Model Simplification using Image and Geometry-Based Metrics

A Thesis
Presented to
The Academic Faculty

by

Peter Lindstrom

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
November 28, 2000

Model Simplification using Image and Geometry-Based Metrics

Approved:

Greg Turk, Chairman

Irfan Essa

Larry F. Hodges

Jarek Rossignac

Hugues Hoppe, Microsoft Research

Date Approved 11/28/00

Acknowledgements

The work presented here was funded in part by a Link Foundation fellowship and an NSF CAREER award (CCR–9703265). I would like to extend my sincere appreciation to these sponsors.

I am forever grateful for the guidance and support of my advisor, Greg Turk. During our years working together, Greg has been an outstanding advisor and a good friend, and I thank him for motivating and inspiring me to pursue good and relevant research. I also owe Greg for his many suggestions on how to improve my thesis, and for his part in making sure that it came together in a timely manner.

I would like to thank Jarek Rossignac for spending many long afternoons discussing research ideas with me, and for his insightful and often thought provoking comments and suggestions on my research. Thanks to Larry Hodges—an Elon alum like myself—for his part in helping me choose the right graduate school, and for his guidance during my early years as a graduate student. Irfan Essa brought valuable expertise in human vision to my committee, and I wish to thank both Irfan and Larry for their help with my thesis work. I am indebted to Hugues Hoppe for his perceptive comments on my research, and for his attention to technical detail. I also thank Hugues for generously volunteering to provide simplified models and results from his own simplification methods. These results are included in several of the comparisons made both in this thesis and in my other publications.

Without an assortment of polygonal models, there would have been nothing to test my algorithms on. I would particularly like to thank Marc Levoy and the Stanford graphics group for making several polygonal data sets available, including the bunny, Buddha, and dragon models. Marc also provided me with the Saint Matthew data set, which appears courtesy of the Digital Michelangelo Project. The horse model and several other data sets were obtained free of charge from Cyberware. The turbine blade model is an isosurface extracted from volume data provided by Kitware. I would also like to thank Michael Garland at UIUC and the researchers at UNC and CNUCE for making their simplification software and tools available.

I wish to thank my friends and colleagues at Georgia Tech, many of whom helped critique and improve my research. I would especially like to thank geometry group members Huong Quynh Dinh, Davis King, David Koller, F. S. Nooruddin, James O'Brien, Jack Tumblin, Ben Watson, and Gary Yngve. Thanks also to the GVU staff for always being around and helping out.

A special thanks to my parents for making this long journey possible. Their advice, support, and encouragement have been invaluable throughout the course of my education. Thanks to my sister Elisabet for always cheering me on and for reminding me to "have fun."

Finally, I thank Kerrie Hudzinski for her constant love and support, and for her patience and understanding during the sometimes difficult process of writing this thesis.

Contents

Summary

This thesis describes methods for simplifying complex polygonal surfaces and optimizing their approximations. Such densely sampled surfaces often arise from range scanning, isosurface extraction, and approximation of smooth curved surfaces. Because of their complexity, these models must often be reduced by orders of magnitude using simplification to allow interactive display, efficient storage and transmission, and faster processing.

In this thesis, I describe three different simplification algorithms, each with a different application domain. The first method, *memoryless simplification*, uses simple geometric heuristics based on changes in volume and area for ordering a sequence of edge collapses and optimizing the positions of the resulting vertices. It differs from most existing simplification methods in that the error metric is defined with respect to the partially simplified model, as opposed to the original. Because no information about the original model is retained, the algorithm is both memory efficient and fast, making it suitable for simplifying models with a few thousand to a few million triangles. Somewhat surprisingly, this method consistently performs better than a wide range of previous algorithms in the mean error sense, while being among the fastest.

In order to handle extremely large models that are too complex to fit in main memory, I propose a method for *out-of-core simplification*. This method is able to process an arbitrarily large mesh in a single pass, while building up an in-core representation of a greatly simplified version. The algorithm is very easy to implement, is able to process and simplify up to 100,000 triangles per second on conventional workstations, and produces simplified models that are of significantly higher quality than similar out-of-core methods. The improvement in quality is largely due to the use of *error quadrics*; a generalization of the error metric used in memoryless simplification.

For many computer graphics applications, preserving the *visual appearance* of the original model during simplification is the most important goal. Simplification methods have traditionally relied on using geometric deviation as a measure of similarity, and typically combine several different metrics for preserving surface attributes, such as normals, color, and texture, in an ad hoc manner. I present a method for *image-driven simplification* that uses rendered images of an object and an image metric to directly capture and preserve the visual appearance, which allows the effects of surface attributes, visibility, and rendering parameters and viewing conditions to be explicitly accounted for during simplification. This method produces models of higher visual quality than existing geometry-based algorithms.

Based on the image-driven method for simplification, I have developed a new method for *mesh optimization* which accepts an already simplified mesh and attempts to find a mesh with the same number of vertices that is more globally optimal with respect to an image metric. This method performs multidimensional simultaneous optimization of vertex positions and surface attributes, while also making local connectivity changes to the mesh where appropriate. This technique produces the most visually pleasing approximating surfaces

of the methods described here.

Finally, I describe a new image metric for image-driven simplification and optimization that is motivated by *human visual perception*. This metric draws upon previous work in vision and psychology, but has been designed mainly for use in polygonal simplification. By taking such domain-specific knowledge into account, several simplifying assumptions can be made, thereby greatly improving the speed of evaluating the metric without significantly compromising its perceptual accuracy. The metric successfully exploits the limitations in the human visual system, and is able to predict imperceptible differences where they occur, making it particularly suitable for measuring differences between very detailed textured polygonal models. This new metric has been integrated and used with my image-driven mesh simplification and optimization methods, occasionally allowing substantial improvements in model quality over more simple image metrics for certain types of objects.

Chapter 1

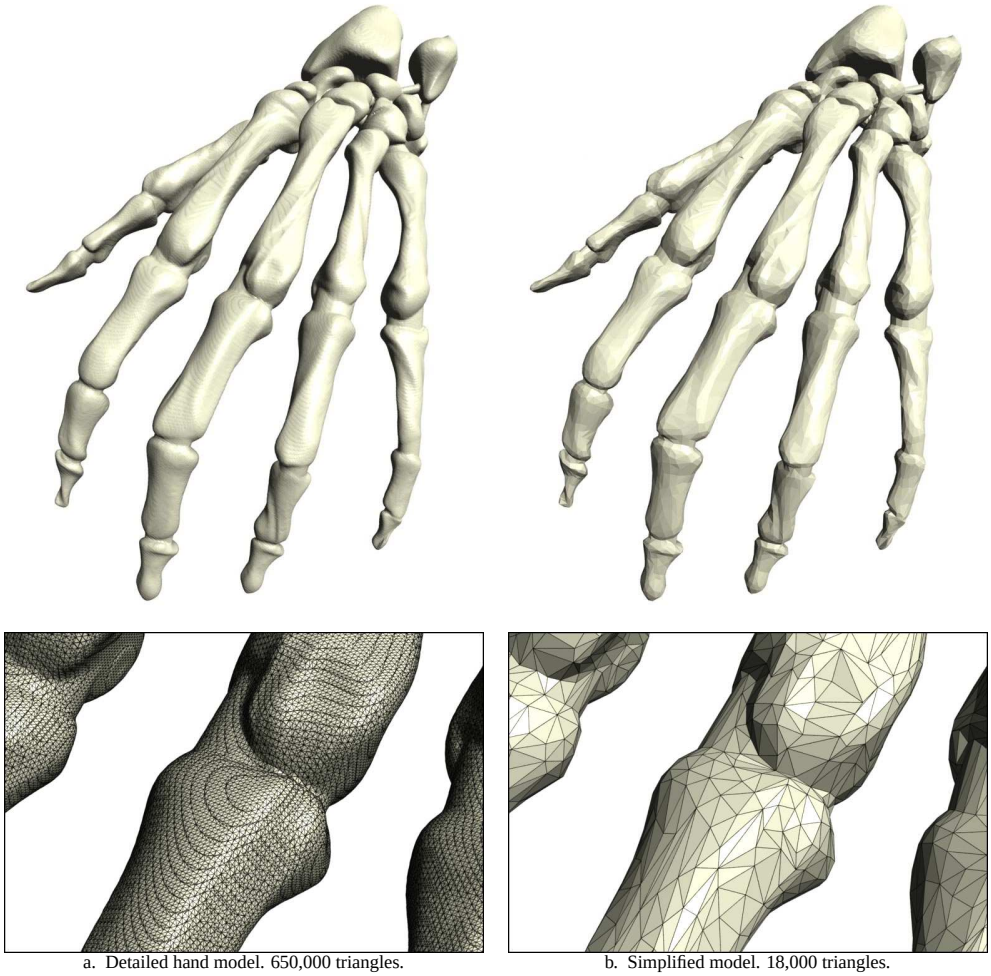
INTRODUCTION

1.1 Background and Motivation

Polygonal models have traditionally been used in computer graphics as the fundamental primitive for representing surfaces of three-dimensional objects. Such surface representations are used in a wide variety of applications, including visual simulation, architectural walk-throughs, medical visualization, computer aided design (CAD), and entertainment. In essence, polygon meshes constitute the lowest common denominator for representing surfaces, allowing objects of arbitrary geometry and topology to be represented at any desired accuracy, and are typically easy to construct from other surface representations. In particular, polygons and polygonal meshes are the main, and in some cases the only, primitives supported by conventional graphics hardware and software rendering systems. For these reasons, the majority of algorithms for manipulating surfaces have been designed for polygonal meshes, and virtually all contemporary methods for model creation, including range scanning and surface reconstruction from points, computer vision algorithms such as shape from shading, isosurface extraction from volumetric data such as CT scans and implicit functions, and modeling systems such as CAD, produce polygonal surfaces.

