One vertex and one face have been removed, and the hole is now a triangle rather than a square. Now, by contracting the highlighted pair $(\mathbf{v}_6, \mathbf{v}_7)$, we arrive at the approximation M^2 which has only six triangular faces. Notice that the hole is now closed. Assuming that contractions were selected based on purely geometric criteria, this example illustrates how the topology of the model may be implicitly simplified by iterative contraction. For application domains in which topological alterations are not desirable, the basic algorithm can be augmented by additional mechanisms that guarantee topological preservation [31, 52].

Note that, as a result of iterative contraction, we establish a correspondence between the vertices of the original model and the vertices of each intermediate approximation. Table 3.1 lays out the correspondence created by the simplifi-

	Original Vertices							
Model	1	2	3	4	5	6	7	8
M^0	1	2	3	4	5	6	7	8
M^1	1	2	3	4	5	6	7	**5**
M^2	1	2	3	4	5	6	**6**	**5**

Table 3.1: Vertex correspondences for example shown in Figure 3.3.

cation of Figure 3.3. This correspondence can be used to construct geomorphs (§2.2.1) between approximations. Disregarding the placement of vertices, the list of vertex assignments completely determines the transformation of the original model into any of the approximations[1]. This is important because of the mapping it provides between the original surface and the resulting approximation. Every vertex and vertex neighborhood on the approximation corresponds to a set of vertices and a set of faces, respectively, on the original surface. As we will see in Chapter 7, the correspondence resulting from iterative contraction induces a hierarchical structure on the surface which provides a useful multiresolution surface representation.

3.2.1 Selecting Candidates

The initial step of the algorithm is to select a set of candidate vertex pairs. The most natural choice is simply to select all edges of M^0. If the model consists of a single connected component, this may be the best choice. However, consider the case of a model with more than one connected component. We may be able to produce better approximations by considering non-edge pairs as well. Figure 3.4a shows a model composed of 100 closely spaced, individual cubes

(a) Grid of 100 cubes (b) using edges only (c) with added pairs

Figure 3.4: Contrast the results of simplifying this model using edges only (b) with the results of simplifying using edges as well as selected non-edge pairs (c).

arranged in a grid. Simplifying this model by considering edges alone produces approximations like that shown in Figure 3.4b. As you can see, the individual

[1] Consequently, it is a *simplicial transformation* or simplicial mapping [3, 4, 87].

components are individually reduced to nothing. This does not provide a good representation of the original. On the other hand, consider the approximation shown in Figure 3.4c. By adding selected non-edge pairs for consideration, the algorithm was able to join the separate components into a single component. The result is a much more faithful approximation of the original.

Building a candidate set containing some non-edge pairs in addition to edges may be beneficial; however, the question remains of how to select these non-edge pairs. Popović and Hoppe [144] report having considered several schemes including "binning, octrees, and k-closest neighbor graphs". They settled on the set of edges of the Delaunay tetrahedralization of the vertex set which connect separate components of the model. The very simple technique which I have proposed [68] is to select all pairs $(\mathbf{v}_i, \mathbf{v}_j)$ which satisfy the conditions:

1. $(\mathbf{v}_i, \mathbf{v}_j)$ is an edge, or
2. $(\mathbf{v}_i, \mathbf{v}_j)$ is not an edge and $\|\mathbf{v}_i - \mathbf{v}_j\| < \tau$

where τ is a threshold parameter specified by the user. Conceptually, I place an open ball of radius τ around every vertex, and form pairs with all other vertices that fall within that ball. This bears some resemblance to the partitioning schemes used by vertex clustering methods [161, 167, 130], particularly the method of Low and Tan [128]. However, instead of directly merging all vertices within a particular cell, my algorithm simply forms pairs of vertices for later consideration. Whether any of these pairs will actually be contracted is determined by the cost metric at the heart of the iterative algorithm. The Delaunay-based scheme has the advantage of providing some level of automatic adaptation to the local scale of the model. On the other hand, the distance threshold scheme can potentially provide more pairs for the algorithm's consideration, and the τ parameter provides strict control over the size of gaps which may be bridged by non-edge pairs. A scheme for selecting adaptive τ values, such as the one developed by Erikson and Manocha [54], can produce higher quality results.

In Section 6.4 I will consider the use of non-edge pairs more closely. In particular, we will see that the benefits of this technique are somewhat limited. In short, I have found that greedy edge contraction produces consistently good results on many kinds of models. Greedy contraction of arbitrary pairs is comparatively less robust. It is relatively more dependent on the structure of the model and on the choice of which non-edge pairs to consider.

3.3 Assessing Cost of Contraction

Having settled on the algorithmic framework discussed in Section 3.2, an important issue remains: how to measure the cost of a contraction. Since we are concerned with producing approximations which remain faithful to the original, the cost of a contraction should reflect how much that contraction changes the surface. For rendering applications, we might select a cost metric which reflects

the extent to which the appearance of the model changes (§2.3.1). However, for reasons outlined in Section 2.3, I will follow the common practice in the field and adopt a cost metric which measures the amount of geometric change introduced into the model as a result of a single edge contraction.

Recall that in Section 2.3.2 I described the Hausdorff distance $E_{max}(M_1, M_2)$ and the average distance $E_{avg}(M_1, M_2)$ which measure the difference between two polygonal models M_1 and M_2. One natural, and common, approach is to use these metrics directly to measure the error introduced by individual contractions. For practical reasons, the sampled forms of E_{avg} (2.8) and E_{max} (2.7), where distance measurements are only sampled at a discrete set of points, are used. Hoppe *et al.* [95, 90] measure the non-symmetric[2] E_{avg} error resulting from each possible contraction. This produces high quality results, but it quite slow. Others selected the localized form of the Hausdorff error metric E_{max} [179, 106, 22]. Because of the localization of the error metric, these algorithms are somewhat faster but the quality of their results is also somewhat lower. These two kinds of algorithms can produce high quality results because they explicitly track and seek to minimize the geometric error between the current approximation and the original surface. Still, these techniques do not necessarily produce approximations which minimize their chosen error metrics because they are greedy algorithms. This type of error metric requires us to make several expensive distance-to-surface (Eq. 2.4) measurements for each contraction under consideration. The price we must pay for this level of accuracy is an algorithm that is rather slow. Simplification of a very large surface model might require several hours.

Another common, and generally less expensive, approach is to generate some local surface approximant and assign contraction costs based on distances to this approximant. For instance, the importance of a vertex might be measured by its distance to a plane fit through its neighborhood [172]. We might also consider higher order approximants, such as quadric patches [70], but the cost of computing distances to the patch increases rapidly with higher orders. It also becomes difficult to fit higher order patches to local neighborhoods. Not only must there be enough points to uniquely determine the patch, but the neighborhood generally needs to be plane-projectable. For instance, while we can fit quadric surfaces to arbitrary sets of points [145], we are not guaranteed that the result will be a patch [190]; it might be a two-sheeted hyperboloid. Therefore, it is customary to project the local surface onto a plane, treat it as a height field $f(u, v)$, and fit a quadric patch to it using a least squares method [70, 113]. Even if we only use an approximating plane, this kind of metric has a significant drawback. It only relates the error of the current approximation to the previous one rather than to the original surface. Thus, we might introduce only small incremental errors at each iteration, but we have no guarantee that they are not accumulating into large total errors. While this situation can be remedied to some extent [171], it appears difficult to propagate incremental errors across iterations in such a way as to guarantee a tight estimate of error.

[2] Distances are only measured from points on M to M'.

In order to achieve very rapid simplification, we might also consider using a much simpler measure of contraction cost. A particularly simple approach is to construct a heuristic based on a combination of very inexpensive measurements such as edge length [198], dihedral edge angle [5, 137, 154], and local curvature [75, 89, 114]. Rather than defining a real notion of error, these metrics are based on observations such as "small edges tend to be less important." However, like those based on local approximants, these metrics provide no connection between the current approximation and the original surface. While each iteration may appear to entail only a minor change to the model, these incremental changes may result in approximations which deviate substantially from the original. Finally, these heuristics are often targeted towards specific classes of models, such as smooth surfaces [5, 75, 89] or CAD parts [154], and typically do not generalize well to other kinds of models.

3.3.1 Plane-Based Error Metric

As I outlined earlier, the primary goal of my work has been an algorithm capable of rapidly generating high-quality approximations. This requires an error metric which is cheap to evaluate. A greedy contraction algorithm requires $O(n)$ iterations to simplify a model with n faces to a fixed level. Consequently, it will evaluate the error metric many times during simplification. While it must be inexpensive, the metric must still provide a useful characterization of the approximation error in order to produce good results. It also seems generally desirable that the metric provide a measure of error between the current approximation and the original surface, not simply with the approximation produced during the previous iteration. To support a wide range of input surfaces, such as non-manifolds, it is also important that the error metric not make too many assumptions about the structure of the surface mesh.

The error metrics described in the previous section do not fit all these requirements. Metrics based on careful distance measurements such as E_{avg} or E_{max} produce good results, but are expensive to evaluate. Those based on local approximants are generally not applicable to non-manifold surfaces. Heuristics based on simple measurements, like dihedral edge angles, do not provide enough information about the error between the current approximation and the original surface to produce consistently good results. The error metric which I have developed, the *quadric error metric*, tries to meet these competing requirements of economy, accuracy, and generality. I will present the ideas underlying this metric in the remainder of this section, and describe the details of the metric itself in subsequent sections.

Following Ronfard and Rossignac [158], I associate a set of planes with every vertex of the model; however, as we will see (§3.4) this set is purely conceptual. The standard representation of a plane is the set of all points for which $\mathbf{n}^\mathsf{T}\mathbf{v} + d = 0$ where $\mathbf{n} = [a\ b\ c]^\mathsf{T}$ is a unit normal (i.e., $a^2 + b^2 + c^2 = 1$) and d is a scalar constant. Given a plane in this form, the squared distance of a vertex

$\mathbf{v} = [x\ y\ z]^\mathsf{T}$ from the plane is given by the equation

$$D^2(\mathbf{v}) = (\mathbf{n}^\mathsf{T}\mathbf{v} + d)^2 = (ax + by + cz + d)^2 \tag{3.1}$$

For a vertex $\mathbf{v}$ with an associated set of planes P, I define the error at that vertex to be the sum of squared distances of the vertex to all the planes in the corresponding set

$$E_{plane}(\mathbf{v}) = \sum_i D_i^2(\mathbf{v}) = \sum_i (\mathbf{n}_i^\mathsf{T}\mathbf{v} + d_i)^2 \tag{3.2}$$

This definition of error differs somewhat from the one given by Ronfard and Rossignac [158]. They measured maximum squared distance over the set rather than the sum of squared distances. While tracking the maximum distance (§4.5) leads to a clearer connection to the Hausdorff distance, the use of summation leads to a convenient implicit representation (§3.4) which provides significant performance benefits.

This error metric fits quite easily into the algorithmic framework of Section 3.2. In the original model, each vertex is assigned a set of planes. Each set is initialized with the planes determined by the faces incident to the corresponding vertex. When a pair is contracted into a single vertex, the resulting set is the union of the two sets associated with the endpoints. At first, it appears that we will be required to explicitly track these sets of planes. This can require a sizeable amount of storage that does not diminish as simplification progresses.

To see how this error metric might work in practice, consider the example shown in Figure 3.5. In this 2-D example, every segment defines a line. The

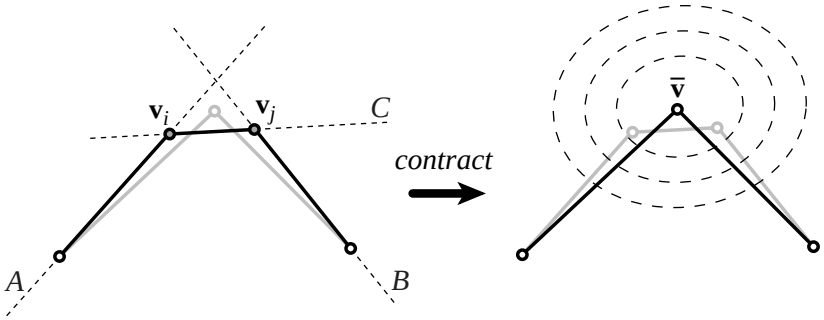


Figure 3.5: Measuring contraction cost in 2-D.

vertices $\mathbf{v}_i, \mathbf{v}_j$ have associated sets of lines $P_i = \{A, C\}$ and $P_j = \{B, C\}$ respectively. Note that the error at these vertices is $E_{plane}(\mathbf{v}_i) = E_{plane}(\mathbf{v}_j) = 0$ since they both lie on all the lines in their respective sets. Upon contracting this pair, we would move the remaining vertex to some new position $\bar{\mathbf{v}}$. The set of lines associated with this vertex would be $\bar{P} = P_i \cup P_j = \{A, B, C\}$. Assuming for the moment that E_{plane} provides an accurate measurement of error, we

would naturally like to choose a position $\bar{\mathbf{v}}$ which minimizes the sum of squared distances to the lines in $\bar{P}$. This vertex, which lies at the center of the triangle formed by the lines A, B, C, is shown in gray. Performing the resulting contraction $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$ produces the approximation pictured on the right. Note that $E_{plane}(\bar{\mathbf{v}}) > 0$ since there is no point at which the lines A, B, C intersect. The elliptical curves around $\bar{\mathbf{v}}$ represent isocontours $E_{plane}(\bar{\mathbf{v}}) = \epsilon$ for various values of ϵ. Note that $\bar{\mathbf{v}}$ lies at the center of these concentric ellipses. In 3-D, the analogous isosurfaces are quadric surfaces, hence the name "quadric error metric."

3.4 Quadric Error Metric

As I indicated in the previous section, the sets of planes associated with each vertex are purely conceptual. In fact, I use a much more convenient and compact representation of error. The foundation of the error metric which I have developed is the fact that we can rewrite the equation for D^2 (3.1) into a new form as follows:

$$\begin{aligned} D^2(\mathbf{v}) &= (\mathbf{n}^\mathsf{T}\mathbf{v} + d)^2 && (3.3)\\ &= (\mathbf{v}^\mathsf{T}\mathbf{n} + d)(\mathbf{n}^\mathsf{T}\mathbf{v} + d) && (3.4)\\ &= (\mathbf{v}^\mathsf{T}\mathbf{n}\mathbf{n}^\mathsf{T}\mathbf{v} + 2d\mathbf{n}^\mathsf{T}\mathbf{v} + d^2) && (3.5)\\ &= \left(\mathbf{v}^\mathsf{T}(\mathbf{n}\mathbf{n}^\mathsf{T})\mathbf{v} + 2(d\mathbf{n})^\mathsf{T}\mathbf{v} + d^2\right) && (3.6) \end{aligned}$$

where $\mathbf{n}\mathbf{n}^\mathsf{T}$ is the outer product matrix

$$\mathbf{n}\mathbf{n}^\mathsf{T} = \begin{bmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{bmatrix} \tag{3.7}$$

This has close connections with various quadratic distance metrics (§4.1.4) and the least squares method of normal equations (§4.2).

I will define a *quadric* Q as a triple

$$Q = (\mathbf{A}, \mathbf{b}, c) \tag{3.8}$$

where $\mathbf{A}$ is a 3×3 matrix, $\mathbf{b}$ is a 3-vector, and c is a scalar. The quadric Q assigns a value $Q(\mathbf{v})$ to every point in space $\mathbf{v}$ by the second order equation

$$Q(\mathbf{v}) = \mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{v} + 2\mathbf{b}^\mathsf{T}\mathbf{v} + c \tag{3.9}$$

I use the term *quadric* because the isosurfaces $Q(\mathbf{v}) = \epsilon$ are quadric surfaces — second degree implicit surfaces which include ellipsoids, paraboloids, hyperboloids, and planes [15]. I will explore this geometric interpretation in much greater detail in Chapter 4.

Quadrics provide a very convenient representation for the squared distance $D^2(\mathbf{v})$ of a point $\mathbf{v}$ to a particular plane. For a given plane $\mathbf{n}^\mathsf{T}\mathbf{v} + d = 0$, I define its *fundamental quadric* Q to be

$$Q = (\mathbf{n}\mathbf{n}^\mathsf{T}, d\mathbf{n}, d^2) \tag{3.10}$$

Recalling the formula for D^2 (3.6) given earlier, we see that the value of this quadric is precisely the squared distance $Q(\mathbf{v}) = D^2(\mathbf{v})$ of $\mathbf{v}$ to the given plane.

The addition of quadrics can be naturally defined component-wise: $Q_i(\mathbf{v}) + Q_j(\mathbf{v}) = (Q_i + Q_j)(\mathbf{v})$ where $(Q_i + Q_j) = (\mathbf{A}_i + \mathbf{A}_j, \mathbf{b}_i + \mathbf{b}_j, c_i + c_j)$. Thus, given a set of fundamental quadrics, determined by a set of planes, the quadric error E_Q is completely determined by the sum of the quadrics Q_i:

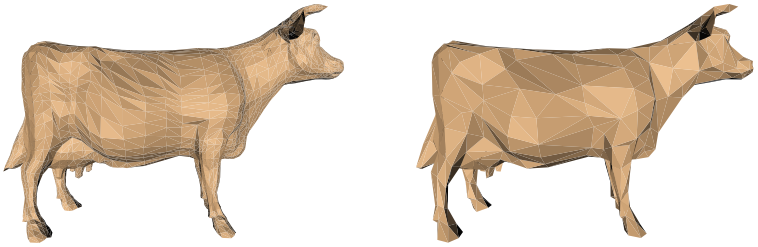
$$E_Q(\mathbf{v}) = \sum_i D_i^2(\mathbf{v}) = \sum_i Q_i(\mathbf{v}) = Q(\mathbf{v}) \tag{3.11}$$

where $Q = \sum_i Q_i$. In other words, to compute the sum of squared distances to a set of planes, we only need one quadric which is the sum of the quadrics defined by each of the individual planes in the set. When contracting the edge $(\mathbf{v}_i, \mathbf{v}_j)$, the resulting quadric is merely $Q = Q_i + Q_j$. Furthermore, the cost of a contraction $(\mathbf{v}_i, \mathbf{v}_j) \to \bar{\mathbf{v}}$ is $Q(\bar{\mathbf{v}}) = Q_i(\bar{\mathbf{v}}) + Q_j(\bar{\mathbf{v}})$.

The efficiency of the quadric error metric is one of its primary features. Each vertex has exactly one quadric, and this is the only additional memory required by the error metric. Specifically, it requires 10 coefficients to store the symmetric 3×3 matrix $\mathbf{A}$, the 3-vector $\mathbf{b}$, and the scalar c. The cost of evaluating the error of a vertex is also a fairly small constant cost. This is in contrast to the related metric of Ronfard and Rossignac [158] which requires time linear in the size of the plane set.

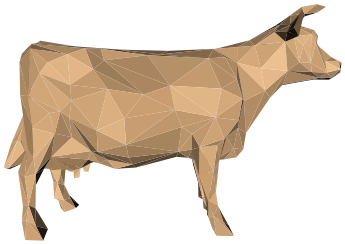
In addition to being efficient, the quadric error metric can also produce quality results. Figures 3.6 and 3.7 show a single example; all models are flat shaded to make their structure more apparent. The original cow model (a) has 5804 faces. As you can see, the features of the cow are preserved fairly well even with only 300 faces (e). Using my implementation of the algorithm, these approximations can be generated in about a half second on a 200 MHz PentiumPro machine. More complete details about the performance of the simplification algorithm are located in Chapter 7.

I have derived the quadric error metric from a conceptual metric based on sets of planes (§3.3.1). Evaluating the quadric Q for a plane is completely equivalent to evaluating the squared distance to that plane using the plane equation (3.1). However, the quadrics which are built up during simplification are not entirely identical to the sets of planes that would be constructed. When we form the union of two plane sets $P_1 \cup P_2$, any duplicate planes occur only once in the resulting set. But when we add two quadrics $Q_1 + Q_2$, duplicate planes remain. Any given plane may be counted up to three times. Each triangle in the original model contributes its plane to each of its vertices. As pairs are contracted, a single vertex may accumulate all three instances of this plane in its quadric. This multiple counting could be eliminated using an inclusion–exclusion rule

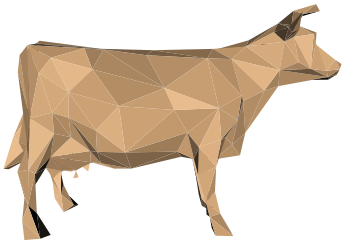


(a) 5804 faces

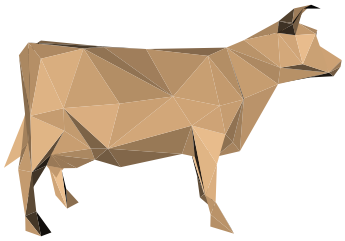
(b) 1000 faces



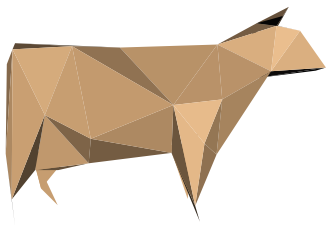
(c) 800 faces



(d) 500 faces

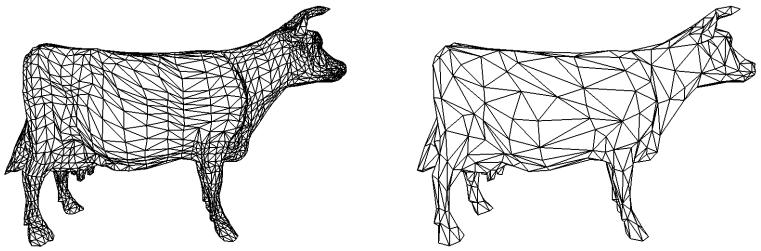


(e) 300 faces



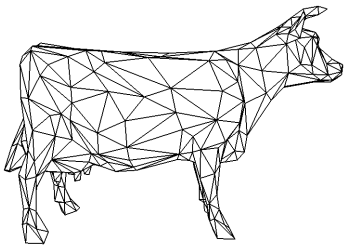
(f) 100 faces

Figure 3.6: Several approximations of cow model (a) constructed with the quadric error metric.

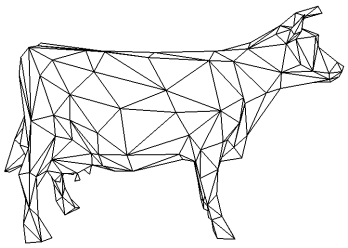


(a) 5804 faces

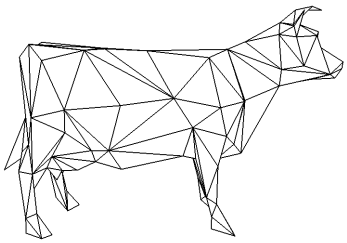
(b) 1000 faces



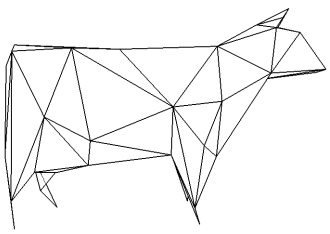
(c) 800 faces



(d) 500 faces



(e) 300 faces



(f) 100 faces

Figure 3.7: Wireframe versions of models in Figure 3.6.

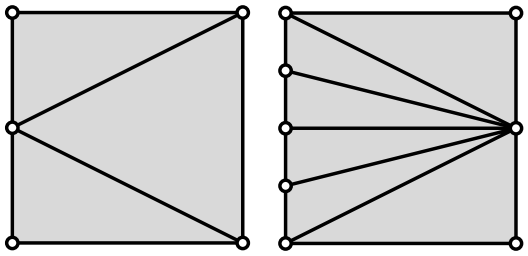
[107] for adding quadrics; however, it would complicate an otherwise simple algorithm. In fact, multiple counting is arguably beneficial. Consider the set of faces represented by the quadric Q; these faces form a connected region on the surface. Faces which intersect the region everywhere, along a single edge, and at a single vertex are triply, doubly, and singly counted, respectively. This has the effect of giving full weight to faces within the region of influence and of discounting faces along the boundary of that region. In any case, the E_{plane} metric, which explicitly tracks sets of planes, is really only a heuristic to begin with. The fact that the quadric metric E_Q is not exactly equivalent to it simply means that E_Q is a slightly different heuristic than E_{plane}.

This question of multiple counting leads us to a much more important issue. The summation of uniformly weighted quadrics, defined by the faces of the original model, is too dependent on the input tessellation. What is really needed is a more careful definition of the quadric error metric.

3.4.1 Normalized Quadric Metric

The form of the quadric error metric given by (3.11) is generally the most convenient for discussion. However, it is not the most appropriate in practice. It can be too heavily influenced by the structure of the mesh because every plane in the summation is given equal weight. For instance, this may produce undesirable results when some triangles are very large while others are very small.

Generally speaking, we are more concerned with the shape of the surface than the specific way in which it is tessellated. Suppose we take a region on the surface and tessellate it in two different ways, both of which have exactly the same geometry. Each face determines a single fundamental quadric. If



(a) (b)

Figure 3.8: Two different triangulations of a planar region.

we add up the quadrics for each face, we have the total quadric for the entire region. Consider the planar region shown in Figure 3.8. Since it is a plane,

each triangle has the same quadric Q. For triangulation (a), the total quadric[3] for the region is $9Q$; however, the total quadric for triangulation (b) is $18Q$. This is an undesirable result. Instead, the quadric metric should adhere to the following principle: the total quadric associated with a region should be the same, no matter which geometrically equivalent tessellation is chosen.

A natural way to try to achieve the desired tessellation invariance is to weight all the fundamental quadrics so that they always sum to the same result. Generalizing the earlier quadric definition, I will define a *weighted quadric* Q as

$$Q = (w\mathbf{A}, w\mathbf{b}, wc) \tag{3.12}$$

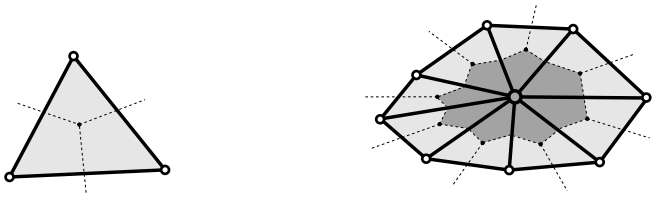
$$Q(\mathbf{v}) = \mathbf{v}^{\mathsf{T}}(w\mathbf{A})\mathbf{v} + 2(w\mathbf{b})^{\mathsf{T}}\mathbf{v} + (wc) \tag{3.13}$$

where w is a scalar weight factor. Suppose we are given a set of fundamental quadrics and associated weights for each. The quadric error metric, generalized to include weighting, becomes

$$E_Q = \sum_i w_i Q_i(\mathbf{v}) = \left(\sum_i w_i Q_i\right)(\mathbf{v}) \tag{3.14}$$

The unweighted error metric (3.11) assigns a uniform weight $w_i = 1$ to all fundamental quadrics. Weighting by angle, an occasionally suggested alternative [52], is also insufficient. In this case, the quadric contributed by a face to one of its corners is weighted by the angle θ formed by the edges of the face at this vertex. But this produces the same kind of tessellation-dependent results as uniform weighting. For example, the total quadrics for the meshes in Figure 3.8 would be (a) $3\pi Q$ and (b) $6\pi Q$.

Given a face f, we can imagine carving it into three fragments by splitting



(a) Face split into three fragments (b) Fragments adjoining a vertex

Figure 3.9: Each face is divided into three fragments (a), determining three fundamental quadrics. A vertex accumulates the fundamental quadrics for all adjacent fragments.

it at an internal point and along each edge (see Figure 3.9a). If Q is the quadric

[3] Recall that each fundamental quadric is counted three times.

determined by the plane of f, we can construct three fundamental quadrics w_1Q, w_2Q, w_3Q — one for each fragment. Each vertex $\mathbf{v}$ of f, instead of receiving an instance of the quadric Q, will receive the fundamental quadric w_iQ corresponding to the fragment of f adjacent to $\mathbf{v}$ (see Figure 3.9b). I propose to compute the fundamental weights w_i according to the area of the fragments — to use *area-weighted* quadrics. One straightforward way to define these fragments is to divide each face into three equal parts, and this is the method which I have adopted as the default policy in my implementation. We can optionally incorporate a form of angle weighting into this framework as well. A face of area w can be divided into fragments of size $w_i = w\theta_i/\pi$ where θ_i is the angle of the corner made at the relevant vertex.

Applying the area-weighted method to the example shown in Figure 3.8, we see that the total quadric for both triangulation (a) and (b) is wQ where w is the area of the region. The total quadric is independent of the tessellation, which is precisely the desired result. Area-weighted quadrics also have another very useful property. Consider subdividing a given tessellation more and more finely. In the limit as the area of individual triangles approaches 0, the summation of area-weighted quadrics becomes an integration of quadrics over a surface region. As we will see in Section 4.4, this will allow us to formulate the quadric error metric on differentiable surfaces.

Finally, note that (3.14) does not involve a division by $\sum_i w_i$. This averaging step would have the undesirable effect of making the error metric scale invariant. Consider two identical pyramids, one the size of a thimble and one the size of a large building. If these two pyramids were placed side by side on a level field, the larger one is obviously a much more significant feature. Therefore, its removal should incur a correspondingly higher error than the removal of the smaller pyramid. In other words, the error metric should be scale dependent.

3.4.2 Homogeneous Variant

In my original presentation of the quadric error metric [68], I used an alternate notation. We can treat the quadric Q as a homogeneous matrix where

$$\mathbf{Q} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^\mathsf{T} & c \end{bmatrix} \tag{3.15}$$

Given this matrix, we can evaluate $Q(\mathbf{v})$ using the quadratic form

$$Q(\mathbf{v}) = \tilde{\mathbf{v}}^\mathsf{T}\mathbf{Q}\tilde{\mathbf{v}} \qquad \text{where} \qquad \tilde{\mathbf{v}} = [\mathbf{v}\ 1]^\mathsf{T} = [x\ y\ z\ 1]^\mathsf{T} \tag{3.16}$$

An area-weighted quadric would simply be the scaled matrix $w\mathbf{Q}$.

This homogeneous quadric notation makes another useful property of the quadric error metric apparent. A model can be transformed by a linear transformation, after the initial quadrics have been computed, and the quadrics can

be transformed appropriately. Suppose $\mathbf{R}$ is a linear transformation applied to the surface, and that all quadrics Q_i were computed prior to the transformation of the surface. To apply a quadric to a transformed vertex, we can map the vertex back to the pre-transformed space:

$$Q(v) = (\mathbf{R}^{-1}\tilde{\mathbf{v}})^{\mathsf{T}}\mathbf{Q}(\mathbf{R}^{-1}\tilde{\mathbf{v}})$$

which we can rewrite as

$$Q(v) = \tilde{\mathbf{v}}^{\mathsf{T}}\left((\mathbf{R}^{-1})^{\mathsf{T}}\mathbf{Q}(\mathbf{R}^{-1})\right)\tilde{\mathbf{v}}$$

This gives us the transformation rule for quadrics

$$\mathbf{Q} \rightarrow (\mathbf{R}^{-1})^{\mathsf{T}}\mathbf{Q}\mathbf{R}^{-1}$$

which is the standard quadric surface transformation rule [15].

On the whole, I have found this homogeneous representation less convenient than the form presented earlier. It requires us to move back and forth between regular and homogeneous coordinates. It also leads to a slightly less efficient implementation because all matrix operations involve 4×4 rather than 3×3 matrices. For matrix inversion in particular, this can lead to a measurable increase in running time.

3.5 Vertex Placement Policies

When considering the contraction of an edge $(\mathbf{v}_i, \mathbf{v}_j)$, we need some way of choosing the target position $\bar{\mathbf{v}}$. There are two primary policies to choose from, and the choice between them must be made with the intended application in mind. We must trade space efficiency against approximation quality.

Subset placement is the simplest strategy that we can adopt. We simply select one of the endpoints as the target position. In other words, we will contract one endpoint into the other. To choose between endpoints, we merely need to find the smaller of $Q(\mathbf{v}_i)$ and $Q(\mathbf{v}_j)$. Under this policy, any approximation which we produce will use a subset of the original vertices in their original positions.

We can often produce better approximations using *optimal placement*. For a given quadric Q, we can try to find the point $\bar{\mathbf{v}}$ such that $Q(\bar{\mathbf{v}})$ is minimal. Of course, $\bar{\mathbf{v}}$ is optimal only in the sense that it minimizes $Q(\bar{\mathbf{v}})$; it does not imply an optimal approximation. Since $Q(\bar{\mathbf{v}})$ is quadratic, finding its minimum is a linear problem; the minimum occurs where $\partial Q/\partial x = \partial Q/\partial y = \partial Q/\partial z = 0$. Taking partial derivatives, we see that the gradient of Q is

$$\nabla Q(\mathbf{v}) = 2\mathbf{A}\mathbf{v} + 2\mathbf{b}$$

Solving for $\nabla Q(\mathbf{v}) = 0$, we find that the optimal position is

$$\bar{\mathbf{v}} = -\mathbf{A}^{-1}\mathbf{b} \tag{3.17}$$

and its error is

$$Q(\bar{\mathbf{v}}) = \mathbf{b}^{\mathsf{T}}\bar{\mathbf{v}} + c = -\mathbf{b}^{\mathsf{T}}\mathbf{A}^{-1}\mathbf{b} + c \tag{3.18}$$

These are instances of well-known formulas for the minimum of a positive definite quadratic form [183, pg. 347].

Naturally, a unique optimal position may not exist. If $\mathbf{A}$ is singular, its inverse does not exist, and we cannot solve for $\bar{\mathbf{v}}$ using (3.17). In this situation, the set of points for which $Q(v)$ is minimal forms either a line or a plane. This may occur, for instance, when all the planes in Q are parallel (see Section 4.1 for more details). For the cases in which $\mathbf{A}$ is singular, we need some fallback strategy for selecting the position $\bar{\mathbf{v}}$. In my own implementation, I use the following three phase approach:

1. Attempt to compute $\bar{\mathbf{v}}$ using (3.17).

2. If $\mathbf{A}$ is singular, find the optimal position along the line segment $(\mathbf{v}_i, \mathbf{v}_j)$.

3. If this is not unique, select the better of $\mathbf{v}_i$ and $\mathbf{v}_j$ (i.e., subset placement).

In practice, some care must be taken in detecting whether or not $\mathbf{A}$ is singular. Theoretically, a singular matrix has determinant $\det \mathbf{A} = 0$ [183]. However, due to the limits of floating point precision, we must test whether a matrix is *nearly* singular. If $\mathbf{A}$ is nearly singular, the position of $\bar{\mathbf{v}}$ will tend towards infinity. One simple strategy is to regard as singular any matrix for which $\det \mathbf{A} \approx 0$, based on a suitable threshold for nearness to 0. However, this does not correlate well with the near singularity of $\mathbf{A}$; a more careful approach relies on measuring the norm of $\mathbf{A}$ [71].

Whether optimal or subset placement is preferable depends on the intended application. Optimal placement will tend to produce approximations which fit the original more closely. The resulting meshes also tend to be better shaped — triangles are more equilateral and their areas are more uniform. Consequently, this is the best choice for generating fixed approximations of an original. However, if we are more interested in producing an incremental representation (§7.1) such as progressive meshes [90], subset placement may be preferable. The overall fit of the models will be somewhat inferior, but we can save significantly on storage. With optimal placement, we must store delta records with each contraction to encode the new vertex position. Using subset placement, we can eliminate such overhead entirely. By adopting the convention that contractions are always ordered such that the result of contracting the pair $(\mathbf{v}_i, \mathbf{v}_j)$ is to move $\mathbf{v}_j$ to the position of $\mathbf{v}_i$, we require no additional storage to specify the new vertex position. This issue of overhead becomes even more important when surfaces possess material properties (Chapter 5). Since this overhead grows linearly with the number of attribute elements, the space savings of subset placement can become substantial.

3.6 Discontinuities and Constraints

Discontinuities of a model, such as creases, open boundaries, and borders between differently colored regions, are often among its most visually significant features. Therefore, their preservation is critical for producing quality approximations. The fundamental quadric algorithm can already handle shape discontinuities (e.g., creases), and it can easily accommodate boundary curves as well.

Surface shape discontinuities (where there is only C^0 continuity) are implicitly preserved by the error metric as described. For example, consider the sharp edges of a cube. A point on the edge of a cube will have contributing planes from both adjoining faces of the cube. Since these planes are perpendicular, the cost of moving the point along the edge is much lower than moving it away from the edge. Consequently, the algorithm will be strongly biased against altering the shape of these edges.

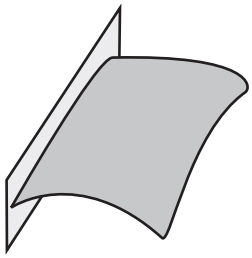
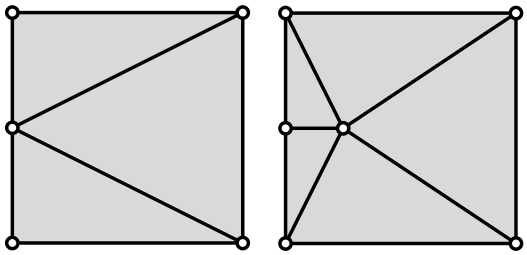


Figure 3.10: Sample boundary constraint plane. Every edge along the boundary defines a single constraint plane.

In contrast, the basic algorithm ignores boundary curves. Fortunately, we can easily incorporate boundary constraints into the existing framework. During initialization, my implementation flags all boundary edges. For each face adjacent to a given boundary edge, it computes a plane perpendicular to the face through the edge. The perpendicular plane defines a boundary *constraint plane* (see Figure 3.10). We can form a quadric for this plane, just as with a regular face plane. To form a constraint from this quadric, I weight it using a large penalty factor[4], and add it into the initial quadric for each of the endpoints. The primary attraction of this approach is that it allows the iterative core of the algorithm to preserve boundaries without any special-case logic. Once the constraint planes are added to the initial quadrics, the algorithm proceeds in exactly the same manner as before.

When using area-weighted quadrics, the constraint quadrics must be properly weighted. Perhaps the most obvious approach is to weight each constraint quadric by the area of the face attached to the boundary edge. However, this makes the resulting quadric dependent on the tessellation near the boundary.

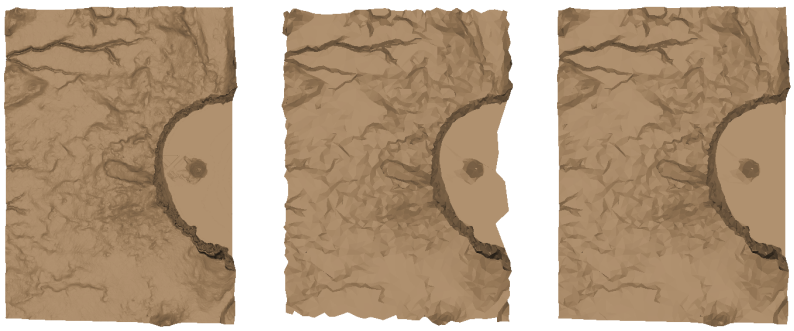
[4] My implementation uses a default penalty factor of 1000.



(a) (b)

Figure 3.11: Two different triangulations of a planar region.

Consider the two triangulations shown in Figure 3.11. Suppose that the left-most side of the box is an open boundary. In both triangulations, there are two identical edges along this boundary. However, the area of the faces adjoining these edges in triangulation (b) is much smaller than in triangulation (a). Consequently, if the constraint quadrics are weighted by face area, they would be assigned a substantially higher weight in triangulation (a). The convention which I propose is to weight the quadric by the squared length of the boundary edge. For the example in Figure 3.11, this would result in equal quadrics for both triangulations. Squared edge length is preferable to length because the squared length has the same units of measure as area.



(a) Original model (b) Unconstrained (c) Constrained

Figure 3.12: Crater Lake simplified with and without boundary constraints.

Figure 3.12a shows a model of the western half of Crater Lake; note the open boundary surrounding the terrain. The example shown in Figure 3.13, a

(a) Original model (b) Unconstrained (c) Constrained

Figure 3.13: Simplification of a femur. Note how the boundary of the unconstrained approximation has receded.

section of a femur bone, has an open boundary on the left side. Without any modification, the basic simplification algorithm will produce approximations such as shown in Figures 3.12b and 3.13b. The boundaries of the objects have been significantly eroded. This is clearly unacceptable. Figures 3.12c and 3.13c illustrate the result of using boundary constraints. They each have the same number of faces as the unconstrained approximations; however, they are clearly superior approximations because the boundaries have been properly preserved.

This technique can also be applied to arbitrary contours through the surface. For instance, a terrain approximation system might like to introduce constraints along the banks of a river or along the contour of a road to preserve these important features. Boundaries may also occur in discrete surface attributes. Consider a map 4-colored by country. Each edge dividing two faces of different colors can be marked as a boundary. This would cause the algorithm to try to faithfully preserve the borders between separate regions. Because of the nature of the error metric, we may also want to treat edges with a very small dihedral angle as discontinuities (see Section 4.5.2 for details).

Using the same kind of mechanism, we can also apply point constraints. Given a point $\mathbf{p}$, we can construct a constraint quadric

$$Q = (\mathbf{I}, -\mathbf{p}, \mathbf{p}^{\mathsf{T}}\mathbf{p}) \tag{3.19}$$

where $\mathbf{I}$ is the 3×3 identity matrix. The value $Q(\mathbf{v})$ measures the squared distance of $\mathbf{v}$ to the point $\mathbf{p}$. A constraint of this sort might be useful for simplifying a model such as the cow shown in Figure 3.6. We could apply a point constraint at the tips of the horns which would have the effect of pinning the tips in place. It seems unlikely that the addition of this kind of constraint could be completely automated, but it provides a useful control mechanism for a user to specify important extremal points.

The primary advantage of this constraint approach is that all the special processing occurs during initialization. Once the initial quadrics at the vertices have been constructed, iterative simplification proceeds as usual. However, there are naturally limits to this technique. These constraints merely bias the simplification process, rather than providing any hard guarantees on boundary preservation. We must also make the assumption that boundaries are reasonably sparse in comparison to the total surface. If, for instance, every face is assigned

a slightly different color, or if we have a triangulated regular grid where every other triangle is a hole, the results will not be good. There will be too many constraints; it will become difficult for the algorithm to discriminate between the available edges to contract.

3.7 Consistency Checks

The quadric error metric is the means by which my algorithm selects contractions to perform and the location at which to place the resulting vertex. However, a given contraction may potentially introduce undesirable inconsistencies or degeneracies into the mesh. We can combat this problem by applying a set of consistency checks to a proposed contraction. If it fails one of these checks, we can either add a large penalty factor (as with boundary constraints) or discard the contraction entirely. If we only penalize "bad" contractions, there is no guarantee that it will not be performed if all other contractions have higher penalties. However, it does ensure that the algorithm will still make progress even when all contractions are considered "bad". This is particularly relevant if we are applying several different checks. On the other hand, discarding contractions will prevent inconsistencies from occurring, but we may not necessarily be able to make progress. Discarded contractions will only be reconsidered when their local neighborhood is changed and they are consequently reevaluated.

The most common consistency check is related to the problem of mesh inversion. Consider the contraction shown in Figure 3.14. For this particular choice

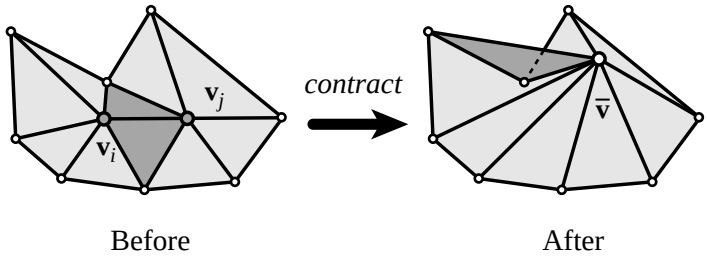
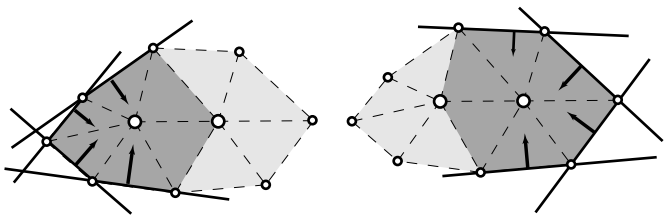


Figure 3.14: An edge contraction which causes the mesh to fold over on itself.

of the position $\bar{\mathbf{v}}$, the mesh folds over onto itself (the darkened area). One popular approach to detecting this situation is to examine the normals of the faces adjoining $\mathbf{v}_i$ and $\mathbf{v}_j$ before and after the contraction [68, 90, 158, 114]. If a face's normal changes by more than some significant threshold, we can regard this face as having "flipped" as a result of the contraction. A contraction fails this check if any of the local faces flip. While this approach does, in principle, prevent fold-over, it requires us to pick a suitable threshold value.