Recent advances in scanning technology and storage capacity have lead to an explosion in the availability and complexity of polygonal models, which often consist of thousands or even millions or billions of polygons [1, 90]. Another contributing factor to model complexity is the fact that polygonal surfaces are piece-wise linear in nature, and a relatively dense sampling of surface points is therefore needed to accurately represent non-linear, curved surfaces. Since the speed and memory requirements of most algorithms for polygonal meshes depend on the mesh complexity, i.e. the number of polygons, these massive data sets greatly exceed the capabilities of current display systems and mesh processing tools. In order to cope with large polygonal data sets, *model simplification* is often used, in which a highly detailed surface is approximated with a smaller number of polygons. While the simplified surface generally isn't an exact replica of the original model, a small loss in fidelity is often considered an acceptable trade-off for the elevated display rates, reduced storage requirements, and improved processing speeds afforded by less detailed models. Figure 1.1 illustrates how a densely sampled surface can be approximated using simplification to produce a visually similar model with many fewer polygons.

The use of simplified models to improve display rates was first proposed by Clark [25]. By creating multiple representations of different complexity for each polygonal object in the scene, an appropriate *level*



a. Detailed hand model. 650,000 triangles.

b. Simplified model. 18,000 triangles.

Figure 1.1: Comparison of a highly detailed model, produced by isosurface extraction, and a less detailed approximation, generated by the simplification method described in Chapter 4. The close-ups of the center knuckle show the fine and nearly uniform tessellation in the original mesh, and a triangulation in the simplified model that is better adapted to the geometry of the surface.

of detail can be chosen for each object, based on factors such as distance, relative motion, visibility, semantic importance, etc. [48]. Distant objects generally require less detail due to foreshortening, while a greater amount of detail is needed to accurately represent objects close to the viewer. Figure 1.2 illustrates the concept and use of multiple levels of detail. Since the publication of Clark's original paper a quarter century ago, a large number of methods have been developed for automatically constructing and managing levels of detail for use in interactive graphics applications, particularly in flight simulators [30, 162] and terrain visualization systems [36, 45, 93], and for visual simulation in general [43, 125].

a. 69,451 triangles. b. 14,535 triangles. c. 2,899 triangles.

Figure 1.2: Three levels of detail of the bunny model, shown at increasing distances. When viewed from a sufficiently large distance, the loss in detail in the simplified models is imperceptible. The three models shaded in a darker color are appropriate levels of detail for their corresponding distances.

Simplification is also used extensively in medical and scientific visualization—applications in which enormous amounts of 3D data are often generated. Medical imaging devices such as CT and MRI scanners are used to produce high resolution three-dimensional density fields. The most common method for visualizing such data is to extract isosurfaces at different density levels. However, such surfaces often consist of millions of polygons and cannot be displayed and interacted with in real-time. A more unconventional data acquisition approach was used in the recent *Visual Human* project [1], in which a human cadaver was frozen and cut into thousands of parallel slices. These slices were then imaged and assembled into a volumetric data set consisting of over 10 billion voxels, with isosurfaces of similar polygonal complexity. Such gigantic data sets are also becoming commonplace in scientific computing, where data sources include remote sensing for geology, oceanography, astronomy, etc., and large finite element simulations, such as computational fluid dynamics (CFD), weather modeling, and material stress analysis. Such simulations are run daily on supercomputers around the world, resulting in immense data sets that require simplification.

Other examples of data sources for polygonal models include modeling software for computer aided design and computer games. This traditional approach to modeling, where the user designs a model from scratch, is often complemented by "reverse engineering" existing 3D objects using scanning hardware. Surface representations of real objects can be obtained using laser range scanning [12, 90], mechanical touch probes, and computer vision techniques [15, 136]. Some of these techniques generate a number of point

samples of the surface, which must be triangulated to form a continuous surface [3, 10, 71]. Regardless of whether the model is acquired via scanning or synthesized using a modeling package, the resulting triangulated surface often needs to be simplified for further use.

Even though very powerful simplification algorithms are now available, level of detail creation for certain applications is still done manually. When polygons are at a premium, such as in computer games for low-end graphics systems, skilled artists, who make the best possible use of a very limited polygon budget, often do the modeling by hand [142]. If automatic simplification is used at all, it is used to quickly produce rough approximations that are then improved and further decimated manually. Probably the most difficult problem in simplification is to produce very coarse approximations that make the most out of a meager polygon budget. The majority of existing simplification algorithms tend to break beyond a certain point of simplification, where model quality degrades rapidly. To partially address this problem, promising new techniques have been developed that either implicitly or explicitly combine simplification with more costly rendering techniques, such as hybrid geometry- and image-based rendering [2, 35, 47], silhouette clipping [56, 132], and geometry enhancing texture-based approaches such as bump, normal, displacement, and environment mapping [27, 33, 59, 87]. An important goal in this thesis is to push the envelope in simplification quality and produce more pleasing models, without having to rely on specialized rendering techniques to make up for the loss in visual quality.

In recent years, simplification has seen uses in areas other than interactive graphics. In particular, many modeling algorithms and surface representations rely on the ability to create a sequence of *progressive meshes* [66] that allow a surface of any desired complexity to be produced on demand. Examples of applications include model compression [66, 78, 113], remeshing for subdivision [11, 37, 61, 84], and creation of multiresolution representations to perform signal processing on meshes [60, 88, 119]. The "MAPS" algorithm by Lee et al. [88], which constructs a one-to-one mapping between a detailed and a simplified mesh, has seen widespread use in a variety of applications, such as compression [78], mesh morphing [86], and remeshing/surface fitting [87]. Other examples of applications of simplification include graphics acceleration using "silhouette clipping" [132] and improved efficiency of global illumination algorithms [130, 157]. In summary, simplification has become an integral and ubiquitous tool in several graphics disciplines, and continued effort is being made to design faster and higher quality simplification methods for a very diverse set of applications.

1.2 Problem Statement

The ultimate goal of automatic simplification is to produce the *optimal* approximation of a given complexity, that is, the model with a prescribed number of polygons that minimizes the approximation error or loss in quality. For this goal to be meaningful, an *error metric* must be defined that allows the quality of a simplified model to be measured. It is important to recognize that there is no "Holy Grail" in simplification—no metric or simplification technique is the best possible for all problems. Rather, some applications require different metrics than others, and designing metrics that are appropriate for their problem domain and are

computationally easy to evaluate is an important part of simplification research.

In many graphics applications, *visual similarity* is an appropriate error measure. While preserving appearance has been an implicit goal in the work of others, most existing methods resort to simple heuristics based on geometric distances as an indirect indication of visual quality. In this thesis, I will address this problem in a more direct manner by using rendered images of models and image metrics to quantify and preserve visual similarity.

In addition to quality, another important consideration is the resource utilization of a simplification algorithm. Processor cycles and memory are finite resources, and—as in most areas of computing—there is generally a trade-off between resource utilization and solution quality. As the increase in complexity of polygonal meshes is currently outpacing the improvements in processor speed and available memory, resource utilization is becoming an increasingly important aspect of simplification algorithms. I will describe two methods for which computational and storage efficiency are paramount, and which deal with models of very high complexity.

While model simplification is a reasonably narrow research topic, this thesis covers a fairly broad scope of simplification techniques that have very different approaches and goals. Instead of solving any one single problem, each of these techniques has its own niche within a spectrum of problem domains and applications, ranging from very high quality, lower complexity meshes—for applications like video games and low-end graphics devices—to extremely large meshes—for applications such as scientific visualization and high resolution data acquisition. I will discuss each of these problems in more detail in the following sections.

1.3 Contributions

This thesis describes three different methods for performing simplification, as well as an optimization method for improving the quality of already simplified meshes. Below, I will provide a brief background of the problem that each of these algorithms attempts to solve and describe their individual contributions.

1.3.1 Out-of-Core Simplification

As scanning devices, computer simulations, and modeling packages have lead to increasingly larger models, users of polygonal models have recently been faced with the problem of dealing with meshes that are significantly larger than available main memory. Because of this memory shortage, conventional simplification methods, which typically require reading and storing the entire model in main memory, can not be used, and *out-of-core* approaches to simplification are needed. Not only do out-of-core methods have to be very memory efficient, but they must also be much faster than existing in-core methods in order to reduce models with millions to billions of triangles in a reasonable amount of time.

In [92], I described a method that simplifies arbitrarily large meshes in a single pass, while providing a fairly high level of geometric quality. This algorithm uses *vertex clustering* [127] as the method of coarsening the object, in conjunction with a *quadric error metric* [50] to optimize the geometry of the simplified surface. The simplicity of this algorithm makes it very fast, allowing models with a billion triangles to be simplified

in just a few hours. To my knowledge, this algorithm is the first published method for doing fast, high-quality out-of-core simplification of arbitrarily large models. It is my hope that it will bring an important and largely unsolved problem to people's attention and inspire future research in this area. This method and an extended analysis of it are presented in Chapter 5.

1.3.2 Memoryless Simplification

Efficient resource utilization is important even for in-core methods. Quite simply, the more memory efficient a method is, the larger the models it can simplify. The majority of published methods measure errors with respect to the original model, and must either keep a copy of this detailed model in memory or maintain a history of operations that allow such errors to be estimated. The memory requirements of these methods are often dominated by the cost of storing this additional information, which limits the size of models that can be simplified in-core.