In my implementation, I have chosen to use a more careful check [23, 52] which appears to perform more reliably in practice. For every face around $\mathbf{v}_i$,



(a) Zone around $\mathbf{v}_i$ (b) Zone around $\mathbf{v}_j$

Figure 3.15: Perpendicular planes through the edge loop around $\mathbf{v}_1, \mathbf{v}_2$ define a zone in which $\bar{\mathbf{v}}$ must fall to prevent fold-over.

excluding the faces shared with $\mathbf{v}_j$, there is an edge opposite $\mathbf{v}_i$. If we place a plane perpendicular to the face through the edge, the position $\bar{\mathbf{v}}$ must lie on the same side of the plane as the vertex $\mathbf{v}_i$. The same criterion applies to the faces surrounding $\mathbf{v}_j$. Figure 3.15 illustrates these planes; the arrows indicate the zone in which $\bar{\mathbf{v}}$ must fall.

There are also other kinds of checks which we can apply to proposed contractions. Recall that a single edge contraction can alter the topology of the object, as illustrated in Figure 3.3. For some applications, the topology of the surface should not be changed. We can apply additional checks to prevent such changes [52]. Sliver triangles, ones which have very small angles, are undesirable in some applications such as finite element analysis. Guéziec [76, 77] suggested a measure of triangle compactness

$$\gamma = \frac{4\sqrt{3}w}{l_1^2 + l_2^2 + l_3^2} \tag{3.20}$$

where the l_i are the lengths of the edges and w is the area of the triangle. This will assign a compactness of 1 to an equilateral triangle and 0 to a triangle whose vertices are colinear. Using this heuristic, we can penalize contractions which produce triangles whose compactness γ falls below some threshold.

3.8 Alternative Contraction Primitives

Vertex pair contraction is only one instance of a simplification primitive compatible with the quadric error metric. Because the quadric error metric is based solely on adding together quadrics associated with individual vertices of the model, it can be used with generalized contraction operations. Given any set of vertices $\{\mathbf{v}_{i_1}, \mathbf{v}_{i_2}, \ldots, \mathbf{v}_{i_k}\}$, the quadric for this set is $Q = \sum_j Q_{i_j}$, and we can select an optimal vertex position using Equation 3.17. Thus, for example, the quadric error metric could be used in conjunction with clustering algorithms such as Rossignac–Borrel [161] and Low–Tan [128].

The most natural alternative to pair contraction is face contraction, illustrated in Figure 3.16. The algorithm which I described in Section 3.2 can easily

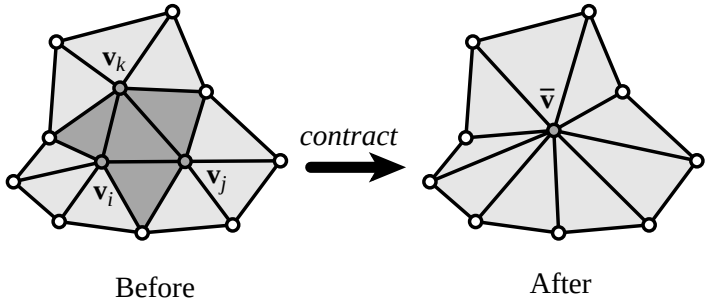


Figure 3.16: Face $(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k)$ is contracted. The darkened triangles become degenerate and are removed.

be adapted to use face, rather than pair, contraction. The resulting method would be more similar to the face-based algorithm described by Hamann [80] and Gieng, Hamann, *et al.* [70]. As we will see in Chapter 6, this face-based algorithm is slightly more efficient, but produces inferior results, when compared to the pair-based algorithm.

3.9 Summary of Algorithm

In Section 3.2 I described the high-level simplification framework upon which my algorithm is based. Having described the specific details of my algorithm, I can now present the following more detailed outline:

1. Select as candidates any pair $(\mathbf{v}_i, \mathbf{v}_j)$ such that:
   (a) $(\mathbf{v}_i, \mathbf{v}_j)$ is an edge, or
   (b) $(\mathbf{v}_i, \mathbf{v}_j)$ is not an edge and $\|\mathbf{v}_i - \mathbf{v}_j\| < \tau$.
2. Allocate a quadric Q_i for each vertex $\mathbf{v}_i$.
3. For each face $f_i = (j, k, l)$, compute a quadric Q_i (3.10). Add this fundamental quadric to the vertex quadrics Q_j, Q_k, and Q_l, weighted appropriately (§3.4.1).
4. For each candidate pair $(\mathbf{v}_i, \mathbf{v}_j)$:
   (a) Compute $Q = Q_i + Q_j$.
   (b) Select a target position $\bar{\mathbf{v}}$ (§3.5).
   (c) Apply consistency checks and penalties (§3.7).
   (d) Place pair in heap keyed on cost $Q(\bar{\mathbf{v}})$.

5. Repeat until the desired approximation is reached:
   (a) Remove the pair $(\mathbf{v}_i, \mathbf{v}_j)$ of least cost from the heap.
   (b) Perform contraction $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$.
   (c) Set $Q_i = Q_i + Q_j$.
   (d) For each remaining pair $(\mathbf{v}_i, \mathbf{v}_k)$, compute target position and cost as in step 4; update heap.

Having completed the outline of the basic simplification algorithm, I will devote the next chapter to the analysis of the quadric error metric. In Chapter 5, we will see that by using a generalized quadric error metric, this algorithm can also accommodate surfaces with material properties as well. An analysis of the performance of this algorithm, including running time, memory consumption, and empirical results, can be found in Chapter 6. Finally, Appendix A contains specific details on my own implementation of this algorithm.

Chapter 4

Analysis of Quadric Metric

The simplification algorithm described in Chapter 3 is built around two core components. First is the process of iterative vertex pair contraction. Several other algorithms (§2.4.3) have been developed using essentially the same greedy framework. The second component, and where my algorithm differs most substantially from other related methods, is the quadric error metric. In this chapter, I will examine the properties of this error metric in much greater detail. As we will see, the quadrics which my algorithm uses have several interesting geometric properties. Most notably, they characterize the local shape of the surface, a notion I will formalize by demonstrating their connection with surface curvature.

4.1 Geometric Interpretation

When describing the simplification algorithm, I derived quadrics algebraically (§3.4) from the formula for the sum of squared distances of a point to a set of planes. While this is the most convenient formulation for computational purposes, it does not provide us with much intuition about what these quadrics measure and how they behave. For that, we need to explore the geometric interpretation of quadrics.

4.1.1 Quadric Isosurfaces

Recall that a given quadric $Q = (\mathbf{A}, \mathbf{b}, c)$ assigns an error value $Q(\mathbf{v}) = \mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{v} + 2\mathbf{b}^\mathsf{T}\mathbf{v} + c$ to every point $\mathbf{v} \in \mathbf{R}^3$. Consider the level surface $Q(\mathbf{v}) = \epsilon$. This is the set of all points whose error with respect to Q is ϵ. Using the homogeneous notation for quadrics (§3.4.2), the equation for this isosurface is the quadratic form $\tilde{\mathbf{v}}^\mathsf{T}\mathbf{Q}\tilde{\mathbf{v}} = \epsilon$. By the definition of a quadric (§3.4), $\mathbf{Q}$ is a 4×4 symmetric

matrix. Therefore, the level surface is defined by the equation

$$\tilde{\mathbf{v}}^{\mathsf{T}}\mathbf{Q}\tilde{\mathbf{v}} = q_{11}x^2 + 2q_{12}xy + 2q_{13}xz + 2q_{14}x + q_{22}y^2 + 2q_{23}yz + 2q_{24}y + q_{33}z^2 + 2q_{34}z + q_{44} = \epsilon \quad (4.1)$$

This is a second order equation in x, y, z and it defines a *quadric surface*. Surfaces of this type include ellipsoids, paraboloids, hyperboloids, and planes. Blinn [15] surveys the main algebraic properties of quadric surfaces.

The connection between quadric surfaces and squared distances to sets of planes was first recognized by Fermat, who stated the following theorem [33]:

> If a point move in such a way that the sum of the squares in given directions from some given planes bears a fixed ratio to the sum of the squares of its distances in given directions from certain other given planes, its locus is a spheroid or conoid.

A classical result in linear algebra and projective geometry [135] states that a quadratic form A in three variables x, y, z can be written as a weighted sum of the squares of three independent linear forms $A = a_1L_1^2 + a_2L_2^2 + a_3L_3^2$. Since distance to a plane is a linear form, the sum of the squares of three such forms determines a quadratic form. Of course, the quadrics used in my simplification algorithm are the sum of squares of many more than three linear forms.

A quadric $\mathbf{Q}$ generated during simplification is symmetric by construction. It is also *positive semi-definite*[1] [183]. In other words, all its eigenvalues λ_i are non-negative ($\lambda_i \geq 0$). Consider an eigenvector $\mathbf{x} \neq 0$ of $\mathbf{Q}$. Since it is an eigenvector, there is some eigenvalue λ for which $\mathbf{Q}\mathbf{x} = \lambda\mathbf{x}$. And by the definition of $\mathbf{Q}$ (§3.4.2), we know that $\mathbf{x}^{\mathsf{T}}\mathbf{Q}\mathbf{x} \geq 0$; a sum of squared (real) distances can never be negative. Using the fact that $\mathbf{x}$ is an eigenvector, we can rewrite this condition as $\mathbf{x}^{\mathsf{T}}\mathbf{Q}\mathbf{x} = \mathbf{x}^{\mathsf{T}}(\lambda\mathbf{x}) = \lambda(\mathbf{x}^{\mathsf{T}}\mathbf{x}) \geq 0$. Since $\mathbf{x} \neq 0$, $\mathbf{x}^{\mathsf{T}}\mathbf{x} > 0$ and thus we can conclude that $\lambda \geq 0$.

Because $\mathbf{Q}$ is positive semi-definite, its isosurfaces will be (potentially degenerate) ellipsoids. The principal axes of these ellipsoids are defined by the eigenvectors of the matrix $\mathbf{A}$ and an ellipsoid's extent one of these axes is determined by the corresponding eigenvalue. The ellipsoids are degenerate, or "open", when some eigenvalues of $\mathbf{A}$ are 0; in other words, when $\mathbf{A}$, and hence $\mathbf{Q}$, are singular. If all the planes represented by Q are parallel to the same line, then exactly one eigenvalue is 0, and the isosurfaces are infinite cylinders centered about this line. If Q represents a set of parallel planes, then two eigenvalues are 0, and the isosurfaces are parallel planes. The equation for finding the optimal vertex placement (§3.5) corresponds to finding the center of the ellipsoid isosurfaces. When the ellipsoids are degenerate (i.e., $\mathbf{A}$ is non-invertible), the "center" is either a line or a plane.

The isosurfaces of the fundamental quadric defined by a single plane P are pairs of planes parallel to P. The matrix $\mathbf{A}$ of this quadric has a single non-zero eigenvalue, and the corresponding eigenvector is the normal of the plane P. As

[1] Or, equivalently, *non-negative definite*.

further quadrics are added, the isosurfaces will become true ellipsoids when $\mathbf{A}$ is non-singular. This will occur when the normals represented by the quadric span 3-D space. Furthermore, once a quadric produced by my simplification algorithm has become non-degenerate, it will remain so.

Figure 4.1 shows quadric isosurfaces computed for three simplified models. Each of the quadrics shown is centered about a vertex of the current approximation. If a vertex is moved within its ellipsoid, the quadric error assigned to that vertex will be bounded by ϵ.

Notice that the quadrics characterize the local shape of the surface. Consider the example of the cube model pictured in Figure 4.1. For vertices along the edges, the ellipsoids are cigar shaped, elongated in the direction of the crease. In contrast, on the sides of the cube, the quadrics are very thin and roughly circular, like pancakes. The quadric at the corner is ball-shaped. Similar features occur on the blobby model and on the bunny. Intuitively, we can see that the quadrics are elongated in directions of low curvature and thin in directions of high curvature. I will discuss this insight in greater detail in Section 4.2.2 and present a more formalized analysis in Section 4.4.

4.1.2 Visualizing Isosurfaces

Before continuing with the analysis of quadrics, I would like to briefly outline the techniques I use to display them. Given a quadric Q and an error value ϵ, we would like to draw the surface $\mathbf{x}^\mathsf{T}\mathbf{Q}\mathbf{x} = \epsilon$. I have used two methods, both based on factorization of the quadric matrix.

The primary technique which I use is based on the Cholesky decomposition [183, 148]. If $\mathbf{Q}$ is a symmetric positive definite matrix, we can factor it into $\mathbf{Q} = \mathbf{R}^\mathsf{T}\mathbf{R}$, where $\mathbf{R}$ is an upper triangular matrix. And we know that $\mathbf{Q}$ is symmetric positive definite exactly when its isosurfaces are ellipsoids. By substituting this factorization into the standard quadric equation we get

$$\mathbf{x}^\mathsf{T}\mathbf{Q}\mathbf{x} = \mathbf{x}^\mathsf{T}(\mathbf{R}^\mathsf{T}\mathbf{R})\mathbf{x} = (\mathbf{R}\mathbf{x})^\mathsf{T}(\mathbf{R}\mathbf{x})$$

Now, let $\mathbf{y} = \mathbf{R}\mathbf{x}$. Note that $\mathbf{y}^\mathsf{T}\mathbf{y} = \epsilon$ is the equation of a sphere. The matrix $\mathbf{R}$ transforms the ellipsoid isosurface into a sphere of radius ϵ; thus, $\mathbf{R}^{-1}$ transforms this sphere into an ellipsoid isosurface. This provides a convenient technique for rendering isosurfaces under systems like OpenGL [197]. We can transform a sphere of radius ϵ into the appropriate isosurface by applying the linear transformation $\mathbf{R}^{-1}$.

An alternative method is to use the spectral decomposition [183]. Assuming that $\mathbf{A}$ is symmetric positive definite, which is the most important case for display, it can be factored into $\mathbf{A} = \mathbf{R}^\mathsf{T}\mathbf{D}\mathbf{R}$ where $\mathbf{D}$ is a diagonal matrix of eigenvalues and the rows of $\mathbf{R}$ are the corresponding eigenvectors. Using the matrices $\mathbf{R}^\mathsf{T}$ and $\mathbf{D}$ we can rotate and scale a unit sphere into an ellipsoid isosurface. We must independently find the center and perform an additional translation on the ellipsoid. This technique is slower than the Cholesky method. However, it has one notable advantage. The x, y, z axes of the unit sphere will be transformed into the principal axes of the ellipsoid.

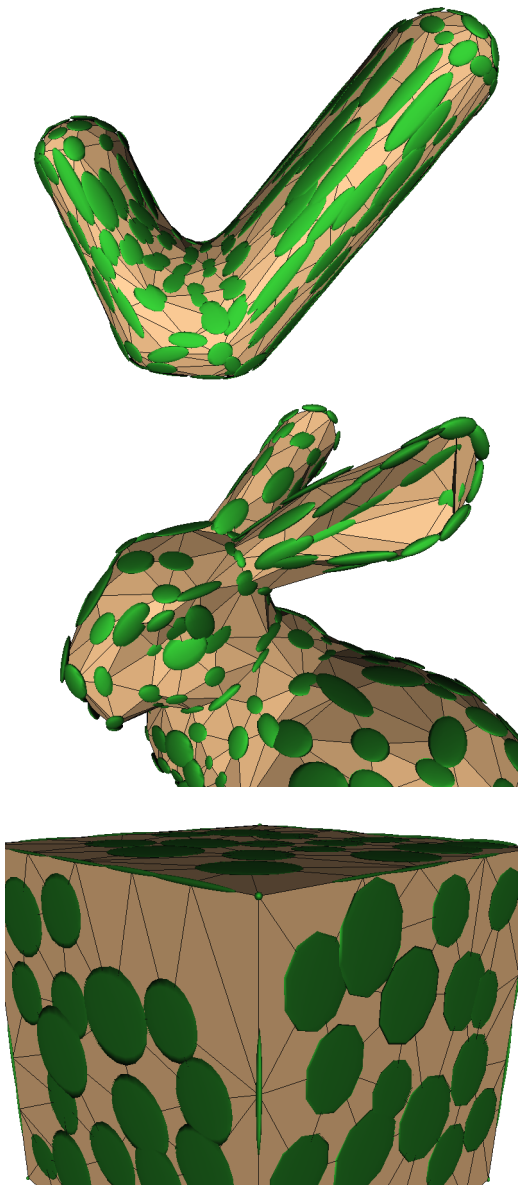


Figure 4.1: Quadric error isosurfaces on three simplified models: blobby "V", bunny, and the corner of a cube. Note how the ellipsoids follow the local shape of each surface.

4.1.3 Volumetric Quadric Construction

My standard definition of the quadric error metric, developed in Section 3.4, is formulated in terms of measuring the sum of squared distances to some set of planes. As we have seen in Section 4.1, the quadrics themselves can be interpreted as a set of quadric isosurfaces in 3-D space. Now, I will describe a useful alternative interpretation of the quadric metric as a volumetric measure of the difference between the approximation and the original surface. This construction is inspired by the "volume optimization" metric recently introduced by Lindstrom and Turk [124], which I will demonstrate is equivalent to a form of my quadric metric.

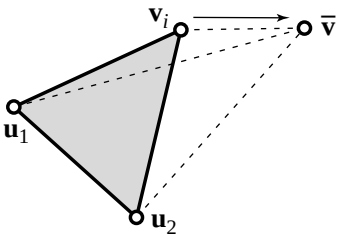


Figure 4.2: As $\mathbf{v}_i$ moves towards $\bar{\mathbf{v}}$, the triangle attached to it sweeps out a tetrahedron in space.

Suppose that we are about to perform an edge contraction $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$. Let us consider a triangle $T = (\mathbf{u}_1, \mathbf{u}_2, \mathbf{v}_i)$ where $\mathbf{u}_k \neq \mathbf{v}_j$. As Lindstrom and Turk observed, while the vertex $\mathbf{v}_i$ moves from its original position to the final position $\bar{\mathbf{v}}$, the triangle T sweeps out a tetrahedron in space (see Figure 4.2).

We can easily derive a formula for the volume of this tetrahedron. The unscaled face normal of T is

$$\mathbf{m} = (\mathbf{u}_1 - \mathbf{v}_i) \times (\mathbf{u}_2 - \mathbf{v}_i). \tag{4.2}$$

and the area of T is half the length of $\mathbf{m}$

$$w = \frac{1}{2}\|\mathbf{m}\| \tag{4.3}$$

We can also compute the perpendicular distance of $\bar{\mathbf{v}}$ from the plane of T by the equation

$$h = (\bar{\mathbf{v}} - \mathbf{v}_i)^{\mathsf{T}} (\mathbf{m}/\|\mathbf{m}\|) \tag{4.4}$$

The volume of the swept tetrahedron is simply

$$V = \frac{1}{3} wh \tag{4.5}$$

$$= \frac{1}{6} (\bar{\mathbf{v}} - \mathbf{v}_i)^{\mathsf{T}} \mathbf{m} \tag{4.6}$$

Now, we can write the squared volume of the swept tetrahedron as

$$V^2 = \frac{1}{36}\left((\bar{\mathbf{v}} - \mathbf{v}_i)^\mathsf{T}\mathbf{m}\right)\left((\bar{\mathbf{v}} - \mathbf{v}_i)^\mathsf{T}\mathbf{m}\right) \tag{4.7}$$

$$= \frac{1}{36}(\bar{\mathbf{v}} - \mathbf{v}_i)^\mathsf{T}(\mathbf{m}\mathbf{m}^\mathsf{T})(\bar{\mathbf{v}} - \mathbf{v}_i) \tag{4.8}$$

$$= \frac{\|\mathbf{m}\|^2}{36}\left(\bar{\mathbf{v}}^\mathsf{T}\mathbf{A}\bar{\mathbf{v}} + 2\mathbf{b}^\mathsf{T}\bar{\mathbf{v}} + c\right) \tag{4.9}$$

where

$$\mathbf{A} = \mathbf{m}\mathbf{m}^\mathsf{T} \tag{4.10}$$

$$\mathbf{b} = -\mathbf{m}\mathbf{m}^\mathsf{T}\mathbf{v}_i \tag{4.11}$$

$$c = \mathbf{v}_i{}^\mathsf{T}(\mathbf{m}\mathbf{m}^\mathsf{T})\mathbf{v}_i \tag{4.12}$$

This is precisely the same as a fundamental quadric constructed using the definition of Section 3.4.1 weighted by a factor of $w^2/9$.

If we weight each fundamental quadric Q by $w^2/9$, where w is the area of the contributing face T, we obtain a quadric such that $Q(\bar{\mathbf{v}})$ is the squared volume of the tetrahedron formed by connecting $\bar{\mathbf{v}}$ to T. Quadrics which are the sum of a set of fundamental quadrics measure the sum of these squared volumes. Consider a vertex $\mathbf{v}$ on an approximation. It corresponds to some set of faces on the original surface. We can connect $\mathbf{v}$ to each face in this set, forming a set of tetrahedra. The error $Q(\mathbf{v})$ measures the sum of squared volumes of these tetrahedra. Thus, by placing $\mathbf{v}$ at a position which minimizes $Q(\mathbf{v})$, we are minimizing this sum of squared volumes. Note that this is not the same as minimizing the squared volume between the approximation and the original because the tetrahedra in this set will, in general, overlap with each other.

4.1.4 Quadratic Distance Metrics

The quadric error metric is algebraically very closely related to a number of previous distance metrics. As demonstrated by (4.1), the quadric metric belongs to the well-studied class of functions known as *quadratic forms* [36]. A quadratic form is a second order polynomial in k variables and can always be written in the form $\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}$ where $\mathbf{x}$ is a k-vector and $\mathbf{A}$ is a $k \times k$ symmetric positive definite matrix. To conform more closely to the traditional notation of quadratic forms, we can recast the quadric metric in the centered form

$$Q(v) = (\mathbf{v} - \bar{\mathbf{v}})^\mathsf{T}\mathbf{A}(\mathbf{v} - \bar{\mathbf{v}}) \tag{4.13}$$

provided that $\mathbf{A}$ is in fact positive definite, and hence a unique center $\bar{\mathbf{v}} = -\mathbf{A}^{-1}\mathbf{b}$ exists. Distance metrics based on quadratic forms have several convenient properties, and they have been used in a variety of contexts.

The squared length of a vector in standard Euclidean geometry is given by the inner product $\mathbf{x}^\mathsf{T}\mathbf{x}$. A more general formulation is to define the squared length of a vector using quadratic form $\mathbf{x}^\mathsf{T}\mathbf{M}\mathbf{x}$. If $\mathbf{M}$ is the identity matrix,

it reduces to the Euclidean distance metric. The matrix $\mathbf{M}$ is referred to as a *Riemannian metric tensor* and $\sqrt{\mathbf{x}^\mathsf{T}\mathbf{M}\mathbf{x}}$ is a Riemannian metric. The field of Riemannian geometry [111, 115] is based on investigating the structure of surfaces where distances are measured using this type of metric.

Quadratic forms also play an important role in statistical modeling. Suppose that $\mathbf{x}$ is a k-element vector-valued random variable, and that n observations $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are given. Let $\bar{\mathbf{x}}$ be the mean of $\mathbf{x}$

$$\bar{\mathbf{x}} = E[\mathbf{x}] = \frac{1}{n}\sum_k \mathbf{x}_k \tag{4.14}$$

The *sample covariance matrix* [50, 101] $\mathbf{Z}$ of $\mathbf{x}$ is defined by

$$\mathbf{Z} = \frac{1}{n-1}\sum_i (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})^\mathsf{T} \tag{4.15}$$

Each diagonal element z_{ii} of this matrix measures the variance of x_i, the components of $\mathbf{x}$. The non-diagonal entry z_{ij} is the covariance of x_i and x_j. If $z_{ij} = 0$, the components x_i and x_j are statistically independent. The isosurfaces of a Gaussian distribution with covariance matrix $\mathbf{Z}$ are hyperellipsoids in k-space. Each hyperellipsoid is the set of all points whose density is constant, and it is determined by the isosurfaces of the quadratic form $(\mathbf{x}-\bar{\mathbf{x}})^\mathsf{T}\left(\mathbf{Z}^{-1}\right)(\mathbf{x}-\bar{\mathbf{x}})$. This form, which is the squared *Mahalanobis distance* of the vector $\mathbf{x}$ from the mean $\bar{\mathbf{x}}$, has a structure which is clearly related to the quadric error metric (4.13). In particular, note that in both cases the matrix of the quadratic form is the sum of a set of outer products. In Section 4.2.2, I will return to the connection between quadrics and covariance matrices as a means of analyzing the shape of the quadric error isosurfaces.

4.2 Quadrics and Least Squares Fitting

The interpretation of the quadric metric as a family of ellipsoidal isosurfaces (§4.1.1) is intuitively appealing. It is particularly helpful because it allows us to easily visualize the metric during simplification. As I suggested earlier, it appears that there is some connection between surface shape and the shape of these isosurfaces. In this section, I will explore this connection in greater detail. I will derive the quadric-optimal position $\bar{\mathbf{v}}$ using the method of least squares fitting, and then cast the shape of the ellipsoidal isosurfaces in terms of the covariance of face normals on the surface.

4.2.1 Analysis of Optimal Placement

The quadric error metric has the useful property that we can easily locate points which minimize the quadric error. Originally, I presented the equation for the optimal position $\bar{\mathbf{v}}$ (3.17) from the algebraic perspective of finding the minimum of a quadratic function. We have also seen that this point lies at the center of

the ellipsoidal isosurfaces of the quadric. I will now show that this optimal position can also be derived by an application of the least squares method of normal equations [113, 117, 71].

Suppose that we are given a set of k planes defined by equations of the form $\mathbf{n}_i^\mathsf{T}\mathbf{v} + d_i = 0$. Let $\mathbf{N}$ be the $k \times 3$ matrix of normals

$$\mathbf{N} = \begin{bmatrix} \mathbf{n}_1^\mathsf{T} \\ \mathbf{n}_2^\mathsf{T} \\ \vdots \\ \mathbf{n}_k^\mathsf{T} \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ & \vdots & \\ a_k & b_k & c_k \end{bmatrix} \tag{4.16}$$

and let $\mathbf{d}$ be the corresponding k-vector of offsets

$$\mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_k \end{bmatrix} \tag{4.17}$$

Suppose that we would like to find a point $\mathbf{v} = [x\ y\ z]^\mathsf{T}$ which lies at the intersection of these planes. Such a point would satisfy the equation $\mathbf{N}\mathbf{v}+\mathbf{d} = 0$. Rewriting this equation as $\mathbf{N}\mathbf{v} = -\mathbf{d}$, we get the following linear system of equations for $\mathbf{v}$:

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ & \vdots & \\ a_k & b_k & c_k \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -d_1 \\ -d_2 \\ \vdots \\ -d_k \end{bmatrix} \tag{4.18}$$

If $k > 3$, this system of equations will, in general, be over-constrained. It is unlikely that all the planes will intersect at a single point. In order to find the point which best fits this set of planes, we can apply the standard least squares method and solve the system of *normal equations* [113]

$$\mathbf{N}^\mathsf{T}\mathbf{N}\mathbf{v} = -\mathbf{N}^\mathsf{T}\mathbf{d} \tag{4.19}$$

In practice, the QR decomposition is preferred for solving large least squares problems because it is more numerically stable than the method of normal equations [117, 71, 105]. However, it is the system of normal equations which clearly demonstrates the connection between the quadric error metric and least squares fitting. By expanding the product $\mathbf{N}^\mathsf{T}\mathbf{N}$, we find that

$$\mathbf{N}^\mathsf{T}\mathbf{N} = \sum_{i=1}^{k} \mathbf{n}_i\mathbf{n}_i^\mathsf{T} \tag{4.20}$$

The right-hand side, the sum of the outer products of the normal vectors, is precisely the definition of the $\mathbf{A}$ matrix component of a quadric (§3.4). Similarly,

by expanding $\mathbf{N}^\mathsf{T}\mathbf{d}$ we find that it is identical to $\mathbf{b}$, the linear component of the quadric metric

$$\mathbf{N}^\mathsf{T}\mathbf{d} = \sum_{i=1}^{k} d_i \mathbf{n}_i = \mathbf{b} \tag{4.21}$$

Substituting $\mathbf{A} = \mathbf{N}^\mathsf{T}\mathbf{N}$ and $\mathbf{b} = \mathbf{N}^\mathsf{T}\mathbf{d}$ into (4.19), we see that the system of normal equations becomes simply

$$\mathbf{A}\mathbf{v} = -\mathbf{b} \tag{4.22}$$

The solution to this system, $\mathbf{v} = -\mathbf{A}^{-1}\mathbf{b}$, is precisely the formula for optimal placement defined in Section 3.5. Thus, the optimal position $\bar{\mathbf{v}}$ found by minimizing the quadric metric is the least squares optimal point which best fits the set of planes represented by the quadric.

4.2.2 Principal Components of Quadrics

We have now seen several ways of interpreting the optimal position $\bar{\mathbf{v}}$ which minimizes the quadric error $Q(\bar{\mathbf{v}})$. Geometrically, it lies at the center of the ellipsoidal isosurfaces of Q. By definition, it minimizes the sum of squared distances to some set of planes. And as we have just seen, it can also be derived from the least squares normal equations. If we weight the fundamental quadrics appropriately, it is the position which minimizes the sum of squared volumes of a set of tetrahedra connecting the vertex to triangles on the original surface. However, the ellipsoidal isosurfaces also have a particular shape, and the nature of this shape is a crucial feature of the quadric metric. Recalling the quadric isosurfaces shown in Figure 4.1, I observed in Section 4.1.1 that the ellipsoids seem to stretch out in directions of low curvature and are squeezed in directions of high curvature. In order to understand this relationship between the shape of the ellipsoids and surface curvature, we must analyze the eigenvalues and eigenvectors of the quadric matrix $\mathbf{A}$.

The spherical Gauss map [111] of a surface transports every point on the surface to the position on the unit sphere corresponding to the unit surface normal at that point (see Figure 4.3). For a polygonal model, all points on a given triangular face have the same normal, thus each triangle is mapped to a single point on the unit sphere. The spherical image of a set of triangles is a set of points. If the surface is continuous, then its spherical image will be a continuous patch on the unit sphere.

Since a given quadric Q represents a discrete set of faces, and hence a discrete set of k unit normals, it corresponds to some set of points on the unit sphere. We can construct the covariance matrix for these normals

$$\mathbf{Z} = \frac{1}{k-1}\sum_{i}(\mathbf{n}_i - \bar{\mathbf{n}})(\mathbf{n}_i - \bar{\mathbf{n}})^\mathsf{T} \tag{4.23}$$

where $\bar{\mathbf{n}}$ is the mean normal. Suppose that we order the eigenvectors $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ of $\mathbf{Z}$ in decreasing order of their corresponding eigenvalues. A well-known result

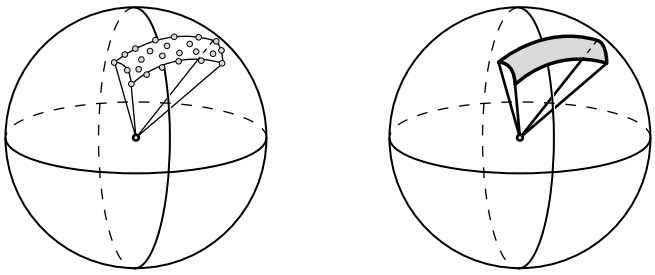


Figure 4.3: Gauss mapping of a surface region to the unit sphere. A triangulated surface produces a discrete set of points while a continuous surface patch produces a continuous patch on the sphere.

from principal component analysis [101] is that the first eigenvector $\mathbf{e}_1$, having the largest eigenvalue, is the direction in which the points $\mathbf{n}_i$ spread out the most around the mean $\bar{\mathbf{n}}$ (i.e., the direction of maximum variance). The smallest eigenvector $\mathbf{e}_3$ is the direction of minimum variance, the direction in which the points are spread out the least.

For the moment, I am only interested in the directions of the eigenvectors of $\mathbf{Z}$ and not the actual values of the corresponding eigenvalues. Therefore, we can ignore the $1/(k-1)$ averaging term. If we take the mean of the normals to be $\bar{\mathbf{n}} = 0$, then the equation for the covariance matrix (4.23) becomes

$$\mathbf{Z} = \sum_i \mathbf{n}_i \mathbf{n}_i^{\mathsf{T}} = \mathbf{A} \tag{4.24}$$

the matrix component of the quadric Q. So, we can regard $\mathbf{A}$ as the covariance matrix of the normals $\mathbf{n}_i$ with mean 0. We know that the eigenvectors of $\mathbf{A}$ define the axes of the ellipsoidal isosurfaces. But now we also see that these eigenvectors are the principal components of the corresponding set of unit normal vectors. It is interesting to note that covariance matrices of this sort have been used by others to define tangent planes [94, 122, 13] and principal directions of curvature [122, 13] for collections of points sampled from a surface.

If the set of faces represented by Q is from a fairly smooth surface region, the corresponding normals will all lie within a small region of the Gauss map (as in Figure 4.3). Intuitively, this should mean that the direction of maximum variance around $\bar{\mathbf{n}} = 0$, the direction in which the normals are spread out most around the origin, will be along the average normal direction. The next largest principal component should be the direction along which the points spread out the most along the sphere, and the final principal component should be the direction in which the normals spread out the least. Observe that normals will spread out further on the Gauss map when the surface is more highly

curved. This suggests that, assuming that we are considering a reasonably smooth surface, the eigenvectors of the quadric matrix **A** should roughly be in the directions of the average normal, the direction of maximum curvature, and the direction of minimum curvature. These will be the axes of the resulting ellipsoidal isosurface. The extent of the ellipsoid along a given axis will be inversely related to the corresponding eigenvalue. For instance, the ellipsoid will extend the farthest along the axis corresponding to the smallest eigenvalue. Therefore, the ellipsoid should stretch out the farthest in the direction of least curvature and stretch out the least in the average normal direction.

4.3 Differential Geometry

The argument which I have just presented confirms the intuition of Section 4.1.1: the shape of the quadric error isosurfaces is related to surface curvature. As I have indicated, this conclusion rests on the assumption that the set of faces from which the quadric was derived are sufficiently smooth. This suggests a further avenue for analyzing the quadric metric. I will assume that the surface is in fact a differentiable manifold. To arrive at a more formalized, and more precise, understanding of quadrics, I will apply the theory of local differential geometry. But before delving into this analysis, a quick overview of the necessary background is in order.

A comprehensive introduction to differential geometry is clearly far beyond the scope of this work. Fortunately, there are a wide variety of books available on the subject. The classic text of Hilbert and Cohn-Vossen [88] provides an excellent introduction to the intuitive side of the subject matter with a minimum of formalism. Besl and Jain [14] give a nice overview of the essential material, and they discuss some computational techniques. For a more comprehensive and systematic treatment of the subject, I have found Kreyszig's text[2] [111] — an expanded version of an earlier book [112] — to be fairly useful.

Let us assume that we are given a closed differentiable manifold surface M which has been divided into a set of patches. A given surface patch is defined by the mapping

$$\mathbf{x} = \mathbf{x}(u, v) = [f_1(u, v)\ f_2(u, v)\ f_3(u, v)]^\mathsf{T} \tag{4.25}$$

where (u, v) range over a region of the Cartesian 2-plane and the functions f_i are of class C^2. We shall be concerned with the surface in the neighborhood of a point $\mathbf{p} = \mathbf{x}(u_0, v_0)$. By convention, all functions of $\mathbf{x}$ and its derivatives are implicitly evaluated at (u_0, v_0).

[2] This book uses the more modern tensor notation. Willmore [195] provides a fairly easy to read introduction using the somewhat dated classical notation.

4.3.1 Fundamental Forms

The partial derivatives of the patch function $\mathbf{x}$

$$\mathbf{x}_1 = \mathbf{x}_u = \partial\mathbf{x}/\partial u \quad \text{and} \quad \mathbf{x}_2 = \mathbf{x}_v = \partial\mathbf{x}/\partial v \tag{4.26}$$

span the tangent plane of the surface at $\mathbf{p}$, provided we make the standard assumption that $\mathbf{x}_1 \times \mathbf{x}_2 \neq 0$. Consequently, we can write the unit surface normal $\mathbf{n}$ at the point $\mathbf{p}$ as

$$\mathbf{n} = \frac{\mathbf{x}_1 \times \mathbf{x}_2}{\|\mathbf{x}_1 \times \mathbf{x}_2\|} \tag{4.27}$$

Let $\mathbf{t}$ be a vector tangent to the surface $\mathbf{x}$ at the point $\mathbf{p}$. We know that we can write it as a linear combination $\mathbf{t} = \mathbf{x}_1\,\delta u + \mathbf{x}_2\,\delta v$. The direction vector $\mathbf{u} = [\delta u\ \delta v]^{\mathsf{T}}$ provides a convenient representation for tangent vectors. This is frequently expressed in differential form as $d\mathbf{x} = \mathbf{x}_1\,du + \mathbf{x}_2\,dv$.

The squared length of a tangent vector in the direction $\mathbf{u}$ is measured using the *first fundamental form*[3] $\mathrm{I}(\mathbf{u})$. If we define the matrix

$$\mathbf{G} = \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} \qquad \text{where } g_{ij} = \mathbf{x}_i \cdot \mathbf{x}_j \tag{4.28}$$

then $\mathrm{I}(\mathbf{u})$ is given by the quadratic form[4] $\mathbf{u}^{\mathsf{T}}\mathbf{G}\mathbf{u}$. Again considering a tangent vector as a differential, this length is often written as $d\mathbf{x} \cdot d\mathbf{x}$. Since the dot product is commutative, it is clear that $g_{ij} = g_{ji}$ and thus $\mathbf{G}$ is symmetric. It is also customary to denote the determinant $\det \mathbf{G} = g_{11}g_{12} - g_{12}^2$ by g. Our assumption that $\mathbf{x}_1 \times \mathbf{x}_2 \neq 0$ implies that $g \neq 0$. We can also measure surface areas using the first fundamental form. Given a region F on the surface, its area is given by the integral

$$\iint_F dA = \iint \sqrt{g}\,du\,dv \tag{4.29}$$

The differential $dA = \sqrt{g}\,du\,dv$ is referred to as the element of area of the surface.

The *second fundamental form* $\mathrm{II}(\mathbf{u})$ measures the change in the normal vector $\mathbf{n}$ in the direction $\mathbf{u}$. Using the matrix

$$\mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \qquad \text{where } b_{ij} = \mathbf{n} \cdot \mathbf{x}_{ij} = -\mathbf{n}_i \cdot \mathbf{x}_j. \tag{4.30}$$

we can write $\mathrm{II}(\mathbf{u})$ as the quadratic form[5] $\mathbf{u}^{\mathsf{T}}\mathbf{B}\mathbf{u}$. The common differential notation for this is $-d\mathbf{x} \cdot d\mathbf{n}$.

There are many uses of these fundamental forms in surface theory, but for our limited purpose here, only one is of significance. Together, the two fundamental forms characterize the local curvature of the surface.

[3] Also referred to as the *metric tensor*.
[4] Classical notation: $ds^2 = E\,du^2 + 2F\,du\,dv + G\,dv^2$. Here $\mathbf{G} = \left[\begin{smallmatrix} E & F \\ F & G \end{smallmatrix}\right]$.
[5] Classical notation: $L\,du^2 + 2M\,du\,dv + N\,dv^2$. Here $\mathbf{B} = \left[\begin{smallmatrix} L & M \\ M & N \end{smallmatrix}\right]$.

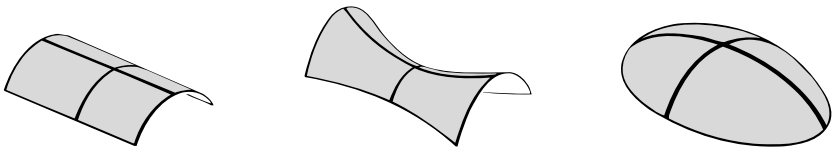
4.3.2 Surface Curvature

The normal curvature κ_n in the direction $\mathbf{u}$ is

$$\kappa_n = \frac{\mathrm{II}(\mathbf{u})}{\mathrm{I}(\mathbf{u})} = \frac{\mathbf{u}^\mathsf{T}\mathbf{B}\mathbf{u}}{\mathbf{u}^\mathsf{T}\mathbf{G}\mathbf{u}} \tag{4.31}$$

Unless the curvature is equal in all directions, there must be a direction $\mathbf{e}_1$ in which the normal curvature reaches a maximum and a direction $\mathbf{e}_2$ in which it reaches a minimum. These directions are called *principal directions* and the corresponding curvatures κ_1, κ_2 are the *principal curvatures*. The principal directions and curvatures are also the eigenvectors and eigenvalues, respectively, of the Weingarten map $\mathbf{G}^{-1}\mathbf{B}$. Given these principal curvatures, we can define the Gaussian curvature $K = \kappa_1\kappa_2$ and the mean curvature $H = \frac{1}{2}(\kappa_1 + \kappa_2)$. Note that there is a sign ambiguity present in the curvatures κ_1 and κ_2. If we flip the surface normal, the signs of the curvatures will change.

A given point on the surface can be classified according its principal curvatures. A point at which κ_n is equal in all directions is called an *umbilic* point; for example, every point on a sphere is an umbilic point. In the special case



(a) Parabolic ($K = 0$) (b) Hyperbolic ($K < 0$) (c) Elliptic ($K > 0$)

Figure 4.4: The three non-umbilic point classifications.

that $\kappa_n = 0$ in all directions, such as on a plane, the point is called a *flat* point. For non-umbilic points, the principal curvatures are well-defined. Figure 4.4 illustrates the three resulting categories.

4.4 Differential Analysis of Quadrics

I suggested earlier that there was a relationship between the shape of quadric isosurfaces and surface curvature (§4.1.1) and explored this connection through principal component analysis of the quadric matrix (§4.2.2). To extend this result, I will apply the local theory of differential geometry to analyze the quadric error metric on smooth surfaces. In particular, I will show that, under suitable assumptions, the eigenvalues of the quadric matrix $\mathbf{A}$ are related to the squares of the principal curvatures and that its eigenvectors are the corresponding principal directions. This provides further mathematical justification for why the

quadric error metric performs well; it works because it accumulates information about the structure of the surface.

4.4.1 Framework for Analysis

Suppose that we are given a differentiable manifold surface M. We can regard this smooth surface as the limit of the infinite subdivision of some triangulated model. As we continually subdivide triangles on the surface, the area of these triangles will approach 0 in the limit, and the resulting surface will be smooth. This is a common method for defining manifold surfaces in the field of topology [4]. This is a useful setting because, as the polygonal input for simplification becomes more and more finely tessellated, the local behavior of the quadric error metric will approach its behavior on differentiable manifolds.

The actual construction of such limit surfaces requires some care. As Schwarz [173] observed, polyhedral subdivision does not necessarily produce a unique limit. Suppose we construct a tessellation of a unit height, unit radius cylinder

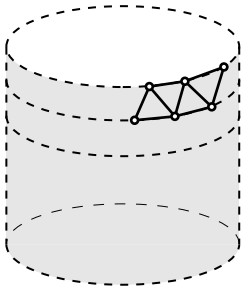


Figure 4.5: Triangulation of a unit height, unit radius cylinder.

by dividing its height into k equal parts and each slice into n equal parts. This will produce a triangulation of the cylinder with $2kn$ triangles (see Figure 4.5). If we set $k = n^s$ and allow n to approach infinity, the surface area of the resulting surface will depend on the value of s. In fact, this area may not even converge at all (when $s = 3$ for instance). Further details can be found in many differential geometry and calculus books [110, 111]. To avoid this kind of ill-defined subdivision, I will require that the distribution of points on the surface should always remain uniform and that the lengths of edges in all directions of non-zero curvature must go to zero in the limit. In the end, this is a relatively minor detail. For the purposes of this analysis, it is enough that we are given a differentiable manifold, and that we know this corresponds to the subdivision limit of some polyhedron.

For every point $\mathbf{p}$ on the surface M we can define a unique tangent plane. Just as with planes defined by triangular faces, we can measure the squared

distance of a given point $\mathbf{v}$ to the tangent plane of $\mathbf{p}$. This distance is

$$D^2 = \left((\mathbf{v}-\mathbf{p})^\mathsf{T}\mathbf{n}\right)^2 \tag{4.32}$$

where $\mathbf{n}$ is the unit surface normal at $\mathbf{p}$. As before, we can rewrite this equation and express D^2 using the standard quadric equation (3.9):

$$\begin{aligned} D^2 &= (\mathbf{v}^\mathsf{T}\mathbf{n} - \mathbf{p}^\mathsf{T}\mathbf{n})(\mathbf{v}^\mathsf{T}\mathbf{n} - \mathbf{p}^\mathsf{T}\mathbf{n}) \\ &= \mathbf{v}^\mathsf{T}(\mathbf{n}\mathbf{n}^\mathsf{T})\mathbf{v} - 2(\mathbf{n}\mathbf{n}^\mathsf{T})\mathbf{p}^\mathsf{T}\mathbf{v} + \mathbf{p}^\mathsf{T}(\mathbf{n}\mathbf{n}^\mathsf{T})\mathbf{p} \\ &= \mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{v} + 2\mathbf{b}^\mathsf{T}\mathbf{v} + c \\ &= Q(\mathbf{v}) \end{aligned}$$

where

$$Q = (\mathbf{A}, \mathbf{b}, c) = (\mathbf{n}\mathbf{n}^\mathsf{T}, -\mathbf{A}\mathbf{p}, \mathbf{p}^\mathsf{T}\mathbf{A}\mathbf{p}) \tag{4.33}$$

This has the same structure as a quadric defined on a polygonal model. In fact, this equation can be used as an alternate formula for constructing the fundamental quadric for a polygonal face.

4.4.2 Quadrics and Curvature

The effect of simplification by contraction is to partition the original surface into connected sets of faces, and to replace each of these sets with a vertex neighborhood. In the limit of smooth surfaces, this corresponds to partitioning the surface into a set of continuous regions. If a given region F is sufficiently small, we can always find a point $\mathbf{p}_0$ such that the region is approximated to second order by a surface patch of the form[6]

$$\mathbf{p}(u,v) = \left[u \;\; v \;\; \frac{1}{2}(\kappa_1 u^2 + \kappa_2 v^2)\right]^\mathsf{T} \tag{4.34}$$

where κ_1, κ_2 are the principal curvatures at $\mathbf{p}_0$. The coordinate frame has its origin at $\mathbf{p}_0 = \mathbf{p}(0,0)$ and its axes are determined by the principal directions and the surface normal $\mathbf{e}_1, \mathbf{e}_2, \mathbf{n}_0$ at $\mathbf{p}_0$. If $\mathbf{p}_0$ is an umbilic point, it is sufficient to pick two arbitrary, orthogonal "principal" directions. Figure 4.6 illustrates this local parameterization of the surface.
The matrix of the first fundamental form for this surface patch is

$$\mathbf{G} = \begin{bmatrix} 1+\kappa_1^2 u^2 & \kappa_1\kappa_2 uv \\ \kappa_1\kappa_2 uv & 1+\kappa_2^2 v^2 \end{bmatrix} \qquad g = 1 + \kappa_1^2 u^2 + \kappa_2^2 v^2 \tag{4.35}$$

and the unit surface normal is

$$\mathbf{n} = \frac{\mathbf{p}_u \times \mathbf{p}_v}{\|\mathbf{p}_u \times \mathbf{p}_v\|} = \left[-\kappa_1 u \;\; -\kappa_2 v \;\; 1\right]^\mathsf{T}/\sqrt{g} \tag{4.36}$$

[6] This is a height field, also referred to as a *Monge patch*.

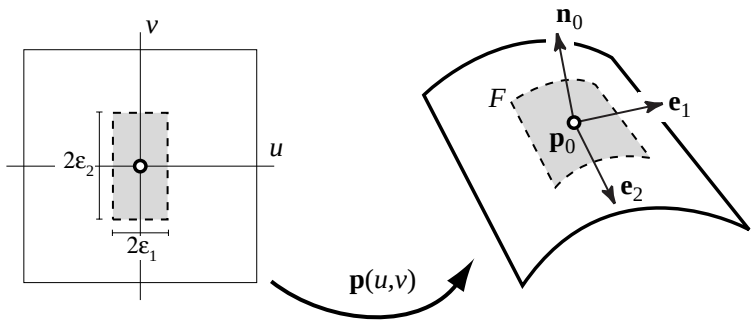


Figure 4.6: Local parameterization of the surface about $\mathbf{p}_0$.

Let $\mathbf{m}$ be the non-unit normal $\mathbf{m} = \mathbf{p}_u \times \mathbf{p}_v = \sqrt{g}\,\mathbf{n}$.

For polygonal surfaces, where a set of faces is replaced by a vertex neighborhood, the quadric associated with the resulting vertex is the area-weighted sum of the fundamental quadrics of the original set of faces. In the limit as triangle size goes to 0, this will become an integration over all the points in the region, where the tangent plane at each point determines a quadric. I will refer to this as the *integral quadric* of the region. The remainder of this section will be devoted to proving the following result:

> For a sufficiently small region about $\mathbf{p}_0$, the two smallest eigenvalues of the integral quadric matrix $\mathbf{A}$ are proportional to the squares of the principal curvatures at $\mathbf{p}_0$, and the corresponding eigenvectors are the principal directions.

For the sake of simplicity, let us assume that F is the image of a rectangular region $-\epsilon_1 \le u \le \epsilon_1, -\epsilon_2 \le v \le \epsilon_2$. An elliptical region produces nearly identical results, differing only in the constants of proportionality, but is less convenient to work with. The integral quadric $Q = (\mathbf{A}, \mathbf{b}, c)$ for this region, formed by integrating the fundamental quadrics for each point in F, is given by

$$\mathbf{A} = \iint_F \mathbf{n}\mathbf{n}^\mathsf{T} dA \tag{4.37}$$

$$\mathbf{b} = \iint_F -(\mathbf{n}\mathbf{n}^\mathsf{T})\mathbf{p}\, dA \tag{4.38}$$

$$c = \iint_F \mathbf{p}^\mathsf{T}(\mathbf{n}\mathbf{n}^\mathsf{T})\mathbf{p}\, dA \tag{4.39}$$

where integration of matrices and vectors is defined by integrating each scalar component separately.

As before, I will focus on the matrix $\mathbf{A}$ since it is the component of the quadric that determines the shape of the error isosurfaces. First, we can rewrite

its equation by expanding the element of area dA:

$$\mathbf{A} = \iint \mathbf{n}\mathbf{n}^{\mathsf{T}} \sqrt{g}\, du\, dv \tag{4.40}$$

Note that, since $\mathbf{n} = \mathbf{m}/\sqrt{g}$, this equation can be simplified as follows

$$\mathbf{A} = \iint \left(\frac{\mathbf{m}}{\sqrt{g}}\right)\left(\frac{\mathbf{m}}{\sqrt{g}}\right)^{\mathsf{T}} \sqrt{g}\, du\, dv = \iint \frac{\mathbf{m}\mathbf{m}^{\mathsf{T}}}{\sqrt{g}}\, du\, dv \tag{4.41}$$

The matrix $\mathbf{m}\mathbf{m}^{\mathsf{T}}$ can easily be derived from the patch equation (4.34).

$$\mathbf{m}\mathbf{m}^{\mathsf{T}} = \begin{bmatrix} \kappa_1^2 u^2 & \kappa_1\kappa_2 uv & -\kappa_1 u \\ \kappa_1\kappa_2 uv & \kappa_2^2 v^2 & -\kappa_2 v \\ -\kappa_1 u & -\kappa_2 v & 1 \end{bmatrix} \tag{4.42}$$

However, the $\sqrt{g}$ in the denominator is problematic. In order to evaluate this integral, I will use the Taylor series approximation (about $\mathbf{p}_0$)

$$\frac{1}{\sqrt{g}} \approx 1 - \frac{1}{2}\kappa_1^2 u^2 - \frac{1}{2}\kappa_2^2 v^2 + O(u^4 + u^2 v^2 + v^4) \tag{4.43}$$

Using this approximation, the formula for $\mathbf{A}$ becomes

$$\mathbf{A} = \int_{-\epsilon_2}^{\epsilon_2} \int_{-\epsilon_1}^{\epsilon_1} \mathbf{m}\mathbf{m}^{\mathsf{T}} (1 - \frac{1}{2}\kappa_1^2 u^2 - \frac{1}{2}\kappa_2^2 v^2)\, du\, dv \tag{4.44}$$

for sufficiently small ϵ_i.

Without going through the details, let me simply summarize the result of carrying out this integration. The matrix $\mathbf{A}$ is a diagonal matrix, largely because of the choice of the principal directions $\mathbf{e}_1, \mathbf{e}_2$ for the local coordinate frame. Assuming that ϵ_1 and ϵ_2 are sufficiently small, we can ignore terms of ϵ_i^6 and higher, in which case the diagonal entries of $\mathbf{A}$ are simply

$$a_{11} = \frac{4}{3}\epsilon_1^3 \epsilon_2 \kappa_1^2 \tag{4.45}$$

$$a_{22} = \frac{4}{3}\epsilon_1 \epsilon_2^3 \kappa_2^2 \tag{4.46}$$

$$a_{33} = 4\epsilon_1\epsilon_2 - \frac{2}{3}\epsilon_1\epsilon_2(\epsilon_1^2\kappa_1^2 + \epsilon_2^2\kappa_2^2) \tag{4.47}$$

Since $\mathbf{A}$ is diagonal, the diagonal entries are also its eigenvalues and the coordinate axes are its eigenvectors. As you can see, the eigenvalues of $\mathbf{A}$, for a sufficiently small neighborhood, are proportional to the squares of the principal curvatures κ_1, κ_2, and the corresponding eigenvectors are the principal directions. The final, and largest, eigenvalue is essentially proportional to the area of F and its corresponding eigenvector is in the direction of the surface normal $\mathbf{n}_0$. This agrees with the result developed in Section 4.2.2 through principal component analysis of the quadric matrix.

This formula for the integral quadric matrix $\mathbf{A}$ (4.45) suggests that locally planar regions will produce isosurfaces which are two parallel planes, that parabolic regions ($K = 0$) will produce cylindrical isosurfaces, and that other regions ($K \neq 0$) will result in ellipsoidal isosurfaces. The longest (major) axis of the ellipsoids will be aligned with the direction of minimum curvature $\mathbf{e}_2$, and the shortest (minor) axis will be aligned with the local surface normal $\mathbf{n}_0$. The remaining axis will be aligned with the direction of maximal curvature $\mathbf{e}_1$. This agrees with the earlier analysis of quadrics. Also note that, since the eigenvalues are related to the squared curvatures, the quadric metric does not distinguish between elliptic ($K > 0$) and hyperbolic ($K < 0$) regions.

To complete the analysis of the integral quadric, we can evaluate $\mathbf{b}$ and c using the same procedure. Carrying out this integration results in the values

$$\mathbf{b} = \left[0 \;\; 0 \;\; \frac{2}{3}\epsilon_1\epsilon_2(\epsilon_1^2\kappa_1 + \epsilon_2^2\kappa_2) \right]^{\mathsf{T}} \tag{4.48}$$

$$c = \frac{1}{5}\kappa_1^2\epsilon_1^5\epsilon_2 + \frac{2}{9}\kappa_1\kappa_2\epsilon_1^3\epsilon_2^3 + \frac{1}{5}\kappa_2^2\epsilon_1\epsilon_2^5 \tag{4.49}$$

We now have a complete quadric Q. Applying the formula for the optimal position $\bar{\mathbf{v}}$ (§3.5), we find that

$$\bar{\mathbf{v}} = \left[0 \;\; 0 \;\; -\frac{1}{6}(\kappa_1\epsilon_1^2 + \kappa_2\epsilon_2^2) \right]^{\mathsf{T}} \tag{4.50}$$

and its error with respect to Q is

$$Q(\bar{\mathbf{v}}) = \frac{4}{45}(\kappa_1^2\epsilon_1^5\epsilon_2 + \kappa_2^2\epsilon_1\epsilon_2^5) \tag{4.51}$$

An equation of the form $A(\kappa_1^2 + \kappa_2^2) + 2B\kappa_1\kappa_2$, where A, B are constants, describes the potential energy of an elastic thin plate [36]. Energy terms of the form $\kappa_1^2 + \kappa_2^2$ are often used for producing nicely smoothed, or fair, surfaces [191]. It is interesting to note that the quadric error $Q(\bar{\mathbf{v}})$ has this form. This suggests that minimizing the quadric error has the effect of producing nicely smoothed approximations, provided that the input is suitably smooth itself.

This differential analysis of quadrics can be extended to consider the structure of the meshes produced by minimizing the quadric error [85]. Suppose that we reparameterize the region F in terms of an aspect ratio $\rho = \epsilon_1/\epsilon_2$ and the mean size $\epsilon = \sqrt{\epsilon_1\epsilon_2}$. Holding ϵ fixed, we can show that the aspect ratio

$$\rho = \left| \frac{\kappa_2}{\kappa_1} \right|^{1/2} \tag{4.52}$$

minimizes the quadric error (4.51). Results from the field of function approximation theory [138, 37, 156] demonstrate that this is the aspect ratio of triangles in the L_2-optimal triangulation of a bivariate function $f(u, v)$. This suggests that, in the limit as triangle size goes to 0 and for sufficiently small neighborhoods,

minimizing the quadric error metric produces triangulations with optimal aspect ratios. While the quadric-based simplification algorithm works with polygonal, as opposed to smooth, surfaces and greedy contraction is not guaranteed to minimize the quadric error, there is some evidence that it does tend towards this theoretical aspect ratio [85].

4.5 Connection with Approximation Error

Up until this point, my analysis of the quadric metric has focused primarily on the structure of the error isosurfaces. As we have seen, these isosurfaces are (potentially degenerate) ellipsoids. Under suitable assumptions, the axes of these ellipsoids align with the principal directions of curvature on the surface, and the extent of the ellipsoid along its axes is inversely related to the surface curvature in that direction. Now let us turn our attention to a somewhat different issue. I have proposed that the Hausdorff distance E_{max} (2.7) and the average squared distance E_{avg} (2.8) provide good measurements of the geometric error between between a model M and its approximation M'. In this section, I will examine how the quadric error metric E_Q relates to these measures of approximation error.

Let us begin by noting an important difference between these metrics. The quadric metric E_Q defines a separate error value at each *vertex* while the approximation error metrics E_{max} and E_{avg} define a single error value between two *surfaces*. We could artificially define a total quadric error by, for example, summing the quadric errors assigned to each vertex, but there is no good foundation for doing this. Also, recall that for each vertex $\mathbf{v}$ on the approximation M', there is some associated set of faces R whose vertices have been contracted into $\mathbf{v}$. The quadric error is only based on the faces in this set. Therefore, it is most appropriate to compare the quadric metric with the localized distance $d_{\mathbf{v}}(R)$ (2.4) which measures the distance from $\mathbf{v}$ to the closest point in the region R as opposed to the closest point over all of M.

To gain some insight on these error metrics, consider a simple 2-D example as shown in Figure 4.7. The original model, shown in black, consists of three line segments. We are considering contracting the top segment, producing an approximation shown in gray. In the figure, we can see the familiar elliptical contours of the quadric error assigned to the resulting vertex. As drawn, the new vertex is placed at the position which minimizes the quadric error, at the center of the concentric isocontours.

Now let us compare this with the isocontours of the E_{max} metric (Figure 4.8) and the E_{avg} metric (Figure 4.9). Again, these contour plots show the error values assigned to the approximation as a function of the position of the resulting vertex. The approximation shown in each figure is the quadric-optimal one shown in Figure 4.7. Note that the E_{max}-optimal and E_{avg}-optimal positions lie very close to, but slightly above and the the right of, the position which minimizes the quadric metric. The E_{avg} contours are smoother than the E_{max} contours because they represent a sum of squared distances rather than the max-

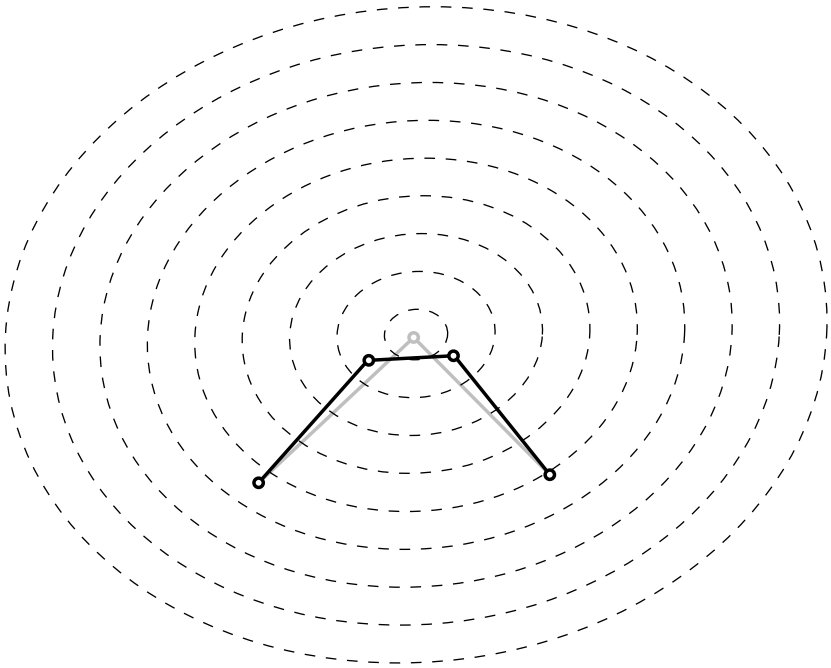


Figure 4.7: Isocontours of the quadric error E_Q.

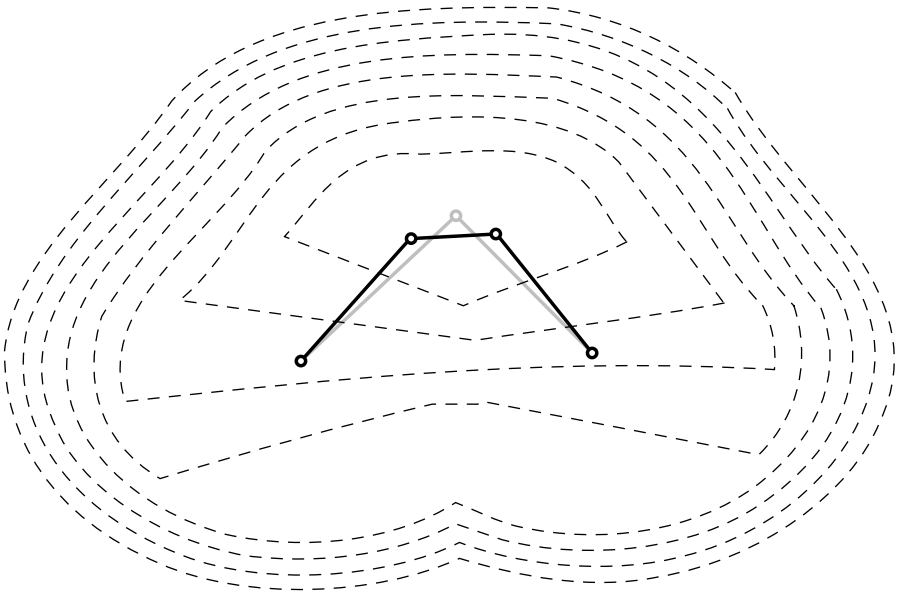


Figure 4.8: Isocontours of the Hausdorff distance E_{max}.

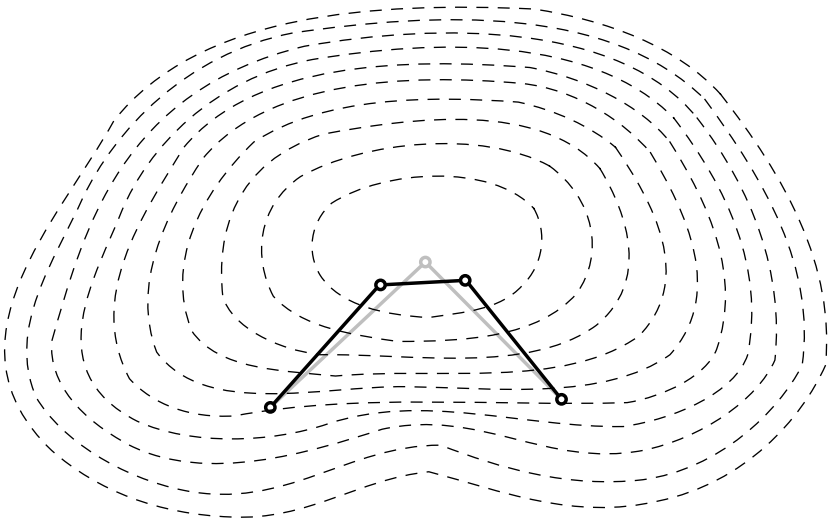


Figure 4.9: Isocontours of the average squared distance E_{avg}.

imum distance. Both metrics have a more complex structure than the quadric error, but the quadric error contours demonstrate some level of similarity, particularly to the inner E_{avg} contours which are roughly elliptical. However, these similarities would decrease as the model becomes more complex. Both the E_{avg} and E_{max} contours, which accumulate more global and more accurate information, would become more complex as more segments were added, while the quadric contours always remain ellipses.

Ideally, we could find some direct connection between the quadric metric and the E_{avg} approximation error. Ronfard and Rossignac [159] were able to show that their metric, which measures the maximum squared distance to a set of planes, was related to the localized Hausdorff distance E_{max} sampled at the vertices of M and M'. However, this argument relies on the use of a subset placement policy — the vertices of the approximation must be a subset of the original vertices — and it requires bounds on dihedral angles between planes. They achieved this bound by adding "auxiliary planes" at sharp corners (see Section 4.5.2). It does not appear that this argument extends to the quadric metric. Just as the L_∞ norm is considered stronger than the L_2 norm because provable bounds are easier to derive (§2.3.2), it appears easier to analyze the metric based on the maximum as opposed to the sum of distances to planes.

4.5.1 Undesirable Optimal Placement

In principle, the quadric error metric can allow points to move arbitrarily far away from the surface without assessing them a large error. Consider the three open cylinders shown in Figure 4.10. Suppose that we want to contract the

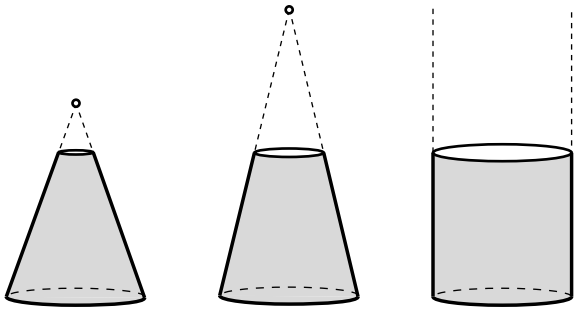


Figure 4.10: Three truncated cone surfaces. Integrating the quadrics along the upper rim, we find the optimal position lies at the apex of the cone.

entire upper rim into a single vertex. We would integrate the quadrics for all the points around the rim. For the first two surfaces, all the tangent planes along the rim intersect at a unique point. This point must have an error of 0 since it lies on all the tangent planes. Therefore, it is the point of minimal quadric error, and that is the position to which we would contract all the points on the rim. However, as the angle of the cylinder approaches vertical, this optimal point moves off to infinity, yet all the time it has zero error according to the quadric metric.

Of course, this problem does not really occur in practice. For open cylinders, the boundary constraint planes prevent the points from leaving the rim. For closed cylinders, the faces of the caps perform a similar function. And if we are using subset placement, we know that the vertices will never deviate from the surface and that the approximation must lie within the convex hull of the original surface.

4.5.2 Ambiguity of Sharp Angles

Because of the nature of the error metric, edges with a very small dihedral angle can be problematic [159] as well. Figure 4.11 provides a 2-D illustration of the situation. A vertex **v** is connected to two very different sets of line segments, a

Figure 4.11: Vertex **v** connected to two sets of line segments.

very flat line (a) and a very sharp corner (b). The initial quadric assigned to **v** is identical in both cases because its incident segments lie along the same lines. As the lines containing these segments approach each other, the segments will

become colinear. Curve (a) will become flat and curve (b) will become a single spike. The quadric error metric will allow points to move within these lines arbitrarily without error. In the first case, this is acceptable since the curve lies in this plane. However, in the second case, this will allow points to move far away from the underlying curve.

One way of alleviating this ambiguity in the error metric, which was suggested by Ronfard and Rossignac [159], is to treat edges with very sharp dihedral angles as discontinuities. We can add a constraint plane[7] (§3.6) through the edge that is perpendicular to the plane which bisects the dihedral angle. This has the effect of "rounding off" these sharp edges. Note that in the case of sharp dihedral angles, my algorithm would not attempt to compute an optimal position. As these angles become smaller, the solid angle bounding the local face normals becomes smaller, and the quadric matrix $\mathbf{A}$ becomes nearly singular. Thus, around very sharp angles it will always use a fixed placement scheme.

[7] Ronfard and Rossignac call this an *auxiliary plane*.

Chapter 5

Extended Simplification Algorithm

Many models have surface properties beyond simple geometry. In computer graphics, colors and textures are particularly common. To produce approximations which faithfully represent the original, we must maintain these properties as well as the surface geometry. The basic error metric presented in Chapter 3 only considers surface geometry when simplifying models. In this chapter, I will present a natural generalization of the quadric error metric which incorporates surface properties defined as vertex attributes.

5.1 Simplification and Surface Properties

Let us assume that each vertex, in addition to its position in space, has some associated continuous scalar values which describe other properties. As with geometric position, these values will be linearly interpolated over the faces of the model. For rendering applications, RGB color and 2-dimensional texture parameters are of particular interest; surface normals might also be treated as attributes. Finally, let us assume that the difference between two property values is measured by the Euclidean distance between them.

As an ongoing example, let us consider a Gouraud-shaded model for which each vertex has an associated color value $\mathbf{c} = [r\ g\ b]^\mathsf{T}$ where $0 \leq r, g, b \leq 1$. Figure 5.1 illustrates a simple example of a surface of this type. The surface itself is a section of the unit sphere. Every vertex has a single color value, and these color values are linearly interpolated over the faces of the mesh. Clearly, a visually faithful approximation of this surface will have to preserve the shading of the surface as well as its shape.

One approach to preserving appearance is to decouple attribute preservation from geometric simplification [25, 133, 178]. We can simplify the surface using any simplification algorithm, although an algorithm that can maintain correspondences between regions of the approximation and regions of the original

Figure 5.1: A section of a sphere Gouraud-shaded with a swirl pattern. Each vertex has a single associated color.

would be preferable. Having produced a simplified surface, we can resample attributes of the original model (e.g., color) into a texture map defined on the approximation. If the target rendering system supports bump mapping or normal mapping, this can be an effective technique for retaining high levels of visual fidelity [25, 30]. For these methods to work, each triangle of the approximation must be assigned a set of texels in the texture space. To avoid wasting texture space, these regions must also be tightly packed. Once the texel correspondence is established, the colors (or other attributes) on the original surface can be sampled into the texture. The primary attraction of this approach is that texturing is widely supported in rendering systems, and it decouples the resolution of the geometry (vertex count) from the resolution of the attributes (texture dimensions). And if the resampling process is completely decoupled from simplification [25], it can be used with separate simplification components that would otherwise produce approximations without any attributes at all. The current approaches for generating this mapping into texture space produce highly fragmented textures [178, 25] — neighboring triangles on the surface will not occupy neighboring regions in the texture. This has some unfortunate consequences. It means that we cannot construct image pyramids [194] for the resulting textures, because neighboring texels may be mapped onto entirely separate parts of the model. It also makes this approach ill-suited for multiresolution modeling applications, since each level of detail would require its own individual texture map. This would make incremental representations such as progressive meshes (§7.1.2) prohibitively expensive because of the amount of texture storage that would be required.

In many circumstances, it is more convenient to incorporate vertex attributes directly into the simplification process. This is particularly true for simplification systems being used to generate multiresolution models. The fundamental operation of my simplification algorithm, and other contraction-based methods,

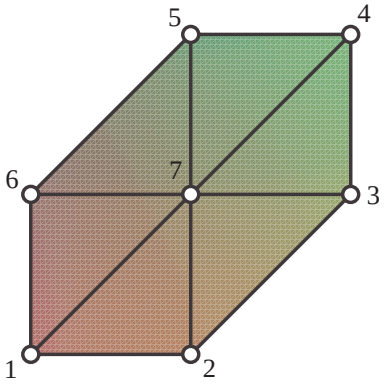
is to evaluate a contraction $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$. This involves two primary steps: (1) select a position $\bar{\mathbf{v}}$ and (2) evaluate the cost of performing the contraction. Integrating attributes involves selecting appropriate attributes for a vertex at position $\bar{\mathbf{v}}$ and extending the error metric to account for error in attributes as well as position. In this scheme, geometry and attributes are *coupled* together into a single error metric.

The most common approach [90, 30, 54] for integrating geometry and attributes in this fashion would seem to be *loose coupling*. Geometry is treated as the primary attribute. For a given contraction $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \bar{\mathbf{v}}$, the position $\bar{\mathbf{v}}$ is chosen, as before, using purely shape-based criteria such as the quadric error metric (§3.4). Given this fixed position, corresponding attributes are selected using another process, and the geometric and attribute error are measured separately. For a Gouraud-shaded surface, we would select a position $\bar{\mathbf{v}}$ and corresponding color $\bar{\mathbf{c}}$. We would then measure the error of this contraction as, for instance, $E_{dist}(\bar{\mathbf{v}}) + E_{rgb}(\bar{\mathbf{c}})$. The alternative to this approach is *tight coupling*. In this case, the best position $\bar{\mathbf{v}}$ and its corresponding attributes are selected simultaneously, taking the correlation between position and other attributes into account. This is the approach which I have adopted. By selecting the best combination of position and color, say, we can produce a mesh which better conforms to color features. While the exact structure of the mesh does not significantly affect texture mapping, it is much more important for Gouraud-shaded surfaces where edges of the mesh must align with color discontinuities to allow accurate color interpolation.

If we are using a subset placement policy (§3.5), there is little distinction between tight and loose coupling. We may pick either $\mathbf{v}_1$ along with all its attributes or $\mathbf{v}_2$ and all its attributes. The only issue is how the attribute error is measured. However, when using optimal placement, we allow the position $\bar{\mathbf{v}}$ to move freely in order to achieve better surface approximations. If $\bar{\mathbf{v}}$ is constrained to lie along the segment $(\mathbf{v}_i, \mathbf{v}_j)$, we can simply linearly interpolate attributes along the segment. However, the optimal positions will, in general, lie near the original surface, but not on it. Consequently, we cannot simply interpolate the property values of the endpoints; we need some way to synthesize entirely new values based on the new position. To do this, I use higher dimensional quadrics whose extra coefficients implicitly encode the cross correlation between the various properties.

5.2 Generalized Error Metric

I will treat each vertex as a vector $\mathbf{v} \in \mathbf{R}^n$. The first three components of $\mathbf{v}$ will be spatial coordinates, and the remaining components will be property values. For consistent results, the model can be scaled so that it lies within the unit cube in $\mathbf{R}^n$. This ensures that the various properties (including position) have roughly the same scale. Consider the example shown in Figure 5.2a, a triangulated hexagon. In this case, we would use 6-dimensional vectors of the form $[x\ y\ z\ r\ g\ b]^\mathsf{T}$. The 6-vectors for this example are listed in Figure 5.2b.



(a) Simple colored mesh

Vertex	Position			RGB Color		
1	[0	0	0	0.7	0.3	0.3]
2	[1	0	0	0.7	0.4	0.3]
3	[2	1	0	0.5	0.5	0.3]
4	[2	2	0	0.3	0.5	0.3]
5	[1	2	0	0.3	0.4	0.3]
6	[0	1	0	0.5	0.3	0.3]
7	[1	1	0	0.5	0.4	0.3]

(b) Table of vertices

Figure 5.2: A triangulated hexagon with color values at each vertex.

Having placed the vertices of the model in n-dimensional space, I can now formulate an extended version of the quadric error metric. Consider the triangle $T = (\mathbf{p}, \mathbf{q}, \mathbf{r})$. In the case of a colored surface, we would have vertices $\mathbf{p} = [p_x\ p_y\ p_z\ p_r\ p_g\ p_b]^\mathsf{T}$ and so forth. Since three linearly independent points always define a 2-dimensional plane, the vertices of this triangle determine a 2-plane in $\mathbf{R}^n$. Given this plane, we can construct a quadric that will measure the squared distance of any point in $\mathbf{R}^n$ to this plane.

Consider the 2-plane containing T. It can be defined by a point and two orthogonal vectors. Let us arbitrarily pick the point $\mathbf{p}$, and let $\mathbf{h} = \mathbf{q} - \mathbf{p}$ and $\mathbf{k} = \mathbf{r} - \mathbf{p}$. Applying a Gram-Schmidt orthogonalization [183], we can compute two orthonormal vectors $\mathbf{e}_1, \mathbf{e}_2$ (see Figure 5.3) defined by the equations

$$\mathbf{e}_1 = \mathbf{h}/\|\mathbf{h}\| \tag{5.1}$$

$$\mathbf{e}_2 = \frac{\mathbf{k} - (\mathbf{e}_1 \cdot \mathbf{k})\mathbf{e}_1}{\|\mathbf{k} - (\mathbf{e}_1 \cdot \mathbf{k})\mathbf{e}_1\|} \tag{5.2}$$

This gives us two unit-length vectors which form two axes of a local frame with $\mathbf{p}$ as the origin. In principle, we could compute an entire local frame with axes $\mathbf{e}_1, \ldots, \mathbf{e}_n$. However, to formulate the generalized quadric error metric, we will only need to explicitly compute $\mathbf{e}_1$ and $\mathbf{e}_2$.

Now, consider an arbitrary point $\mathbf{v} \in \mathbf{R}^n$. We are interested in the squared distance D^2 of $\mathbf{v}$ from the plane of T. The squared length of the vector $\mathbf{u} = \mathbf{p} - \mathbf{v}$ can be decomposed as

$$\|\mathbf{u}\|^2 = \mathbf{u}^\mathsf{T}\mathbf{u} = (\mathbf{u}^\mathsf{T}\mathbf{e}_1)^2 + \cdots + (\mathbf{u}^\mathsf{T}\mathbf{e}_n)^2 \tag{5.3}$$

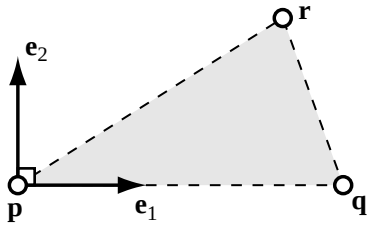


Figure 5.3: Orthonormal vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ define a local frame, with origin $\mathbf{p}$, for the 2-plane defined by the triangle $(\mathbf{p}, \mathbf{q}, \mathbf{r})$.

This is simply the generalized form of the Pythagorean Theorem. We can rewrite this equation as

$$(\mathbf{u}^\mathsf{T}\mathbf{e}_3)^2 + \cdots + (\mathbf{u}^\mathsf{T}\mathbf{e}_n)^2 = \|\mathbf{u}\|^2 - (\mathbf{u}^\mathsf{T}\mathbf{e}_1)^2 - (\mathbf{u}^\mathsf{T}\mathbf{e}_2)^2 \tag{5.4}$$

Note that the left hand side is the squared length of $\mathbf{u}$ along all the axes perpendicular to the plane of T. In other words, it is the squared perpendicular distance of $\mathbf{v}$ to the 2-plane of T. This is precisely the distance we are interested in measuring.

Applying (5.4), we can compute the distance D^2 as

$$D^2 = \|\mathbf{u}\|^2 - (\mathbf{u}^\mathsf{T}\mathbf{e}_1)^2 - (\mathbf{u}^\mathsf{T}\mathbf{e}_2)^2 \tag{5.5}$$

$$= \mathbf{u}^\mathsf{T}\mathbf{u} - (\mathbf{u}^\mathsf{T}\mathbf{e}_1)(\mathbf{e}_1{}^\mathsf{T}\mathbf{u}) - (\mathbf{u}^\mathsf{T}\mathbf{e}_2)(\mathbf{e}_2{}^\mathsf{T}\mathbf{u}) \tag{5.6}$$

By expanding and collecting terms, we arrive at the following formula

$$\begin{aligned} D^2 = \mathbf{v}^\mathsf{T}\mathbf{v} - 2\mathbf{p}^\mathsf{T}\mathbf{v} + \mathbf{p}\cdot\mathbf{p} \\ - \mathbf{v}^\mathsf{T}(\mathbf{e}_1\mathbf{e}_1{}^\mathsf{T})\mathbf{v} + 2(\mathbf{p}\cdot\mathbf{e}_1)\mathbf{e}_1{}^\mathsf{T}\mathbf{v} - (\mathbf{p}\cdot\mathbf{e}_1)^2 \\ - \mathbf{v}^\mathsf{T}(\mathbf{e}_2\mathbf{e}_2{}^\mathsf{T})\mathbf{v} + 2(\mathbf{p}\cdot\mathbf{e}_2)\mathbf{e}_2{}^\mathsf{T}\mathbf{v} - (\mathbf{p}\cdot\mathbf{e}_2)^2 \end{aligned} \tag{5.7}$$

This has exactly the same structure as the quadric error metric. We can write it as $D^2 = \mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{v} + 2\mathbf{b}^\mathsf{T}\mathbf{v} + c$ where

$$\mathbf{A} = \mathbf{I} - \mathbf{e}_1\mathbf{e}_1{}^\mathsf{T} - \mathbf{e}_2\mathbf{e}_2{}^\mathsf{T} \tag{5.8}$$

$$\mathbf{b} = (\mathbf{p}\cdot\mathbf{e}_1)\mathbf{e}_1 + (\mathbf{p}\cdot\mathbf{e}_2)\mathbf{e}_2 - \mathbf{p} \tag{5.9}$$

$$c = \mathbf{p}\cdot\mathbf{p} - (\mathbf{p}\cdot\mathbf{e}_1)^2 - (\mathbf{p}\cdot\mathbf{e}_2)^2 \tag{5.10}$$

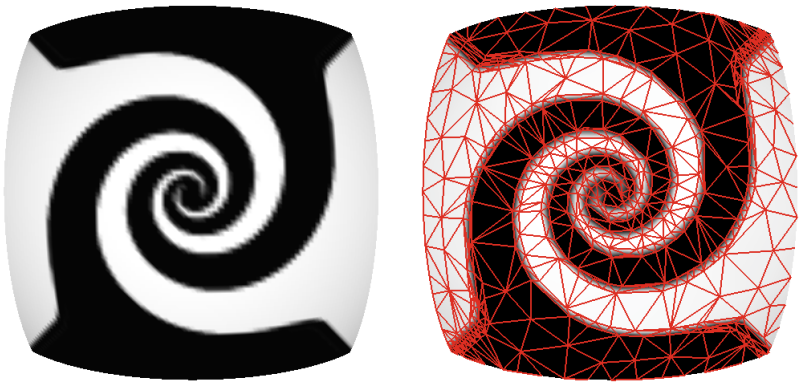
In this generalized quadric, $\mathbf{A}$ is an $n \times n$ matrix, and $\mathbf{b}$ is an n-vector. As a concrete example, the $\mathbf{A}$ matrix for the central vertex shown in Figure 5.2 is

$$\mathbf{A} = \begin{bmatrix} 0.06 & 0 & 0 & 0 & -0.59 & 0 \\ 0 & 0.23 & 0 & 1.15 & 0 & 0 \\ 0 & 0 & 6.00 & 0 & 0 & 0 \\ 0 & 1.15 & 0 & 5.77 & 0 & 0 \\ -0.59 & 0 & 0 & 0 & 5.94 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6.00 \end{bmatrix}$$

It is computed by summing the quadrics for the six surrounding faces (without area weighting). By examining the non-zero off-diagonal elements, we can see how the quadric records the high correlation between r & y and between g & x.

Because this generalized quadric error metric has exactly the same form as before, we can substitute it into the basic algorithm (§3.9) with very little effort. The generalized quadrics are constructed in a different manner, but they are used in an identical way. In particular, the fundamental steps of selecting the optimal position and updating quadrics by addition are exactly the same. Only the dimension of the matrices and vectors involved has changed.

In Figure 5.4, we can see the results of applying my algorithm using generalized quadrics. The approximation is about 5% the size of the original. Notice



(a) 18,050 faces (b) 1,000 faces

Figure 5.4: Original surface (a) and its approximation (b) using $[x\ y\ z\ r\ g\ b]^{\mathsf{T}}$ simplification. Notice that the mesh edges follow the color contours.

that the structure of the mesh conforms closely to the color pattern. The mesh is fairly sparse in areas of constant color. Near the borders of the color pattern, not only is the triangle density higher, but the edges are properly aligned with the curve of the swirl pattern. Further examples can be found in Section 6.3.

5.3 Assessing Generalized Quadrics

This generalized metric allows the optimal placement policy to be used with surfaces with material properties. It provides a means to synthesize new property values at vertex positions which do not lie anywhere on the original surface. Since, in most cases, my algorithm produces the best quality approximations using optimal placement, this is a significant advantage.

Unfortunately, for every property, we must add extra dimensions to the vertices and quadrics. Consequently, as we add more and more properties to a surface, the size of the quadric matrix grows quadratically in the number of attribute elements, and the running time will also increase. The space requirements for a few sample cases are summarized in Table 5.1. The necessary

Surface type	Vertex type	dim $\mathbf{A}$	No. unique coeff.
Geometry only	$[x\ y\ z]^\mathsf{T}$	3×3	$\binom{5}{2} = 10$
Geometry & 2-D texture	$[x\ y\ z\ s\ t]^\mathsf{T}$	5×5	$\binom{7}{2} = 21$
Geometry & Gouraud color	$[x\ y\ z\ r\ g\ b]^\mathsf{T}$	6×6	$\binom{8}{2} = 28$
Geometry, color & normal	$[x\ y\ z\ r\ g\ b\ u\ v\ w]^\mathsf{T}$	9×9	$\binom{11}{2} = 55$

Table 5.1: Summary of common extended quadric types.

overhead is not extreme. For instance, I have run tests on models with both color and surface normals at every vertex; this requires 9-dimensional quadrics. A 30,000 face model can still be simplified in under 30 seconds on a 200 MHz PentiumPro system.

We must also be able to deal with constraints on the range of property values. RGB colors, for instance, must be kept within the color cube. While my algorithm will not generate colors far outside the cube, since they would very poorly fit the data, it may generate colors that are slightly below 0 or above 1. As others have done [90], I merely clamp the offending colors to the nearest point on the color cube. Since only small distances are involved, this should not introduce any appreciable artifacts. The same caution must be applied to texture coordinates. Similarly, surface normals must be rescaled to have unit length.

As I have presented it, the generalized error metric weights all vertex attributes equally. However, for optimal results one might wish to selectively weight certain attributes more than others. It seems doubtful that there is a single optimal weighting for a given set of attributes, but it can easily be offered as a user-selectable preference. Individual attribute components can also be easily weighted. For example, we might weight the individual RGB values by the standard coefficients for computing luminance values from RGB colors [59].

5.3.1 Attribute Continuity

I have assumed that property values vary continuously over the surface, but in practice, this is not always the case. For example, consider simply wrapping a texture around a cylinder. There will be a seam where s wraps from 0 to 1. Consequently, every vertex along this seam must have two separate texture coordinates, and this will introduce a discontinuity at every one of these vertices.

To represent and simplify this textured cylinder with the current algorithm, we would need to replicate each vertex along the seam; each copy would have

separate texture coordinates but identical positions. In other words, we must force the surface to be a topological plane rather than a cylinder. Using boundary constraints to maintain the seam, this might produce acceptable results.

However, this will not work for all cases. A model might conceivably have a distinct piece of texture for each face, and this would create a discontinuity at every edge. As mentioned in Section 3.6, the algorithm does not work well when constraints proliferate to this extent. A more complete solution would require that we allow multiple values for each property at each vertex, which would require multiple corresponding quadrics for each vertex.

5.3.2 Euclidean Attributes

I have also assumed that the metric for measuring the difference in attributes is Euclidean. More specifically, I have assumed that attribute values at the vertices will be linearly interpolated over the faces of the model.