In [94], I proposed a method for doing *memoryless simplification*. Contrary to previous methods, this technique does not require any information about the original model. Instead, the approximation error is measured as the amount of change incurred in each iteration between pairs of successive meshes. Similar to most contemporary simplification methods, the memoryless method uses the *edge collapse* operation to iteratively coarsen the mesh. The memoryless error metric, which is based on local changes in volume over the surface and changes in area near surface boundaries, is used to order the sequence of such edge collapses, and the new mesh vertices introduced in each iteration are chosen such that the error is locally minimized. The memoryless algorithm is described in depth in Chapter 4.

1.3.3 Image-Driven Simplification

Most existing polygonal simplification methods attempt to exploit redundancy in the surface representation, i.e. by reducing the complexity in regions that are relatively flat (where the surface is approximately linear), and spending relatively more polygons in regions that are more curved and detailed. In fact, nearly all previous algorithms for simplification are driven by the goal of minimizing some measure of geometric deviation between the two surfaces. Some impose a maximum tolerance on the deviation and simplify the surface as long as the tolerance is met [28, 57, 81]. Others are designed to reduce the average distance between the two surfaces [50, 72, 82]. Such geometric measures of similarity are important for applications that require that the two meshes are geometrically close, e.g. for tasks such as path planning, machining, and accurate collision detection. A more common goal, especially in most computer graphics applications, is that the two models *appear* similar when rendered. While the amount of geometric error indirectly influences similarity in appearance, it is but one of several contributing factors. Surface attributes such as texture, color, and normals also play important roles in model appearance, as do silhouettes, surface visibility (e.g. self-occlusion and transparency), choice of rendering style (e.g. lighting and shading models), and interpolation techniques for geometry and surface attributes (e.g. subdivision and normal interpolation). To achieve a higher degree of visual quality, an entirely different type of metric is needed, that better accounts for the

properties of a model that influence its visual appearance.

In this thesis, I propose the use of *image metrics* and a collection of rendered images of an original and a simplified object, taken from a sphere of camera positions surrounding them, to measure their visual similarity. This is a more direct approach to measuring visual similarity, and avoids the problem of having different metrics for geometric, parametric, and attribute error, and having to combine these error terms into a single metric. Instead, image metrics provide a unified measure of similarity, and directly account for the complex interactions between these components. Based on this approach, I have developed an *image-driven simplification* algorithm that is able to better preserve the visual appearance of the original model. This algorithm, which also is based on edge collapse, was first published in [97], and is described in Chapter 7.

A main benefit of the approach described here is that it allows for any image metric to be plugged into the simplification method. As we learn more about human visual perception, computational models can be built to better estimate how we perceive differences between rendered images of objects. Since the human eye is not a perfect imaging device, such perceptual metrics can exploit the limitations of human vision and guide the simplification to preserve more detail in regions that are perceptually important. In Chapter 6, I will present a new image metric that is inspired by previous work [103] in the computer vision field and is fast to evaluate, making it suitable for use in simplification.

1.3.4 Image-Driven Mesh Optimization

My image-driven simplification algorithm often produces simplified models of high visual quality. However, like virtually all previous simplification algorithms, it is a *greedy* method in that it always chooses the "cheapest" edge to collapse. This method also places new vertices in a manner that locally minimizes the error for any single edge collapse. Such an approach, however, is not guaranteed to lead to a *globally optimal* mesh, for which the positions of the vertices and the connectivity of the mesh are collectively in the best possible configuration. Not only that; my image-driven simplification method uses a geometry-based heuristic for placing new vertices and computing new surface attributes. Such an approach, based on geometric error, is not likely to yield optimal results with respect to any given image metric.

In order to further improve the appearance of a mesh, a global optimization method is needed that alters the geometry, surface attributes, and connectivity of the mesh by minimizing the error measured by the image metric. Inspired by the mesh optimization work by Hoppe et al. [72] and drawing upon my own work on image-driven simplification, I have developed a new optimization technique that accepts an already simplified mesh and makes repeated small modifications to it to improve its visual appearance, while keeping its number of vertices fixed [96]. This *image-driven mesh optimization* method uses the *downhill simplex method* to optimize the geometry, wrapped into an outer combinatorial optimization of the mesh connectivity. The image metric is able to produce difference images that are the basis for an *oracle*. This oracle suggests what parts of the mesh to devote more effort to in order to most efficiently improve the quality of the mesh. Used in conjunction with any simplification method, this optimization technique can be added as a postprocessing step that rapidly improves the visual quality of the simplified mesh. This method is described in Chapter 8.

1.3.5 Summary of Contributions

The algorithms presented here perform differently in terms of quality and speed. Because of the inherent trade-off between computational efficiency and simplification quality, it is important to recognize that each of these methods is best suited to a particular problem domain. Listed in order of increasing quality and decreasing speed and model size, the algorithms presented in this thesis and the major contributions of this work as a whole are:

- An out-of-core simplification method that accepts arbitrarily large models and simplifies them in a single pass. This method is fast, memory efficient, and produces models of high geometric quality.
- An in-core simplification method that requires no information about the original model to be retained during simplification. This "memoryless" approach to simplification has proven to result in models of higher quality than those produced by methods that are less memory and compute efficient.
- A simplification algorithm that uses rendered images and an image metric to better preserve the *visual appearance* of a model. This entirely new approach to simplification provides a unified metric for measuring visual similarity, and often leads to models of substantially higher visual quality than geometry-based methods.
- A technique based on image metrics for improving the appearance of simplified models by performing global optimization of mesh connectivity and geometry.
- A new perceptually motivated image metric that is computationally simple to evaluate and is appropriate for use in simplification and optimization.
- Empirical evaluations and comparisons between my new algorithms and several well-known simplification methods.

Collectively, the techniques presented in this thesis complement each other well and span a spectrum of applications and uses, from fast simplification of very large models, to higher quality simplification of very low polygon count models. In subsequent chapters, I will make several comparisons of the methods, contrast them, and suggest how they can sometimes be used together to make the best possible use of them.

1.4 Overview

The thesis is roughly divided into one chapter for each algorithm, with an additional chapter related to image metrics to cover the overlap between the image-driven simplification and optimization algorithms. Before describing the algorithms, I will cover the notation and terminology used in the remainder of the thesis in Chapter 2. In Chapter 3, I will discuss previous research related to simplification, level of detail, optimization, and metrics. Because my out-of-core and image-driven simplification algorithms both make use of my memoryless simplification method, I will discuss this latter method first in Chapter 4. This chapter is followed by

a description of the out-of-core algorithm (Chapter 5). The remainder of the thesis describes algorithms that use image metrics to simplify and optimize polygonal models. Chapter 6 is intended as an introduction to using image metrics for comparing and simplifying models. §6.2, in particular, describes efficient algorithms and data structures for partially updating images and then quickly re-evaluating the metrics. §6.3 ends the chapter with a description of a new perceptually-based image metric. The next chapter, Chapter 7, describes the remaining pieces of the image-driven simplification algorithm. Chapter 8 extends this algorithm to global optimization of meshes. Chapter 9 concludes the thesis and discusses future work.

Chapter 2

NOTATION

In order to most effectively communicate the ideas presented in subsequent chapters, I will make frequent use of mathematical notation and terminology. In this chapter, I will briefly cover some of the most common symbols, terms, and conventions used in this thesis. I will assume that the reader is familiar with introductory set theory, linear algebra, calculus, and geometric modeling, and that, unless explicitly stated, the meanings and definitions of the terms used here are commonly accepted and available elsewhere.

2.1 Algebraic Topology and Sets

An n-simplex is a topological entity defined by $n+1$ vertices, e.g. a 0-simplex is a vertex, a 1-simplex is an edge, a 2-simplex is a triangle, etc. The *faces* of an n-simplex are the $(n-1)$-simplices that bound it, e.g. the edges of a triangle are its faces, while the faces of an edge are its two vertices. I will use small letters to denote individual simplices and capital letters for sets of simplices and other elements. The following operators will be used frequently in the text to refer to the faces and simplices surrounding a simplex:

$\lvert S\rvert$	cardinality (number of elements) of S
$\neg S$	complement of S
$S \cup T$	union
$S \cap T$	intersection
$S \setminus T = S \cap \neg T$	set difference
$\lceil s \rceil$	set of $(n+1)$-simplices that the n-simplex s is a face of
$\lfloor s \rfloor$	set of $(n-1)$-simplices that are faces of the n-simplex s
$\lceil S \rceil = \bigcup_{s \in S} \lceil s \rceil$	generalization of $\lceil \cdot \rceil$ to sets ($\lfloor S \rfloor$ is defined similarly)
∂S	boundary edges of S

See Figure 2.1 for graphical illustrations of these operators.

2.2 Linear Algebra and Vector Calculus

Scalars are written in lower-case italics, e.g. c, column vectors are written in lower-case boldface, e.g. $\mathbf{b} = \left[\begin{smallmatrix} b_1 \\ b_2 \end{smallmatrix}\right]$, and matrices are written in capital boldface, e.g. $\mathbf{A} = \left[\begin{smallmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{smallmatrix}\right]$. $\mathbf{I}$ denotes the identity matrix.