Perceptual color spaces are not Euclidean in RGB. However, the resulting approximations will presumably be displayed as Gouraud-shaded surfaces. Consequently, for display, colors will be linearly interpolated over faces. Thus, regardless of the non-Euclidean nature of RGB color space, these assumptions coincide with the way the results will be displayed. It is also interesting to note that any color space conversion which is a linear transform (e.g., RGB to CIE XYZ) can be incorporated into the extended quadrics via the standard quadric transformation rule (§3.4.2).

Surface normals are another attribute type which might not seem to conform with the Euclidean assumption. However, I contend that the algorithm will typically treat normals appropriately. Consider the Gauss mapping of the surface, where every vertex is mapped to the point on the unit sphere corresponding to the unit surface normal at that vertex. Simplifying surface normals as part of the generalized quadric metric is essentially equivalent to simplifying this spherical surface with the basic algorithm. So, as long as normals don't change too rapidly, the extended algorithm will produce good surface normals.

Chapter 6

Results and Performance Analysis

In the preceding chapters, I have described my surface simplification algorithm and analyzed the nature of the quadric error metric. This chapter is devoted to analyzing both the efficiency of my algorithm and the quality of its results. I will begin with some theoretical analysis of the time and space complexity of my algorithm. But the majority of the material I will present is focused on an empirical analysis of the running time of simplification and the geometric quality of the resulting approximations. Note that all surfaces, except for those with RGB color or texture attributes, are shown flat-shaded with one constant color per face. While smooth shading is probably more common in practice, this flat shading makes the structure of the surface approximations more apparent.

6.1 Time and Space Efficiency

One of my primary design goals has been to produce an efficient simplification algorithm, and the algorithm which I have developed is indeed quite fast. Based on published comparisons [26, 124], it appears to be about 10–100 times faster than other algorithms which can produce roughly comparable approximations. Uniform vertex clustering [161] is quite likely more efficient; however, the lower quality of its results places it in another category.

In this section, I will examine the efficiency of my algorithm. Note that the algorithm can be divided into two distinct phases. First is the *initialization* phase, where candidates are selected, the initial quadrics are created, and the candidates are ordered in a heap (§3.9, steps 1–4). Second is the *simplification* phase, where a sequence of pair contractions are selected and applied (§3.9, step 5). I will generally treat these phases separately.

Throughout the following analysis, I will assume that an input model M

having r vertices and n faces[1] is given. Furthermore, I will assume that the goal is to produce an approximation M' having only m faces, and I will let p be the number of candidate pairs selected during initialization.

6.1.1 Time Complexity

In order to efficiently simplify very large models, it is important that the asymptotic time complexity [34] of the algorithm be reasonably low. In this section, I will analyze the complexity of my algorithm under the assumptions that

1. the number of faces removed in one iteration is bounded by a constant k,
2. $O(n)$ candidate pairs are selected, and
3. the maximum vertex degree is bounded by a constant g.

The first assumption essentially requires that the surface be sufficiently similar to a manifold, in which case $k = 2$. Non-manifold edges are allowed, but there should be no edge with $O(n)$ adjacent faces. This is clearly a reasonable requirement, satisfied by almost all surfaces likely to be encountered in practice. I add the second assumption for cases in which we are selecting vertex pairs in addition to the edges. In principle, we could select all possible $O(n^2)$ pairs of vertices. However, the much more reasonable case involves selecting all $O(n)$ edges plus an optional set of $O(n)$ non-edge pairs. The final condition is important because various parts of the algorithm require looking at each edge attached to a particular vertex. We want these operations to have constant cost. While any particular model might not have bounded vertex degree, it is almost always possible to alter the mesh so that it satisfies this condition.

The algorithm begins with the initialization phase. Constructing all the initial quadrics takes $O(n)$ time. Selection and evaluation of the candidate pairs also requires $O(n)$ time. Placing all the resulting candidates in a heap requires $O(n \log n)$ time. Thus, the total complexity of initialization is $O(n \log n)$.

Now, consider iteration i of the simplification phase. We need to select the minimum cost pair, contract it, and update the local neighborhood. Table 6.1 summarizes the costs for these individual operations. Since we have assumed that g is a constant, the cost for a single iteration is thus $O(\log(n - ki))$. Summing over all iterations, the total cost for the simplification phase is

$$\log n + \log(n - k) + \log(n - 2k) + \ldots + \log m$$

Since k is a constant, we can replace this by its upper bound

$$\log n + \log(n - 1) + \log(n - 2) + \ldots + \log m$$

which is simply

$$\log \frac{n!}{m!} = \log n! - \log m! = O(n \log n - m \log m)$$

Thus, the overall complexity of my algorithm is $O(n \log n)$.

[1] Note that $n \approx 2r$ if M is a manifold (§2.1.1).

	Time Complexity	
Operation	For iteration i	Total
Select min cost pair	$O(\log(n-ki))$	$O(n\log n - m\log m)$
Contract pair	$O(g)$	$O(n-m)$
Update neighborhood	$O(g\log(n-ki))$	$O(n\log n - m\log m)$
TOTAL	$O(\log(n-ki))$	$O(n\log n - m\log m)$

Table 6.1: Time complexity of operations in simplification phase.

6.1.2 Memory Usage

In addition to running time, memory consumption is also an important aspect of efficiency. This is particularly true for very large models with more than a million faces.

	Data Item	Stored As	Size (bytes)
Vertices	Position	$3r$ floats	$12r$
	Face links	$3n$ words	$12n$
	Pair links	$2p$ words	$8p$
	Quadrics + areas	$11r$ doubles	$88r$
	Subtotal		$100r + 12n + 8p \approx 148r$
Pairs	Endpoints	$2p$ words	$8p$
	Target $\bar{\mathbf{v}}$	$3p$ floats	$12p$
	Cost $Q(\bar{\mathbf{v}})$	p floats	$4p$
	Heap backlinks	p words	$4p$
	Subtotal		$28p \approx 84r$
Faces	Vertices	$3n$ words	$12n$
	Normal	$3n$ shorts	$6n$
	Subtotal		$18n \approx 36r$
TOTAL			$100r + 30n + 36p \approx 268r$

Table 6.2: Breakdown of memory consumption for simplification algorithm. Totals show result of assuming that $n \approx 2r$ and $p \approx 3r$.

Table 6.2 summarizes the memory usage of my algorithm. The data items are categorized by whether they are associated with vertices, pairs, or faces. Recall that there are r vertices, p pairs, and n faces. For closed manifolds, there are two faces for every vertex ($n \approx 2r$) and three edges for every vertex ($p \approx 3r$). These proportions hold quite closely for many surfaces; the totals in Table 6.2

show the result of making this approximation.

Each vertex stores its position, links to pairs and faces which use it, and the associated quadric. Each pair stores links to its endpoint, the target position and cost of contracting it, and a link to its position in the heap. These "backlinks" are necessary for efficient updates of the heap after performing a contraction. Note that we could save 4.5% by using subset placement only; tracking the position $\bar{\mathbf{v}}$ would no longer be necessary. Each face stores links to its corners and a surface normal. These face normals are not strictly necessary, but they are convenient to have and only account for 2.2% of overall memory consumption.

Quadrics account for 33%, the single largest component, of total memory consumption. It is tempting to reduce memory requirements by using single precision, rather than double precision, floating point values. However, simply replacing `double` variables with `float` variables can cause significant numerical problems. Single precision floating point does not have sufficient precision to accurately track the quadric coefficients. It is possible that by carefully managing numerical values, such as by transforming the model to be centered at the origin and lie within the unit cube, that `float`s could be safely used. However, I have not tested any of these strategies; it has proved far more convenient to simply continue using double precision values.

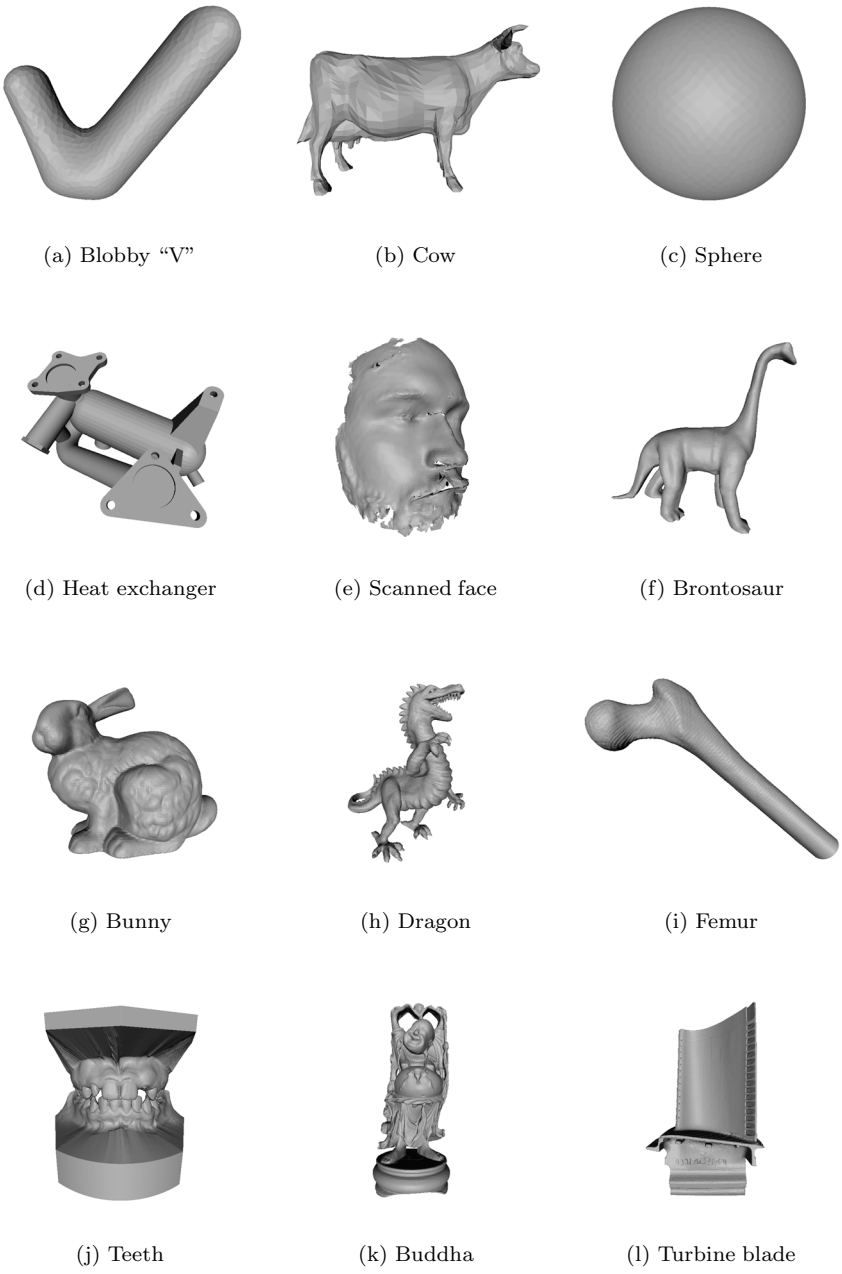
6.1.3 Empirical Running Time

The time complexity analysis in Section 6.1.1 has established that running time will not grow prohibitively large when processing large models. However, more important in practice is the actual running time of the algorithm. Many iterative contraction algorithms have $O(n \log n)$ running times. But the various error metrics which they use can have widely varying costs. My quadric error metric is relatively cheap; consequently, actual running times for my implementation are quite low.

To demonstrate the performance of my algorithm, I have selected the twelve surface models shown in Figure 6.1. The models are arranged in order of increasing size, ranging from 3842 to 1.7 million faces, and they represent several kinds of objects. Models (a) and (c) are very simple, smooth surfaces. The heat exchanger (d) and turbine blade (l) are machine parts. The face scan (e) is data acquired from a stereo vision system. Models (f–h, k) are all small figures scanned by a laser scanning system. The femur (i) and dental models (j) are datasets encountered in medical applications.

The overall complexity of these models, and the time necessary to simplify them, is summarized in Table 6.3. The running times listed reflect complete decimation to 0 faces. I have excluded the time required to read and write files to disk, since these vary significantly based on the model format used. The models are divided into three size groups: small, medium, and large.

For most of the models, I ran the timing tests on a system with a 200 MHz Intel PentiumPro processor and 160 MB of main memory. This level of computer is readily available at the consumer level today. For the three largest models, I ran performance tests on an SGI Onyx machine with a 195 MHz R10000



(a) Blobby "V" (b) Cow (c) Sphere

(d) Heat exchanger (e) Scanned face (f) Brontosaur

(g) Bunny (h) Dragon (i) Femur

(j) Teeth (k) Buddha (l) Turbine blade

Figure 6.1: Twelve surface models, arranged in order of increasing size, used in performance tests. Note that all models are flat-shaded. See Table 6.3 for a summary of model complexity and performance results.

			Running Time (s)		
Model	Vertices	Faces	Init.	Simpl.	Total
Blobby "V"	1923	3842	0.129	0.168	0.297
Cow	2904	5804	0.198	0.269	0.467
Sphere	4004	8004	0.294	0.361	0.655
Exchanger	5500	11,036	0.38	0.548	0.928
Face	14,445	28,906	1.02	1.632	2.652
Brontosaur	23,984	47,904	1.749	2.916	4.665
Bunny	34,834	69,451	2.612	4.435	7.047
Dragon	54,296	108,588	3.685	7.087	10.772
Femur	76,794	153,322	5.447	10.048	15.495
Teeth†	212,192	424,376	11.83	43.76	55.59
Buddha†	543,644	1,085,634	33.75	149.10	182.85
Turbine†	882,954	1,765,388	52.16	257.93	310.09

Table 6.3: Running times for complete decimation to 0 faces for models in Figure 6.1. Only initialization and simplification times are shown; input–output time is excluded. Tests were performed on a PentiumPro 200 MHz († 195 MHz R10000) system.

processor and 1 GB of main memory; the Intel machine did not have enough memory to accommodate models (k–l). My implementation causes excessive paging when there is not enough physical memory available because it exhibits poor memory locality. At each iteration, it tries to pick the best available contraction, and the location of the best contraction tends to vary widely over the surface from one iteration to the next. Note that, according to comparative tests that I have run, my software has roughly similar running times on both the R10000 and PentiumPro machines.

Figure 6.2 shows more complete running time information on the small-size models (a–d). It shows total running time as a function of output approximation size. The solid line indicates initialization time; in other words, the running time when the desired output size m is equal to the input size n. Running time increases as the size of the desired approximation m decreases; more iterations, and more work, are required to remove larger numbers of triangles. Note that despite the logarithmic factors in the theoretical time complexity, the running time tends to be linear in m in practice.

Similar running time data is shown in Figure 6.3 for the mid-size models (e–i). Again note that the running times behave quite linearly. The slope of these lines is also essentially similar across these different models.

The bunny model (g) provides a useful point of comparison with other simplification algorithms, since it has been widely used as an example model in

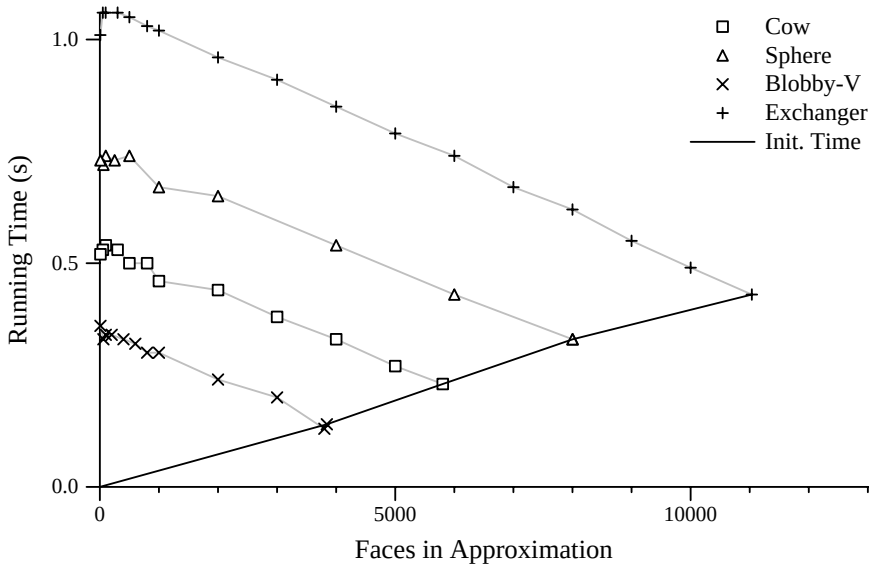


Figure 6.2: Running times for small-size models ($n < 15,000$ faces).

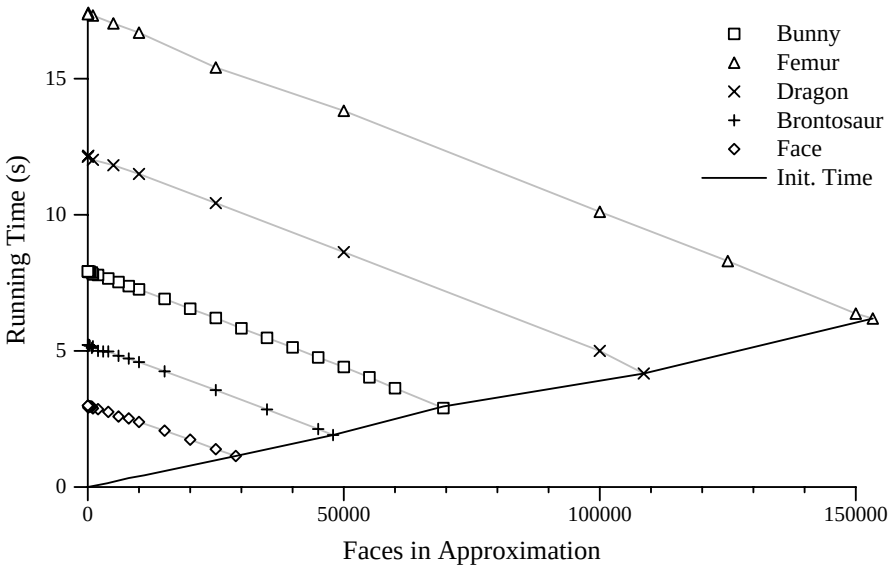


Figure 6.3: Times for medium-size models (n =15,000–150,000 faces).

the literature. As shown in Table 6.3, my system requires at most 7 seconds to produce a simplified bunny. An additional 3 seconds are required to input the original geometry, and outputting a 1000 face approximation requires 0.4 seconds. Thus, my software can generate an approximation of about 1000 faces in a total of 10.4 seconds. For this bunny model, Lindstrom and Turk [124] report running times[2] of 2585 seconds for mesh optimization [95], 500 seconds for Hoppe's progressive mesh construction algorithm [90], 325 seconds for the JADE software [22], and 149 seconds for their own memoryless algorithm. Clearly, my algorithm is substantially faster[3] than these related techniques. The accompanying error measurements indicate that, while my algorithm does not produce the highest quality approximations, its results are competitive.

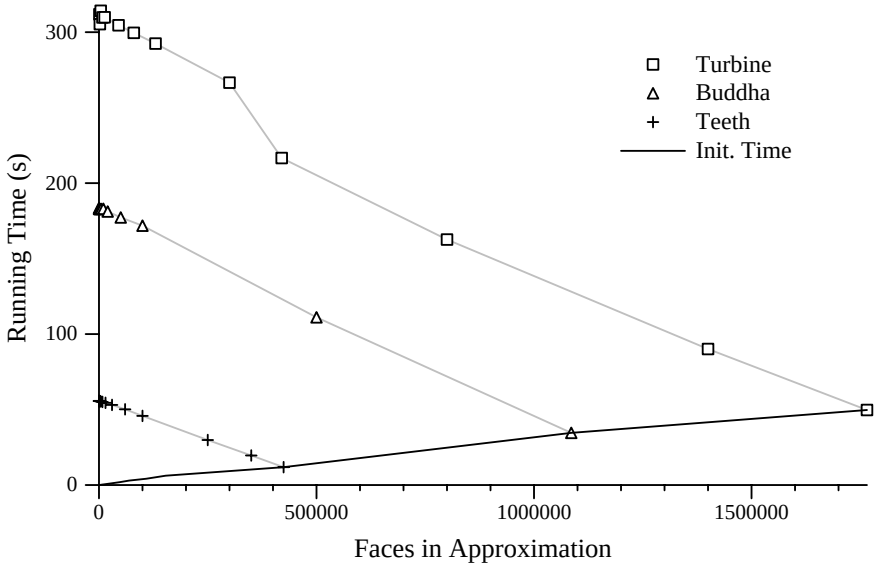


Figure 6.4: Running times for large-size models ($n > 400,000$ faces).

It is also feasible to simplify very large models using my algorithm. Figure 6.4 shows the running times for models (j–l). Note that the running time continues to behave quite linearly. The only significant exception is a jump in the running time on the turbine blade between 400,000 and 300,000 faces. This appears to be caused by a period in which vertices of rather high degree are created. Nevertheless, only 310 seconds are required to simplify this very large model. With an additional 40 seconds for input and output, the total running time to produce a 13,000 face approximation is about 350 seconds. In contrast, Lindstrom and Turk [124] report that their algorithm required just under an hour to produce a similar approximation, and that the other algorithms they

[2] On a 195 MHz R10000 Onyx with 1 GB of memory.

[3] The 32 second running time reported in their comparison for my algorithm is based on an older, and less efficient, implementation.

tested were unable to handle a model of this size.

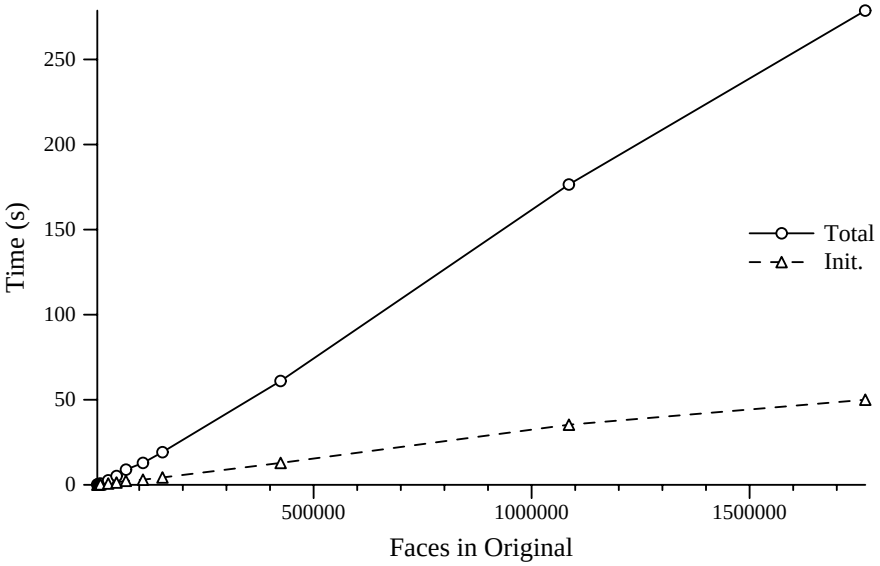


Figure 6.5: Time for total simplification as a function of input size n.

Figure 6.5 provides a slightly different view of running time. The simplification process was run to completion on each sample model; in other words, until all available pairs had been contracted and no faces remained. For greater consistency, these running times were all generated on the R10000 test platform. The previous figures show running times as a function of m for fixed input size. Here we can see how running times vary as a function of input size n. As in the earlier figures, the running time appears nearly linear in the input size, despite the logarithmic terms in the theoretical complexity. This suggests that, in practice, the amount of movement of elements in the heap remains relatively low as simplification proceeds. Notice that total running time is growing much faster than initialization time. For small models, initialization can take around half of the total time (see Figure 6.2). However, as model sizes increase, the process of iterative contraction comes to dominate the running time.

Finally, the choice of placement policy (§3.5) can affect the running time of my algorithm. Figure 6.6 shows running times on the bunny model for five separate strategies. As we would expect, optimal placement is the most expensive policy, and subset placement is the cheapest. Falling between these two extremes are the intermediate policies of selecting the optimal position along the pair segment, and selecting the best of the endpoints or the midpoint. For comparison, I have also shown the running time for face-based contraction (§3.8) using optimal placement. This is faster than all the edge-based policies, primarily because there are 30% fewer faces than edges. However, the price of this increased efficiency is a higher approximation error (see Section 6.2.3).

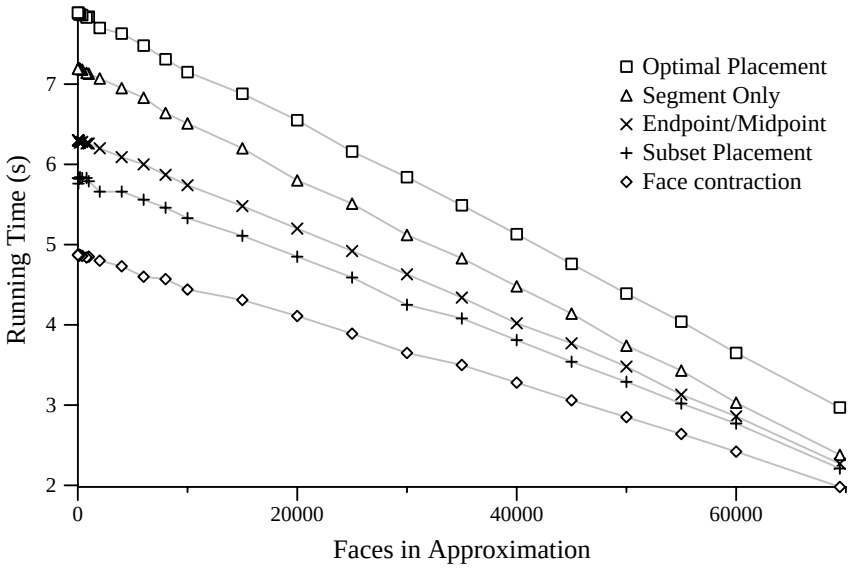


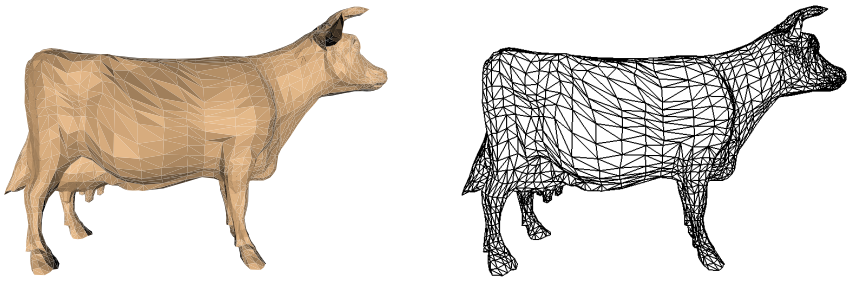
Figure 6.6: Time to simplify the bunny model under different policies. Four vertex placement policies for the edge-based algorithm are shown along with the face-based algorithm (using optimal placement).

6.2 Geometric Quality of Results

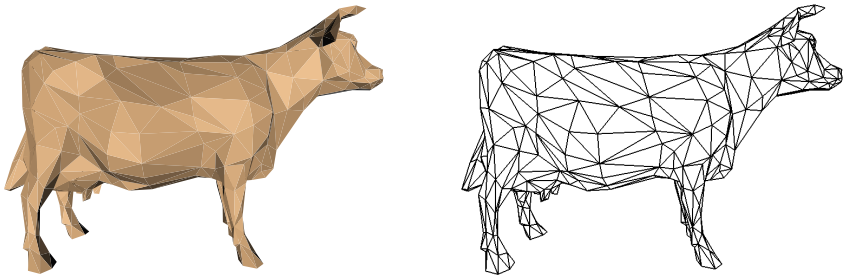
As demonstrated in the previous section, my algorithm can rapidly produce approximations. Even large models with a million faces can be simplified in about 3 minutes. But such efficiency would be useless if the resulting approximations did not remain faithful to the original models. In this section, I will demonstrate that the models produced by my algorithm are in fact of fairly high quality. I will provide some visual examples of the results of quadric-based simplification as well as numerical measurements of geometric error.

Figure 6.7a shows a surface model of a cow, in both shaded and wireframe views. At 5804 faces, this model is not over-tessellated. Most of the features of the surface are described with a fairly small number of triangles. Thus, by simplifying this surface by any significant degree, we must necessarily remove surface features. Note that triangle density is higher around the head and along the legs.

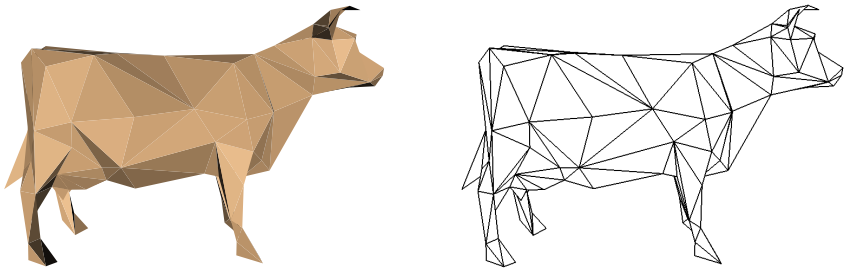
Two different approximations of 1000 and 300 faces are shown in Figures 6.7b–c, respectively. At 1000 faces (17% of the original), the overall shape of the model remains largely unchanged. All the major extremities remain, and have the same, albeit slightly simplified, shape. However, all the finer details of the surface have been removed. In comparison to the original (a), the distribution of triangles is roughly uniform over the surface. The 300 face approximation (5% of the original) is still quite recognizable, but the shape of the extremities



(a) Full Model; 5804 faces

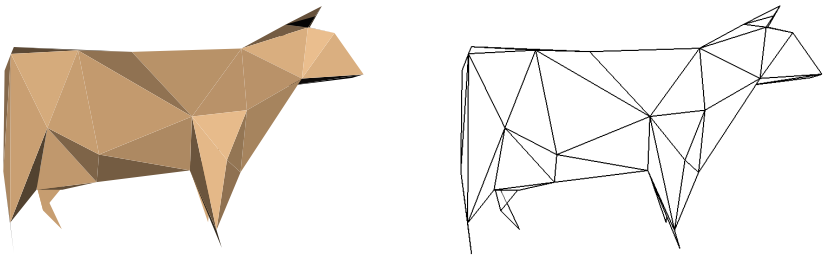


(b) 1000 face approximation

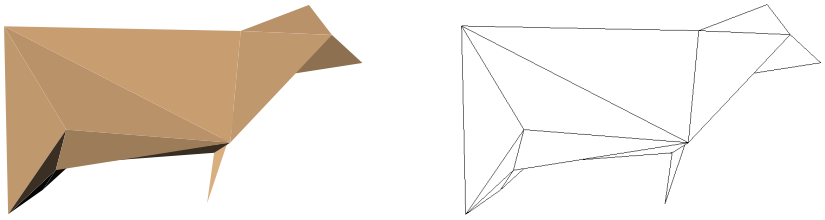


(c) 300 face approximation

Figure 6.7: Sample approximations of the cow model.



(a) 100 face approximation



(b) 30 face approximation

(c) 4 face approximation

Figure 6.8: Further approximations of the cow model.

is becoming more noticeably simplified. For example, the udders have been completely removed and the ears and horns have been simplified substantially.

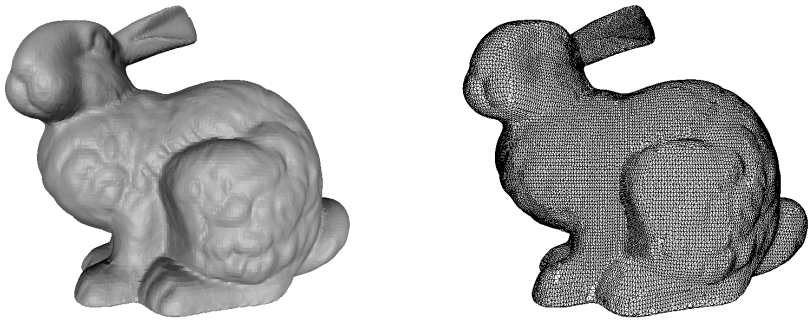
Figure 6.8 shows more aggressive reductions of the cow model to 100, 30, and 4 faces. At the 100 face level, large-scale features of the model are being either removed or substantially simplified. The tail has been removed, the ears and horns have been merged together, and the legs and head have been reduced to very simple shapes. This process has progressed even further in the 30 face approximation. The hind legs have been merged together, and one of the front legs has been removed entirely. This is a highly abstracted version of the original cow, although it does retain a roughly similar size and shape. However, displayed at a very small size, of say 16 pixels, this might be an entirely suitable replacement for the original The final approximation, containing only 4 faces, is no longer even recognizably a cow. While earlier approximations retained a similar size and volume to the original, the volume of this approximation is noticeably smaller. Although not visible from this angle, it is substantially thinner than the original model.

A more interesting example is shown in Figure 6.9, a scanned model of a (possibly chocolate) bunny. The original model (a) has just under 70,000 faces. The surface is fairly densely sampled. Indeed, although the surface is flat-shaded, the triangles are small enough that it appears smooth.

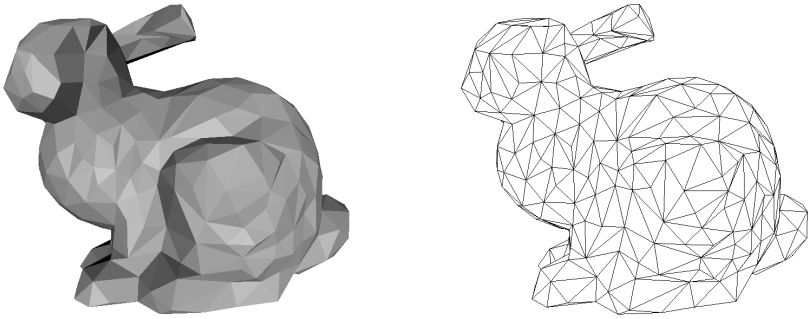
The 1000 face approximation (b) faithfully preserves the shape of the surface. Nearly all the fur texture and facial details have been removed. However, critical features such as the neck and leg lines, and the shape of the head and ears, have been maintained. If a texture map were applied to both (a) and (b), there would be relatively little perceptual difference. A more aggressive approximation using only 150 faces is also shown (c). At this point, most of the detail of the object has been removed, yet the primary features of the model, such as the head, ears, and tail, remain in a recognizable form.

The dragon model shown in Figures 6.10 and 6.11 is a more complex surface, also generated from a scanner. It has a larger number of triangles than the bunny model, and it has more features. Figure 6.11a illustrates the density of the original mesh, also shown in much greater detail in Figure 2.3. Looking closely at Figure 6.10a, we can see striations in the original surface; there is a particularly apparent one along the lower jaw. Notice how even these striations have been preserved in the 25,000 face approximation. This indicates that, as is common with scanned data, the original model was significantly over-tessellated for display at this resolution. At 5000 faces, the main features of the model, such as the teeth, still retain their basic shape. However, notice that the fine details of the surface, including the striations, have been removed. This agrees with the observation of Section 4.4.2 that the quadric error metric attempts to produce smoothed approximations of smooth surface regions. In the final approximation, containing only 1000 faces, the facial detail and fangs have disappeared. However, all the extremities remain and the larger scales along the back have also been preserved.

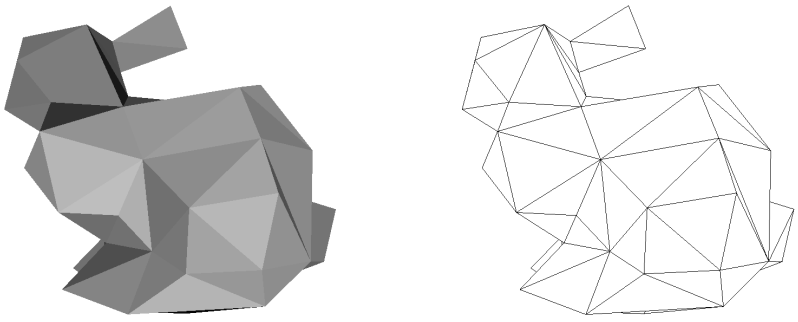
Figure 6.12a shows a scanned dental model containing about 420,000 faces. While quite accurate, it is too bulky to allow direct interactive manipulation by



(a) Original model; 69,451 faces



(b) 1000 face approximation



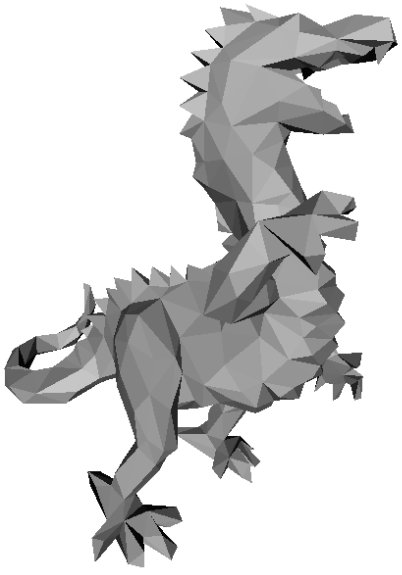
(c) 150 face approximation

Figure 6.9: Sample approximations of the bunny model.

(a) Original model; 108,588 faces

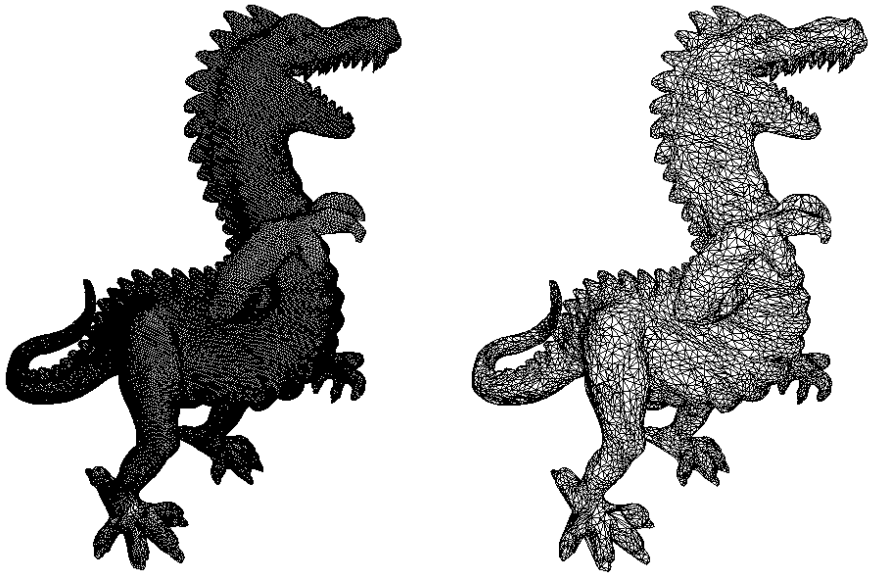
(b) 25,000 face approximation

(c) 5000 face approximation



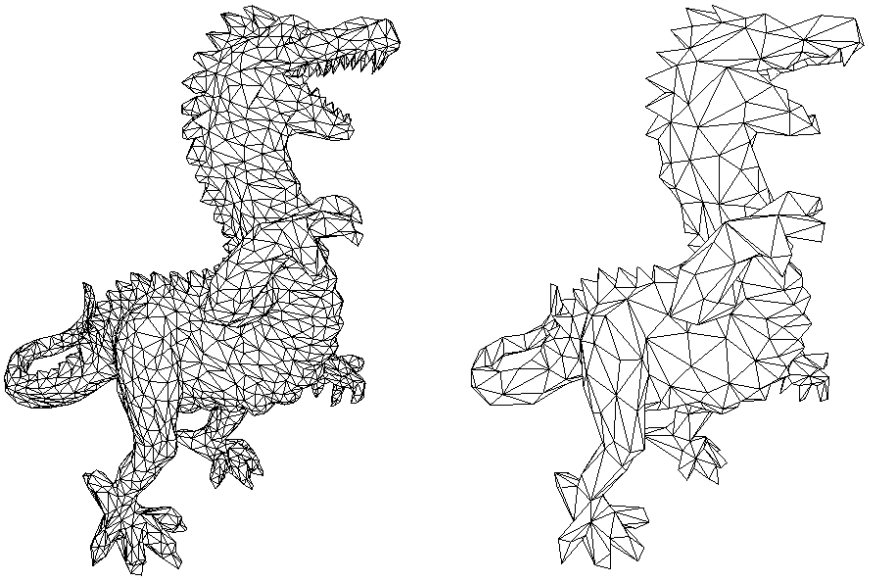
(d) 1000 face approximation

Figure 6.10: Sample approximations of the dragon model.



(a) Original; 108,588 faces

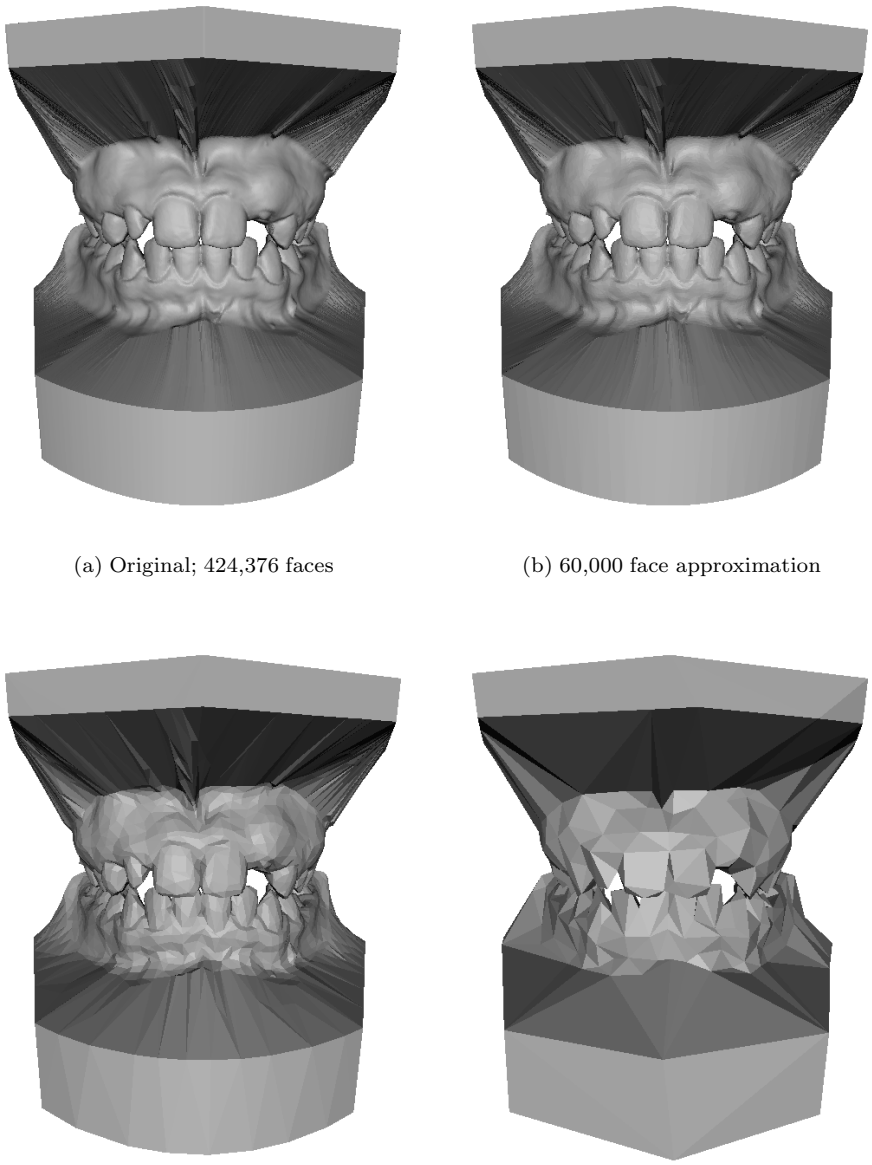
(b) 25,000 face mesh



(c) 5000 face mesh

(d) 1000 face mesh

Figure 6.11: Meshes for surfaces shown in Figure 6.10.



(a) Original; 424,376 faces

(b) 60,000 face approximation

(c) 8000 face approximation

(d) 1000 face approximation

Figure 6.12: The dental model and three sample approximations.

a user (i.e., an orthodontist). Because this surface is uniformly tessellated, it wastes many triangles on the flat bases above and below the teeth. The 60,000 face approximation (b) is much more economical with its triangles, and retains almost all the detail of the original surface. The 8000 face approximation (c) sacrifices some quality, but it can be rendered fairly comfortably on even modest machines without graphics acceleration. Finally, the 1000 face approximation maintains the overall structure of the model; all the teeth are still present. It would be a suitable replacement if the model were being viewed from a distance, which is admittedly unlikely given the probable application of this dataset. However, if we wanted to compute some gross property of the surface, such as volume for instance, we could produce a good first estimate from this model. This estimate could be refined by refining the model, and in the end, we would achieve a result much more quickly than if we had begun with the original data.

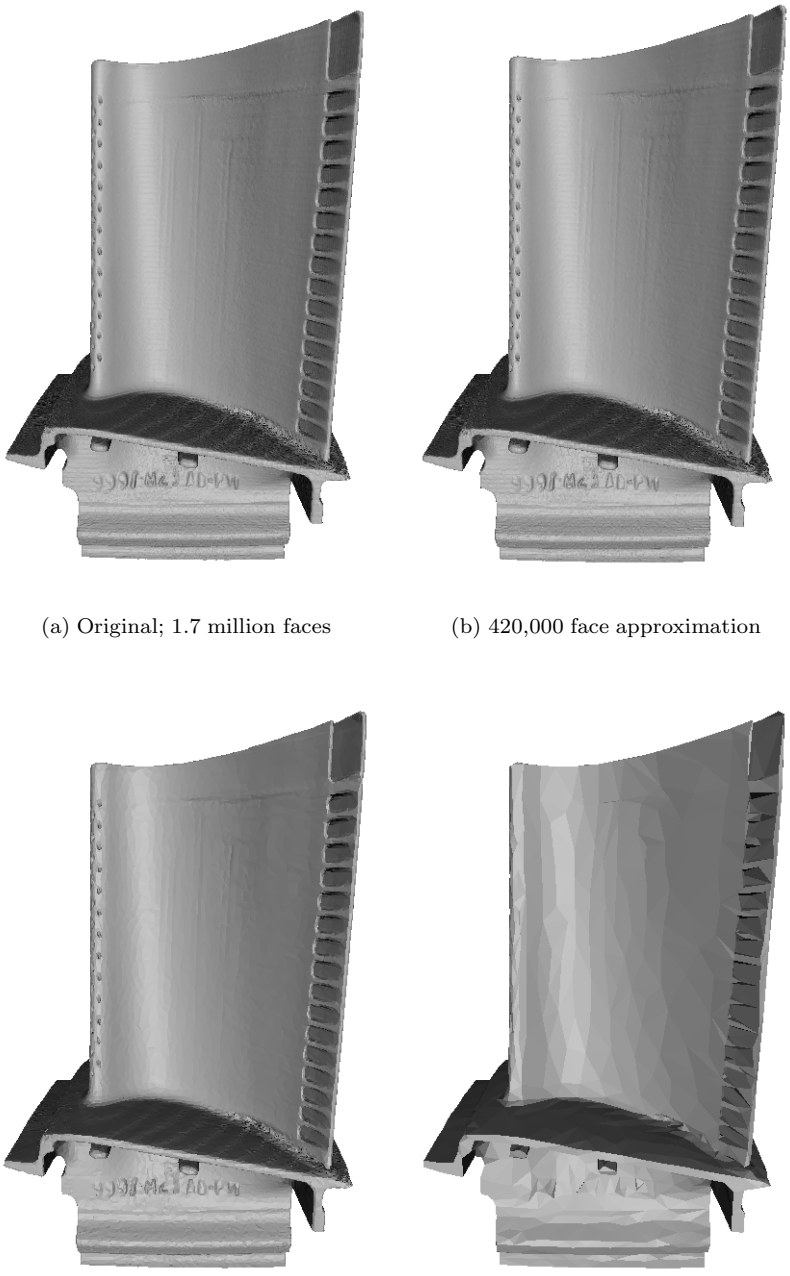
The final example model is the turbine blade shown in Figure 6.13. This surface appears to have been extracted from a volumetric dataset. At over 1.7 million faces, this model (a) is quite large. While not visible, it also has a fairly complex internal structure, with hollow tubes running through the interior. This surface is very densely sampled. The 80,000 face approximation (c) is a substantial reduction, but it remains very faithful to the original. In addition, some of the noise present in the plate below the blade has been smoothed away. Even at 8000 faces (d) the simplification preserves the primary features of the model while smaller features, such as the label on the base, have been removed. Visually, these approximations compare quite favorably to those of Lindstrom and Turk [124] and appear, at low resolutions, to be significantly smoother than those generated by Schroeder [171, Fig. 9]. Many other algorithms are incapable of simplifying a model of this size, either because of prohibitive memory consumption or extremely long running times [124].

6.2.1 Approximation Error

The pictures of sample approximations which I have just presented provide useful visual information about the quality of the results generated by my algorithm. But to gain a more precise understanding of approximation quality, we need to measure the geometric approximation error of the results as well. All error measurements in this section are based on the sampled E_{avg} (2.8) metric defined in Section 2.3.2. For each of the two surfaces, the corresponding set of sample points consists of the vertices of the model and an additional 100 samples taken over each face.

Figure 6.14 shows the approximation error E_{avg} as a function of approximation complexity for five sample models. To allow for comparisons between models, all error values are normalized by the squared[4] diameter of the models. As we might expect, the blobby "V" model and the sphere exhibit the lowest error profile during simplification. Both these models are very smooth and can be approximated efficiently with fairly simple meshes. Of all the models shown,

[4] The diameter is squared because E_{avg} measures squared distances.



(a) Original; 1.7 million faces

(b) 420,000 face approximation

(c) 80,000 face approximation

(d) 8000 face approximation

Figure 6.13: Sample approximations of the very large turbine blade model.

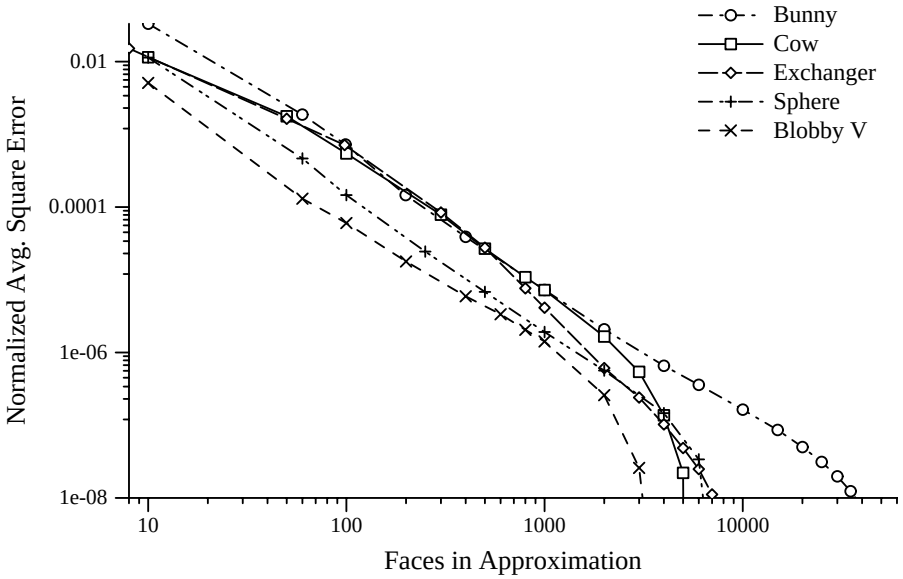


Figure 6.14: Approximation error E_{avg} as a function of approximation size m.

the cow seems to exhibit the highest rate of error growth during early phases of simplification (at the right end of the graph). This also seems quite sensible; the cow has a comparatively more complex shape and fewer than 6000 triangles to begin with. Notice that after the initial phases of simplification, all the error curves become nearly linear, with the sole exception of the heat exchanger. Since this is a log–log graph, this suggests that the error and number of faces of a particular approximation may be related by a power law. Ignoring the first one to two data points, where the curves are less linear, all of these curves are quite closely fit by a function of the form $E_{avg} = \alpha m^{-2}$.

Figure 6.15 provides an alternate view of the error measurements shown in Figure 6.14. However, instead of showing error as a function of approximation size, it shows error as a function of the remaining fraction of the original faces. Keep in mind that, in this case, the horizontal axis uses a linear scale, as opposed to the log scale in Figure 6.14. Here we see that the cow has the highest error profile. Again, this is because of the relative complexity of the model in comparison to the number of triangles in the original. However, the bunny model exhibits the lowest error profile in this graph. This indicates that the original model was somewhat over-tessellated in relation to the complexity of its shape. A significant number of the original faces can be removed without introducing sizeable error.

Notice that the overall trend in approximation errors shown is fairly consistent. All the curves have the same basic shape, even though the models being simplified are quite different. One important trend that becomes apparent is

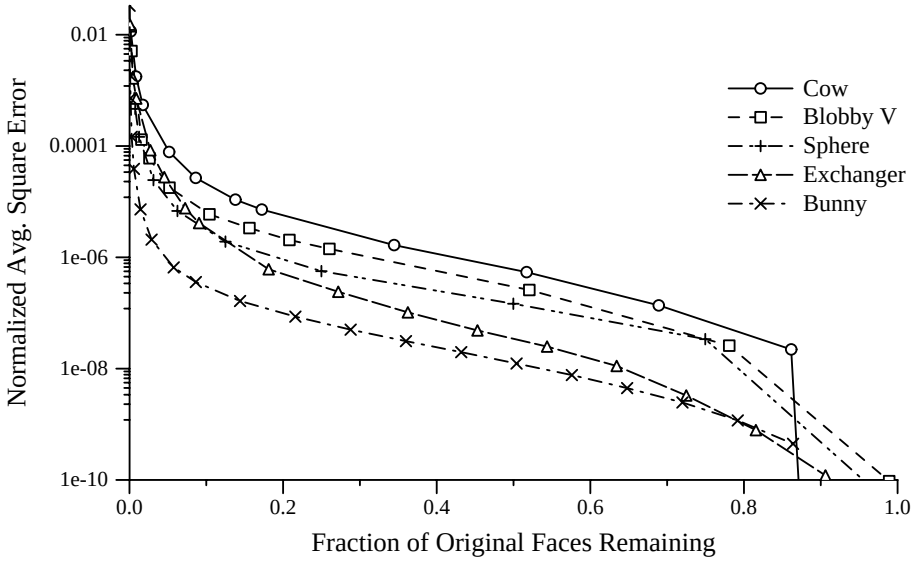


Figure 6.15: Approximation errors from Figure 6.14 as a function of the fraction of original faces remaining.

that the approximation error remains very low for quite some time. All but one of the models have an approximation error below 10^{-6} even when only 30% of the original faces remain. Near the end of simplification, when only a very small fraction of the original triangles remain, the error increases very rapidly. This occurs because various parts of the model begin to collapse. For example, consider the dragon in Figure 6.10d. In the larger approximations, the shape of the teeth has been simplified but all the teeth remain in the model. However, at this level of simplification (1000 faces) the teeth have begun to disappear entirely. As larger and larger features are removed, the error increases more rapidly.

Finally, I have claimed that my quadric-based simplification algorithm provides a good compromise between the speed of simplification and the quality of the resulting approximations. Figure 6.16 illustrates this tradeoff[5]. Unlike most performance comparisons [26, 124] which focus on error as a function of approximation size (as in Figure 6.14), this graph shows running time as a function of error. The graph reflects the simplification of the bunny model using several different algorithms. Running times reflect performance on an R4400 SGI Indigo2, and the error values, generated by a metric essentially identical to E_{avg}, were measured using the Metro [27] model comparison tool. Corresponding points on the graph represent approximations with roughly identical vertex

[5] This data is taken, with permission, from the survey of Cignoni *et al.* [26]. They present the standard graphs of error vs. approximation size rather than this time vs. error tradeoff graph.

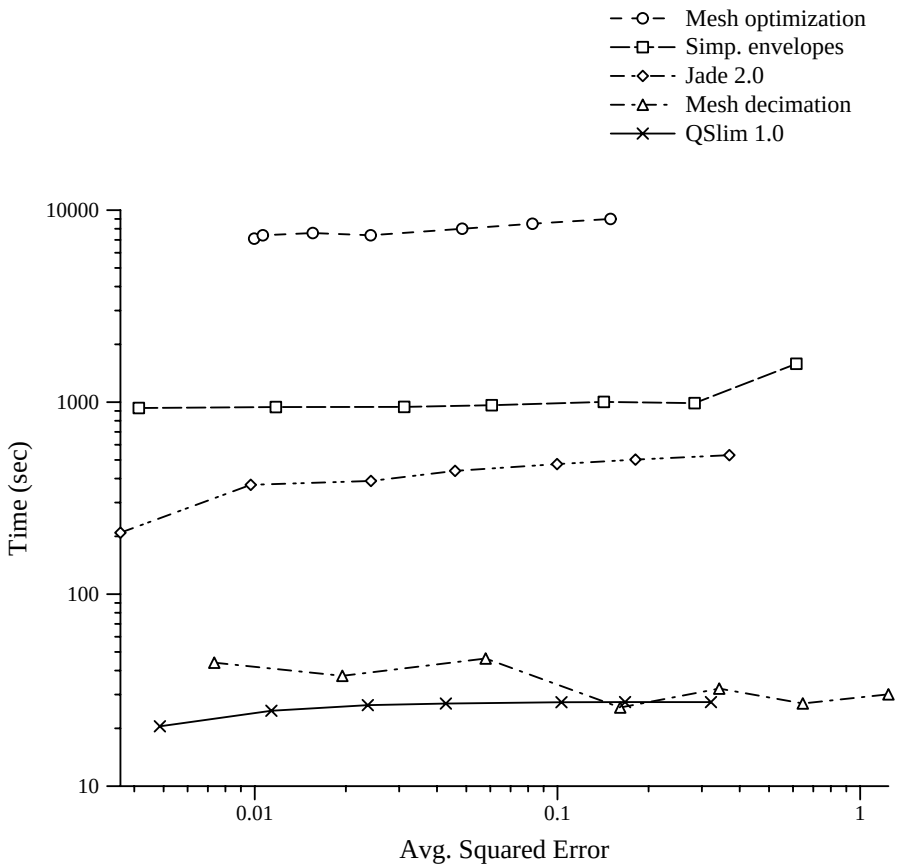


Figure 6.16: Comparison with other methods: Tradeoff of approximation error vs. running time for simplifying bunny model.

counts. From left to right, the approximations contain 50%, 25%, 10%, 5%, 2%, 1%, and 0.5% of the original vertices. A good way to read this graph is to consider a particular vertical line. This will tell you, for a given error, how long it will take to produce an approximation with that error.

As we would expect, the mesh optimization procedure of Hoppe *et al.* [95] is by far the most expensive algorithm tested, requiring 1–3 orders of magnitude more time than the other algorithms. However, at low approximation levels, it also produces the best results. For instance, the last point on each curve represents an approximation with 0.5% of the original vertices. Mesh optimization produces the lowest error, and thus the highest quality result, at this level. Interestingly, it produces the highest error at the 50% reduction level.

The next two algorithms below mesh optimization are the the simplification envelopes algorithm of Cohen *et al.* [31], and the JADE vertex decimation system by Ciampalini *et al.* [22]. Simplification envelopes provides tightly controlled error bounds on the approximations it generates, and it preserves the topological genus of the object. The JADE system uses an error metric based on the localized Hausdorff distance (§2.3.2). In this respect, it is similar to several other algorithms [179, 106, 108] which probably produce similar results. By comparison to mesh optimization, these algorithms provide a significant savings in running time, but produce approximations with higher error at higher levels of simplification.

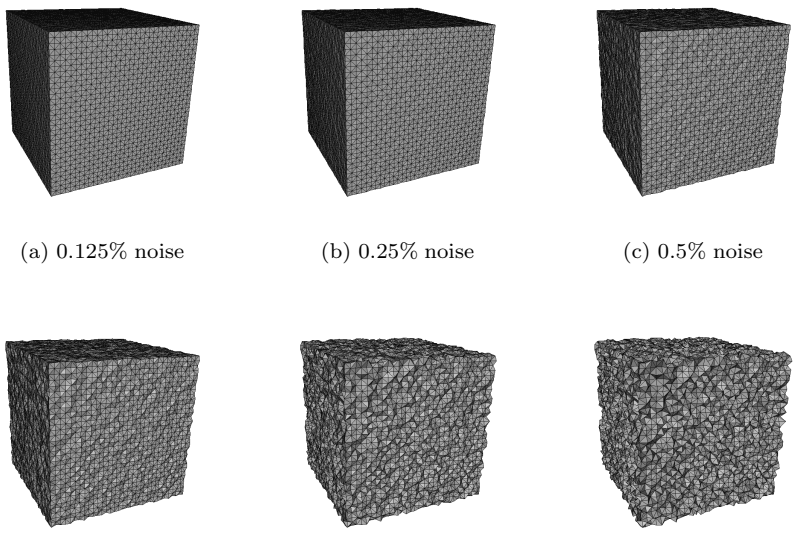
The final two algorithms are the vertex decimation algorithm of Schroeder *et al.* [172], and my own QSlim implementation of quadric-based simplification [68]. Both are substantially faster than the other algorithms. But while the error of the approximations generated by my algorithm is quite competitive with the other algorithms, the error resulting from vertex decimation grows quite rapidly. In fact, at the 0.5% reduction level, vertex decimation produces the highest error, while only mesh optimization produces a lower error than my quadric-based system.

This example demonstrates quite clearly the mix of speed and approximation quality provided by quadric-based simplification. It is able to rapidly produce approximations whose error is competitive with significantly more expensive methods. Vertex decimation, which is comparable in running time, produces approximations of appreciably higher error for a given number of output faces. As an aside, I should mention that the data in this graph reflect the performance of the first release of my QSlim software [68]. The performance data which I have detailed in previous sections are based on the second release of that software, which is approximately twice as fast as the original.

6.2.2 Performance on Noisy Data

There is no explicit mechanism for handling noisy data in the quadric error metric. Quadrics are constructed directly from the faces of the original surface. Any noise will be treated as a surface feature. In application areas where noise is common, prefiltering of the surface normals would help to reduce noise in the approximation process. However, without regard to the presence of noise,

the quality of approximations degrades gracefully for noisy models. In fact, one side-effect of quadric-based simplification is a smoothing process which can partially remove some noise.



(a) 0.125% noise (b) 0.25% noise (c) 0.5% noise

(d) 1% noise (e) 2% noise (f) 3% noise

Figure 6.17: Noisy cubes — original 12,288 triangle resolution.

Consider the cubes shown in Figure 6.17. Each model has approximately 12,000 faces. They are generated from a unit cube by iteratively subdividing the initial 12 triangles. After subdivision completed, all the newly created vertices were displaced by a random amount along their average surface normal. The size of this displacement is measured as a percentage of the model diameter. In this example, noise levels range from 0.125%–3%. Notice that models (a–c) are reasonably close to cubical, while model (d) is noticeably rough, and models (e–f) appear quite noisy.

To see how my simplification algorithm deals with this noisy data, I produced several approximations for each of these cubes. For each approximation, I measured the approximation error E_{avg} between that approximation and the underlying unit cube (as opposed to the original model). Figure 6.18 shows the results of this experiment. Notice that as simplification proceeds, the measured error between the approximation and the unit cube initially decreases. In effect, the surface is being smoothed and noise is being removed. The lends support to the analysis of quadric errors at the end of Section 4.4.2. I suggested that the form of the error for vertices positioned by minimizing the quadric error is related to a surface smoothing operation. We can see this kind of smoothing

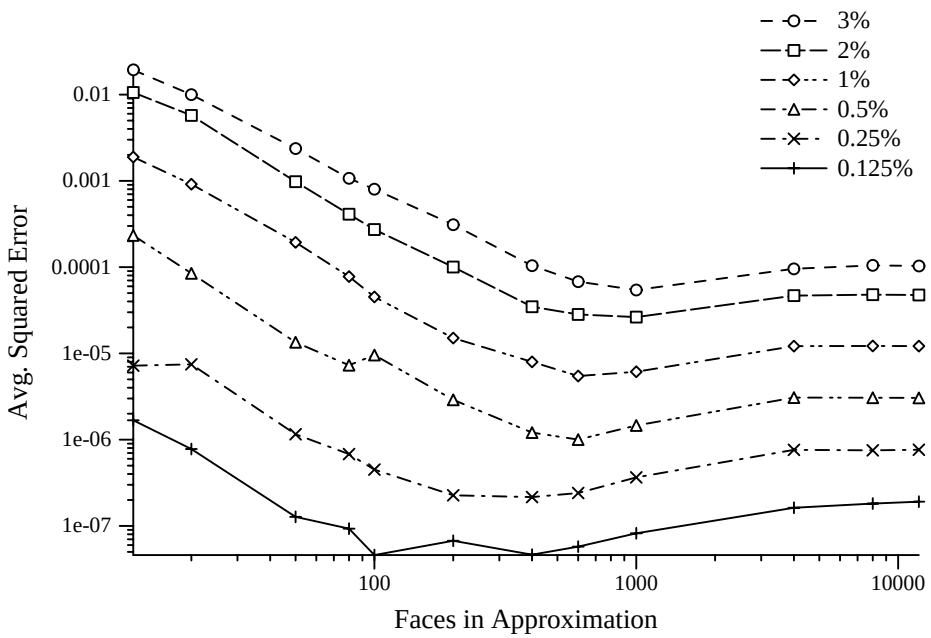


Figure 6.18: Approximation error for noisy cubes (see Figure 6.17).

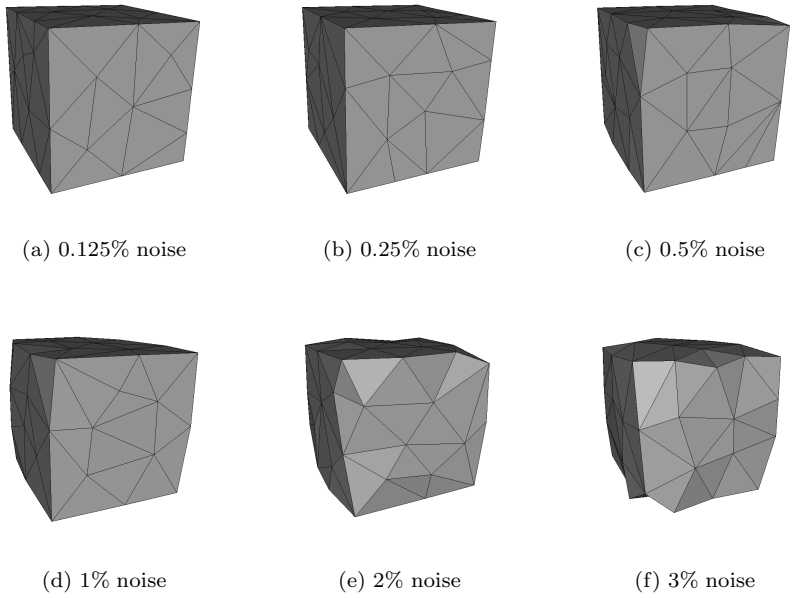
happening in this example. Naturally, the error will ultimately increase as fewer and fewer triangles are used, but for the tests with less than 0.5% noise, the maximum error reached remains very small (less than 10^{-5}).

Figure 6.19 shows the 100 face approximations generated by simplifying the cubes in Figure 6.17. As you can see, models (a–b) are almost perfectly cubical and model (c) exhibits a few small deviations, such as the rounding of the closest visible corner. Model (d) is more noticeably bent around the corners, and models (e–f) are certainly less cubical, although they do have identifiable sides and they are roughly the size of the unit cube.

A minimal triangulation of the unit cube contains 12 faces, and the corresponding approximations are shown in Figure 6.20. The two models with the least noise are almost exactly cubical; as shown in Figure 6.18, their approximation errors are very low. In effect, the process of simplification has successfully removed the noise that was originally added to these models. Models (c–e) appear to be increasingly warped cubes, but each still has four essentially planar quadrilateral sides. The final model (f) no longer retains the shape of a cube, and it appears somewhat shrunken.

6.2.3 Effect of Alternate Policies

As illustrated by Figure 6.6, the choice of placement policy affects the running time of the system. Naturally, it also affects the quality of the results. Figure



(a) 0.125% noise (b) 0.25% noise (c) 0.5% noise

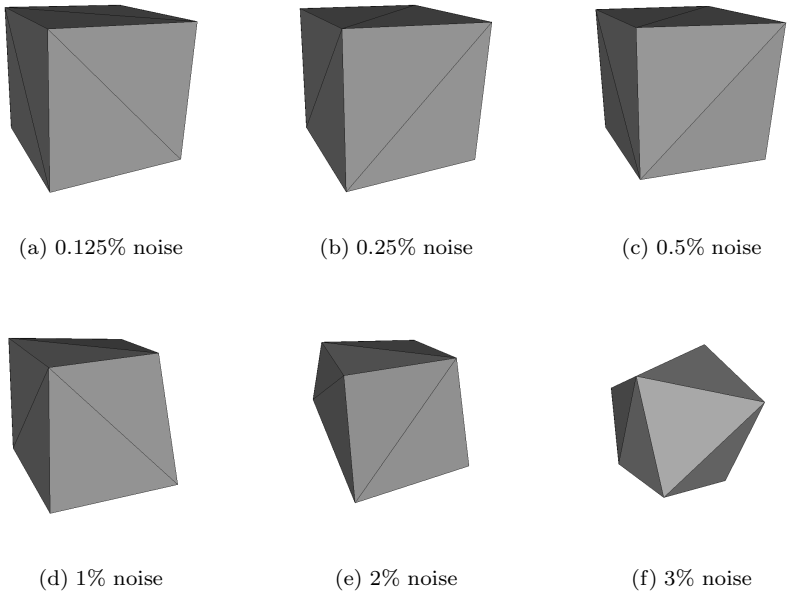
(d) 1% noise (e) 2% noise (f) 3% noise

Figure 6.19: Noisy cubes from Figure 6.17 simplified to 100 triangles.

6.21 shows the results of simplifying the cow model using three different policies: edge contraction with optimal placement, edge contraction with subset placement, and face contraction using optimal placement. Each approximation contains 300 faces, or 5% of the original triangle count.

By visual inspection, we can see that edge-based optimal placement (a) produces the best result. In particular, the definition of the head and horns is superior to the alternatives. Notice how the head has been rounded out, and the rear leg twisted, by subset placement (b). At this level of approximation, face contraction (c) produces results similar to edge contraction, both using optimal placement. The horns and head are not quite as well defined, but the overall quality is competitive.

Figure 6.22 provides a broader view. Initially, face contraction produces the worst results; however, when approaching lower face counts, it performs somewhat better than edge-based subset placement. With one exception, edge-based optimal placement performs consistently better than the other policies. This difference is even more apparent when we examine the actual error values, as in Table 6.4. Subset placement tends to produce a 30–50% higher error than optimal placement. Face-based optimal placement, while it has initially twice the error of edge-based optimal placement, narrows the gap at lower face counts.



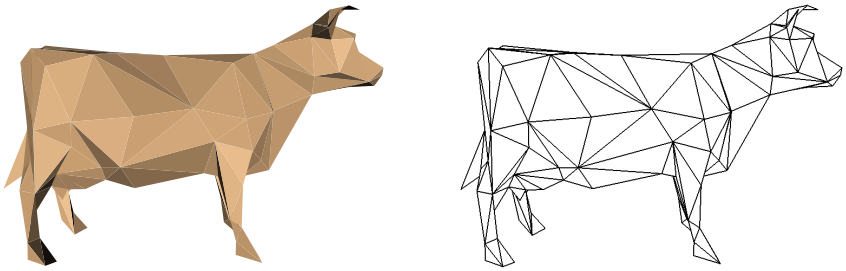
(a) 0.125% noise (b) 0.25% noise (c) 0.5% noise

(d) 1% noise (e) 2% noise (f) 3% noise

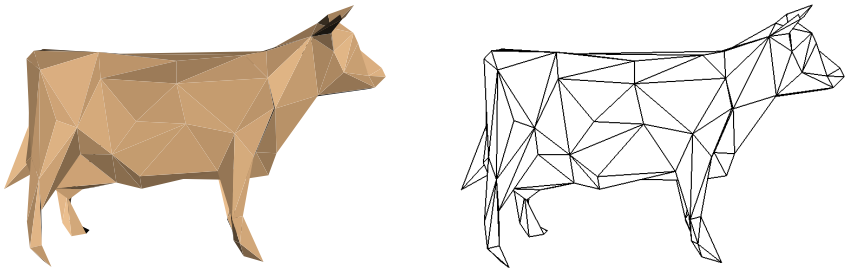
Figure 6.20: Noisy cubes from Figure 6.17 simplified to 12 triangles.

	Approximation Error E_{avg}				
Faces	Optimal	Subset	% change	Face	% change
10	0.00545769	0.00784432	43.7	0.00645922	18.4
100	0.000259639	0.000352787	35.9	0.00029681	14.3
500	1.27412e-05	2.03091e-05	59.4	1.73388e-05	36.1
1000	3.43394e-06	5.18089e-06	50.9	5.03303e-06	46.6
3000	2.57919e-07	3.39024e-07	31.4	4.17302e-07	61.8
5000	1.05293e-08	1.12268e-08	6.62	2.26776e-08	115

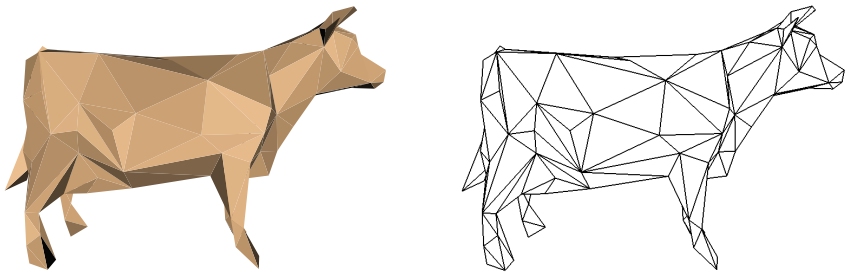
Table 6.4: Numerical error values shown in Figure 6.22 along with percentage change from optimal placement.



(a) Edge-based decimation; Optimal placement



(b) Edge-based decimation; Subset placement



(c) Face-based decimation; Optimal placement

Figure 6.21: Different approximations, all with 300 faces, of the cow using different simplification policies.

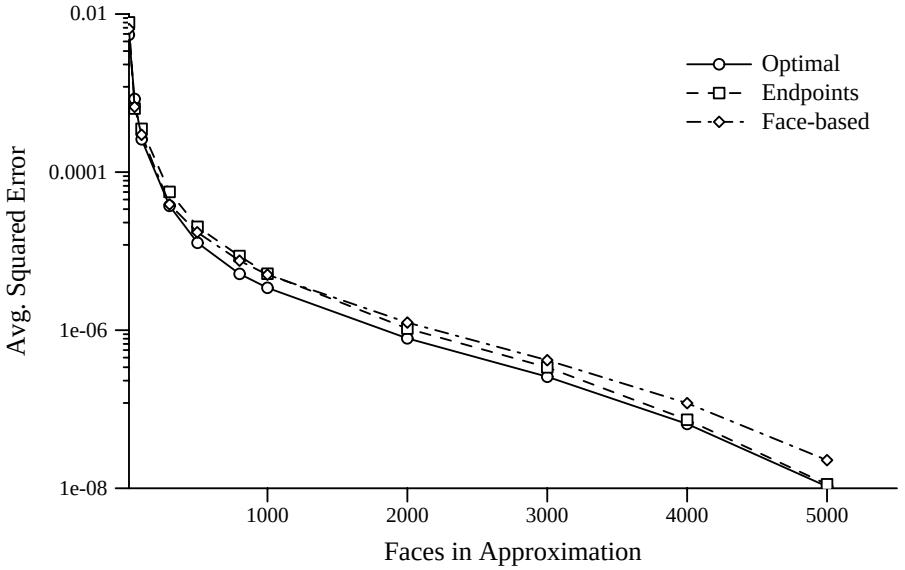


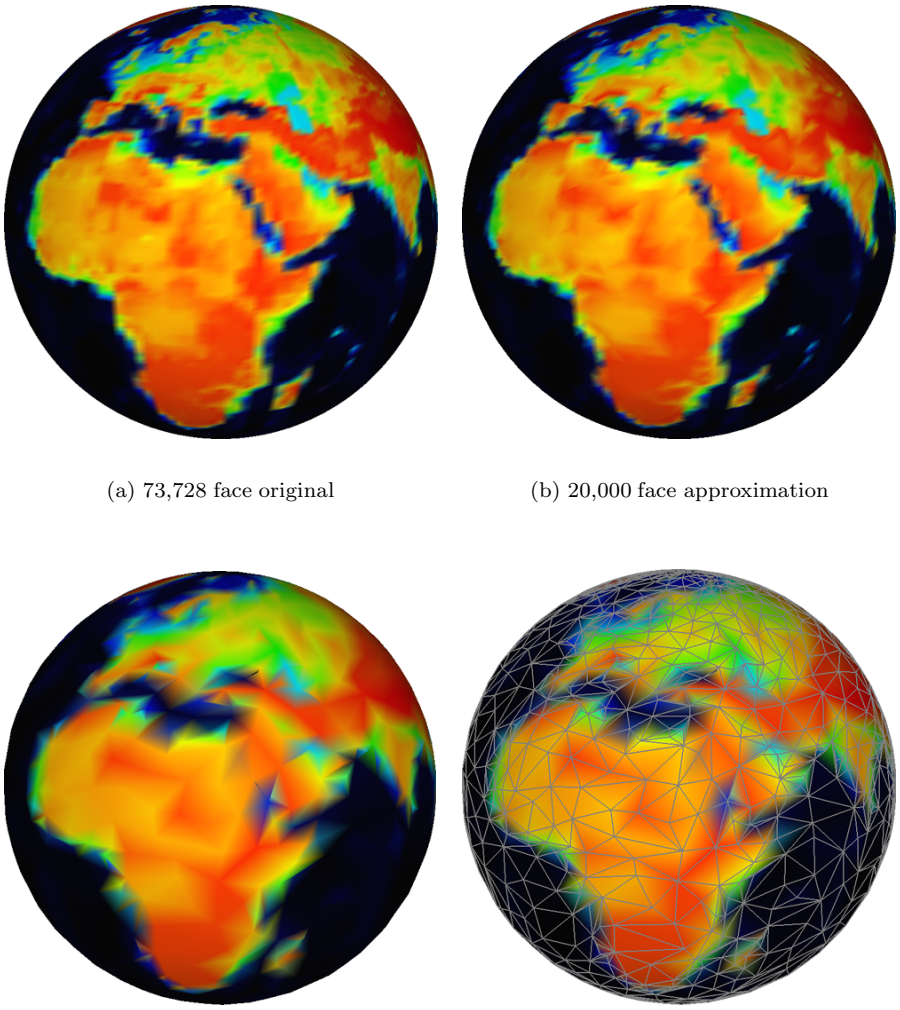
Figure 6.22: Error E_{avg} for cow approximations using different policies.

6.3 Surface Property Approximations

All the sample approximations examined in previous sections consist of purely geometric surfaces. By applying the extended quadric metric developed in Chapter 5, my simplification algorithm can also produce approximations for surfaces with per-vertex attributes. In this section, I will consider two of the most common attribute types: RGB color and texture coordinates. As we will see, the extended error metric provides an effective means for producing approximations of such surfaces.

Consider the Gouraud-shaded example shown in Figure 6.23. The surface itself is a sphere built from points on a latitude–longitude grid. Each vertex is assigned a color based on the elevation of the Earth's surface at that point. Since it is so simple, the surface shape is preserved very accurately. More importantly, the coloring of the surface is also preserved well. The coastlines, which represent the greatest color discontinuities, are simplified but still quite discernible. Note that larger triangles appear in areas of constant or linear color variation, such as oceans, while smaller triangles occur along the coastlines.

Figure 6.24 shows another Gouraud-shaded example. Each vertex has one RGB value which was computed by a radiosity system. All visible colors are due to the radiosity solution; no display-time lighting calculations are being performed. Radiosity meshes such as this are a prime application of my algorithm. Even the simplest geometries are often carved up into very small polygons to achieve high-quality shading. The resulting surfaces are ripe for simplification; typically, the surface geometry is heavily over-sampled and the



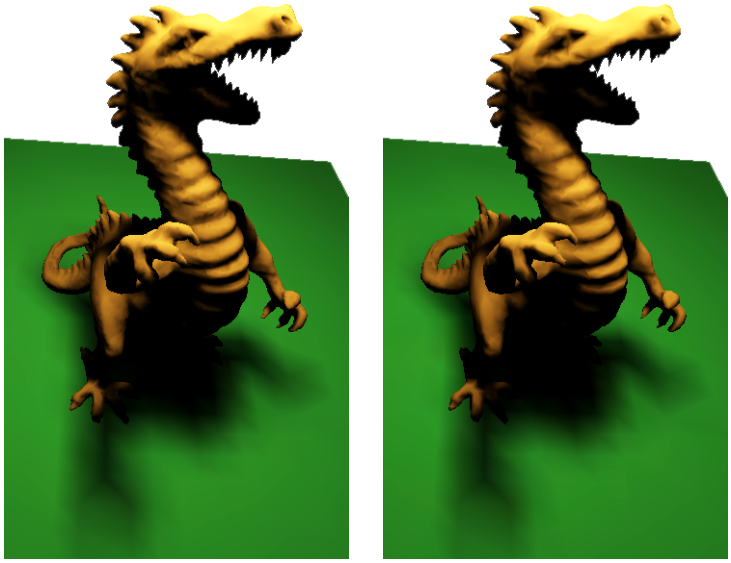
(a) 73,728 face original

(b) 20,000 face approximation

(c) 3000 face approximation

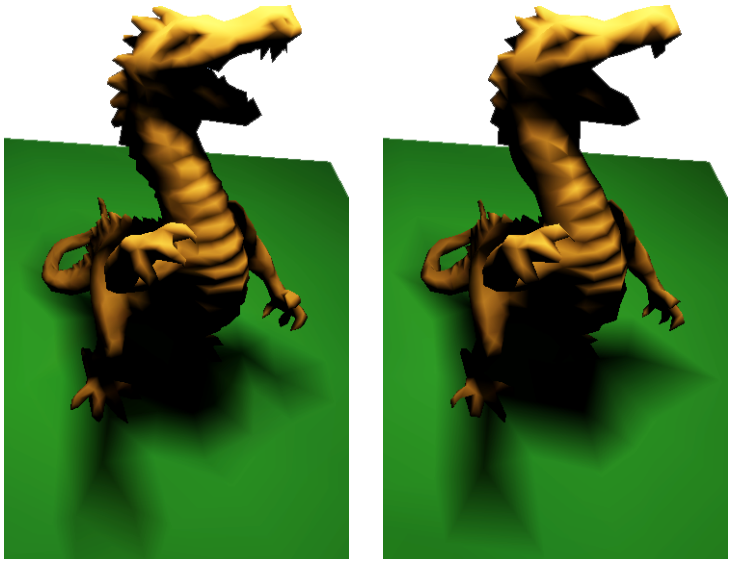
(d) 3000 face mesh

Figure 6.23: A Gouraud-shaded sphere, where each vertex has been colored according to the elevation of a point on the surface of the Earth.



(a) Original; 50,761 faces

(b) 10,000 face approximation



(c) 3000 face approximation

(d) 1500 face approximation

Figure 6.24: A scene shaded with a radiosity solution. Each vertex has one resulting RGB color; no other lighting calculations are being performed.

colors vary smoothly over large areas. For example, the base plane outside the shadow of the dragon is colored with a nearly constant color, yet it contains many triangles.

The original model shown in Figure 6.24a contains just over 50,000 faces. The dragon is a pre-simplified version of the model shown in Figure 6.1h. The approximation (b) contains only 10,000 faces, yet preserves the surface shape and the shading as well. Models (c–d) are more radically simplified approximations. Geometric features such as the teeth, and shading features such as the shadow under the neck, are gradually removed.

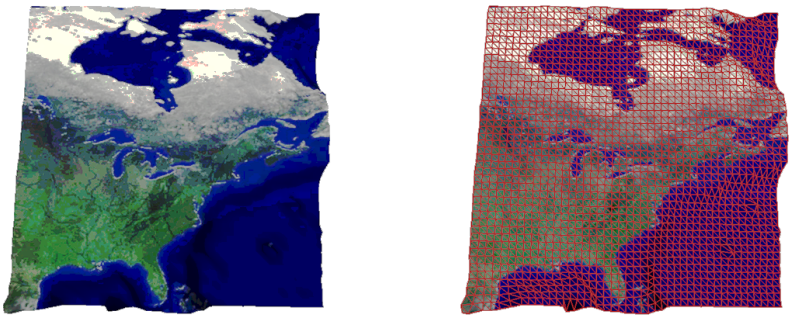
The generalized quadric error metric can just as easily be used to simplify surfaces with texture maps. In Figure 6.25a, we are looking down at a square height field of the eastern half of North America. The surface is textured with a satellite photograph with height given by altitude and bathymetric data.

Figure 6.25b shows the result of simplifying this surface using optimal placement without regard for the texture coordinates. For each contraction, we simply propagate the texture coordinates of $\mathbf{v}_1$. As we would expect, this produces very poor results. Moving vertices without synthesizing correct texture coordinates causes the texture to warp like a rubber sheet. For example, note the substantial distortion around Florida and New England.

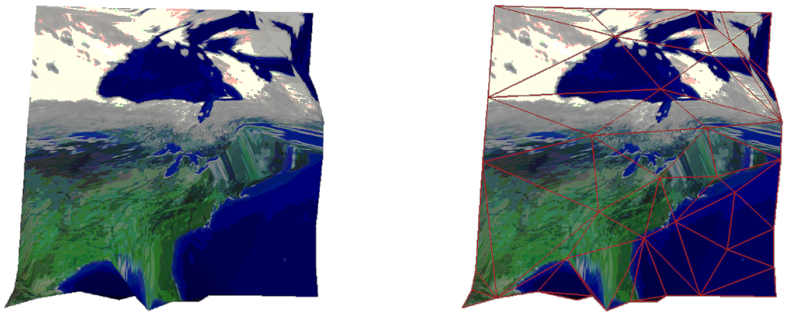
In contrast, the approximation shown in Figure 6.25c, with the same number of faces, was produced using an *xyzst* extended quadric. The algorithm produces nearly the same geometric surface as before. However, by using the extended metric, I have allowed the algorithm to synthesize appropriate texture coordinates for the new vertices. Unlike colored surfaces (such as Figure 6.24), the edges of the mesh do not align with features in the texture. The simplification algorithm never accesses the pixels of the texture; it merely tries to update texture coordinates so that the texture is mapped onto the surface using the same parameterization as in the original.

In the example of Figure 6.25, there is a direct correspondence between (x, y) and (s, t). This would provide an alternate way to synthesize new texture coordinates: for any vertex position, we can use x and y to compute appropriate s and t values. While this only works for height field surfaces, a more general alternative presents itself. For a proposed contraction $(\mathbf{v}_i, \mathbf{v}_j) \to \bar{\mathbf{v}}$, we can project the position $\bar{\mathbf{v}}$ onto the nearest point in the neighborhood of $\mathbf{v}_i$ and $\mathbf{v}_j$ prior to performing the contraction. This projected point will fall within one of the triangles of the local neighborhood. Each vertex of this triangle will have an associated attribute, and we could synthesize an attribute value for $\bar{\mathbf{v}}$ by linearly interpolating the value at its projection from the vertices of the surrounding triangle. As the following example demonstrates, this method of *local reprojection* works well when texture coordinates are an affine function of position. But when the texture parameterization is more non-linear, the generalized quadric error metric produces substantially better results.