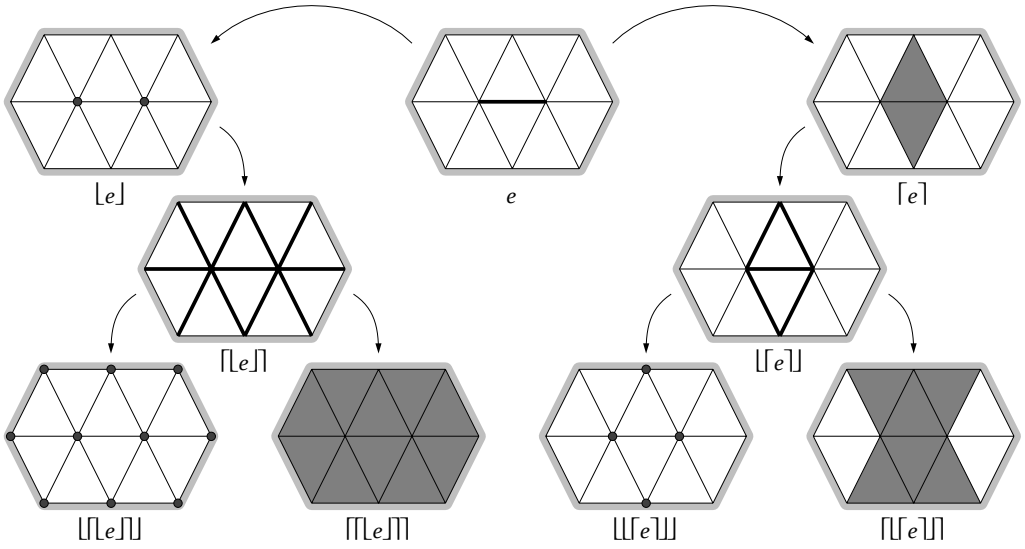


Figure 2.1: The simplex operators $\lfloor s \rfloor$ and $\lceil s \rceil$.

Row vectors are written as transposed column vectors.

$\delta_{ij} = \left\{ \begin{smallmatrix} 1 \text{ if } i = j \\ 0 \text{ otherwise} \end{smallmatrix} \right.$	Kronecker delta
∇f	gradient
$\|\mathbf{a}\|_p = \left(\sum_i \|a_i\|^p\right)^{1/p}$	L_p norm
$\|\mathbf{a}\|_2 = \|\mathbf{a}\|$	vector magnitude (L_2/Euclidean norm)
$\|\mathbf{a}\|_\infty = \max_i\{\|a_i\|\}$	L_∞ norm
$\mathbf{A}^\mathsf{T}$	transpose
$\|\mathbf{A}\| = \det \mathbf{A}$	determinant
$[\mathbf{a}, \mathbf{b}, \mathbf{c}] = \det [\mathbf{a} \quad \mathbf{b} \quad \mathbf{c}]$	scalar triple product
$\mathbf{a}^\mathsf{T}\mathbf{b}$	inner/scalar product (a scalar)
$\mathbf{a} \times \mathbf{b}$	cross/vector product (a 3-vector)
$\mathbf{a}\mathbf{b}^\mathsf{T}$	outer/matrix product (a matrix)
$\mathbf{a} \wedge \mathbf{b}$	exterior/wedge product (a multivector)
$[\mathbf{a}\times] = -[\mathbf{a}\times]^\mathsf{T} = \left[\begin{smallmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{smallmatrix}\right]$	cross product as linear operator; $[\mathbf{a}\times]\mathbf{b} = \mathbf{a} \times \mathbf{b}$

I will make frequent use of block matrix notation, e.g.

$$\begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{c}^\mathsf{T} & d \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ c_1 & c_2 & d \end{bmatrix}$$

denotes a 3×3-matrix constructed by concatenating a 2×2-matrix $\mathbf{A}$, two 2-vectors $\mathbf{b}$ and $\mathbf{c}$, and a scalar d.

As is common in computer graphics, I will use homogeneous coordinates as a shorthand notation for affine transformations. Given a vector $\mathbf{x}$, I will write $\bar{\mathbf{x}}^\mathsf{T} = \begin{bmatrix} \mathbf{x}^\mathsf{T} & 1 \end{bmatrix}$ to denote a homogeneous vector in one

higher dimension. This notation allows affine transformations $\mathbf{A}\mathbf{x} + \mathbf{b}$ to be written as as $[\mathbf{A} \quad \mathbf{b}]\,\bar{\mathbf{x}}$.

Some authors make a point of distinguishing between vectors and points. While this distinction has its merits, I have chosen not to separate the two concepts as one can trivially convert between these entities. Instead, I will treat all vertices as vectors, which allows operations on vertices to be written in a more straightforward manner using conventional linear algebra notation.

2.3 Triangle Meshes

A *triangle mesh* is piece-wise linear surface made up of a set of interconnected triangles (Figure 2.2).[1] The triangle mesh is the most general representation of a polygonal surface since all polygons can be broken down into triangles [112]. Therefore, I will refer only to triangle meshes from this point on. The triangle mesh can be separated into its *connectivity*—the incidence relationships between the simplices in the mesh—and its *geometry*, which is specified by the positions of its vertices in $\mathbb{R}^3$. The connectivity information allows one to traverse the mesh from one triangle to another and query which simplices belong to and are adjacent to others. For a discussion of efficient mesh representations, see [19].

An edge is said to be *manifold* if it has exactly two incident triangles, a *boundary* edge has one incident triangle, while an edge with three or more incident triangles is *non-manifold*. A vertex is similarly classified as boundary, manifold, or non-manifold depending on the classification of its incident edges, with a couple of exceptions. First, a vertex v is non-manifold if either any of its incident edges $\lceil v \rceil$ are non-manifold, or if $\lfloor \lceil \lceil v \rceil \rceil \rfloor \setminus \lceil v \rceil$ does not form a single connected loop of edges. A vertex is a boundary vertex if it is not non-manifold and it has two or more incident boundary edges. Otherwise, the vertex is manifold, or *interior*. A triangle mesh is said to be manifold if all of its vertices and edges are manifold, or *manifold with boundary* if it has some boundary edges but no non-manifold simplices.

In the mathematical field of topology, an intuitively similar definition is used to classify general surfaces; a point on the surface is said to be manifold if there is a small neighborhood centered around the point that is homeomorphic to a disk, while a boundary point has a neighborhood around it that is homeomorphic to a half-disk. Note that this definition of manifold depends on the geometry of the surface. Many polygonal surfaces are not true *embeddings*, that is, they interpenetrate themselves in one or more places. However, when discussing the manifold quality of triangle meshes, I will refer only to the connectivity aspect of being manifold.

The *orientation* of a triangle is determined by the order in which its vertices are permuted. Permutations of the same parity refer to the same triangle, e.g. $(i, j, k) \Longleftrightarrow (j, k, i)$ but (i, j, k) and (i, k, j) are triangles of opposite orientation. A manifold mesh (with boundary) is said to be *orientable* if there exists a configuration of orientations for its triangles such that, for each edge (i, j), there is no pair of triangles $\{(i, j, k), (i, j, l)\}$. That is, if (i, j) is part of a triangle, then the opposite edge (j, i) must be part of the neighboring triangle. The *Möbius strip* is an example of a non-orientable surface.

[1] Some authors allow isolated edges and vertices to be part of the mesh. I will not consider such mixed dimension structures, but assume that *regularization* is used to remove dangling edges and vertices from the mesh.

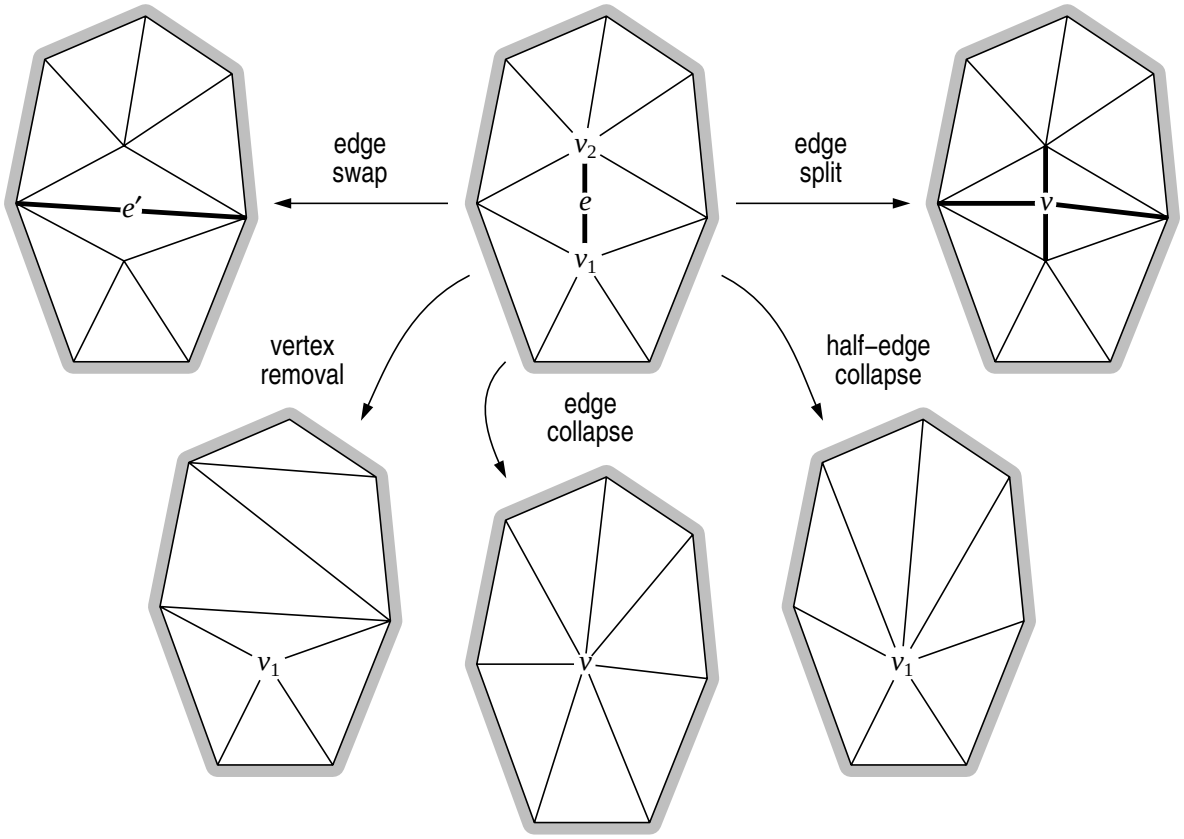


Figure 2.2: Local connectivity operations.