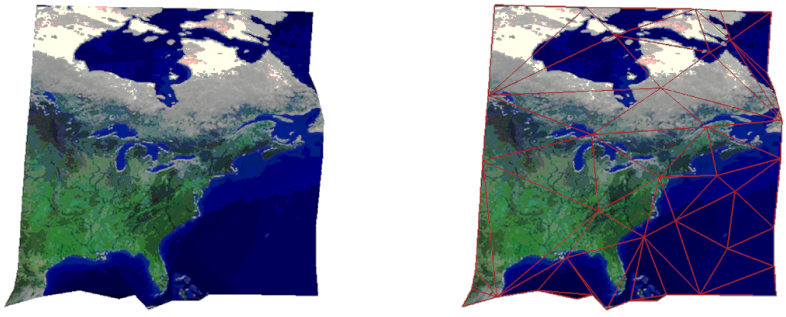
Consider the two height field surfaces shown in Figure 6.26. Each surface is a piece of a hemisphere, curving up toward the viewpoint, defined by tessellating the domain $x, y \in [0, 1]$ with 19,602 triangles. Geometrically, the surfaces are identical and they have been textured with the same checkboard image, but



(a) 3872 face model

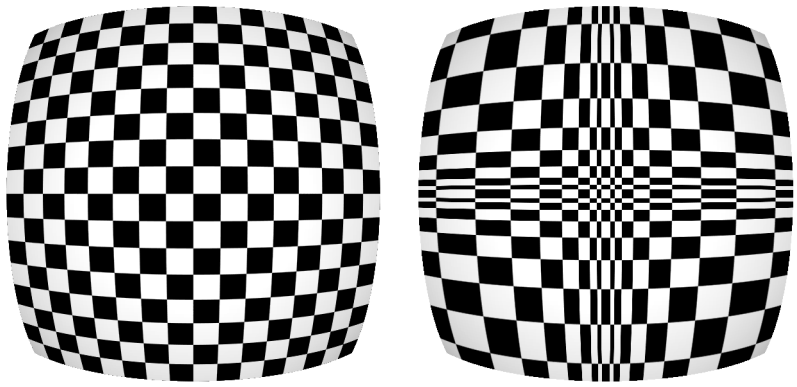


(b) Simplified without synthesizing new texture coordinates



(c) Simplified using generalized quadric metric

Figure 6.25: Simplifying a texture mapped model of North America. Using the generalized quadric metric, we can synthesize new texture coordinates during simplification.



(a) Direct mapping (b) Warped mapping

Figure 6.26: Two identical spherical sections (19,602 triangles) textured with the same image. Only the texture parameterization differs between them.

the mappings of this texture onto the surfaces differs significantly. On the one hand, model (a) is parametrized using a direct correspondence

$$s = x \quad \text{and} \quad t = y \tag{6.1}$$

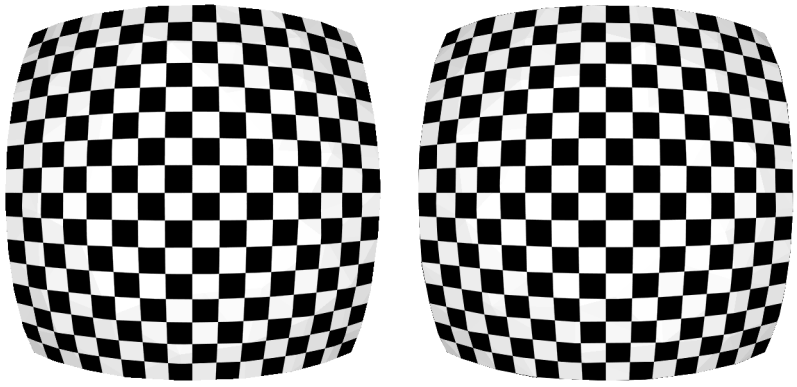
between a vertex's position (x, y) and its texture coordinate (s, t). This is the same simple mapping used in Figure 6.25. On the other hand, model (b) is parametrized using the more complex mapping

$$s = |x - 0.5|^{1/4} \quad \text{and} \quad t = |y - 0.5|^{1/4} \tag{6.2}$$

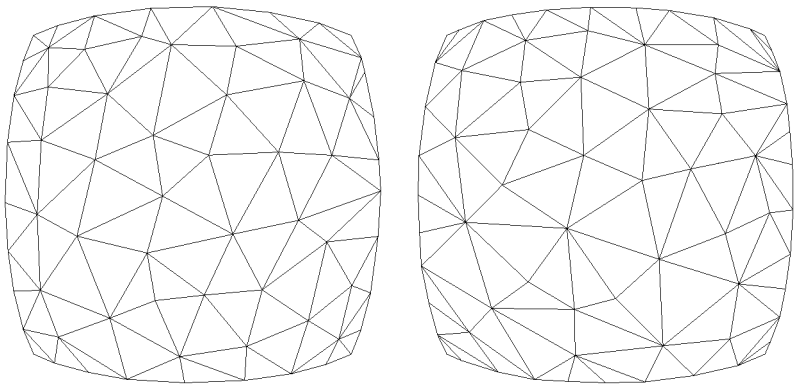
which results in a warping of the texture image. It also causes the same corner of the image to be applied to each of the four quadrants of the surface.

Figure 6.27 compares the results of simplifying the example shown in Figure 6.26. On the left, the surface is simplified using the purely geometric quadric metric. Once a position $\bar{\mathbf{v}}$ is chosen for a particular contraction, a new texture coordinate is synthesized using local reprojection. On the right is the result of applying the simplification algorithm using a generalized $xyzst$ quadric. As you can see, even with only 100 faces, both approximations appear virtually identical to the original. The meshes produced by each method are quite similar; they both distribute faces fairly evenly over the entire surface.

Local reprojection and extended quadrics produce nearly identical results in cases where the texture parameterization is an approximately affine function of position. However, if the mapping is more nonlinear, as in Figure 6.26b, extended quadrics perform much better. Figure 6.28 demonstrates the resulting approximations generated by these two methods. It is quite clear that local

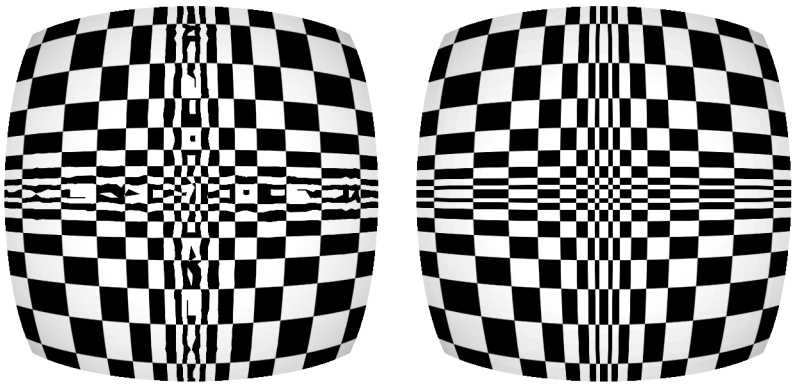


(a) 100 face approximation

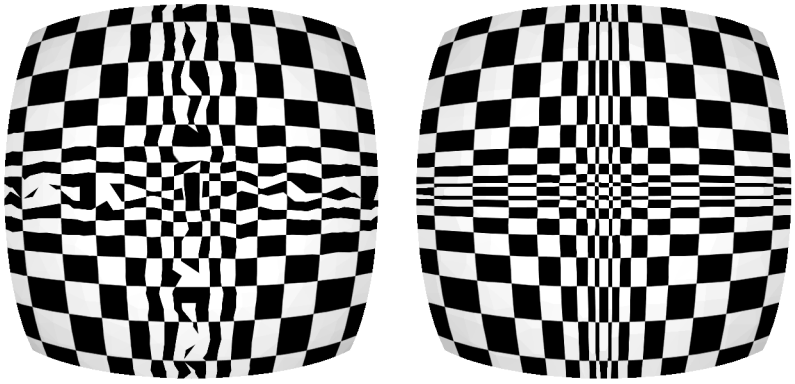


(b) 100 face mesh

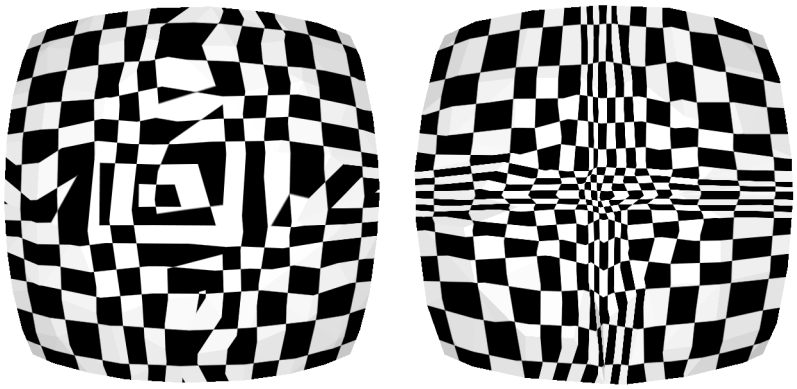
Figure 6.27: Approximations of an affine texture parameterization using local reprojection *(left)* and the extended quadric metric *(right)*.



(a) 4500 face approximation



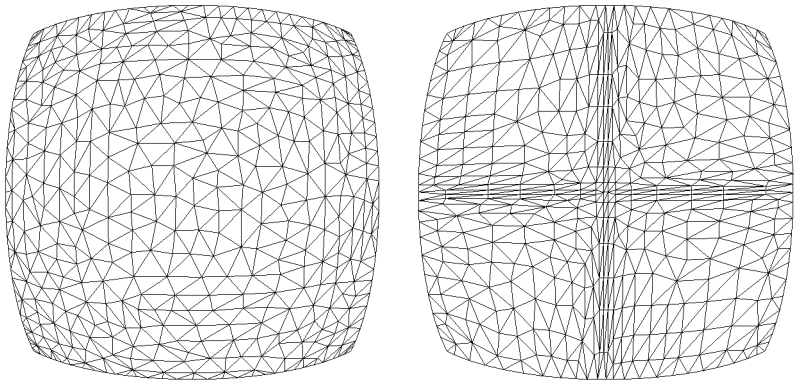
(b) 800 face approximation



(c) 100 face approximation

Figure 6.28: Approximations of a highly nonlinear texture parameterization using local reprojection *(left)* and the extended quadric metric *(right)*.

reprojection does a poor job of properly maintaining the texture parameterization. At the 800 face level, it results in a noticeably garbled texture whereas the approximation produced using extended quadrics remains quite faithful to the original. Only at around 100 faces does the texture parameterization begin to degrade during extended quadric-based simplification. This is largely because there are simply no longer enough triangles to accurately preserve both the open contour of the surface and the texture mapping.



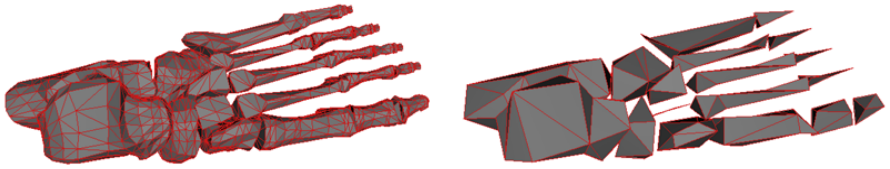
(a) Local reprojection (b) Quadric-based

Figure 6.29: Meshes of 800 face approximations from Figure 6.28. Extended quadrics lead to a mesh that conforms to the geometry as well as the texture parameterization.

Figure 6.29 provides some insight into why the extended quadric method produces substantially better results when the texture mapping is warped. Consider for a moment the texture coordinate s as a function of x given in (6.2). This function has very high curvature near $x = 0.5$, but its curvature becomes much lower (i.e., it becomes fairly flat) as x approaches 0 and 1. If we were to generate a piecewise-linear approximation of this function, it would be clear that a good approximation will have more segments in the area where the curvature is large and fewer segments in the areas where the curvature is small. The same reasoning applies in this case. The curvature of the texture parameterization is much higher near the middle and along the central lines of the surface than it is along the boundary. Therefore, a good approximation of this parameterization should concentrate more triangles in these regions. This is exactly what the extended quadrics lead the algorithm to do; in contrast, local reprojection continues to generate a fairly uniform grid. Looking more closely, we can see that the triangles concentrated along the central lines also stretch along the direction in which the parameterization is changing the least.

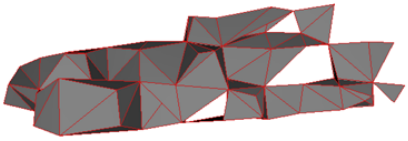
6.4 Contraction of Non-Edge Pairs

One of the novel features introduced by my algorithm [68] is the generalization of iterative contraction to consider arbitrary pairs of vertices instead of merely edges (§3.2.1). By contracting non-edge vertex pairs together, the algorithm can aggregate separate components together. When the model is composed of many disjoint components, this can result in significantly better approximations. All previous results shown in this section were generated using edges alone ($\tau = 0$). Now, let us consider the effect of using non-edge pairs as well.

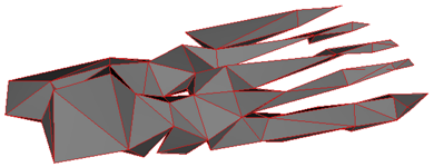


(a) Original 4204 face model

(b) 250 faces ($\tau = 0$)



(c) 262 faces (clustering)



(d) 250 faces ($\tau = 0.318$)

Figure 6.30: Approximating a model with many separate components.

Figure 6.30 demonstrates the benefits which aggregation via pair contractions can provide. The original model consists of the bones of a human's left foot, and is composed of 4204 triangles. There are many separate bone segments, some of which even interpenetrate (obviously an error in model reconstruction). However, when viewed from a distance, the separation between many these bone segments is imperceptible. By selectively joining bones together, we should be able to produce a model which uses fewer triangles for a given error tolerance. When edge contractions alone are used (b), the small segments at the ends of the toes collapse into single points; this creates the impression that the toes are slowly receding back into the foot. Simplifying this model by uniform clustering on a $11 \times 4 \times 4$ grid (c) produces a model where separate components have been fused together. However, the bones have been joined together rather indiscriminately, and the quality of the resulting approximation is poor. In contrast, performing iterative contraction considering both edges and additional non-edge

pairs (d) produces a model where separate bone segments have been merged in a much more desirable manner. The toes are being merged into single long segments. As a side note, this model now contains 61 non-manifold edges. If the underlying simplification algorithm did not support non-manifold models, producing this result would be appreciably more difficult.

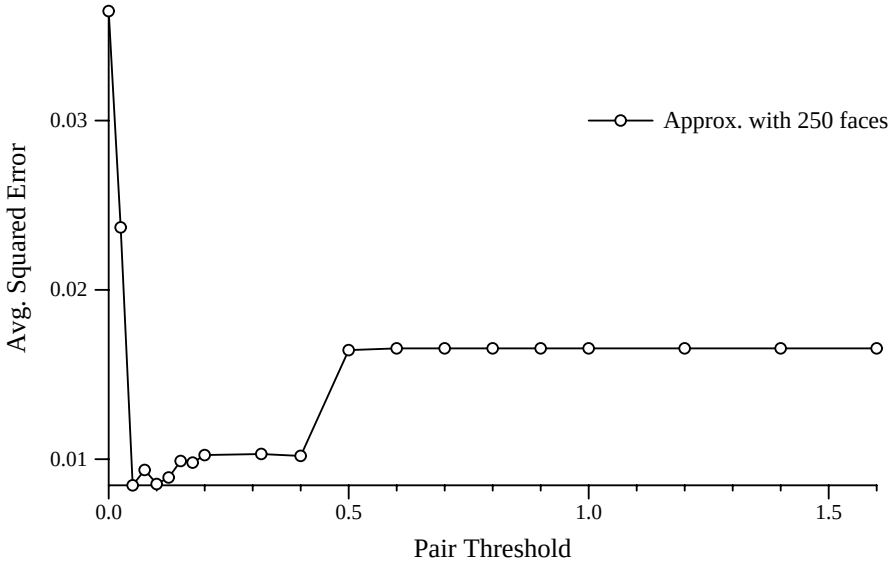
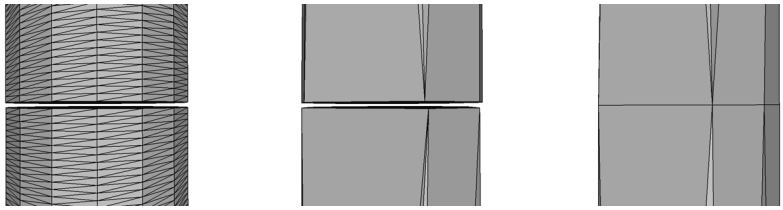


Figure 6.31: Affect of pair thresholds on approximation error E_{avg} for 250 face approximations of the foot model.

The benefits of aggregation are not solely visual. Aggregation can also produce objectively lower approximation error. Figure 6.31 shows the approximation error E_{avg} of 250 face approximations of the foot model as a function of the selection threshold τ. A fairly wide range of values all produce objectively better approximations than are achieved by using edges along ($\tau = 0$). However, this also exhibits one of the weaknesses of the selection method. It is fairly sensitive to the choice of the threshold τ. Increasing the threshold does not always improve the quality of the approximation. For some models, there may be no value for $\tau > 0$ for which the error is less than for $\tau = 0$. This is in part due to the nature of the quadric error metric. Locally, the distance to a set of planes is a reasonable characterization of the distance to a set of faces. However, because planes have infinite extent and faces do not, the error metric becomes less reliable as we move farther away from the neighborhood in question.

Aside from this sensitivity to the selection threshold, aggregation by pair contraction also has one another clear weakness. Its success depends on the relative positions of vertices on separate components. Consequently, contraction using non-edge pairs does not perform as reliably as contraction using edges only, which performs well on a wide range of models.



(a) Original cylinder (b) Edges only (c) Plus non-edges

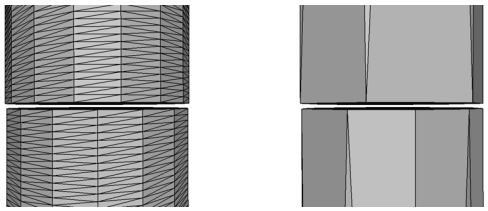
Figure 6.32: Two vertical capped cylinders in close proximity (viewed from the side). Simplifying with edges only (b) will never join these cylinders. By considering non-edges as well (c), we can merge them into one cylinder.

Consider the model shown in Figure 6.32a. It consists of two closed cylinders whose ends are very close together. Suppose we had a model consisting of a chain of several such cylinders. For example, this might represent a set of stacked cans. At a distance, the gaps between them would be nearly invisible, and we could approximate them all by a single cylinder.

Simplifying this model using edge contraction produces results like that shown in Figure 6.32b. The separate cylinders will never be merged. By considering additional pairs of vertices, in this case the pairs of vertices that span the gap, we can achieve approximations like Figure 6.32c. The two cylinders have been fused into one. At very low levels of detail, this will allow us to more faithfully preserve the shape of the original. However, there are two difficulties with the results of aggregating the cylinders in this way.

First, the caps of the cylinders have not been removed in model (c). Corresponding vertices along the rim of the cylinders have been joined; the faces of the caps are now simply hidden within the merged cylinder. Ideally, we would like to remove these faces, but even if they were removed from the model they would continue to affect the error metric. The planes of the caps have been accumulated by the quadrics of the vertices on the rim. These planes, which are perpendicular to the surface of the cylinder, will prevent vertices from moving away from the rims. In other words, the seam where the cylinders are merged will never disappear from the model.

The second problem with this method of aggregation is that it relies on being able to match up pairs of vertices to span gaps. Figure 6.33 illustrates this problem. Model (a) is identical to the model in Figure 6.32a, except that the upper cylinder has been rotated. The vertices along the rims of the cylinders are no longer aligned. Applying the same process which produced Figure 6.32c now produces Figure 6.33b. The cylinders are no longer merged. In this very simple case, expanding the threshold τ can once again result in the cylinders being merged. But in cases where there is no clear correspondence between vertices, components will not be reliably aggregated.



(a) Twisted original (b) Approximation

Figure 6.33: The two cylinders shown in Figure 6.32; one has been rotated slightly. Using the same threshold τ no longer results in merging of the cylinders.

Chapter 7

Applications

Surface simplification algorithms, such as the one I have described in the preceding chapters, are designed to produce approximations from initial surface models. They may be used to produce a single final approximation, or a series of approximations for use as a discrete multiresolution model. Iterative contraction algorithms, such as my own, induce a certain hierarchical structure on the surface determined by the sequence of contractions itself. This structure leads quite naturally to continuous multiresolution representations. In this chapter, I will examine the structure induced by iterative contraction algorithms in general. In addition, I will also explore some of the potential applications which are made possible by the efficiency of my algorithm in particular.

7.1 Incremental Representations

Starting with the original model M^0, iterative simplification algorithms generate a sequence of approximations

$$M^0 \rightarrow M^1 \rightarrow M^2 \rightarrow \cdots \rightarrow M^k$$

arriving at the final approximation M^k. Note that this differs from the progressive mesh notation used by Hoppe [90] where M^0 is the base mesh which, by a sequence of refinements, is transformed into the original mesh M^k. An *incremental representation* is one which encodes the original model M^0, the final model M^k, and all the intermediate approximations $M^1, \ldots, M^{k-1}$. This is a multiresolution representation because it allows us to extract a fairly wide range of levels of detail. However, the available approximations are restricted to exactly those which were generated during simplification.

7.1.1 Simplification Streams

The direct by-product of iterative contraction is an incremental representation which I will term a *simplification stream*. During the process of simplification,

we generate the sequence of models

$$M^0 \stackrel{\phi^1}{\rightarrow} M^1 \stackrel{\phi^2}{\rightarrow} M^2 \stackrel{\phi^3}{\rightarrow} \cdots \stackrel{\phi^k}{\rightarrow} M^k$$

where each step $M^{i-1} \rightarrow M^i$ corresponds to the application of a single contraction ϕ^i. Thus each intermediate approximation M^i can be expressed as the result of applying some prefix of the total sequence of contractions

$$M^i = [\phi^i](M^0) = (\phi^i \circ \cdots \circ \phi^1)(M^0) \tag{7.1}$$

Suppose that along with the original mesh M^0 we store a representation of each contraction ϕ^i. By simply applying (7.1), we can reconstruct any intermediate model M^i in addition to the original and final models. In essence, by storing a record of the simplification process, we can re-apply the same contraction sequence but choose an earlier stopping point.

At a minimum, the pieces of information that must be recorded for a contraction ϕ^i are

1. identifiers for the vertices $(\mathbf{v}_j, \mathbf{v}_k)$ being contracted, and
2. the final position $\bar{\mathbf{v}}$.

Note that the actual size of this data may vary. If we are using an optimal placement strategy (§3.5), $\bar{\mathbf{v}}$ might require as much as 3 floating point numbers to determine the new vertex position. And if the vertices have associated material attributes, further data will be required for each attribute. On the other hand, with a fixed placement strategy, $\bar{\mathbf{v}}$ can be encoded implicitly by the order of the vertices $(\mathbf{v}_j, \mathbf{v}_k)$.

Simplification streams do provide an incremental representation of the surface. By applying some prefix of the contraction sequence, we can reconstruct any of the intermediate approximations generated during simplification. However, their practical utility is limited. Lau *et al.* [114] used simplification streams of bounded size as a cache of recently generated approximations for run-time simplification. Recording the last k contractions allows their system to refine the current approximation by at most k steps. If further refinement is required, simplification begins again from the original model. For certain limited applications, this might yield acceptable results, but simplification streams are unsuitable for representing a wide range of approximations. Since we store the entire original model M^0 plus the contraction sequence, the resulting representation is necessarily larger than the original model. If we assume that our original model is very large and our desired approximation is quite small, certainly a common case, we are faced with a more significant problem. In order to reconstruct a small approximation, we must apply a large number of contractions to a large model. Thus, the smaller the approximation the greater the time required to extract it. We would clearly prefer reconstruction cost to be proportional to the desired approximation size. Fortunately, a closely related representation can solve both of these problems.

7.1.2 Progressive Meshes

The progressive mesh (PM) structure, originally introduced by Hoppe [90, 92], provides the same functionality as a simplification stream. However, it has two important advantages. First, the resulting representation can actually be smaller than the original model. Second, reconstruction time is proportional to the desired approximation size.

A progressive mesh is, in essence, a reversed simplification stream. It exploits the fact that the contraction operator is invertible. For each contraction ϕ^i we can define a corresponding inverse ψ^i. This operation, called a *vertex split*, is an inverse of ϕ^i such that $\psi^i(M^i) = M^{i-1}$. Thus, we begin with the final approximation M^k and produce a sequence of models

$$M^k \xrightarrow{\psi^k} \cdots \xrightarrow{\psi^3} M^2 \xrightarrow{\psi^2} M^1 \xrightarrow{\psi^1} M^0$$

terminating at the original model. The model M^k is called the *base mesh*. Along with this base mesh, we can store the vertex split sequence $\psi^k, \psi^{k-1}, \ldots, \psi^1$. Each item in the sequence must encode the vertex being split, positions for the two resulting vertices, and which triangles to introduce into the mesh [90]. We can reconstruct some intermediate approximation M^i by applying a prefix of the vertex split sequence:

$$M^i = [\psi^{i+1}](M^k) = (\psi^{i+1} \circ \cdots \circ \psi^k)(M^k) \tag{7.2}$$

Not only does this encode many different levels of detail of the original model, but Hoppe [90] demonstrated that progressive meshes are also an effective technique for compressing the input geometry.

Progressive meshes are generally developed as the result of iterative edge contraction. Naturally, they can also be used with alternative contraction primitives (§3.8). For instance, a progressive mesh based on face contraction has exactly the same structure as one based on edge contraction. However, because edge contraction is the finest-grained contraction primitive, it generates progressive meshes with the largest number of intermediate models. Simplification by pair contraction, and hence progressive meshes, can also be generalized to operate on arbitrary simplicial complexes [144].

7.1.3 Interruptible Simplification

A single simplification run produces a final approximation M^k and a corresponding progressive mesh as well. Using this progressive mesh, we can reconstruct any intermediate approximation between the base mesh M^k and the original model M^0. But we cannot use the progressive mesh to extract an approximation which is simpler than the base mesh M^k. Without any additional information, we would have to perform another simplification phase, beginning with M^k as the initial model, in order to produce a simpler approximation M^l. This can be expensive, particularly if we are using a costly algorithm such as Hoppe's [90]. Furthermore, by starting from the already simplified model M^k, we are quite

likely to produce a worse approximation than if we had begun with the initial model M^0.

My quadric error metric (§3.4) provides a convenient way around this problem. Recall that every vertex $\mathbf{v}_i$ has an associated quadric Q_i. The error at the vertex $\mathbf{v}_i$ is defined to be $Q_i(\mathbf{v}_i)$. With the exception of consistency checks performed on the current approximation, the sequence of contractions performed by my algorithm is completely determined by the quadrics associated with the vertices of the current approximation. These quadrics record all the information which the system has retained about the original model. Therefore, if we record the quadrics $Q_1, \ldots, Q_m$ of the vertices of M^k, we can resume simplification at a later time, and the resulting approximation will be exactly the same one which we would have obtained by beginning directly with M^0. In other words, provided that we record the quadrics for all the active vertices, we can halt the simplification process and resume it later without any loss of information. Using double precision values, storing the 10 coefficients of each quadric would require 80 bytes. Thus, storing all the quadrics for an m-vertex approximation would require $80m$ bytes.

In fact, this property allows quadric-based simplification to be combined with other algorithms. Suppose that we want to simplify a model in separate phases, alternating between quadric-based simplification and some other method. As long as the other algorithm properly maintains the quadrics at each vertex, my simplification algorithm can be stopped and resumed without any loss of information. Consider a very large, very densely over-sampled model. It may be that a very inexpensive algorithm, say one based on local heuristics such as dihedral edge angle, can produce good approximations of this model down to 50% of its original complexity. If this algorithm correctly tracks the accumulation of quadrics at the individual vertices, it can be run as an initial simplification phase, and the resulting model and quadrics can then simply be passed to a quadric-based procedure for further simplification.

7.2 Simplification and Spanning Trees

As outlined in the previous section, the sequence of contractions computed during simplification can be used to construct an incremental multiresolution representation of the model. But the process of iterative contraction actually induces further structure on the surface. In this section, we shall see that iterative edge contraction is a close analog of a minimum spanning tree algorithm.

Let us consider a model M as a graph. A *spanning tree* is an acyclic subgraph of M which connects all the vertices of M. A *spanning forest* is a collection of disjoint trees which collectively cover all the vertices of M. Suppose that every edge i has an associated weight, or cost, w_i. The weight of a graph is the sum of the weights of its edges, and a *minimum spanning tree* is a spanning tree whose total edge cost is minimal.

Figure 7.1 provides a simple illustration of the connection between iterative contraction and spanning trees. In the right-hand column is a sequence of

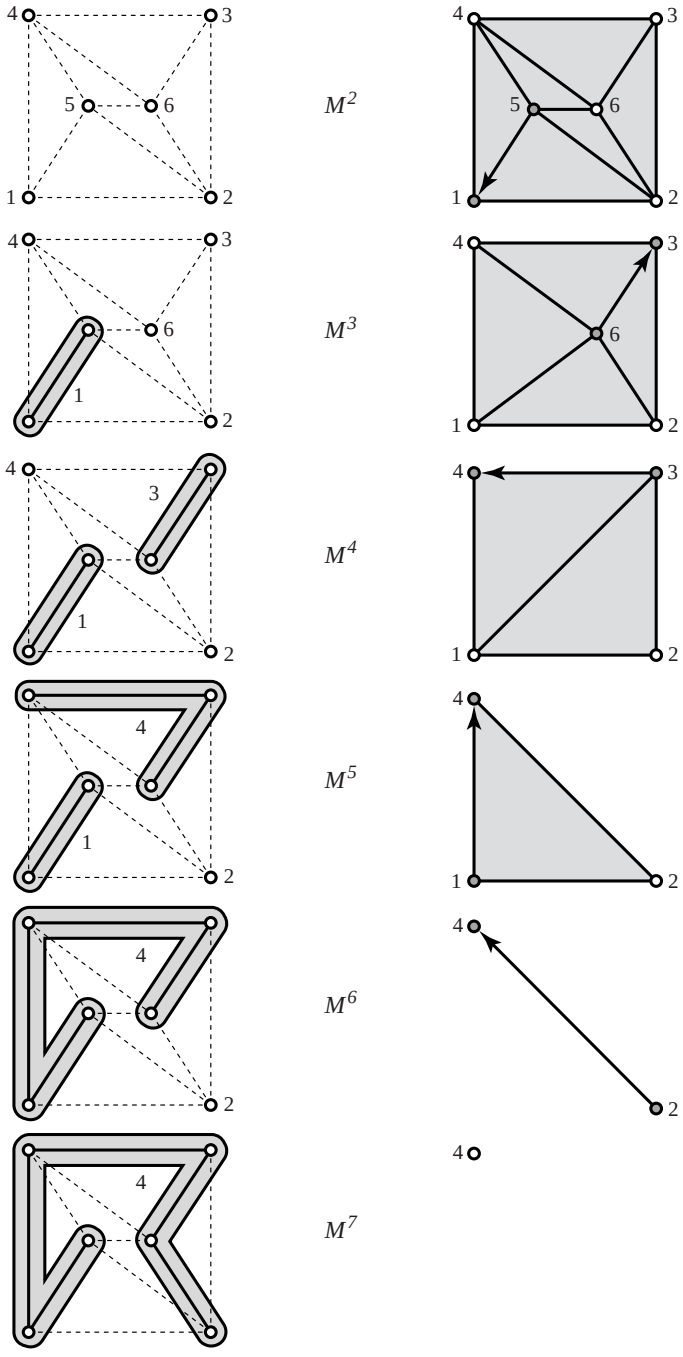


Figure 7.1: Simplification process and corresponding spanning tree.

approximations $M^2, \ldots, M^7$. In this example, we are using a fixed placement strategy; in other words, each contraction is of the form $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \mathbf{v}_i$. I will indicate this by writing the contraction in the more compact form $\mathbf{v}_j \rightarrow \mathbf{v}_i$. For each model, the edge being contracted to produce the next model is drawn as an arrow indicating the contraction in the same manner. Note that I have labeled the initial model M^2 because it corresponds to the final model shown in Figure 3.3.

Each vertex in a given approximation corresponds to some set of vertices in the original model. In particular, it corresponds to itself plus all the vertices which have been contracted into it. These sets are disjoint and they completely partition the original set of vertices. In the original model M^2, each vertex $\mathbf{v}_i$ corresponds to the singleton set $\{\mathbf{v}_i\}$. After the contraction $\mathbf{v}_5 \rightarrow \mathbf{v}_1$, the vertex $\mathbf{v}_1$ corresponds to the set $\{\mathbf{v}_1, \mathbf{v}_5\}$ in the original model. In the left-hand column of Figure 7.1, I have indicated the structure of these sets by enclosing them in shaded regions.

It is easy to show inductively that the construction of these sets is equivalent to the construction of a spanning forest. In the initial model, each set contains a single vertex. The set of all vertices is clearly a spanning forest. In general, a single edge contraction joins exactly two sets together. Assuming that each set is already a spanning tree, connecting these trees by a single edge (the edge being contracted) results in a larger spanning tree. Notice, however, that there is not one unique spanning tree corresponding to the simplification process. When two regions are merged, any duplicate edges connecting them to the same region are removed. Thus, an edge connecting two vertices in the approximation corresponds to multiple edges in the original graph. For example, when producing the tree for M^6 in Figure 7.1, there are three different edges which connect the corresponding parts of the spanning forest.

This view of iterative contraction is very similar to the minimum spanning tree algorithm of Cheriton and Tarjan [20, 185]. The primary difference is that in their algorithm each edge is assigned a constant weight, but in the simplification algorithm the weights assigned to the edges change over time. In my algorithm, the weight of an edge is a function of the quadrics associated with its endpoints.

We can construct a directed graph by creating an edge $\mathbf{v}_j \rightarrow \mathbf{v}_i$ whenever we perform the corresponding contraction. This records the same sort of information as found in Table 3.1. Figure 7.2 illustrates the resulting graphs for models M^5 and M^7. By starting at a node and following the arcs, we arrive at the currently active vertex which has accumulated the node at which we began. This structure is a common representation for disjoint sets [34, 185]; it is often referred to as a disjoint-set (or union–find) forest. In fact, these structures are exactly those used by Cheriton and Tarjan [20] to track disjoint sets of vertices in their minimum spanning tree algorithm. When using these graphs to simply track disjoint sets, it is common practice to apply a path compression heuristic — whenever we traverse a path, we update all the nodes encountered to point directly to the root of their tree. Consequently, all nodes will ultimately point directly to the root. Path compression leads to more efficient set membership queries; however, the additional structure of the uncompressed graph has some

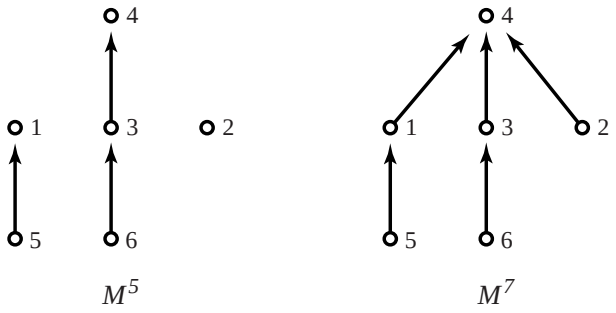


Figure 7.2: Disjoint-set graph for two models shown in Figure 7.1.

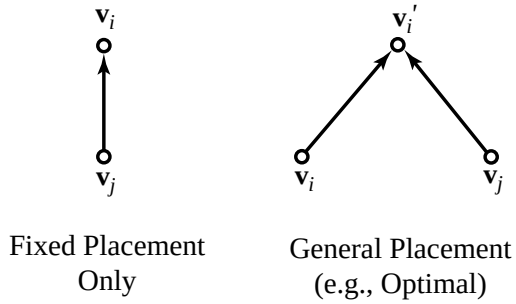


Figure 7.3: Graph structure of a single contraction.

very useful multiresolution applications.

Edge contraction has also been used explicitly for constructing minimum spanning trees by Karger *et al.* [104]. This technique is similar to the algorithm of Xia and Varshney [198], which simplifies the surface by iteratively contracting maximal independent sets of edges.

7.3 Vertex Hierarchies

Progressive meshes provide a useful multiresolution structure, but they are somewhat restrictive. They only allow us to reconstruct models which were generated during the original simplification process. This is because we always perform contractions in the order in which they were discovered, but this total ordering of contractions is not necessary. By using a less restrictive partial ordering, we can achieve a much more flexible multiresolution representation which will allow us to generate novel approximations that were never constructed during simplification.

Let us return to the simplification sequence pictured in Figure 7.1. Consider

the contraction $\mathbf{v}_5 \rightarrow \mathbf{v}_1$ (which produces M^3) and the contraction $\mathbf{v}_6 \rightarrow \mathbf{v}_3$ (which produces M^4). With a progressive mesh, we would always perform these two contractions in this order, because that is the order in which they were initially performed during simplification. However, by inspection, we can see that they are independent. We can just as easily perform them in the opposite order, and the resulting meshes would be just as valid as the ones shown in the figure.

These two contractions are interchangeable because the sets of vertices involved in the contraction are disjoint. We can see this fact reflected in the structure of the graphs shown in Figure 7.2. There is no path which includes both contractions $\mathbf{v}_5 \rightarrow \mathbf{v}_1$ and $\mathbf{v}_6 \rightarrow \mathbf{v}_3$. If we interpret these disjoint-set forests as dependency graphs, we can determine whether any given pair of contractions can be performed independently or not.

The construction pictured in Figure 7.2 is only applicable when we are using a fixed placement scheme. In general, we would like to perform contractions $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \mathbf{v}'_i$ where $\mathbf{v}'_i \neq \mathbf{v}_i$. In order to accommodate such general placement, we can treat $\mathbf{v}'_i$ as a new vertex instance and link both $\mathbf{v}_i$ and $\mathbf{v}_j$ to it. This structure is illustrated in Figure 7.3.

The result of applying this rule over the entire simplification process will be a binary forest. Assuming that we simplify a model completely to a single vertex, we will have a binary tree. The resulting graph is a *vertex hierarchy*; Figure 7.4 illustrates the hierarchy resulting from the simplification in Figures 3.3 and 7.1. This vertex hierarchy structure was developed independently by

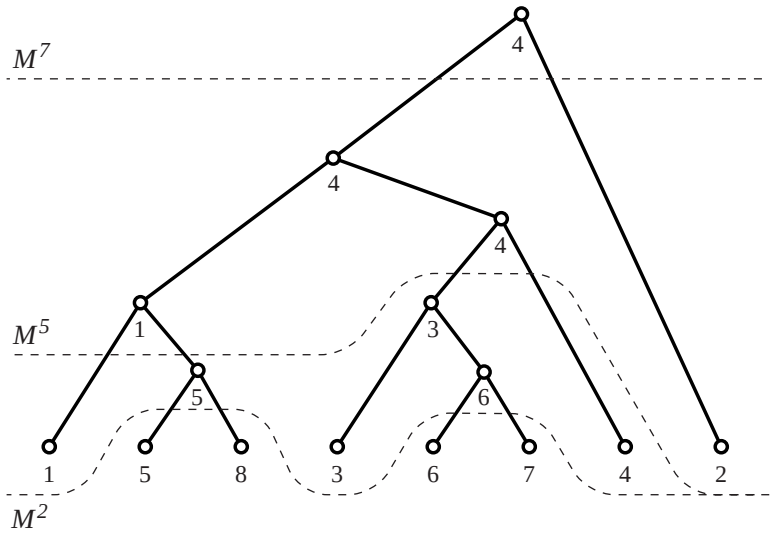


Figure 7.4: The vertex hierarchy resulting from the simplification process shown in Figures 3.3 and 7.1.

myself, Xia and Varshney [198], Hoppe [91, 93], and Luebke and Erikson [130].

In a vertex hierarchy, cuts through the tree correspond to allowable approximations. Figure 7.4 shows the three different cuts corresponding to models M^2, M^5, and M^7. Any given cut divides the tree into some number of components. The component containing the root remains a tree, essentially a pruned version of the original. I will call the leaves of this pruned tree *active vertices*. Each active vertex is the root of some subtree in the hierarchy. For every active vertex, we can perform all the contractions described by that subtree, and we can perform contractions in separate subtrees in any order. The result is a valid approximation of the original model. However, it is not guaranteed to be free of artifacts such as mesh fold-over (§2.3.4). To prevent such degeneracies, we must either perform run-time consistency checks or encode further dependencies in the hierarchy itself.

7.3.1 Adaptive Refinement

The primary application for which vertex hierarchies have been used is view-dependent refinement of models for real-time rendering [91, 93, 130, 198]. Suppose that we have a large surface, such as a terrain, that we would like to render using a continuously varying level of detail. Furthermore, let us assume that there is generally a high degree of frame-to-frame coherence in the viewpoint, such as we would expect in a flight simulator. Now, imagine that our model is represented using a vertex hierarchy, and that we have selected a cut through the tree. Any change in this cut, either up or down the tree, will result in a new, closely related approximation. By maintaining an active cut through the tree, and incrementally adjusting it for each frame, we can adapt the current model to the new viewing conditions.

Vertex hierarchies can also accommodate further synthetic refinement of the model past the original level of detail. Any mesh refinement which can be implemented by edge splitting can be added into the hierarchy. An edge

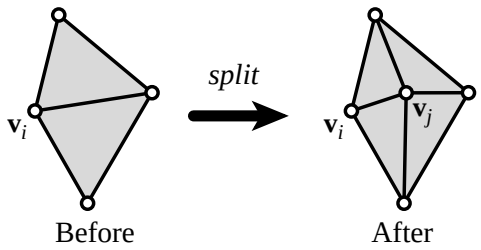


Figure 7.5: Refining a mesh with a single edge split.

split, such as the one shown in Figure 7.5, can be directly encoded as a vertex split operation on the vertex $\mathbf{v}_i$. The leaves of the vertex hierarchy represent the original vertices, but we can add temporary levels below the leaves corresponding

to further refinement of the initial mesh. Consider the example of modeling a large terrain surface. By extracting some information from the input data [189], we could synthesize additional detail, consistent with the form of the surface, using fractal-based edge subdivision.

Naturally, vertex hierarchies are not required to be binary trees. If we allow generalized contraction operations, each node may have an arbitrary number of children [130]. A very deep hierarchy has the disadvantage that many adaptation steps are required to smoothly transition from the bottom of the hierarchy to the top, or vice versa. In some cases, it may be desirable to reduce this cost by compressing entire subtrees so that all the leaves point directly to the root of the subtree and all internal nodes are eliminated. An extreme example of this strategy would be to pick a small number of cuts through the tree, and remove all internal nodes separating them so that each level was linked directly to the next. This sort of *stratified* vertex hierarchy corresponds directly to the discrete multiresolution models discussed in Section 2.2.1.

7.3.2 Structural Invariance of Quadrics

Consider a node $\mathbf{v}$ in a vertex hierarchy. The quadric Q associated with this node is the sum of the quadrics of its child nodes. They, in turn, are defined by the sum over their children. In fact, the quadric Q is precisely the sum of the quadrics of the leaves of the subtree rooted at $\mathbf{v}$. Thus, the quadric Q is independent of the structure of the subtree; it depends only on the leaves.

This structural invariance of quadrics means that we can restructure subtrees in the vertex hierarchy, and the error and optimal position associated with the root do not change. For example, after computing a vertex hierarchy, we might want to rebalance parts of the tree. Because of structural invariance, we know that the optimal positions of nodes outside the subtree are unaffected by the rebalancing. Of course, we still need to use consistency checks to prevent degeneracies such as mesh fold-over.

7.4 Online Simplification

The structure induced by simplification, which I have examined in the preceding sections, is common to all iterative contraction algorithms. I have also highlighted some of the specific properties of my own algorithm, such as support for interruptible simplification and the structural invariance of quadrics. Efficient simplification algorithms, such as mine, also make online simplification feasible. I use the term "online" because, while simplification takes place at run time, it does not necessarily take place in what could be considered "real time."

7.4.1 On-Demand PM Construction

One of the primary applications of progressive meshes is the progressive transmission of models over the network. A common application of this sort might

involve a server which provides network access to a repository of models. For instance, an online store might have models of its merchandise, or a CAD server might provide access to all the parts of a large aircraft.

Using a fairly expensive algorithm, such as Hoppe's [90], we would be more or less required to maintain a repository of precomputed progressive meshes. The server could then provide access to all these models. However, if we use a fairly cheap algorithm, it becomes feasible to produce progressive meshes when the client requests them. In addition, we could produce PMs for novel geometry without having to wait for a lengthy construction phase. Coupling an on-demand PM construction system with a cache of frequently used PMs, could provide an effective and flexible geometry server system.

As a simple experiment, I have set up a Web application which can generate simplified models on demand. Requests are submitted to the server, and VRML models are returned to the client. For models in the range of 20,000 faces or less, the cost of this is negligible. The overhead of simplification is almost completely hidden by transmission times and the startup cost of the client VRML viewer. For models of 150,000 faces or less, there is discernible latency associated with simplification, but it is certainly tolerable. Very large models, containing many hundreds of thousands of faces, currently require too much processing time for any kind of real-time performance.

7.4.2 Interactive Simplification

Fast simplification systems can also provide support for interactive simplification. I have implemented a simplification program that presents the user with a view of the current model and an interface mechanism to move forward and backward through the simplification process. When a simpler model is requested, the model is simplified, and when a more complex model is requested, the corresponding simplification is undone. The experience is similar to having a progressive mesh where the user can select what prefix of the split records to apply. However, in this case there is no fixed record of contractions; simplification is always computed on-demand at run time. In addition to maintaining the current approximation, the system also maintains the corresponding vertex hierarchy.

This kind of control over simplification makes it possible for the user to interact with the simplification process. For example, the user might begin by simplifying a model to 10% of its original size. At this point, they might provide certain feedback to the system to bias the simplification process. They might selectively weight certain vertices more than others, or introduce additional quadric constraints similar to the automatically computed boundary constraints. Having provided this feedback, they might continue simplifying to 5% of the original size.

Chapter 8

Hierarchical Face Clustering

The process of simplification via iterative edge contraction induces a particular structure on the original surface: a hierarchy of vertex neighborhoods. These vertex hierarchies, described in Chapter 7, provide a useful multiresolution representation for reconstructing surface approximations. In this chapter, I will consider the case where hierarchies of surface regions, as opposed to geometric approximations, are the primary structure of interest. And in particular, I will present an algorithm, closely related to the simplification algorithm of Chapter 3, for constructing hierarchies of face clusters.

8.1 Hierarchies of Surface Regions

For applications such as real-time rendering, we are typically concerned with multiresolution models that can provide geometric approximations of the original surface. Depending on the viewing conditions, the run-time system will select the appropriate geometric level of detail to display. However, there are other applications in which we are more interested in the aggregate properties of surface regions. Rather than extracting a single approximation from the hierarchy, we would like to perform computations using the hierarchy itself. For collision detection or ray tracing, we would like a hierarchy of bounding volumes that enclose successively larger regions of the surface. Many kinds of simulation problems, such as radiosity, can be computed on a hierarchy of surface regions. In the specific case of radiosity, which is one of the primary motivating applications for the work described in this chapter, we might want to approximate successively larger regions of the surface with planar elements. This would allow us to use a kind of generalized hierarchical radiosity [81] computation on the surface.

Originally, I described *vertex hierarchies* (§7.3) as a means of encoding the dependencies in the sequence of contractions built during simplification. But we

can also think of them as hierarchies of vertex neighborhoods. At the leaves of the tree are the vertices of the original model. Their neighborhood corresponds to their adjacent faces. When we contract two vertices together, we merge their neighborhoods and construct a new approximate neighborhood for the resulting vertex. Each node in the hierarchy corresponds to a disjoint set of vertices on the original surface. Consequently, it also corresponds to a surface region that is the union of the neighborhoods of all the vertices in its set.

This hierarchy of neighborhoods is itself a useful construction. For instance, Kobbelt *et al.* [109] and Lee *et al.* [118] use the resulting hierarchy to define a multiresolution parameterization of the surface. This parameterization facilitates applications such as multiresolution surface editing.

While useful, these hierarchical neighborhoods have some drawbacks for certain applications. First, they do not form a well-defined partition of the surface. The sets of faces forming the neighborhoods of two adjacent vertices will overlap. Second, the neighborhood surrounding a single vertex is generally nonplanar; a vertex at the corner of a cube is a prime example. Finally, the set of triangles on the original surface corresponding to a particular vertex in the hierarchy may have a very irregular shape. However, for some applications it is preferable that the shape of this region be regular. Standard simplification algorithms provide no means to control this shape.

Face hierarchies are a natural alternative to vertex hierarchies. Instead of iteratively merging vertex neighborhoods, we can iteratively cluster neighboring groups of faces. This allows us to avoid some of the specific drawbacks of vertex hierarchies listed above. Because the clusters are disjoint sets of faces, they partition the surface in a well-defined way. Furthermore, each cluster corresponds to a well-defined surface area: the area of all its constituent faces. Face clusters are also more likely to have a single normal which is a good match for the surface normals of the faces in the cluster.

To highlight the difference between these kinds of hierarchies, consider the example of a cubical object. Near the root of a vertex hierarchy, there will be a level at which we have eight conical neighborhoods, one for each vertex. In contrast, near the top of a face hierarchy we will have 6 planar face clusters, one for each face of the cube. If we are trying to formulate a single planar approximating element, or a single normal for the entire region, we will get a much better result with the face hierarchies than with the vertex hierarchies.

8.2 Face Clustering Algorithm

To construct a face hierarchy, I begin by forming the *dual graph* of the surface. For the moment, I will assume that the surface is a manifold with boundary; in other words, every surface edge has at most 2 adjacent faces. I will also assume that all input polygons have been triangulated. Every face of the surface is mapped to a node in the dual graph, and two dual nodes are connected by an edge if the corresponding faces are adjacent on the surface. I will use the term *face cluster* to refer to a connected set of faces that have been grouped together.

In the dual graph, each node will correspond to a face cluster. Initially, each cluster consists of a single face of the input model. An edge contraction in this graph merges two dual nodes into one. This corresponds to grouping their associated faces, which must necessarily be adjacent, into a single cluster. Thus in general, dual edge contraction corresponds to merging two adjacent face clusters into a single cluster. Figure 8.1 illustrates a simple example. The underlying

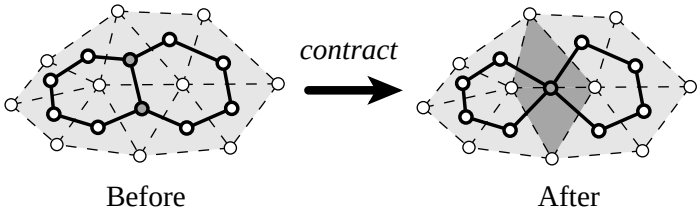


Figure 8.1: Contraction of a single dual edge. The two faces corresponding to the endpoints of the dual edge are merged to form a single face cluster.

mesh and the dual graph are shown in dashed and solid lines, respectively. On the left is a mesh where each dual node corresponds to a single face; in other words, each face is its own cluster. After contracting a single dual edge, the two darkened triangles have been merged into a single cluster.

To construct a complete hierarchy, I use a simple greedy procedure very similar to the simplification algorithm described in previous chapters. Each dual edge is assigned a "cost" of contraction, and the system iteratively contracts the dual edge of least cost. At each iteration i, we will have constructed a partition N^i of the surface into disjoint sets of connected faces. This is in contrast to simplification, where at each iteration we would have constructed an approximate surface. Figure 8.2 shows a simple example of this process. At each iteration, a single dual edge (shown on right) is selected for contraction. The selected edge is shown as a heavy dashed line. The effect of contracting each edge is to merge two face clusters (shown on left). Edges internal to a single cluster are shown as light dashed lines.

Let me emphasize that the geometry of the original surface is *not* altered in any way by this clustering process; every vertex remains in its original position and the connectivity of the mesh is unchanged. Instead, we begin with an initial surface partition N^0 where every face belongs to its own singleton cluster. The process of iterative dual contraction produces a sequence of partitions

$$N^0 \rightarrow N^1 \rightarrow N^2 \rightarrow \cdots \rightarrow N^k$$

with successively fewer clusters. If run to completion, this will produce a single face cluster for each connected component of the surface.

Just as with iterative edge contraction, iterative dual contraction produces a natural hierarchy. When two dual nodes are contracted together (i.e., two face clusters are merged), we can make them both children of their new parent

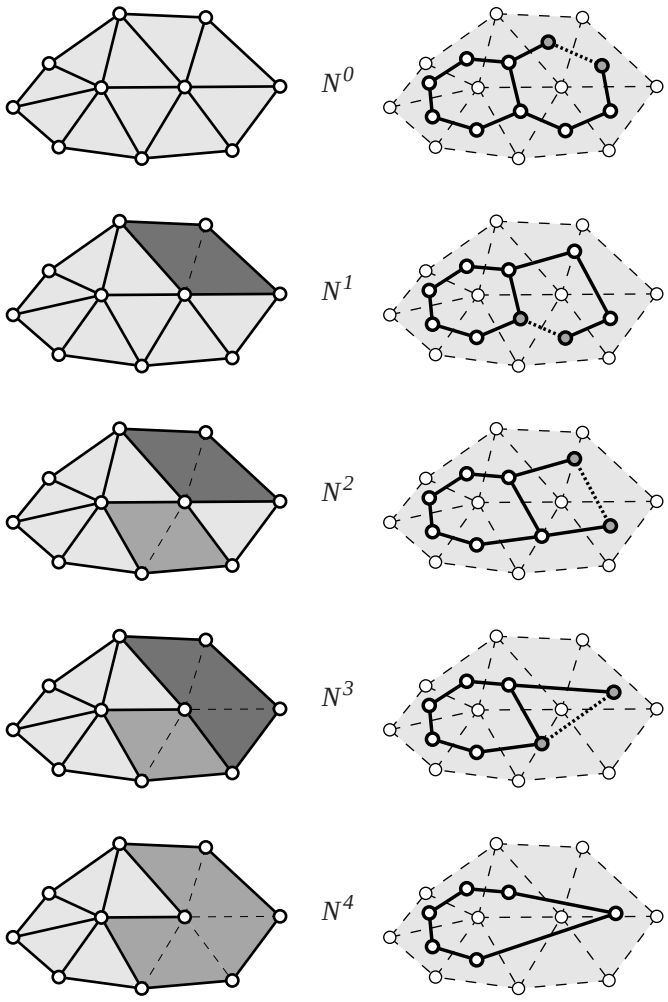


Figure 8.2: Faces of the input surface *(left)* are iteratively clustered together. This corresponds to iterative contraction of edges in the dual graph *(right)*.

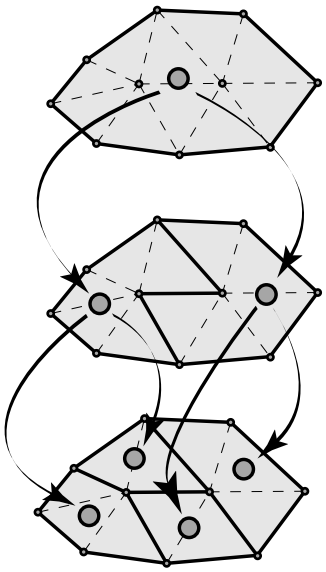


Figure 8.3: A three level face hierarchy. Four leaf clusters below are merged into one root cluster at the top.

node which represents the union of their associated clusters. See Figure 8.3 for a simple example. At the bottom of the hierarchy are four face clusters, each of which already contains two faces. The two leftmost clusters are merged together, forming a single parent node. The two rightmost clusters are also merged; notice that this corresponds to the same dual contraction $N^3 \rightarrow N^4$ shown in Figure 8.2. The two parent clusters, which together partition the mesh into two disjoint sets of faces, are merged together to produce a single root cluster which spans the entire mesh.

8.2.1 Related Methods

Iterative clustering, as a general class of algorithms, has been in use for decades [7]. While a substantial number of algorithms have been developed, most are only tangentially related to the problem of building face cluster hierarchies on surfaces. However, there are a few algorithms which are more clearly related to the problem at hand.

This process of face clustering which I have just described is closely related to the simplification algorithm of Kalvin and Taylor [102, 103]. They also partitioned the surface into a set of disjoint face clusters, or "superfaces." Their algorithm was based on growing a single cluster one face at a time until it exceeded a planarity threshold. In contrast, my algorithm is based on pairwise cluster merging. This has the advantage of producing a hierarchical structure

of clusters.

DeRose *et al.* [46] proposed a related algorithm for generating hierarchies of bounding boxes on subdivision surfaces. They use these hierarchies for collision detection during cloth dynamics simulation. Given an initial quadrilateral mesh, they iteratively merge a maximal independent set of adjoining clusters until only a single cluster remains. Since no criterion, other than adjacency, is used to select which clusters to merge, the individual clusters may or may not contain roughly coplanar elements.

Finally, Willersinn and Kropatsch [193] used the method of dual edge contraction to construct irregular image pyramids. While this method is designed for an entirely different domain — images rather than surfaces — it uses the same formalism of dual contraction as the algorithm which I have developed.

8.2.2 Dual Quadric Metric

In order to evaluate the cost of a dual contraction, we need some idea of what qualities a face cluster should have. Naturally, there are many potential criteria from which to choose. But for many applications, a good criterion is the planarity of the cluster. This means that a given face cluster can be approximated by a planar element without undue inaccuracy. Planarity is the criterion that I will adopt, and as we will see, this criterion can be expressed using a dual form of the quadric error metric.

Every cluster has an associated set of faces $\{f_1, \ldots, f_n\}$ and a set of points $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ determined by the vertices of these faces. Let us suppose that we want to find the least squares best fit plane to this set of points. For a plane specified by a unit normal $\mathbf{n}$ and a scalar offset d, the distance of a point $\mathbf{v}$ to this plane is $\mathbf{n}^\mathsf{T}\mathbf{v} + d$. The *fit error* of a given plane $\mathbf{n}^\mathsf{T}\mathbf{v} + d = 0$ is the average squared distance of all the points in the cluster to the plane

$$E_{fit} = \frac{1}{k} \sum_i (\mathbf{n}^\mathsf{T}\mathbf{v}_i + d)^2 \tag{8.1}$$

The least squares best plane is the one which minimizes this error. Notice that this error formula is nearly identical to the error metric which I have used for simplification. Aside from the averaging factor, the sole difference is that we are summing over a set of points rather than a set of normals. And, just as before, we can evaluate this error using a corresponding set of quadrics:

$$E_{fit} = \frac{1}{k} \sum_i P_i(\mathbf{n}, d) = \frac{1}{k} \left(\sum_i P_i \right) (\mathbf{n}, d) \tag{8.2}$$

where

$$P_i = (\mathbf{A}_i, \mathbf{b}_i, c_i) = (\mathbf{v}_i \mathbf{v}_i^\mathsf{T}, \mathbf{v}_i, 1) \tag{8.3}$$

and

$$P(\mathbf{n}, d) = \mathbf{n}^\mathsf{T}\mathbf{A}\mathbf{n} + 2\mathbf{b}^\mathsf{T}(d\mathbf{n}) + cd^2 \tag{8.4}$$

Every dual node will have an associated fit quadric P_i which requires ten coefficients to represent the symmetric 3×3 matrix $\mathbf{A}$, the 3-vector $\mathbf{b}$, and the scalar c. The cost of contracting two dual nodes together is reflected by the sum of the fit quadrics of each node $(P_i + P_j)(\mathbf{n}, d)$. Note that the form of this dual quadric error (8.4) differs slightly from the original due to the presence of the d and d^2 terms. This error metric is also closely related to the planarity metric of Kalvin and Taylor [102, 103] which measured the maximum distance of any point to the plane, as opposed to the average squared distance.

Finding the optimal plane which minimizes $P(\mathbf{n}, d)$ is not quite as simple as finding the optimum of the original quadric metric. The standard technique, based on principal component analysis (PCA) [101], is to construct the sample covariance matrix (§4.1.4):

$$\mathbf{Z} = \frac{1}{k-1} \sum_{i=1}^{k} (\mathbf{v}_i - \bar{\mathbf{v}})(\mathbf{v}_i - \bar{\mathbf{v}})^\mathsf{T} \tag{8.5}$$

where $\bar{\mathbf{v}}$ is the mean of the vertices $\bar{\mathbf{v}} = \left(\sum_i \mathbf{v}_i\right)/k$. Covariance matrices of normals were a useful tool in analyzing the quadric metric (§4.2.2) and this vertex covariance matrix plays a central role in the dual metric. The three eigenvectors of the matrix $\mathbf{Z}$ determine a local frame with $\bar{\mathbf{v}}$ as the origin. The eigenvector corresponding to the smallest eigenvalue is the normal of the least squares best plane through the set of points $\{\mathbf{v}_i\}$. This method, which is identical to the least squares method of normal equations, is frequently used to estimate or define local tangent planes [94, 122, 13] when reconstructing surfaces from sets of points. Note that the normal computed in this fashion is only unique up to sign. In practice, it is helpful to track the average normal of all the faces in the cluster to resolve this sign ambiguity. For the remainder of the discussion, I will drop the $1/(k-1)$ averaging factor from the covariance matrix formula. This has no effect on the algorithm because I am only interested in the eigenvectors of $\mathbf{Z}$ and the relative scales of its eigenvalues.

The covariance matrix $\mathbf{Z}$ can be expressed in terms of the quadric P. First, we can expand the equation for $\mathbf{Z}$ as:

$$\begin{aligned}
\mathbf{Z} &= \sum_{i=1}^{k} (\mathbf{v}_i - \bar{\mathbf{v}})(\mathbf{v}_i - \bar{\mathbf{v}})^\mathsf{T} && (8.6)\\
&= \sum \mathbf{v}_i \mathbf{v}_i^\mathsf{T} - \sum \mathbf{v}_i \bar{\mathbf{v}}^\mathsf{T} - \sum \bar{\mathbf{v}} \mathbf{v}_i^\mathsf{T} + \sum \bar{\mathbf{v}} \bar{\mathbf{v}}^\mathsf{T} && (8.7)\\
&= \sum \mathbf{v}_i \mathbf{v}_i^\mathsf{T} - \left(\sum \mathbf{v}_i\right) \bar{\mathbf{v}}^\mathsf{T} - \bar{\mathbf{v}} \left(\sum \mathbf{v}_i\right)^\mathsf{T} + k \bar{\mathbf{v}} \bar{\mathbf{v}}^\mathsf{T} && (8.8)\\
& && (8.9)
\end{aligned}$$

By substituting $\bar{\mathbf{v}} = \left(\sum_i \mathbf{v}_i\right)/k$ and collecting terms, we find that

$$\begin{aligned}
\mathbf{Z} &= \sum \mathbf{v}_i \mathbf{v}_i^\mathsf{T} - k(\bar{\mathbf{v}} \bar{\mathbf{v}}^\mathsf{T}) && (8.10)\\
&= \mathbf{A} - \frac{\mathbf{b}\mathbf{b}^\mathsf{T}}{c} && (8.11)
\end{aligned}$$

Thus, the normal of the optimal plane through the set of points can be computed directly from the corresponding fit quadric P. No extra storage is required to track the covariance matrix.

Minimizing the planarity term E_{fit} will naturally tend to merge clusters which are collectively nearly planar. However, a surface may locally fold back on itself. It will seem nearly planar, but the normal of the optimal plane will not be a good fit for all the surface normals in the region. To combat this problem, I will also use an additional error term which measures the average deviation of the plane normal $\mathbf{n}$ from the surface normals:

$$E_{dir} = \frac{1}{w} \sum_i w_i (1 - \mathbf{n}^\mathsf{T} \mathbf{n}_i)^2 \tag{8.12}$$

where w_i is the area of face f_i and $w = \sum_i w_i$ is the total area of the face cluster. We can write this metric as a quadric as well

$$E_{dir} = \frac{1}{w} \sum_i w_i R_i(\mathbf{n}) = \frac{1}{w} \left(\sum_i w_i R_i \right) (\mathbf{n}) \tag{8.13}$$

where

$$R_i = (\mathbf{D}, \mathbf{e}, f) = (\mathbf{n}_i \mathbf{n}_i^\mathsf{T}, -\mathbf{n}_i, 1) \tag{8.14}$$

and

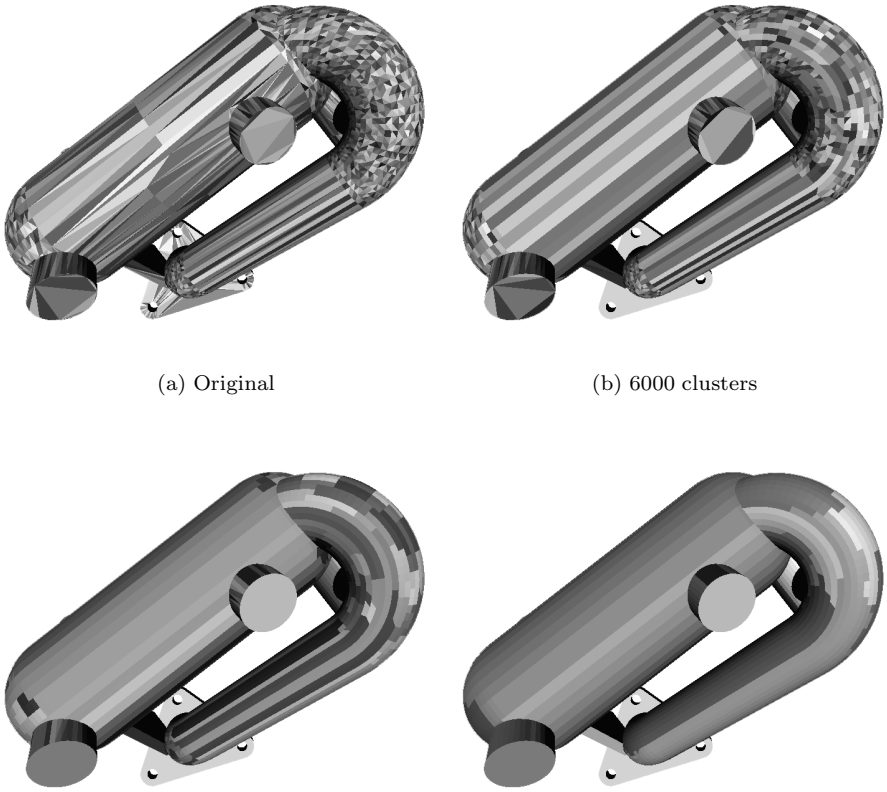
$$R(n) = \mathbf{n}^\mathsf{T} \mathbf{D} \mathbf{n} + 2\mathbf{e}^\mathsf{T} \mathbf{n} + f \tag{8.15}$$

Given these error metrics, the clustering algorithm has essentially the same form as the simplification algorithm. For every dual node, it computes quadrics P and R. For every dual edge, we sum the quadrics of the endpoints, find the optimal plane, and evaluate its error as $E_{fit} + E_{dir}$. Following this initialization step, we place all dual edges in a heap keyed on cost, and greedily contract the minimal cost edge.

Some examples of the results produced by this algorithm are shown in Figure 8.4. The original model (a) has 11,036 faces, each of which corresponds to an individual cluster. The partition of the surface shown in (b) contains only 6000 clusters. Note how the clusters are growing along the cylindrical parts of the surface and curving along the rounded parts. This effect is even more apparent in partition (c) containing 1000 clusters. Note how each planar region has been grouped into a single region. As clustering proceeds (d), the regions expand over more and more of the surface. Naturally, at some point an individual region may no longer be well-approximated by any plane, since it corresponds to a significant part of the model.

8.2.3 Compact Shape Bias

If planarity is our only clustering criterion, then the algorithm described above performs fairly well. However, there are applications where we are also interested



(a) Original

(b) 6000 clusters

(c) 1000 clusters

(d) 200 clusters

Figure 8.4: Sample clusters on the heat exchanger.

in the shape of these regions. In particular, we might want them to have a fairly compact shape; in other words, we might like each cluster to be as nearly circular as possible. It is fairly easy to add a simple compactness heuristic which significantly improves the regularity of the clusters.

Given a cluster with area w and perimeter ρ, I will define the *irregularity* γ of the cluster as a ratio of its squared perimeter ρ^2 to its area w

$$\gamma = \frac{\rho^2}{4\pi w} \tag{8.16}$$

This is the ratio of the squared perimeter ρ^2 to the squared perimeter of a circle with area w. A circle will have irregularity $\gamma = 1$ and larger values of γ correspond to more irregular (less compact) regions. This definition of irregularity is fairly natural and has been widely used in fields ranging from image processing [99] to detecting gerrymandering of Congressional districts [140]. Kalvin and Taylor used this definition of irregularity [102, 103] for directing the construction of superfaces for simplification, and Guéziec [77] proposed a similar measure of triangle compactness which requires fewer operations to compute.

Now suppose we have have two adjacent clusters with irregularity γ_1 and γ_2, respectively. Let γ be the irregularity of the cluster formed by merging them together. I define the shape penalty as

$$E_{shape} = 1 - \frac{\max(\gamma_1, \gamma_2)}{2\gamma} \tag{8.17}$$

If the irregularity of the cluster arising from a dual contraction is significantly worse than the two original clusters, that contraction will incur a penalty ($E_{shape} > 0$). On the other hand, if the irregularity improves substantially it will incur a negative penalty, or bonus. Based on my experience, requiring the irregularity to improve at each iteration over-constrains the clustering algorithm and leads to bad results. Using a penalty term such as E_{shape} seems to be more effective.

At first glance, it might appear that computing the shape penalty E_{shape} is rather expensive. We need to know the area of the clusters, but this is quite easy to track. The area w of the resulting cluster is merely the sum $w = w_1 + w_2$ of the constituent areas. However, we also need to have the perimeters of both original clusters and the final cluster, and perimeters are not additive. Fortunately, there is a fairly straightforward solution to this problem. Suppose we track the perimeter ρ_i for each cluster in the current partition; these values are stored with the relevant dual nodes. When two clusters are merged, the perimeter of the resulting cluster will be the sum of the perimeters of the constituent clusters minus twice the length of the boundary which separated them. Since we are only merging adjacent clusters, the boundary between them corresponds to a single dual edge. Thus, for each dual edge, we can store the length of the boundary separating the two clusters corresponding to its endpoints. By updating these lengths appropriately following a dual contraction, we can compute the perimeter of the merged cluster directly from the perimeters of the merging clusters and the length associated with the dual edge being contracted.

Figure 8.5 demonstrates the effect of using the additional shape penalty. Clustering without bias (a) can produce highly irregular regions. Note the clus-

(a) No shape bias (b) With bias

Figure 8.5: Two partitions of the cow model, each with 1000 clusters. The shape bias produces much more compactly shaped clusters.

ters that stretch all along the rear leg. In terms of finding clusters which are best fit by a plane, this behavior is desirable, but it does not produce compact regions. In contrast, when we use the shape bias term (b), the algorithm produces very regularly shaped regions.

I have now outlined three error terms which constitute the dual quadric error metric E_{DQ}

$$E_{DQ} = \alpha_1 E_{fit} + \alpha_2 E_{dir} + \alpha_3 E_{shape} \tag{8.18}$$

The results shown in this chapter were generated using a uniform weighting $\alpha_1 = \alpha_2 = 1$. The partitions shown in Figure 8.4 were generated without shape bias ($\alpha_3 = 0$) whereas those in Figure 8.5 used a value of $\alpha_3 = 1$.

8.3 Applications

The face hierarchies produced by my clustering algorithm are intended for use in applications which are more concerned with the aggregate properties of surface regions rather than their exact geometry. Any particular partition produced by iterative clustering could potentially be used to compute a simplified surface [103]. However, the hierarchy itself has several useful applications.

8.3.1 Hierarchical Bounding Volumes

One natural application of these face hierarchies is for constructing hierarchical bounding volumes. Recall that, for each cluster in the hierarchy, we compute a

best fit plane. Given this plane, it is a fairly simple task to compute an oriented bounding box which tightly encloses the cluster. A simple, but relatively inefficient, method is to enumerate all the points in the cluster, project into the local frame defined by the plane, and compute an axis-aligned bounding box in this local frame. Alternatively, we can avoid having to traverse large sets of points by tracking the convex hull [147] of the point set [74].

Hierarchies of oriented boxes such as this can be used to efficiently measure the distance from a point to a surface, an operation common in simplification systems. They are also an effective means of accelerating the kind of spatial queries common in applications such as collision detection and ray tracing [8]. They can be constructed over both curves [10] and surfaces [11, 74]. In contrast to spatial partitions such as octrees [165], hierarchies which are attached to the surface are guaranteed to have a size linear in the size of the input model [11].

The bounding box hierarchy constructed in this manner is very similar to the OBBTree structure introduced by Gottschalk *et al.* [74]. They also used PCA to compute best fit planes, and thus oriented bounding boxes. However, they produced the vertex sets by a top–down partition of the vertices of the original model. In contrast, my clustering algorithm computes a bottom–up hierarchical partition of the surface and can subsequently derive bounding boxes for each region. The BOXTREE algorithm of Barequet *et al.* [11] also builds a bottom–up hierarchy of boxes. They rank pairs of boxes for merging based on properties of the resulting boxes, such as volume.

8.3.2 Multiresolution Radiosity

The face cluster hierarchy arising from iterative clustering can also be used to accelerate applications such as radiosity. Willmott [196] has developed a very effective multiresolution radiosity algorithm based on face hierarchies. In this section, I will briefly summarize this method and the performance gains achieved through the use of face hierarchies.

Early radiosity methods had time complexities of $O(n^2)$ or $O(n^3)$ for scenes containing n polygonal elements [32]. Because of the rather high expense of this kind of solution process, hierarchical radiosity methods [81] were developed. The initial input polygons can be adaptively subdivided, producing a hierarchy of surface elements. These methods allow light transport to be simulated at an appropriate level of detail. Nearby surface patches can interact at a fairly fine grain while distant patches can be represented at a coarser level. This algorithm is not as effective on very complex scenes because it can only subdivide the input surfaces. To handle complex surfaces, we need to build higher-level clusters that contain entire surface regions. Consider the scene shown in Figure 8.6 which contains several geometrically complex models. The serpentine dragon on the table, for example, contains 870,000 triangles. But given its relatively small size, an accurate radiosity solution could be computed on a much coarser set of surface patches than these input triangles.

One proposed method for building hierarchies above the input polygon level is volume clustering [176]. Faces of the model are grouped into spatial cells,

Figure 8.6: A museum scene containing 2.7 million input polygons. This radiosity solution was computed on a face cluster hierarchy.

typically axis-aligned boxes, and these cells are grouped into higher and higher levels. This approach appears to work well for clustering *objects*, such as the leaves of a tree. However, it does not perform as well for clustering *surface regions*. A particular region on a smooth surface will tend to have a dominant orientation, which is not captured by enclosing the region in an axis-aligned box. An object-aligned box will produce a much tighter fit to the surface, and it will have an orientation that reflects the orientation of the surface. The iterative face clustering algorithm described in this chapter provides a convenient way to construct a hierarchy of exactly this sort of tight-fitting box.

To demonstrate the advantages of this method, Figures 8.7 and 8.8 illustrate the comparative running times and memory consumption of three different radiosity algorithms. The performance of the algorithms is shown as a function of input scene complexity, which was controlled by using surface approximations generated using quadric-based simplification. All performance tests were run on a 195 MHz R10000 SGI machine with 1 GB of main memory. The progressive radiosity algorithm, which uses all of the input polygons during the solution process, requires rapidly increasing amounts of time and memory as the input complexity increases. Hierarchical radiosity with volume clustering requires substantially less resources, but its requirements also grow fairly rapidly with complexity. This is largely due to the fact that volume clusters do not provide an accurate fit for surface regions. Consequently, the solution computed at high

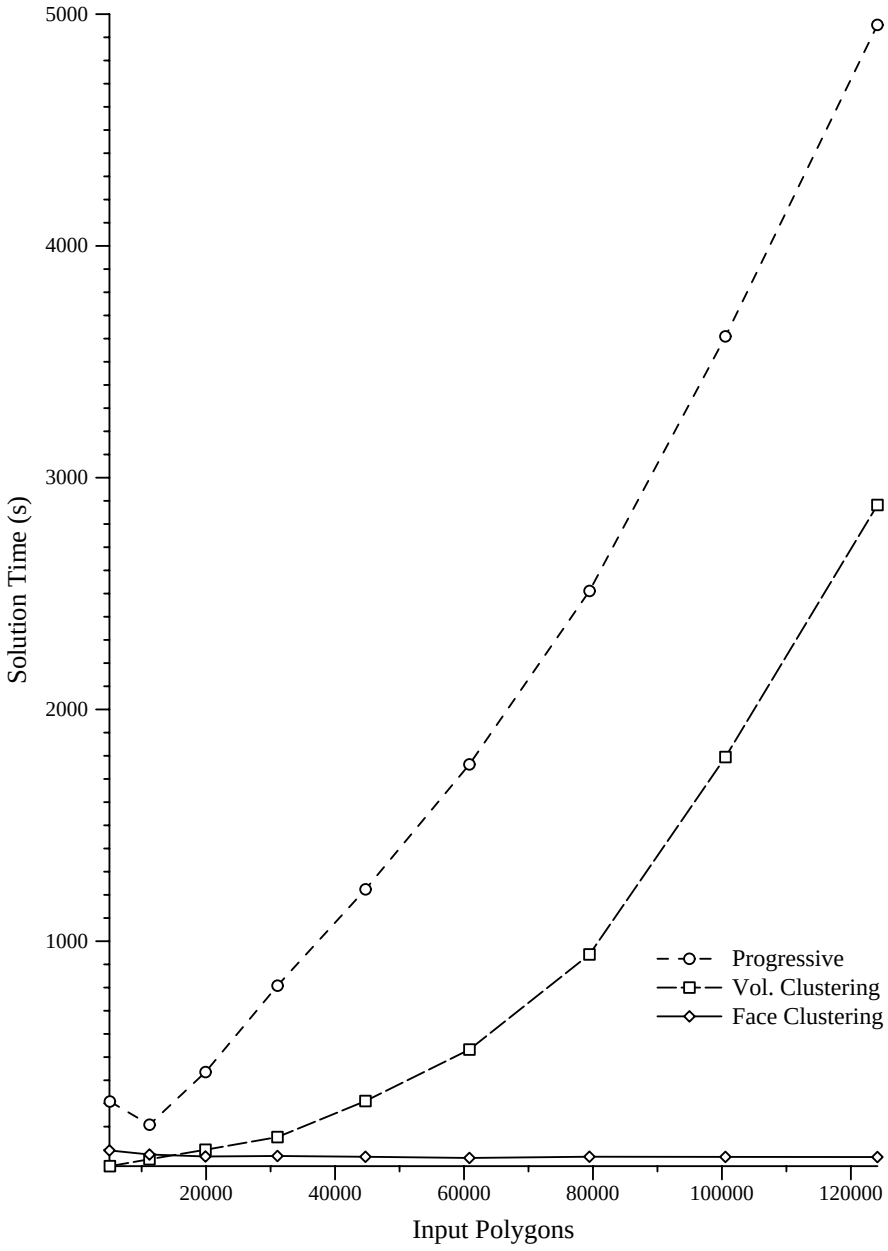


Figure 8.7: Running time of radiosity algorithms on the museum scene.

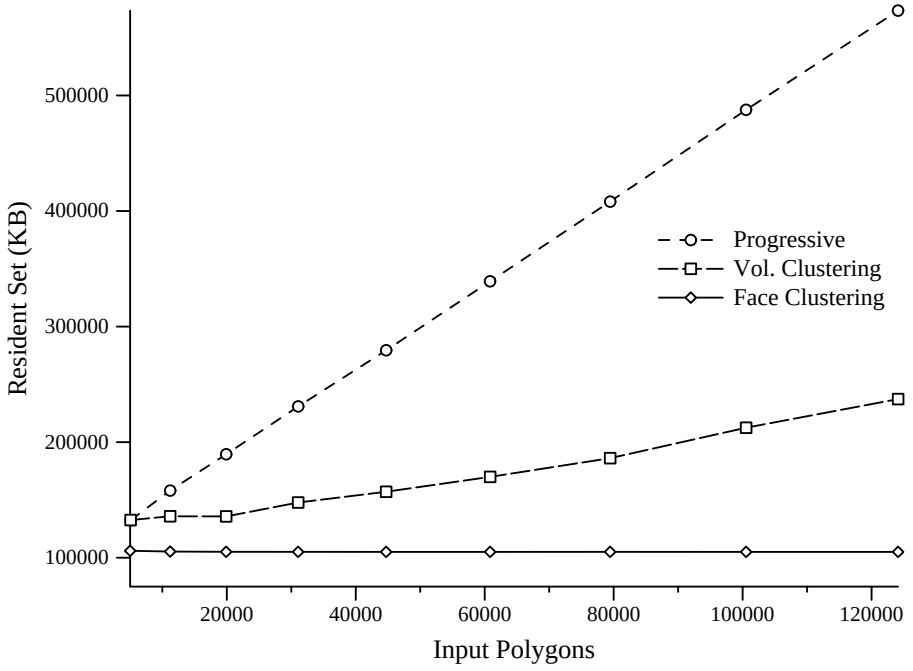


Figure 8.8: Physical memory consumption used during solution process.

levels in the hierarchy is quite inaccurate, and the algorithm is forced to descend far down in the tree to achieve an acceptable solution. In fact, it must typically descend all the way to the level of the input polygons. In contrast, the algorithm based on face cluster hierarchies exhibits essentially unchanging running times once the input complexity has risen to the level of 30,000 polygons. The algorithm is able to find a level in the hierarchy, far above the level of the input triangles, at which an acceptable solution can be computed. Because it never needs to descend deeper into the hierarchy, it requires much less time and far less resident data than the other algorithms.

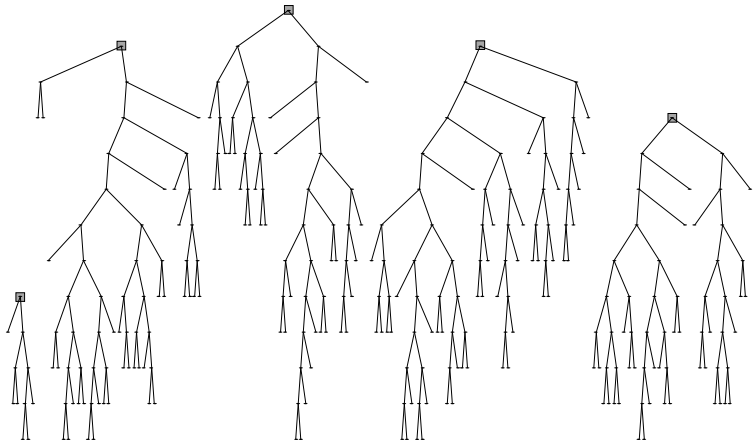


Figure 8.9: An actual face hierarchy used in radiosity simulation on dragon scene in Figure 6.24.

Figure 8.9 shows a face hierarchy used in a radiosity simulation of the scene pictured in Figure 6.24. This is the part of the face hierarchy for the dragon surface which was actually used during the solution phase. The leaves of the hierarchy, corresponding to the input polygons, lie far below the lowest level shown. This demonstrates the primary reason for the efficiency of the face cluster radiosity algorithm; it computes a solution on a very limited piece of the total hierarchy. Also notice that this hierarchy is incomplete, consisting of five separate trees rather than a single tree. For the purposes of radiosity simulation, the very uppermost levels of a complete hierarchy are relatively less useful. Consider the root node for the surface, which represents the entire model as a single cluster. The planar approximation of this cluster does not provide a good fit for the orientations of all the triangles on the original surface. The root nodes shown in Figure 8.9, which correspond to a partition of the dragon into five regions, reflect the orientations of their constituent faces much more accurately.

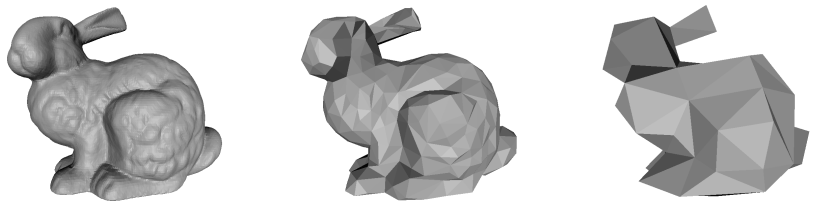
In the preceding discussion, I have glossed over most of the details of the face cluster radiosity algorithm. My goal has only been to present this as one

possible application of the face clustering method developed in this chapter. Further details on the radiosity algorithm and its performance are given by Willmott *et al.* [196].

Chapter 9

Conclusion

I have described an algorithm for the *automatic simplification* of highly detailed polygonal surface models into faithful approximations containing fewer polygons (see Figure 9.1). My empirical tests, as well as those performed by others [26, 124], demonstrate that this quadric-based simplification algorithm provides a good compromise between the speed of simplification and the quality of the resulting approximations. For instance, the approximations shown in Figure 9.1 quite effectively capture the shape of the original and were generated in only 7 seconds on a 200 MHz PentiumPro machine. I have also presented a closely related dual algorithm which can generate hierarchical surface partitions of the model into nearly planar regions.



(a) Original (b) 1000 triangles (c) 150 triangles

Figure 9.1: Sample approximations of a scanned bunny surface model containing 70,000 triangular faces.

9.1 Summary of Contributions

To review, the primary contributions of my work as described in this dissertation are:

- **Quadric Error Metric.** I have developed an error metric which provides a useful characterization of local surface shape. It requires only modest storage space and computation time. Through a simple extension, it can be used on surfaces for which each vertex has an associated set of material properties, such as RGB color and texture coordinates. I have also proven a direct connection between the quadric error metric and surface curvature.

- **Surface Simplification Algorithm.** By combining the quadric error metric with iterative vertex pair contraction, I have developed a fast algorithm for producing high-quality approximations of polygonal surfaces. This algorithm can simplify both manifold and non-manifold models. It is also capable of joining unconnected regions of the model together, thus ultimately simplifying the surface topology by aggregating separate components. In addition to producing single approximations, my algorithm can also be used to generate multiresolution representations such as progressive meshes and vertex hierarchies for view-dependent refinement.

- **Face-Hierarchy Construction.** Finally, I have developed a technique for constructing a hierarchy of well-defined surface regions composed of disjoint sets of faces. This algorithm involves applying a dual form of my simplification algorithm to the dual graph of the input surface. The resulting structure is a hierarchy of face clusters which is an effective multiresolution representation for certain applications, including radiosity.

9.2 Future Directions

There are several ways in which this work could be extended in the future. However, the following avenues appear particularly important or promising.

Broader Range of Models

The broader the class of inputs supported by simplification methods, the more broadly applicable these methods will be. In real world applications, surface models often contain noise of one sort or another. As demonstrated in Section 6.2.2, the quadric error metric can, to some extent, remove noise from the input surface. But further extensions may be possible. If we have some better way of estimating normals than from faces, these normals could be used to construct fundamental quadrics rather than the faces. Alternatively, suppose that we have some information about the distribution of normals. Since quadrics are covariance matrices of normals, it might be possible to incorporate this information directly into the quadrics themselves.

The extended quadric error metric described in Chapter 5 incorporates attributes such as color into the simplification process. But it is based on the assumption that each vertex has a single unique value for each attribute. There are models where this may not be the case. A simple example is the set of

vertex normals on a cube. Vertices at the corners and along the edges of the cube do not have a single unique normal. Rather, there is a normal discontinuity at these points. Discontinuities also frequently arise when texturing objects, such as cylinders, where two ends of the texture come together to form a seam. Vertices at attribute discontinuities could be split into multiple virtual vertices, all with the same position but with separate unique attribute values. With one extended quadric for each of these virtual vertices, and by correctly distinguishing edges joining real vertices and virtual vertices, the extended metric could be generalized to handle these discontinuities.

Higher dimensional quadrics form the basis of the extended quadric metric. A related kind of metric could potentially be applied to higher dimensional data sets. Suppose that we have a volumetric model represented as a tetrahedral mesh with one scalar value per vertex. The edge contraction operation is applicable to all simplicial complexes [144]. We could also define a 4-dimensional quadric at each vertex and proceed with greedy edge contraction.

Finally, all current simplification methods assume that the surface being simplified is rigid. This covers a large class of models used in practice, including those composed of many individual moving, yet rigid, parts such as an engine model. However, there are many other applications where surfaces are changing over time. For example, many animation systems represent characters as surfaces attached to articulated skeletons. As the skeletal joints bend, the surface is deformed. Current simplification methods must be extended to handle this more generalized class of models.

Handling Large Datasets

My algorithm is capable of simplifying large datasets reasonably efficiently. For example, the 1.7 million face turbine blade model shown in Chapter 6 can be simplified in about 5 minutes. However, certain customizations can be made to improve performance on very large datasets (10 million triangles or more).

The primary limiting feature of the algorithm is that it selects the single contraction of minimum cost at each iteration. As shown in Figure 6.5, the cost of this iterative process comes to dominate the total running time of the system. However, for very large models selecting one contraction at a time is probably not necessary. If we are beginning at 10 million faces and producing a 60,000 face approximation, we will need to perform a very long sequence of contractions. We also know that a given sequence of contractions can be reordered (§7.3) without affecting the final result. Therefore, the algorithm could be significantly faster if a large set of contractions were performed at each iteration (much like the original vertex decimation algorithm [172]). In fact, if a constant fraction of the edges are contracted at each iteration, the time complexity of the algorithm can be reduced to $O(n)$. Allowing the simultaneous contraction of many edges also provides a means for parallelizing the simplification algorithm.