In set theory and topology, the term *closed* is used to characterize a set that contains its boundary. In computer graphics and many engineering disciplines, a *closed surface* typically refers to a surface that is "watertight," i.e. it contains and bounds a piece of solid space. I will use this latter definition of "closed" in a topological sense by ignoring the geometry of the surface. Thus, a closed surface is manifold without boundary.

There are many possible representations of triangular surfaces. These can be divided into shared vertex representations, wherein vertices common to a set of triangles are specified as unique entities (typically via an indexing scheme), and *triangle soups*. A triangle soup provides no incidence information. Instead, each triangle is an independent triple of vertices that belong only to that triangle. The term "mesh" is usually used to refer to a shared vertex representation, which is also the terminology used here.

Most simplification methods make use of local connectivity operations to reduce the complexity of the mesh (see Figure 2.2). One such operation is *vertex removal*. In this operation, a vertex and its surrounding edges and triangles are removed, and the resulting hole is then triangulated. Note that there is no unique way to triangulate such a hole, which leaves a great amount of freedom in choosing how to best preserve the surface shape. There is a particular instance of vertex removal, called *half-edge collapse*, for which the

resulting triangulation is unique, however. In this operation, an edge $e = (v_1, v_2)$ is collapsed to one of its vertices (say v_1), effectively removing its other vertex. See Figure 2.2 for a comparison between the vertex removal and half-edge collapse operations. The general *edge collapse* operation is similar to half-edge collapse, in that it contracts an edge e into a single vertex v, but the location of v is not constrained to be one of the two endpoints of e.

In Chapter 8, I will make use of two additional connectivity operations: *edge swap* and *edge split* (Figure 2.2). The edge swap operation is applicable to manifold edges, and can be used to make connectivity changes to a mesh without removing or adding vertices. In the edge split operation, a vertex is inserted at the midpoint of the edge, thereby bisecting it and its incident triangles.

Chapter 3

PREVIOUS WORK

Automatic model simplification has been a very active field of research in the last decade, with dozens of published techniques. Due to this wealth of simplification methods, the review here is necessarily incomplete, but I will attempt to cover the most notable publications in the area. In addition to simplification, there are a number of related fields relevant to the work in this thesis, including level of detail management, optimization, geometry-based metrics, and image metrics. As part of previous work on image metrics, I will discuss computational models of human visual perception, which are the foundation for several perceptually motivated image metrics.

3.1 Simplification

The concept of using multiple levels of detail for 3D models was first proposed by Clark [25] as a means of improving the interactivity of graphics applications. By using coarser levels of detail to render distant objects, the total complexity of a scene can be decreased without appreciable degradation in display quality (Figure 1.2). Subsequent to Clark's seminal paper, level of detail management and surface approximation algorithms were developed for restricted classes of surfaces with simple topologies, such as triangulated *height fields* of the form $z(x, y)$ and parametric surfaces. Such techniques were widely adopted in the flight simulation industry, where expansive terrains at high resolution put a limit on achievable display rates [30, 45, 162].

Simplification algorithms for more general surfaces did not appear until the early nineties. The increasing availability of complex free-form surfaces has been a prime motivator for such algorithms, and until recently few automated techniques were available for acquiring and synthesizing such surfaces. The *marching cubes* algorithm by Lorensen and Cline [99] made it possible to extract dense isosurfaces of volumetric data, while range scanning hardware allowed surfaces of arbitrary topology to be acquired and reconstructed from real objects [12, 31, 71, 90, 148]. With graphical desktop computers becoming more commonplace, computer aided design (CAD) established itself as yet another common source of large polygonal data sets. These advances in model generation allowed very complex polygonal data sets to be created with very little effort. Because of limited processing and rendering speed, these data sets were, and still are, challenging to handle, which has lead to a demand for automated techniques to reduce their complexity. Some of the earliest publications on simplification of free-form polygonal surfaces include the work of Schroeder et al. [135],

Turk [146], Hoppe et al. [72], and Rossignac and Borrel [127]. The techniques used in these algorithms are today the cornerstones in the field, and most existing simplification algorithms are variations on these methods.

3.1.1 Simplification of Geometry

In this section, I will classify a large number of algorithms for doing simplification of triangle meshes. I will focus only on the problem of geometric mesh simplification as an off-line process. Handling meshes with surface properties and doing on-line view-dependent simplification/refinement will be discussed in later sections. The following taxonomy is based on the type of *coarsening operation* used, i.e. the manner in which the surface is modified to reduce its complexity.

3.1.1.1 Vertex Clustering

The original *vertex clustering* approach to simplification, proposed by Rossignac and Borrel [127], is essentially a form of lossy compression using uniform quantization. The main idea is to quantize the vertex coordinates of the input model to integers, whose precision is user-specified. Geometrically this corresponds to building a uniform grid of rectilinear cells, and then merging all vertices within a cell, or *cluster*, to a representative vertex for the cell. All triangles and edges that are completely contained within a cell are merged to a single point and can be eliminated.[1] Thus, a larger grid spacing leads to a coarser simplified model.

Rather than using exact integer positions for the representative vertices, the geometry is improved by optimizing the position of each cluster's vertex. In [127], simple heuristics were used to rate the "visual importance" of each vertex, which was used as a basis for selecting a representative among the vertices within a cluster. Subsequently, Low and Tan [102] proposed a slight variation on this heuristic motivated by a more thorough geometric reasoning. The main contribution of their technique, however, was the use of "floating cells." Rather than using a static grid, these cells are constructed dynamically by picking the most important vertex in the entire data set as the center of a new cell. All vertices that fall in the cell are merged, and a new floating cell is constructed from the next most important remaining vertex.

The vertex clustering technique is attractive in its simplicity and is useful for quickly constructing hierarchies of simplified models. By recursively merging clusters, thus creating "superclusters" of increasing spatial extent, a sequence of fine to coarse meshes can be organized in a tree-like manner, allowing selective refinement for view-dependent simplification [104, 133]. I will further discuss these techniques in §3.1.4.

My out-of-core simplification technique, described in [92] and later in Chapter 5, is based on vertex clustering. A key improvement in this algorithm over [102, 127] is the elimination of the vertex grading phase. Instead, geometric information is recorded in the non-empty cells during a single traversal of the input mesh. This avoids the need of storing the vertex grades and maintaining the connectivity needed to compute them.

[1] In Rossignac and Borrel's original scheme, dangling edges and vertices are not discarded but are maintained in specialized data structures. Appropriate geometric representations can then be used to render such low-dimensional primitives.

3.1.1.2 Face Clustering

A less popular approach to simplification can be thought of as the dual of vertex clustering. This technique is known as *face clustering*. The idea behind this approach is to merge nearly coplanar faces into large clusters of faces. Kalvin and Taylor [75] refer to such clusters as "superfaces." In their algorithm, and also in [65], the mesh is first partitioned into clusters. The interior vertices in each cluster are removed, and the cluster boundaries are simplified. In a final phase, the resulting non-planar superclusters are triangulated, resulting in a simplified model.

Eck et al. [37] use a similar face clustering technique to convert a mesh with irregular connectivity to a coarse base mesh. The base mesh is used to *remesh* the original detailed surface, i.e. to convert it to a mesh of similar complexity that has *subdivision connectivity*. While their method is not explicitly designed to perform model simplification, it produces a multiresolution/wavelet representation for the mesh that lends itself to fine selection of levels of detail [143].

A more recent face clustering algorithm is described by Garland [49] (see also [157]). The goal of this algorithm is to create multiple levels of detail for radiositized models by simplifying them in regions of near constant color. This problem is cast as simplification of the dual graph of a mesh using Garland's *quadric error metric* [50]. I will discuss this metric in detail below.

Models produced by face clustering generally exhibit relatively poor geometric and visual quality. This is mainly because the geometry of the simplified mesh is rarely optimized, and choosing a good triangulation for the patches can be hard. Additionally, it is difficult to create clusters that collectively provide an optimal partitioning of the mesh so as to capture its most important features.

3.1.1.3 Vertex Removal

Perhaps one of the most intuitive approaches to simplification is to iteratively remove pieces of geometry. One such method is based on "plucking" vertices; each time a vertex and its incident triangles are removed, a hole is created, which must then be patched via triangulation (see Figure 2.2). The vertex removal method for arbitrary meshes was first introduced by Schroeder et al. [135]. Their algorithm makes repeated passes over the mesh and selects vertices for removal. By fitting a plane to the vertices surrounding the vertex being considered for removal, the decision whether to remove it is based on the distance between the vertex and the plane, which serves as an error metric. Since removing a vertex in a high curvature region introduces a large error, vertices in flatter regions are preferred candidates for removal. As we shall see later, most simplification methods are implicitly or explicitly driven by preserving high curvature regions, and it is widely believed that the triangulation density should be determined by the curvature of the surface [16, 49, 63, 82, 140, 146].

Because the vertex removal algorithm is orthogonal to the choice of metric, several other simplification algorithms that make use of it have been published. Cohen and co-workers use vertex removal as the coarsening operation in their "Simplification Envelopes" algorithm [28]. In this method, inner and outer offset surfaces are created that constrain the geometry of the simplified surface. Vertices are removed randomly as long as the simplified surface lies within the envelopes, which ensures that the simplified model is within a

specified deviation from the original. Klein et al. [81] similarly guarantee maximum error bounds and coarsen the model using vertex removal.