Performing batch contractions can improve the memory locality of the algorithm as well. The single contractions chosen in consecutive iterations often fall in widely separated portions of the model. Therefore, the algorithm is con-

stantly touching different sections of the model database at each iteration. If we were to perform a batch of contractions in a single region at once, the locality of memory references could be improved substantially.

Improved Error Analysis

I have analyzed the quadric error metric in terms of its connection to surface curvature, and this provides some justification for why quadric-based simplification performs well. However, the error E_{avg} of the resulting approximations may not be bounded. In general, there are cases (§4.5.1) where minimizing the quadric error produces approximations with vertices which can be arbitrarily far from the original surface. The quadric metric itself is also unable to distinguish between very flat and very sharp vertex neighborhoods (§4.5.2). This raises an interesting question. Minimizing the quadric error does not, in general, guarantee approximations of bounded E_{avg} error. But does this adequately reflect its practical performance? There may be suitable restrictions, satisfied by most models encountered in practice, under which minimizing the quadric error does in fact lead to bounded error approximations.

New methods for measuring approximation error are also needed. For rendering applications, similarity of appearance (§2.3.1) is the ultimate goal. It would be helpful to have an appearance-based metric for reliably comparing the visual similarity of two models. Even if we are primarily concerned with preserving the shape of an object, the error metrics E_{max} and E_{avg} have some drawbacks. They do not address the measurement of attribute error. In developing the extended quadric metric, I have assumed that attribute error can be measured as a Euclidean distance. This can easily be incorporated into these metrics, but it is not a strictly accurate way of assessing attribute error. It is also unclear whether metrics like E_{max} and E_{avg} adequately reflect the similarity of an approximation whose topology has been simplified.

More Effective Simplification

There are several areas in which simplification methods, including my own, could be made more effective. One obvious goal is to devise an algorithm which can produce approximations (for general surfaces) which are provably close to optimal. An algorithm which could preserve higher level surface characteristics, such as symmetry, would also be quite useful. Current simplification methods all seem to perform poorly at extremely low levels of simplification, of say less than 50 triangles. This may very well be a weakness of their shared approach: almost all algorithms derive approximations by repeated transformation (e.g., edge contraction) of the original. At these very low levels, a human could presumably do a substantially better job. This suggests that semi-automatic tools which could incorporate some human interaction (§7.4.2) in the final stages of simplification could be quite useful.

It would also be desirable for simplification systems to incorporate better control over topology simplification. Like my quadric-based algorithm, most

methods that allow topological simplification simplify it only implicitly. It would seem that explicit topological simplification might require a more volumetric approach to surface models, and early methods [53, 82] have moved in this direction This may also lead to more effective aggregation of separate components. The method of aggregation via pair contraction, which I and others [144, 54] have used, has not been completely successful. For instance, it would appear to be overly dependent on the placement of the vertices on the original surface (§6.4).

Decoupling Analysis and Synthesis

I believe that the most promising avenue for improving the quality of automatically generated approximations may well be to decouple the analysis and synthesis phases of the simplification process. Consider the quadric-based algorithm. A particular vertex begins with a quadric constructed from its immediate neighborhood. As other vertices are repeatedly contracted into this vertex, it accumulates a quadric that represents ever larger regions of the surface. In some sense, this is a shape analysis process. The algorithm is constructing information about ever larger regions of the surface. However, the current algorithm actually performs a contraction on the mesh immediately after accumulating the quadrics for the endpoints — it immediately synthesizes a new approximation.

Now imagine that we were to decouple these processes into two separate phases. The first phase would be an analysis phase, gaining information about the structure of the surface at ever coarser levels of detail. This might involve accumulating quadrics over progressively larger regions, constructing face cluster hierarchies, or some completely different technique. After this phase is completed, we could begin simplifying the surface. The current algorithm, when considering the contraction of an edge, can only consider the shape of the immediate neighborhood of this edge, represented by the quadrics of the endpoints. It suffers from a certain shortsightedness because it can only assess the local effect of a contraction. However, if an earlier analysis phase had already been performed, it could consider the effect of a contraction at several levels of detail, from the immediately local to the more global.

Bibliography

[1] Pankaj K. Agarwal and Pavan K. Desikan. An efficient algorithm for terrain simplification. In *Proc. ACM-SIAM Sympos. Discrete Algorithms*, pages 139–147, 1997.

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

[3] P. S. Aleksandrov. *Elementary Concepts of Topology*. Dover Publications, New York, 1961. English translation by Alan Farley from the German *Einfachste Grundbegriffe der Topologie*. Author's name also transliterated "Alexandroff".

[4] P. S. Aleksandrov. *Combinatorial Topology*. Dover, New York, 1998. Reprint of original translation published by Graylock Press in 3 volumes.

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

[6] Daniel G. Aliaga and Anselmo A. Lastra. Smooth transitions in texture-based simplification. *Computers & Graphics*, 22(1):71–81, February 1998.

[7] Michael R. Anderberg. *Cluster Analysis for Applications*. Academic Press, New York, 1973.

[8] James Arvo and David Kirk. *An Introduction to Ray Tracing*, chapter A Survey of Ray Tracing Acceleration Techniques, pages 201–262. Academic Press, London, 1989.

[9] I. Babuska and A. Aziz. On the angle condition in the finite element method. *SIAM Journal of Numerical Analysis*, 1976(13):214–227, 1976.

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

[11] Gill Barequet, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. BOXTREE: A hierarchical representation for surfaces in 3D. *Computer Graphics Forum*, 15(3):387–96, 484, 1996.

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

[13] Jens Berkmann and Terry Caelli. Computation of surface geometry and segmentation using covariance techniques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(11):1114–1116, 1994.

[14] Paul J. Besl and Ramesh C. Jain. Invariant surface characteristics for 3D object recognition in range images. *Computer Vision, Graphics, and Image Processing*, 33:33–80, 1986.

[15] Jim Blinn. The algebraic properties of second-order surfaces. In Jules Bloomenthal, editor, *Introduction to Implicit Surfaces*, pages 53–97. Morgan Kaufmann, San Francisco, 1997.

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

[17] Peter J. Burt and Edward H. Adelson. The Laplacian pyramid as a compact image code. *IEEE Trans. on Communications*, 31(4):532–540, April 1983.

[18] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, September 1978.

[19] Andrew Certain, Jovan Popović, Tony DeRose, Tom Duchamp, David Salesin, and Werner Stuetzle. Interactive multiresolution surface viewing. In *SIGGRAPH 96 Conference Proceedings*, pages 91–98, August 1996.

[20] David Cheriton and Robert E. Tarjan. Finding minimum spanning trees. *SIAM Journal of Computing*, 5(4):724–742, December 1976.

[21] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proc. Ninth Annual Symposium on Computational Geometry*, pages 274–280. ACM, 1993.

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

[23] P. Ciarlet and F. Lamour. Does contraction preserve triangular meshes? *Numerical Algorithms*, 13:201–223, 1996.

[24] P. Cignoni, L. De Floriani, C. Montani, E. Puppo, and R. Scopigno. Multiresoultion modeling and visualization of volume data based on simplicial complexes. In *Proceedings 1994 ACM Symposium on Volume Visualization*, pages 19–26, October 1994. ftp://ftp.disi.unige.it/pub/puppo/PS/ACM-VV94.ps.gz.

[25] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. A general method for preserving attribute values on simplified meshes. In *IEEE Visualization 98 Conference Proceedings*, pages 59–66,518, Oct 1998.

[26] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, 1998.

[27] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–74, June 1998. http://vcg.iei.pi.cnr.it/metro.html.

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

[29] Jonathan Cohen, Dinesh Manocha, and Marc Olano. Simplifying polygonal models using successive mappings. In *Proceedings IEEE Visualization '97*, pages 395–402, October 1997.

[30] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *Proceedings SIGGRAPH 98*, pages 115–122, 1998.

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

[32] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, Boston, 1993.

[33] Julian Lowell Coolidge. *A History of the Conic Sections and Quadric Surfaces*. Oxford University Press, London, 1945.

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

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

[36] R. Courant and D. Hilbert. *Methods of Mathematical Physics*, volume 1. Interscience Publishers, New York, 1953.

[37] Eduardo F. D'Azevedo and R. Bruce Simpson. On optimal interpolation triangle incidences. *SIAM J. Sci. Stat. Comput.*, 10(6):1063–1075, 1989.

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

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

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

[41] Leila De Floriani, Paola Magillo, and Enrico Puppo. Efficient implementation of multi-triangulations. In *IEEE Visualization 98 Conference Proceedings*, pages 43–50,517, Oct 1998.

[42] Leila De Floriani, Enrico Poppo, and Paola Magillo. A formal approach to multiresolution hypersurface modeling. In W. Straßer, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*. Springer-Verlag, 1997.

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

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

[45] Hervé Delingette. Simplex meshes: A general representation for 3D shape reconstruction. In *Conf. on Computer Vision and Pattern Recognition (CVPR '94)*, June 1994. http://zenon.inria.fr/epidaure/personnel/delingette/delingette.html.

[46] Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *Proceedings SIGGRAPH 98*, pages 85–94, 1998.

[47] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10(6):356–360, September 1978.

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

[49] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization 97 Conference Proceedings*, pages 81–88, October 1997. http://www.llnl.gov/graphics/ROAM/.

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

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

[52] Herbert Edelsbrunner and Dmitry V. Nekhayev. Topology and curvature preserving surface decimation. Technical Report rgi-tech-97-010, Raindrop Geomagic, Inc., Champaign, IL, 1997.

[53] Jihad El-Sana and Amitabh Varshney. Controlled simplification of genus for polygonal models. In *IEEE Visualization 97 Conference Proceedings*, pages 403–410, 1997.

[54] Carl Erikson and Dinesh Manocha. GAPS: General and automatic polygonal simplification. In *Proc. Symposium on Interactive 3D Graphics '99*, pages 79–88,225, 1999. http://www.cs.unc.edu/~eriksonc/Research/Paper/.

[55] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press, Boston, Third edition, 1993.

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

[57] R. L. Ferguson, R. Economy, W. A. Kelley, and P. P. Ramos. Continuous terrain level of detail for visual simulation. In *Proceedings of the 1990 Image V Conference*, pages 145–151. Image Society, Tempe, AZ, June 1990.

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

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

[60] David R. Forsey and Richard H. Bartels. Hierarchical B-spline refinement. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 205–212, August 1988.

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

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

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

[64] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proc.)*, 1993.

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

[66] Harold N. Gabow, Zvi Galil, and Thomas H. Spencer. Efficient implementation of graph algorithms using contraction. *Journal of the ACM*, 36(3):540–572, July 1989.

[67] Michael Garland and Paul S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Comp. Sci. Dept., Carnegie Mellon University, September 1995. http://www.cs.cmu.edu/~garland/scape.

[68] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Proc.*, pages 209–216, August 1997. http://www.cs.cmu.edu/~garland/quadrics/.

[69] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization 98 Conference Proceedings*, pages 263–269,542, October 1998. http://www.cs.cmu.edu/~garland/quadrics/.

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

[71] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins Univ. Press, Baltimore, Third edition, 1996.

[72] M. Gopi and D. Manocha. A unified approach for simplifying polygonal and spline models. In *IEEE Visualization 98 Conference Proceedings*, pages 271–278,543, Oct 1998.

[73] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *SIGGRAPH 96 Conference Proceedings*, pages 43–54, 1996. http://research.microsoft.com/MSRSIGGRAPH/96/Lumigraph.htm.

[74] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings SIGGRAPH 96*, pages 171–180, 1996.

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

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

[77] André Guéziec. Surface simplification inside a tolerance volume. Technical report, Yorktown Heights, NY 10598, March 1996. IBM Research Report RC 20440, http://www.watson.ibm.com:8080/search_paper.shtml.

[78] André Guéziec, Gabriel Taubin, Francis Lazarus, and William Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. In *IEEE Visualization 98 Conference Proceedings*, pages 383–390,553, Oct 1998.

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

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

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

[82] T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.

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

[84] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. In *Multiresolution Surface Modeling Course Notes*. ACM SIGGRAPH, 1997. http://www.cs.cmu.edu/~ph, draft of Carnegie Mellon University tech. report, to appear.

[85] Paul S. Heckbert and Michael Garland. Optimal triangulation and quadric-based surface simplification. 1999. Submitted for publication.

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

[87] Michael Henle. *A Combinatorial Introduction to Topology*. Dover, New York, 1994. Reprint of 1979 edition of W. H. Freeman.

[88] D. Hilbert and S. Cohn-Vossen. *Geometry and the Imagination*. Chelsea, New York, 1952. Translated from the German *Anschauliche Geometrie* by P. Nemenyi.

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

[90] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96 Proc.*, pages 99–108, August 1996. http://research.microsoft.com/~hoppe/.

[91] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH 97 Proc.*, pages 189–198, August 1997. http://research.microsoft.com/~hoppe/.

[92] Hugues Hoppe. Efficient implementation of progressive meshes. *Computers & Graphics*, 22(1):27–36, 1998.

[93] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization 98 Conference Proceedings*, pages 35–42,516, Oct 1998.

[94] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 71–78, July 1992. http://research.microsoft.com/~hoppe/.

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

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

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

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

[99] Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall International, London, 1989.

[100] Andrew Johnson and Martial Hebert. Control of polygonal mesh resolution for 3-D cmputer vision. Technical report, Robotics Institute, Carnegie Mellon U., February 1997. CMU-RI-TR-96-20, http://www.ius.cs.cmu.edu/IUS/clash_usr0/aej/www/papers/TR-96-20.ps.gz.

[101] I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, New York, 1986.

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

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

[104] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, March 1995.

[105] Åke Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, 1996.

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

[107] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, Reading, MA, Third edition, 1997.

[108] Leif Kobbelt, Swen Campagna, and Hans Peter Seidel. A general framework for mesh decimation. In *Proc. Graphics Interface '98*, pages 43–50, June 1998.

[109] Leif Kobbelt, Swen Campagna, Jens Vorsatz, and Hans-Peter Seidel. Interactive multi-resolution modeling on arbitrary meshes. In *Proc. SIGGRAPH 98*, pages 105–114, 1998.

[110] Jan J. Koenderink. *Solid Shape*. MIT Press, Cambridge, MA, 1990.

[111] Erwin Kreyszig. *Introduction to Differential Geometry and Riemannian Geometry*. Number 16 in Mathematical Expositions. University of Toronto Press, Toronto, 1968. English translation — original German edition published in 1967 by Akademische Verlagsgesellschaft.

[112] Erwin Kreyszig. *Differential Geometry*. Dover, New York, 1991. Reprint of 1959 edition published by University of Toronto Press as *Mathematical Expositions* No. 11.

[113] Peter Lancaster and Kęstutis Šalkauskas. *Curve and Surface Fitting: An Introduction*. Academic Press, London, 1986.

[114] Rynson Lau, Mark Green, Danny To, and Janis Wong. Real-time continuous multi-resolution method for models of arbitrary topology. *Presence: Teleoperators and Virtual Environments*, 7(1):22–35, February 1998. http://www.cs.cityu.edu.hk/~rynson/pub-cgvr.html.

[115] Detlef Laugwitz. *Differential and Riemannian Geometry*. Academic Press, New York, 1965. English translation by Fritz Steinhardt — original German edition, *Differentialgeometrie,* published in 1960 by B. G. Teubner.

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

[117] Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[118] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. In *Proc. SIGGRAPH 98*, pages 95–104, 1998.

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

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

[121] Marc Levoy and Pat Hanrahan. Light field rendering. In *SIGGRAPH 96 Conference Proceedings*, pages 31–42, 1996. http://www.graphics.stanford.edu/projects/lightfield/.

[122] Ping Liang and John S. Todhunter. Representation and recognition of surface shapes in range images: A differential geometry approach. *Computer Vision, Graphics, and Image Processing*, 52:78–109, 1990.

[123] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96*, pages 109–118, August 1996.

[124] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In *IEEE Visualization 98 Conference Proceedings*, pages 279–286,544, Oct 1998.

[125] Charles Loop. Smooth subdivision surfaces based on triangles. Master's thesis, Department of Mathematics, University of Utah, August 1987.

[126] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface reconstruction algorithm. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):163–170, July 1987.

[127] Michael Lounsbery, Tony D. DeRose, and Joe Warren. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Trans. on Graphics*, 16(1):34–73, 1997. http://www.cs.washington.edu/research/graphics/pub/.

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

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

[130] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH 97 Proc.*, pages 199–208, August 1997.

[131] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *1995 Symposium on Interactive 3D Graphics*, pages 95–102. ACM SIGGRAPH, April 1995.

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

[133] Makoto Maruya. Generating a texture map from object-surface texture data. *Computer Graphics Forum*, 14(3):397–405,506–507, 1995. Proc. Eurographics '95.

[134] Nelson L. Max. Computer graphics distortion for IMAX and OMNIMAX projection. In *Nicograph '83 Proceedings*, pages 137–159, December 1983.

[135] E. A. Maxwell. *General Homogeneous Coordinates in Space of Three Dimensions*. University Press, Cambridge, 1951.

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

[137] Stan Melax. A simple, fast, and effective polygon reduction algorithm. *Game Developer*, pages 44–49, November 1998. http://www.gdmag.com/backissue1998.htm#nov98.

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

[139] Atul Narkhede and Dinesh Manocha. Fast polygon triangulation based on Seidel's algorithm. In Alan W. Paeth, editor, *Graphics Gems V*, pages 394–397. Academic Press, Boston, 1995.

[140] Richard G. Niemi, Bernard Grofman, Carl Carlucci, and Thomas Hofeller. Measuring compactness and the role of a compactness standard in a test for partisan and racial gerrymandering. *Journal of Politics*, 52(4):1155–1181, 1990.

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

[142] Ken Perlin and Luiz Velho. Live paint: Painting with procedural multiscale textures. In *SIGGRAPH 95 Proc.*, pages 153–160, August 1995.

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

[144] Jovan Popović and Hugues Hoppe. Progressive simplicial complexes. In *SIGGRAPH 97 Proc.*, pages 217–224, 1997. http://research.microsoft.com/~hoppe/.

[145] Vaughan Pratt. Direct least-squares fitting of algebraic surfaces. In *Proc. SIGGRAPH 87*, pages 145–152, 1987.

[146] P. M. Prenter. *Splines and Variational Methods*. John Wiley & Sons, New York, 1975.

[147] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, New York, NY, 1985.

[148] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Second edition, 1992. http://www.nr.com.

[149] Enrico Puppo. Variable resolution terrain surfaces. In *Proceedings of the Eighth Canadian Conference on Computational Geometry*, pages 202–210, 1996.

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

[151] Matthew M. Rafferty, Daniel G. Aliaga, Voicu Popescu, and Anselmo A. Lastra. Images for accelerating architectural walkthroughs. *IEEE Computer Graphics & Applications*, 18(6), November – December 1998.

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

[153] Ramesh Raskar, Greg Welch, Matt Cutts, Adam Lake, Lev Stesin, and Henry Fuchs. The office of the future: A unified approach to image-based modeling and spatially immersive displays. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 179–188. ACM SIGGRAPH, 1998.

[154] M. Reddy. SCROOGE: Perceptually-driven polygon reduction. *Computer Graphics Forum*, 15(4):191–203, 1996.

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

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

[157] John Rohlf and James Helman. IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics. In *SIGGRAPH '94 Proc.*, pages 381–394, July 1994. http://www.sgi.com/software/performer/.

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

[159] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. Technical Report IBM Research Report RC 20423, IBM T. J. Watson Research, Yorktown Heights, NY, 1996.

[160] Azriel Rosenfeld, editor. *Multiresolution Image Processing and Analysis*. Springer, Berlin, 1984.

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

[162] Y. Rubner and C. Tomasi. Texture metrics. In *Proc. IEEE Intl. Conf. on Systems, Man, and Cybernetics*, pages 4601–4607, 1998.

[163] Holly E. Rushmeier, Charles Patterson, and Aravindan Veerasamy. Geometric simplification for indirect illumination calculations. In *Proc. Graphics Interface '93*, pages 227–236, Toronto, Ontario, May 1993. Canadian Inf. Proc. Soc. http://www.cc.gatech.edu/gvu/people/Phd/Charles.Patterson/research/gsii/gsii.html.

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

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

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

[167] G. Schaufler and W. Stürzlinger. Generating multiple levels of detail from polygonal geometry models. In M. Göbel, editor, *Virtual Environments '95 (Eurographics Workshop)*, pages 33–41. Springer Verlag, January 1995.

[168] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–236, August 1996. Proceedings of Eurographics '96.

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

[170] Peter Schröder and Denis Zorin, editors. *Subdivision for Modeling and Animation*, SIGGRAPH 98 Course Notes, Course 36, 1998.

[171] William J. Schroeder. A topology modifying progressive decimation algorithm. In *IEEE Visualization 97 Conference Proceedings*, pages 205–212,545, 1997.

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

[173] H. A. Schwarz. Sur une définition erronée de l'aire d'une surface courbe. In *Collected Papers*, volume 2, pages 309–311. Berlin, 1890.

[174] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1(1):51–64, 1991.

[175] Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 75–82, 1996.

[176] Brian Smits, James Arvo, and Donald Greenberg. A clustering algorithm for radiosity in complex environments. In *Proceedings of SIGGRAPH '94*, pages 435–442, July 1994.

[177] John Snyder and Jed Lengyel. Visibility sorting and compositing without splitting for image layer decomposition. In *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 219–230. ACM SIGGRAPH, 1998.

[178] Marc Soucy, Guy Godin, and Marc Rioux. A texture-mapping approach for the compression of colored 3D triangulations. *The Visual Computer*, 12(10):503–514, 1996.

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

[180] Oliver G. Staadt and Markus H. Gross. Progressive tetrahedralizations. In *IEEE Visualization 98 Conference Proceedings*, pages 397–402,555, Oct 1998.

[181] Paul Steed. The art of low-polygon modeling. *Game Developer*, pages 62–69, June 1998. http://www.gdmag.com/backissue1998.htm#jun98.

[182] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgann Kaufmann, San Francisco, CA, 1996.

[183] Gilbert Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, San Diego, Third edition, 1988.

[184] Richard Szeliski and Heung-Yeung Shum. Creating full view panoramic mosaics and environment maps. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 251–258. ACM SIGGRAPH, 1997.

[185] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Number 44 in Regional Conference Series in Applied Mathematics. SIAM, Philadelphia, 1983.

[186] Isaac J. Trotts, Bernd Hamann, Kenneth I. Joy, and David F. Wiley. Simplification of tetrahedral meshes. In *IEEE Visualization 98 Conference Proceedings*, pages 287–295, Oct 1998.

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

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

[189] Joost van Lawick van Pabst and Hans Jense. Dynamic terrain generation based on multifractal techniques. In *Proc. Intl. Workshop on High Performance Computing for Computer Graphics and Visualization*, pages 186–203, July 1995.

[190] William Welch. *Serious Putty: Topological Design for Variational Curves and Surfaces*. PhD thesis, CS Dept, Carnegie Mellon U., December 1995. CMU-CS-95-217, ftp://reports.adm.cs.cmu.edu/usr/anon/1995/.

[191] William Welch and Andrew Witkin. Free-form shape design using triangulated surfaces. In *SIGGRAPH '94 Proc.*, pages 247–256. ACM, 1994.

[192] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics With Open Inventor, Release 2*. Addison-Wesley, 1994.

[193] Dieter Willersinn and Walter G. Kropatsch. Dual graph contraction for irregular pyramids. In *Proc. 12th IAPR Intl. Conf. on Pattern Recognition*, volume III, pages 251–256, October 1994.

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

[195] T. J. Willmore. *An Introduction to Differential Geometry*. Oxford University Press, London, 1959.

[196] Andrew J. Willmott, Paul S. Heckbert, and Michael Garland. Face cluster radiosity. 1999. Submitted for publication.

[197] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*. Addison-Wesley Developers Press, Reading, Mass., Second edition, 1997. http://www.opengl.org.

[198] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of Visualization '96*, pages 327–334, October 1996.

Appendix A

Implementation Notes

The purpose of this appendix is to augment the outline of my algorithm and the analysis of its performance with specific details about my implementation. All of the code presented here is written in C++ and is designed to be portable over a wide range of compilers. I have made only minimal use of features such as templates, and I have avoided relatively recent additions to the language such as exceptions and the Standard Template Library. The code listings presented here are not totally complete and have been condensed from my actual code by removing some tangential functionality. I have tried to focus on several key procedures, leaving out various support functions such as constructors and accessors.

My first generation implementation, QSlim 1.0, developed in 1996, forms the basis for my initial results [68] and the performance surveys that have been published since then [26, 124]. The results reported more recently [69] and in this work are based on the second generation QSlim 2.0 implementation. This implementation has been publicly available[1] since March 1999. The code listings presented here are based on this software package.

A.1 Quadric Data Structure

One of the core components of my simplification algorithm is the quadric error metric. The basic metric, supporting geometry without attributes, can be implemented very easily. Each quadric Q is represented by an object of class `MxQuadric3`.

[1] Currently distributed at http://www.cs.cmu.edu/~garland/quadrics/index.html

```
class MxQuadric3
{
private:
    // Ten unique quadric coefficients
    double a2, ab, ac, ad;
    double     b2, bc, bd;
    double         c2, cd;
    double             d2;

    // Associated area for weighting
    double r;

public:
    // Return the (A, b, c) and area components of a quadric, respectively
    Mat3 tensor() const;
    Vec3 vector() const;
    double offset() const;
    double area() const;

    Mat4 homogeneous() const;
};
```

As mentioned in Section 6.1.2, quadrics account for about 33% of the total memory used during simplification. However, converting these `double` coefficients to `float` values can cause significant numerical problems.

An object of this class corresponds to the area-weighted quadric

$$Q = \left(r \begin{bmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{bmatrix}, r \begin{bmatrix} ad \\ bd \\ cd \end{bmatrix}, rd^2 \right)$$

It can also be written in homogeneous form as the matrix

$$\mathbf{Q} = r \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

Note that recording the area term r separately is not strictly necessary. The system always pre-multiplies the area factor into the other coefficients. In other words, the field `a2` will hold the value ra^2. If memory consumption is critical, the area term can be dropped from the structure. However, it can be useful to have immediate access to a measure of the area corresponding to a particular quadric.

Addition of quadrics is performed by adding all corresponding components (10 coefficients and the area r) together. In contrast, scaling operations (e.g., `Q*=2`) scale all the components *except* the area. While this means that `Q*=2` and `Q+=Q` do not produce the same result, it makes quadric scaling more convenient. Premultiplying a quadric by its area can be written simply as `Q*=Q.area()`.

It also means that weighting quadrics by penalty factors, as in boundary constraints, does not alter the associated area term.

The first critical function of a quadric Q is to determine the error $Q(\mathbf{v}) = \mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{v} + 2\mathbf{b}^\mathsf{T}\mathbf{v} + c$. This is evaluated using the following procedure.

```
double MxQuadric3::evaluate(double x, double y, double z) const
{
    return x*x*a2 + 2*x*y*ab + 2*x*z*ac + 2*x*ad
                  + y*y*b2   + 2*y*z*bc + 2*y*bd
                             + z*z*c2   + 2*z*cd
                                        + d2;
}
```

Recall that this will evaluate the properly area-weighted error because all coefficients (`a2`, `ab`, etc.) will be pre-multiplied by the area factor r.

For each contraction, the simplification algorithm attempts to find the position $\bar{\mathbf{v}}$ which minimizes the error $Q(\bar{\mathbf{v}})$. Assuming that we have classes `Vec3` and `Mat3` for 3-dimensional vectors and matrices, respectively, this optimal position can be computed as follows.

```
bool MxQuadric3::optimize(Vec3& v) const
{
    // Compute A^-1 and fail if |det A| < 10^-12
    Mat3 Ainv;
    double det = tensor().invert(Ainv);
    if( FEQ(det, 0.0, 1e-12) )
        return false;

    // A is nonsingular, so v̄ = -A^-1 b
    v = -(Ainv*vector());
    return true;
}
```

The return value of this function indicates whether a unique minimum exists. If it does not, the system will fall back on selecting between the positions of the two vertices being contracted. Also note that, because $\mathbf{A}$ is only a 3×3 matrix, its inverse can be found using a very simple, and efficient procedure.

A.2 Model Representation

Two primary concerns have motivated the design of the representation for surface models. First, it is important that the structure be reasonably efficient in its use of memory. An obvious application of the system will be to simplify very large models of a million or more faces, and these models will necessarily require substantial amounts of storage. Second, my simplification algorithm is meant to support non-manifold surfaces (§2.1.1). Many common adjaceny representations (e.g., winged-edge structures) assume surfaces have manifold topology, and are thus unsuitable for this application.

A.2.1 Geometric Primitives

Polygonal models are constructed from sets of vertices and faces. Each individual vertex and face is assigned a unique integer identifier.

```
typedef unsigned int MxVertexID;
typedef unsigned int MxFaceID;

// The invalid vertex/face identifier.  Normally 0xffffffffU on 32-bit machines.
#define MXID_NIL UINT_MAX
```

They are always referenced by identifier rather than by a direct pointer. As currently implemented, these identifiers are simply indices into arrays of structures. Simply passing pointers, to vertices for instance, is less cumbersome. Instead of passing a single pointer, we need to pass both a model structure and the appropriate integer identifier. However, this approach has some nice benefits. For example, maintaining references as indices allows the underlying model code to move and reallocate blocks of vertices without disturbing other parts of the system.

Vertices are represented using three single precision floating point values.

```
typedef float MxVertex[3];
```

In the full implementation, this is actually a class which wraps some additional functionality around the array. I have chosen to use `float` values rather than `double` values to conserve space.

Faces are simply triples of vertex identifiers.

```
class MxFace
{
private:
    MxVertexID v[3];

public:
    MxVertexID& operator[](int i)       { return v[i]; }

    // Replace all occurrences of ID from with ID to and return the
    // number of replacements made.
    int remap_vertex(MxVertexID from, MxVertexID to);
};
```

Various other utility methods are also supported, but they are not used in the code contained in this appendix.

Lists of faces and vertices are frequently used while manipulating surface models. I represent these as lists of vertex and face identifiers.

```
class MxVertexList;
class MxFaceList;
```

These lists are currently implemented as dynamically resized arrays. In addition to the usual accessor methods, they provide an `add()` method to insert new elements at the end of the list and a `reset()` method to remove all elements from the list.

A.2.2 Surface Model Structures

The surface model representation is divided into two classes. `MxBlockModel` is the core model class. It maintains the set of vertices and faces for the surface.

```
class MxBlockModel
{
public:
    // Vertices and faces are stored in internal arrays. Identifiers
    // are indices into these arrays.  These procedures return
    // the length of these arrays, and hence the next available index.
    unsigned int vert_count() const;
    unsigned int face_count() const;

    // Access vertices and faces by indexing into the underlying arrays.
    MxVertex& vertex(unsigned int i);
    MxFace& face(unsigned int i);
};
```

While I am focusing on geometry alone, this class also tracks any attributes, such as color and surface normals, which might be attached to the model. These properties are accessed using a similar mechanism to that shown for vertices and faces (e.g., `color_count()`).

Models in the surface simplification package are represented using a higher level `MxStdModel` class. Derived from `MxBlockModel`, it provides an important additional level of connectivity information.

```
class MxStdModel : public MxBlockModel
{
private:
    // Get internal marks on faces and vertices
    unsigned char fmark(MxFaceID i) const;
    unsigned char vmark(MxVertexID i) const;

    // Set internal marks on faces and vertices
    void fmark(MxFaceID i, unsigned char m);
    void vmark(MxVertexID i, unsigned char m);

public:
    // Return list of faces connected to a given vertex
    MxFaceList& neighbors(MxVertexID v);

    // Procedures for marking faces
    void mark_neighborhood(MxVertexID, unsigned short mark=0);
    void mark_neighborhood_delta(MxVertexID, short delta);

    // Once faces are marked, we can collect them in lists
    void collect_unmarked_neighbors(MxVertexID, MxFaceList& faces);
    void partition_marked_neighbors(MxVertexID, unsigned short pivot,
                                    MxFaceList& below,
                                    MxFaceList& above);

    // The marking and collecting primitives allow us to extract
    // connectivity information from the model.
    void collect_edge_neighbors(MxVertexID, MxVertexID, MxFaceList&);
};
```

Since this code is intended to handle general non-manifold surfaces, it can make few assumptions about the structure of the surface. Therefore, I have chosen to maintain only the most minimal connectivity information. Every face naturally has a link to the three vertices which form its corners. The `MxStdModel` class adds, for every vertex, a list of faces which are linked to that vertex. These lists are accessed using the `neighbors()` method.

The model class uses a system of marking and collection primitives to support the extraction of higher-level connectivity. Each face and vertex has an 8-bit internal *mark* field attached to it. Every face neighboring a given vertex can be marked using the procedure

```
void MxStdModel::mark_neighborhood(MxVertexID vid, unsigned short mark)
{
    for(unsigned int i=0; i<neighbors(vid).length(); i++)
        fmark(neighbors(vid)(i), mark);
}
```

The related method

```
void MxStdModel::mark_neighborhood_delta(MxVertexID vid, short delta);
```

increments the marks of the faces surrounding the given vertex by `delta`.

Given a set of faces, particular face subsets can be collected based on the value of the associated marks. The most common procedure collects a list of all unmarked faces[2] attached to a given vertex

```
void MxStdModel::collect_unmarked_neighbors(MxVertexID vid,
                                            MxFaceList& faces)
{
    for(unsigned int i=0; i<neighbors(vid).length(); i++)
    {
        unsigned int fid = neighbors(vid)(i);
        if( !fmark(fid) )
        {
            faces.add(fid);
            fmark(fid, 1);
        }
    }
}
```

Alternatively, all the faces surrounding a vertex can be partitioned based on whether their mark values are above or below a certain pivot value.

```
void MxStdModel::partition_marked_neighbors(MxVertexID v,
                                            unsigned short pivot,
                                            MxFaceList& lo,
                                            MxFaceList& hi);
```

To see how this process works, suppose we want to build a list of all the faces adjacent to an edge $(\mathbf{v}_1, \mathbf{v}_2)$. This can be accomplished very easily using the marking and collection primitives.

```
void MxStdModel::collect_edge_neighbors(MxVertexID v1, MxVertexID v2,
                                        MxFaceList& faces)
{
    mark_neighborhood(v1, 1);
    mark_neighborhood(v2, 0);
    collect_unmarked_neighbors(v1, faces);
}
```

A.3 Quadric-Based Simplification

Using the quadric and geometric primitives defined in the previous sections, we can proceed to the implementation of the simplification algorithm itself. Support for quadric-based simplification is divided between the model representation classes and a simplification class. The model structures implement the underlying contraction primitives. The greedy decimation loop driven by the quadric metric is encapsulated in its own class.

[2] Unmarked faces are those whose mark value is 0.

A.3.1 Contraction Primitives

In addition to tracking the connectivity of the surface, the standard model structure provides the fundamental operators for manipulating the surface. Only the contraction operators are of interest here, but other functionality, such as edge and face splitting, is also provided.

Contractions are recorded in the following structure.

```
class MxPairContraction
{
public:
    MxVertexID v1, v2;          // The two vertices being contracted
    float dv1[3], dv2[3];       // Offsets of v1, v2 from v̄

    MxFaceList dead_faces;      // List of faces deleted by contraction
    MxFaceList delta_faces;     // List of faces moved by contraction
    unsigned int delta_pivot;   // Split point in delta_faces
};
```

Faces which were only connected to one of $\mathbf{v}_1, \mathbf{v}_2$ are moved, as opposed to deleted, by the contraction. They are partitioned according to which vertex they were connected to. The `delta_pivot` field records the split between these two sets. All faces listed in `delta_faces` prior to the pivot were originally connected to $\mathbf{v}_1$. All subsequent faces were originally connected to $\mathbf{v}_2$.

The contraction operation is divided into two phases. First, all the fields of the contraction record must be filled in. Once the structure is complete, the contraction can be performed and the surface modified.

```
class MxStdModel : public MxBlockModel
{
public:
    // Fill in the given contraction record.
    void compute_contraction(MxVertexID v1, MxVertexID v2,
                             MxPairContraction *conx,
                             const float *vnew);

    // Perform the given contraction operation.
    void apply_contraction(const MxPairContraction& conx);
};
```

Filling in the contraction record is quite straightforward. Most of the fields can be directly computed from the vertex identifiers and their positions. The primary non-trivial task of this procedure involves filling the lists of faces which would be deleted and changed by the contraction.

```
void MxStdModel::compute_contraction(MxVertexID v1, MxVertexID v2,
                                     MxPairContraction *conx, const float *vnew)
{
    conx->v1 = v1;
    conx->v2 = v2;
    mxv_sub(conx->dv1, vnew, vertex(v1), 3);    // Δv1 = v̄ − v1
    mxv_sub(conx->dv2, vnew, vertex(v2), 3);    // Δv2 = v̄ − v2

    // Clear the face lists
    conx->delta_faces.reset();
    conx->dead_faces.reset();

    // Mark the neighborhood of (v1, v2) such that each face is tagged with
    // the number of times the vertices v1, v2 occur in it.  Possible values are 1 or 2.
    mark_neighborhood(v2, 0);
    mark_neighborhood(v1, 1);
    mark_neighborhood_delta(v2, 1);

    // Now partition the neighborhood of (v1, v2) into those faces which degenerate
    // during contraction (mark=2) and those which are merely reshaped (mark=1).
    partition_marked_neighbors(v1, 2, conx->delta_faces, conx->dead_faces);
    conx->delta_pivot = conx->delta_faces.length();
    partition_marked_neighbors(v2, 2, conx->delta_faces, conx->dead_faces);
}
```

Contractions are performed in three steps. First, the position of vertex $\mathbf{v}_1$ is updated. This changes the geometry of the model. Next, all occurrences of $\mathbf{v}_2$ are replaced by $\mathbf{v}_1$. This updates the connectivity of the mesh. Finally, $\mathbf{v}_2$ and all degenerate faces can be deleted from the model entirely.

```
void MxStdModel::apply_contraction(const MxPairContraction& conx)
{
    MxVertexID v1=conx.v1, v2=conx.v2;
    unsigned int i;

    mxv_addinto(vertex(v1), conx.dv1, 3);    // Move v1 to v1 + Δv1 = v̄

    // Replace all remaining occurrences of v2 with v1
    for(i=conx.delta_pivot; i<conx.delta_faces.length(); i++)
    {
        MxFaceID fid = conx.delta_faces(i);
        face(fid).remap_vertex(v2, v1);
        neighbors(v1).add(fid);
    }

    // Delete degenerate faces and remove them from the neighborhood
    // lists associated with their vertices.
    for(i=0; i<conx.dead_faces.length(); i++)
        unlink_face(conx.dead_faces(i));

    // Having removed all links to v2, we can remove it from the model.
    vertex_mark_invalid(v2);
    neighbors(v2).reset();
}
```

A.3.2 Decimation Procedure

Built on top of these underlying contraction operators is the greedy simplification process, which is presented in this section. For the purposes of this presentation, I have simplified the actual class hierarchy somewhat. The full system supports both iterative edge and face contraction. Consequently, the simplification code is divided between an underlying base class `MxQSlim` and two specialized operator-specific classes. However, the code presented here supports only pair contraction, and I have combined all the relevant functionality into a single class.

The first phase of the simplification process is initialization. During this phase, the quadrics associated with the vertices of the original model are constructed. Once these quadrics have been constructed, all potential candidate pairs can be selected, evaluated, and placed in a heap to efficiently track the minimum cost contraction. Note that throughout this code, the variable `m` is the model being simplified.

```
void MxQSlim::initialize()
{
    // Compute the initial quadrics at each vertex
    collect_quadrics();

    // Add constraint quadrics, if desired.
    if( boundary_weight > 0.0 )
        constrain_boundaries();

    // Collect all edges/pairs for consideration and call compute_edge_info() on each.
    // This will also enter all these pairs into a heap.
    collect_edges();
}
```

Computing the initial quadrics is very simple. Every vertex has a corresponding quadric. For each face, we compute the corresponding fundamental quadric. Once properly area-weighted, this fundamental quadric is added to the quadrics for each of the three corners of the face. After traversing the list of all faces, the initial quadrics will have been completely constructed. Subsequently, other additional quadrics (e.g., boundary constraints) can be added as well.

```
void MxQSlim::collect_quadrics()
{
    unsigned int j;

    // Clear all quadrics.  There is one quadric per vertex.
    for(j=0; j<quadrics.length(); j++)
        quadrics(j).clear();

    for(MxFaceID i=0; i<m->face_count(); i++)
    {
        MxFace& f = m->face(i);
        Vec3 v1(m->vertex(f[0])), v2(m->vertex(f[1])), v3(m->vertex(f[2]));

        // Construct fundamental quadric for this face
        Vec4 p = triangle_plane(v1, v2, v3);
        Quadric Q(p[X], p[Y], p[Z], p[W], m->compute_face_area(i));

        // Area-weight quadric and add it into the three quadrics for the corners
        Q *= Q.area();
        quadrics(f[0]) += Q;
        quadrics(f[1]) += Q;
        quadrics(f[2]) += Q;
    }
}
```

Every candidate edge (or pair) is recorded in a structure that tracks the endpoints of the edge and the position to which the vertices will be contracted to. This class inherits from `MxHeapable` so that it can be tracked in a heap. This base class stores the contraction cost as the heap key, and it records the position of the candidate in the heap to facilitate efficient updates.

```
class MxQSlimEdge : public MxHeapable
{
public:
    MxVertexID v1, v2;
    float vnew[3];          // Target position v̄
};
```

Evaluating a particular edge primarily involves computing the target position $\bar{\mathbf{v}}$. Once this position, and its error $Q(\bar{\mathbf{v}})$ have been computed, the edge can be either placed in the heap or its position in the heap updated.

```
void MxQSlim::compute_edge_info(MxQSlimEdge *info)
{
    compute_target_placement(info);

    if( info->is_in_heap() )
        heap.update(info);
    else
        heap.insert(info);
}

void MxQSlim::compute_target_placement(MxQSlimEdge *info)
{
    MxVertexID i=info->v1, j=info->v2;
    double e_min;

    // Quadric for edge is the sum of quadrics for the endpoints
    Quadric Q = quadrics(Qi);  Q += quadrics(j);

    // Try to compute the quadric-optimal position
    if( Q.optimize(&info->vnew[X], &info->vnew[Y], &info->vnew[Z]) )
        e_min = Q(info->vnew);
    else
    // No unique optimal position.  Select best endpoint position.
    {
        Vec3 vi(m->vertex(i)), vj(m->vertex(j));
        Vec3 best;

        double ei=Q(vi), ej=Q(vj);

        if( ei < ej ) { e_min = ei; best = vi; }
        else          { e_min = ej; best = vj; }

        info->vnew[X] = best[X];
        info->vnew[Y] = best[Y];
        info->vnew[Z] = best[Z];
    }

    // Negate the error because the heap class puts the maximum key on top.
    info->heap_key(-e_min);
}
```

Once initialization is complete, the second phase of simplification begins. This is a simple iterative procedure that runs until the given target number of faces is achieved. At each iteration, the minimum cost contraction is removed from the heap and that contraction is performed.

```
bool MxQSlim::decimate(unsigned int target)
{
    MxPairContraction conx;

    while( valid_faces > target )
    {
        MxQSlimEdge *info = (MxQSlimEdge *)heap.extract();
        // Stop early if candidate edge set is exhausted.
        if(!info)  return false;

        // Perform the selected contraction
        MxVertexID v1=info->v1, v2=info->v2;
        m->compute_contraction(v1, v2, &conx, info->vnew);
        apply_contraction(conx);

        delete info;
    }

    return true;
}
```

The actual contraction of a pair in the mesh is accomplished by the contraction operator discussed in the previous section. However, the internal state of the simplification class must also be updated. In particular, all candidate edges which are connected to either $\mathbf{v}_1$ or $\mathbf{v}_2$ must be updated.

```
void MxQSlim::apply_contraction(const MxPairContraction& conx)
{
    valid_verts--;
    valid_faces -= conx.dead_faces.length();

    // Accumulate the quadric of v2 into the quadric of v1
    quadrics(conx.v1) += quadrics(conx.v2);

    // Every edge currently linked to v2 must be relinked to v1.
    // Duplicate edges will be discarded and removed from the heap.
    update_pre_contract(conx);

    m->apply_contraction(conx);

    // Update v̄ and Q(v̄) for each edge connected to v1
    for(unsigned int i=0; i<edge_links(conx.v1).length(); i++)
        compute_edge_info(edge_links(conx.v1)[i]);
}
```