Vertex removal algorithms also differ in how the resulting holes are triangulated. Techniques for improving the triangulation have been described in [6, 23, 124]. In [6, 23], folds in the mesh are prevented by attempting edge swaps (Figure 2.2) near the removed vertex. The method by Ciampalini et al. [23] will be further discussed below and in Chapter 4.

3.1.1.4 Edge Collapse

The *edge collapse* operation (Figure 2.2), first proposed by Hoppe and co-workers [72], has become one of the most popular coarsening operation in recent years. In this operation, the two vertices of en edge are contracted to a single vertex, thereby eliminating the collapsed edge and its incident triangles. Similar to vertex removal, but contrary to vertex clustering, edge collapse is a fine-grained operation that removes a single vertex. The advantage over vertex removal, however, is that the position of the substitute vertex can be chosen freely, and can be optimized to minimize the associated error metric. In addition, the resulting connectivity is uniquely defined, which eliminates the need to perform a costly triangulation step in each iteration.

The general edge collapse algorithm involves two decisions: (1) where to place the substitute vertex resulting from a collapse operation, and (2) choosing the order of edges to collapse. In general, both of these decisions are made implicitly by specifying an error metric that depends on the position of the substitute vertex. The metric then provides the order of edge collapses, from cheapest to costliest, and the position of the replacement vertex is chosen so as to minimize the metric. It is often the case, however, that other factors are involved in positioning vertices and ordering edges. Examples include topological constraints (e.g. to preserve manifoldness or to limit valence), geometric constraints (e.g. to preserve volume or to avoid folds and triangles with poor aspect ratios), handling of degenerate cases (e.g. where the surface is locally flat), and so on. These issues are secondary, however, and I will limit my discussion here to edge collapse metrics.

Many edge collapse metrics are defined based on some aggregate measure of error (as opposed to maximum error). This is the case in Hoppe's original work [72], as well as in [66], in which the mean square error is used to define a quadratic energy functional. This metric is defined by sampling a large number of points on the surface of the original model, and then measuring the closest (squared) distance of each point to the triangles in the simplified model. A conjugate gradient solver [54] is used to optimize the positions of the vertices in a small neighborhood around the collapsed edge.

Perhaps the key contribution of Hoppe's work is the notion of *progressive meshes* [66]. The idea behind this concept is to represent a detailed mesh M^n as a coarse base mesh M^0 plus a sequence of refinement operations, called *vertex splits*. This information allows the original sequence of edge collapses used to construct the base mesh M^0 to be reversed, making it possible to recover the detailed mesh M^n. Using this representation, a continuum of meshes $M^0 \rightarrow M^1 \rightarrow \cdots \rightarrow M^n$ of increasing detail can be stored with little overhead, allowing fine-grained selection of mesh complexity. Of course, the progressive mesh representation can be used with any method based on edge collapse and related operations.

Ronfard and Rossignac describe another edge collapse method [126] that maintains a list of supporting planes with each vertex. Initially, each vertex is assigned the planes associated with its incident triangles. As two vertices are merged into one by an edge collapse, the new vertex inherits the planes of the merged vertices, and the maximum distance from the new vertex to its supporting planes is used to measure the cost of collapsing the edge. Garland and Heckbert [50] use a slight variation on this technique by measuring the sum of *squared* distances to the supporting planes, and minimize this aggregate *quadric error metric*. While at first a seemingly trivial difference, this approach allows them to to capture all of the plane equations of the supporting planes in a single 4×4 symmetric matrix, rather than having to maintain a long list of planes with each vertex. Whenever an edge is collapsed, the new vertex inherits the sum of quadric matrices of the edge's vertices. This elegant technique has since been adopted and modified by others [41, 69, 94], and I will further analyze the differences between some of these methods in §4.4.8.

It is also possible to bound the *maximum error* between the two surfaces using edge collapse. Guéziec maintains spherical error volumes around the vertices of the partially simplified model. By interpolating the radii of these spheres and taking the union, an error volume for the entire mesh can be constructed, and Guéziec ensures that the simplified surface is contained in this volume. His method, as the method presented in Chapter 4, preserves the volume bounded by the surface in each iteration and minimizes the sum of squared distances of the replacement vertex to the planes of the nearby triangles. Guéziec's method differs from my memoryless method (Chapter 4) in several respects, however. First, his vertex placement scheme uses a uniform weighting of distances, as opposed to mine which uses area-squared weights. Second, his method does not address the degenerate case of planar geometry. Third, and most important, his ordering of edge collapses does not correspond well with his error measure for placing vertices, nor is it related to the maximum deviation between the two surfaces. Instead, he orders the edges by their length, which is a relatively poor heuristic for models other than very dense and nearly uniformly tessellated surfaces. The maximum error bounds provided by this method are also overly conservative as they don't distinguish between geometric and parametric error.

The method by Cohen et al. [26] has a number of interesting characteristics. Similar to Guéziec's method, it provides upper bounds on the maximum error, but uses axis-aligned boxes as the error volume. Their method additionally computes a bijective mapping between the two consecutive meshes in each iteration of edge collapse, allowing errors in the texture parameterization to be tracked. In constructing this mapping, they flatten the neighborhood around an edge in a manner such that the mesh does not fold onto itself. They then choose a replacement vertex within the *kernel* of the polygon associated with the flattened neighborhood using linear programming, and optimize the "vertical" position of the vertex such that the maximum deviation between corresponding points is minimized.

All of the edge collapse methods mentioned above use metrics that measure some form of geometric error. In [97], I proposed a new method that uses *image metrics* to order the sequence of edge collapses to better preserve the visual appearance of a model. This method is described in Chapter 7. I will return to the issue of metrics for simplification and quality evaluation in §3.3.

Not all applications require the generality of the edge collapse, and more specialized versions of it have

been proposed. A more restricted operation is obtained by limiting the position of the replacement vertex to one of the two endpoints of the edge. As a special case of edge collapse, this operation, commonly referred to as *half-edge collapse*, is also a particular instance of vertex removal for which the triangulation is pre-determined. This operation gets its name from the fact that individual *half-edges*, which are simply the two directed halves of an edge, are considered for contraction independently. Half-edge collapse generally results in lower quality meshes than regular edge collapse since it allows no freedom in optimizing the mesh geometry, but has the advantage of having a more concise representation, which among other things is useful for progressive compression. This property is also advantageous for doing view-dependent dynamic level of detail management, as it allows the geometry to be fixed up-front so that it can be transferred to and cached on the graphics card efficiently (see, for example, the *vertex array* primitive in OpenGL [159]). The half-edge collapse has been used in several simplification methods, including [82, 88, 126, 134]. I will compare the performance of half-edge collapse with the more general edge collapse operation in §4.8.3.

Triangle collapse is yet another possible coarsening operation, which was used by Hamann [63] and Gieng et al. [52] for simplification. In this operation, the three vertices of a triangle are merged to a single new vertex. Since this operation is equivalent to performing two consecutive edge collapses, and, like edge collapse, also requires optimization of the substitute vertex, it provides little practical benefit over the more general edge collapse.

3.1.1.5 Vertex Pair Contraction

Vertex pair contraction is an even more general operation than edge collapse in that it allows *any* pair of vertices to be merged, whether they share an edge or not [41, 50, 116]. By allowing topologically disjoint vertices to be collapsed, it is possible to merge disconnected components of a model. While this operation generally introduces non-manifold simplices, merging disjoint pieces of a model may be necessary to achieve drastic simplification rates if the number of separate components is large. Note that any closed manifold connected component must necessarily consist of at least four triangles. By closing small holes in the model and merging components, further simplification is possible.

In order to limit the number of possible candidates for pair contraction, only a subset of pairs, called *virtual edges*, that are spatially close are considered. The actual choice of virtual edges is usually orthogonal to the simplification method itself, and different strategies exist, including Delaunay edges [116] and static thresholding [50]. The work of Erikson and Manocha [41] is most notable for its dynamic selection of virtual edges, which allows increasingly larger gaps between pieces of a model to be merged.

As Garland and Heckbert [50] note, the vertex pair contraction can be generalized to a single atomic merge of any number of vertices. In fact, that is exactly what vertex clustering is, and the vertex pair contraction operator can be seen as the most primitive generalization of both edge collapse and vertex clustering. Since most edge collapse metrics can easily be generalized to pair contraction, I will consider this extension as optional. For simplicity, I will focus on the edge collapse operation and how it is used in different simplification methods.

3.1.1.6 Alternative Methods

In addition to the techniques described above, there have been a few published algorithms that do not neatly fit into these categories. Turk [146] described one of the earliest methods for simplification. His algorithm is based on allocating more vertices to high curvature areas of the surface. This is accomplished by inserting new vertices into the existing mesh that will eventually represent the simplified mesh. These vertices are then allowed to "slide" on the original surface, and are distributed by simulating point-wise repulsion forces that are inversely proportional to the local curvature of the original mesh, thereby ensuring that high curvature regions are sampled more densely. After the system reaches equilibrium, the original vertices are removed, and the resulting holes are triangulated.

Brodsky and Watson [16] present a *refinement* algorithm based on *vector quantization*. Refinement is the opposite of simplification; it begins with a coarse model and successively introduces detail until a model with a desired complexity is obtained. For very large reductions in complexity, refinement may be preferable over simplification, as reaching a coarse model from a very detailed one may take longer than starting from a very simple one. Indeed, Brodsky and Watson report fast running times for their algorithm. It is commonly believed, however, that refinement generally leads to lower quality results than simplification, which is consistent with their findings.

The method by He and co-workers [64] is novel in that it uses an intermediate volumetric representation of the model. By converting the original triangle mesh into a set of voxels, standard low-pass filtering techniques can be used to both remove fine geometric details and close topological holes. Unfortunately, the voxel data has to be converted back to a triangle mesh, which may still be dense. Nooruddin and Turk [111] take a similar approach, but extend the work by He et al. by using *morphological operators*, such as *erosion* and *dilation*, to explicitly remove topologically insignificant features. After converting the model back to a surface, further mesh simplification is performed to coarsen the dense mesh.

The publications by Popovic and Hoppe [116] and Eck et al. [37] have already been mentioned above. While the coarsening operations in these algorithms are common, other features of these algorithms make them unique. The method in [116] differs in the representation of the simplified surface. In this technique, vertex pair contractions sometimes reduce parts of the mesh to individual edges and vertices. In such cases, these simplices are represented using cylinders (for edges) and spheres (for vertices), while ensuring that their surface area matches that of the original geometry. This approach is similar to [127]. The goal in [37] is to remesh the original model by converting it to a wavelet representation with subdivision connectivity. Similar techniques are described in [11, 21, 61, 84, 87, 101, 143]. There is a close connection between lossy wavelet compression and simplification. However, in order to stay on topic, I will focus only on methods specifically designed for simplification, and refer the interested reader to [143].

3.1.2 Simplification of Topology

Several recently published algorithms for simplification have been centered around the issue of simplifying the *topology* of models with complex topological structure. The motivation for topology simplification was

discussed in §3.1.1.5; the amount of geometric simplification possible is limited by the topological complexity of the model. For models with a large number of connected components and holes, such as mechanical part assemblies, isosurfaces of turbulent flow, and large structures such as a building or a ship, it may be necessary to merge geometrically close pieces into individual larger ones to allow further coarsening (see, for example, [42]).

Algorithms based on vertex clustering [102, 104, 127] and vertex pair contraction [41, 50, 116, 133] are by nature topology modifying. However, few of these algorithms are designed specifically to simplify the topology, which is rather a byproduct of these coarsening operations. Instead, both of these coarsening operations modify the topology at the expense of introducing non-manifold simplices. The volumetric methods in [64, 111] circumvent this problem by treating the object as a solid, and simplifying the solid geometry of the model. A manifold surface can then be extracted from this simplified volumetric representation.

Schroeder [134] relies on half-edge collapses to coarsen the mesh, and uses *vertex splits*—the opposite of edge collapse—to "cut" the mesh into different pieces whenever non-manifold situations or geometric constraints would otherwise prevent further simplification. Whereas only limited topological simplification is allowed, e.g. disconnected components cannot be merged, this method can in some cases work around the dead ends reached by manifold preserving edge collapse methods.

The method by El-Sana and Varshney [38] is perhaps the most elaborate scheme for performing genus reducing simplification directly on surfaces. Their technique is able to identify small holes in the solid object, using a technique similar to α-hulls, and eliminate them. Inherent in this approach is also the ability to remove small indentations and bumps on the mesh.

Topology modifying simplification is without question essential for a number of applications. If in addition preserving the manifoldness of a surface is important, then the techniques in [38, 64, 111] are applicable. However, for many graphics applications, manifoldness of the surface is of less importance, and vertex pair contraction and its derivatives are adequate alternatives that are significantly easier to implement.

3.1.3 Preservation of Surface Properties

Real three-dimensional objects are more than simple geometric shapes; they also have texture and color, they react in certain ways to different lighting conditions, they have certain material properties, etc. Therefore, it is quite common for polygonal meshes to be augmented with such surface attributes, which can greatly enhance the appearance and semantic content of an object (see Figure 3.1), and it is often crucial that such information is preserved to the greatest extent possible during simplification. These surface properties can be categorized as discrete, such as surface material or texture indices, and continuous properties, such as normal vectors, RGB color components, and texture coordinates. Discrete attributes are most often assigned to the faces of a mesh, while continuous surface attributes are typically associated with the mesh vertices, and are linearly interpolated over the surface. For methods that introduce new vertices during simplification, a new set of attributes must be assigned to the vertices such that, when interpolated, they match the attributes of the original model well. Such a metric for comparing attributes necessarily depends on the geometry of the two models. While the choice of attribute metric varies between applications, most existing methods implicitly

a. Without vertex colors. b. With vertex colors.

c. Without texture. d. With texture.

Figure 3.1: Example meshes with and without surface properties.

attempt to preserve the *appearance* of the model.

There have recently been several publications of simplification methods that take surface properties into account. Bajaj and Schikore use vertex removal to coarsen a model and maintain bounds on geometric and attribute error [6]. Hoppe extended his original mesh optimization technique to include error terms for surface attributes [66]. Garland and Heckbert also have augmented their error metric to include terms for attributes such as texture coordinates and color [51]. Hoppe recently improved upon their work by proposing an alternative, more compact quadric error metric for measuring differences in geometry and surface attributes, and found that the use of "memoryless" edge collapses (see Chapter 4) further improved model quality [69]. Erikson and Manocha use point clouds in color space and texture coordinate space to choose new material properties after performing a vertex pair contraction [41]. Cohen and co-workers describe an edge collapse algorithm that maintains a mapping between the original and simplified surface [26]. This allows them to maintain error bounds for texture coordinates. Their technique was later extended to emphasize appearance preservation by parameterizing the surface and encapsulating surface attributes and fine details in normal and texture maps [27]. This approach is becoming a viable alternative as support for normal mapping is now being added to off-the-shelf graphics cards [79].

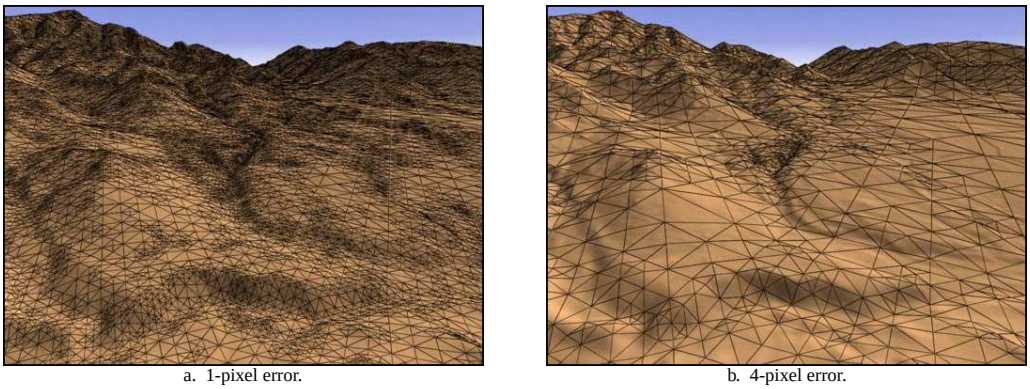
I, too, have developed methods for preserving surface attributes. In §4.7.1 (see also [97]), I will describe a heuristic for computing surface attributes for new vertices resulting from edge collapses. This method, like those of others [51, 69], extends the vertex representation from $\mathbb{R}^3$ to higher dimensions, i.e. each scalar attribute corresponds to a separate dimension, and generalizes the "memoryless" vertex placement scheme from §4.4 (and [94, 95]). As discussed in §3.1.1.4, however, placing vertices and assigning new attributes is but one component of edge collapse methods and their relatives. In addition, an error metric is needed for ordering the edge collapses. It is in this area that my work departs from previous work, in that an *image metric* is used to measure visual similarity. I will describe two methods, one for simplification (Chapter 7) and one for global mesh optimization (Chapter 8), that both use an image metric. Whereas most other techniques [6, 26, 41, 51, 66, 69] are faced with the difficult issue of combining different metrics for geometry and attributes, typically as a weighted sum, or relating geometry and properties in a single high dimensional space, image metrics provide a unified error measure that takes all these factors into account in a more direct and natural manner. Image metrics are in addition sensitive to the *image content* of a texture, while previous methods are concerned only with the texture parameterization.

3.1.4 View-Dependent Simplification

The majority of published work on model simplification has centered around producing static models from an off-line simplification algorithm. Using simplification, a few fixed levels of detail of an object are created, e.g. by successively halving the number of triangles, which are candidates for selection in the run-time display system. When switching from one level of detail to another, visual artifacts such as "popping" often occur because of the relatively large differences between two consecutive levels of detail. In addition, such large differences in complexity make it difficult to regulate frame rates, and do not allow an optimal representation of each object for a given view and rendering budget.

To overcome this problem, *view-dependent simplification* can be used. This technique allows objects to be adaptively simplified—or alternatively refined—on-the-fly based on the observer's viewpoint. I was among the first to use such techniques for real-time terrain rendering [93]. This algorithm is based on vertex removal on a mesh with subdivision connectivity, and in each frame produces a mesh that meets a user-specified screen space error tolerance (Figure 3.2). Duchaineau et al. [36] and Hoppe [68] have more recently described similar algorithms for terrain visualization.

Hoppe [67] was also able to generalize the technique in [93] to irregular meshes of arbitrary topology using his *progressive mesh* representation together with a tree of edge collapse dependencies. His method is also able to coarsen parts of objects that are outside the view volume or are occluded. Another contribution of his work is the use of *geomorphs* to smoothly blend geometry over several frames in order to eliminate popping. Xia and Varshney proposed a run-time system that selects the triangles to display based on their screen-space area, whether they are on the silhouette, and if they are near a specular highlight [160, 161]. More recently, El-Sana and Varshney have enhanced this work using a view-dependence tree to help avoid fold-over and to utilize memory better [39]. The same researchers have also described a method of producing triangle strips



a. 1-pixel error. b. 4-pixel error.

Figure 3.2: View-dependent simplification of terrain. The geometric errors of the simplified surface are projected to screen space and compared against a user-specified threshold.

on-the-fly for view-dependent simplification [40]. Luebke and Erikson perform view-dependent simplification based on a hierarchical data structure that is built off-line using vertex clustering [104]. This framework allows other types of coarsening operations to be used.

The view-dependent methods discussed above use geometric distance measures to perform their off-line simplification preprocessing. In particular, the preprocessing is typically done in a *view-independent* manner, leaving the *run-time* system to construct a view-dependent model by choosing from a set of predefined simplification moves (e.g. edge collapse, vertex removal, etc.). These two components of view-dependent simplification are, in fact, orthogonal, assuming the two methods are based on the same coarsening operation, and that the off-line simplification method is able to provide the information needed to compute the view-dependent metric. For example, it would be entirely possible to build a run-time system based on the edge collapse methods presented in Chapters 4 and 7. My image-driven simplification method, in particular, has proven to produce static models of very high visual quality. Because visual similarity is of prime importance in view-dependent simplification, this algorithm also has the potential of being more effective for this purpose than previous methods.

3.1.5 Out-of-Core Simplification

Spatial and temporal data sets for computer graphics and visualization are becoming larger at a rapid pace. In fact, a substantial fraction of existing data sets are significantly larger than available memory, such as the Visible Human [1] and Digital Michelangelo [90] data sets. In order to process and visualize these large quantities of data, *out-of-core* methods, which do not rely on having instantaneous access to the entire data set, are needed. Example applications and techniques for out-of-core processing include surface reconstruction [10], isosurface extraction [22], terrain visualization [34], ray-tracing [114], and streamline visualization [149].

Even as polygonal data sets have exploded in size over the last few years, little work has been done in the area of out-of-core simplification. This can be attributed in part to the fact that very few published in-core methods can easily be extended to perform out-of-core simplification. One exception is the clustering method

by Rossignac and Borrel explained in §3.1.1.1. While the authors do not explicitly point this out, it is quite straightforward to adapt their algorithm for out-of-core processing. This technique is also the basis for my own out-of-core algorithm, which is described in Chapter 5. This new algorithm improves upon Rossignac and Borrel's method in computational complexity, speed, disk usage, and quality.

Bernardini et al. describe an algorithm that has been specifically designed to perform out-of-core simplification [9]. Their method splits the model up into separate patches that are small enough to be simplified separately in-core using a conventional simplification algorithm. The patch boundaries are left intact to allow the different pieces to be stitched together without cracks after simplification. A new set of patch boundaries is then used as another iteration of simplification is performed, allowing the seams between the previous set of patches to be coarsened. This technique is very similar to the one proposed by Hoppe for creating hierarchical levels of detail for height fields [68] (see §3.1.4). His technique has more recently been generalized to arbitrary meshes by Prince [121]. While conceptually simple, the time and space overhead of partitioning the model and later stitching it together adds to an already expensive in-core simplification process, rendering such a method less suitable for simplifying very large meshes. In Chapter 5, I will describe a more efficient solution to this problem.

3.2 Optimization

Whereas most simplification methods perform some type of optimization, it is very rare that the resulting model is, in fact, optimal. One reason for this is that the space of all possible meshes of a given complexity is enormous, and shortcuts must be taken. The general approach that most simplification methods take is to iteratively coarsen the model by performing a *greedy* [29] sequence of local operations (e.g. edge collapse), and in each iteration optimize a single vertex. Such an approach suffers from the following problems:

1. The position of any single vertex is optimal only with respect to a fixed instance of the remaining mesh. To obtain a truly globally optimal mesh, all vertices must collectively be optimal with respect to each other.

2. Each vertex is optimized only for a single iteration, that is, the vertices in the final mesh are each moved (optimized) at most once.[2]

3. The mesh connectivity is generally suboptimal. Very few methods make any attempt to improve the connectivity, and those that do typically limit themselves to making a few local changes.

4. The order of coarsening operations is generally greedy. By always performing the "cheapest" move, it is possible for a method to paint itself into a corner. Planning ahead, however, rapidly increases the search space.

[2] As a simple illustration of why this can be a problem, consider the 2D case of approximating a circle with four respectively five vertices. Most metrics will suggest that a square and a regular pentagon are the two optimal shapes, yet no single edge collapse will transform a regular pentagon into a square without moving several vertices.

5. For efficiency, most metrics used during simplification are approximate and measure only local errors.

From the points made above, it follows that there is often much room for improvement. However, to reach our goal, a more global approach to optimization, that considers the entire mesh as a whole, is needed. In this section, I will discuss related work on global optimization and, in particular, optimization of meshes.

Before discussing particular problems relating to mesh optimization, it is useful to consider the more general case of multi-dimensional optimization. There are two main categories of optimization: combinatorial (or discrete) and continuous optimization. In combinatorial optimization, the search space is generally finite, but typically increases exponentially with the input size, thus traversing the entire space is seldom practical. Most combinatorial optimization problems are solved by making a sequence of small perturbations to an existing combinatorial state. As an example, operations such as edge collapse and edge swap can be used to perturb the connectivity of a polygonal mesh. However, few general guidelines exist for choosing what perturbations to make and when, and application specific heuristics are often needed. The other category, continuous optimization, can be further divided into methods that use gradient information, and ones that don't. Steepest descent, conjugate gradients, Powell's method, and quasi-Newton methods all use gradient information to determine which direction to move in to approach the optimum as quickly as possible. When the gradient of the objective function cannot be easily computed, which is generally the case for image metrics, techniques that rely on function evaluations alone must be used. Methods such as simulated annealing and genetic algorithms, which both can be used for either discrete or continuous optimization, belong to this camp, as does the *downhill simplex method*, which is the technique used in my optimization method. For an objective function with n parameters, the downhill simplex method constructs a simplex with $n + 1$ vertices in $\mathbb{R}^n$, and evaluates the function at these points. A number of moves are then attempted, such as reflecting one vertex around the hyperplane of the others, or contracting or expanding the simplex in one or more directions, and the function is always evaluated at any newly visited points in space. Because of its generality and ease of implementation, this method has been used to solve a wide variety of problems, including tile shape optimization for Escher-like tilings [76]. The downhill simplex method as well as several other optimization methods are described in [120].

Many objective functions have more than one optimum. In the case of minimization, such functions have two or more local minima, around which there is a finite neighborhood for which the function values increase in every direction. This is generally the case for the image metrics described later. None of the optimization techniques mentioned above, whether for combinatorial or continuous optimization, is able to guarantee that the global and not a local optimum is found. Instead, it is quite common that the optimization converges on a local optimum, from which no further progress is made. In order to escape from such a state, appropriate combinations of sufficiently large changes to the input parameters must be considered to cross any obstructing "hills" in the objective function. While there are no general guidelines for determining when and how such moves should be attempted, we shall see later that some provisions have been made to address this problem in my optimization method.

The work that is most closely related to my image-driven optimization method is that of Hoppe et al. [72]. Similar to my own optimization method, theirs attempts to produce a globally optimal mesh that best approximates an original, detailed mesh. However, whereas my algorithm attempts to improve an already simplified mesh, theirs can also be used for simplification, though this feature is more or less a consequence of their choice of error metric, which includes a term that penalizes meshes with a large number of vertices. This metric, or energy functional, additionally includes a term for surface deviation and a spring term that penalizes long edges. Since the two goals of producing a mesh that is both concise and faithful to the original are in direct conflict with each other, mesh vertices are spent only in areas where they make a significant impact on the geometric error.

The approach in [72] is to make a series of random connectivity moves, e.g. edge collapse, swap, and split operations (Figure 2.2), and accept a move if it lowers the overall mesh energy. Subsequent to each connectivity move, the positions of the vertices near the affected area are simultaneously optimized. The surface error is estimated by sampling a large number of points on the original surface, projecting these onto the approximating surface, and measuring the sum of squared distances. These point-to-point correspondences are maintained during optimization using barycentric coordinates. By expressing the surface error and spring energy as quadratic forms, the optimum is given by a system of linear equations, which is solved quickly using an iterative conjugate gradient solver.

While my optimization method has much in common with the one in [72], there are also many differences, which I will discuss in more detail in Chapter 8. Briefly, the following features of my algorithm distinguish it from [72]:

- uses an image-metric to measure error
- is based on a different continuous optimization method
- uses an oracle to determine where to improve the mesh
- chooses connectivity moves based on recent performance
- keeps the number of vertices in the mesh constant

Blanz and Vetter [13] have recently proposed an algorithm for morphing human faces. In some respects, one of the goals in their work parallels mine. They address the problem of synthesizing a novel 3D face, using a database of a few hundred range scanned faces, that matches the face of a person in a 2D photograph. This is accomplished by first parameterizing the geometry and texture of the faces in the database using principal component analysis, which then allows novel faces to be constructed using linear combinations in this parameter space, effectively providing the means of combining different characteristics of human heads. In addition to these parameters, variables for camera position and orientation, lighting conditions, and photometric properties must be estimated and optimized over. For a given set of these parameters, an image of the resulting face is rendered and compared against the image from the photograph using the pixel-wise root mean square error. To find the optimal match, a stochastic gradient descent technique is used that periodically selects a random subset of parameters to optimize over.

Even though the goal in [13] is very similar to mine, i.e. altering the geometry and texture of an object to match a given set of images, the two parameter spaces and types of models considered are entirely different, making it very difficult to compare the strategies employed in the two methods.

There are a number of related techniques for optimizing meshes in areas other than simplification. Hoppe et al. extended their mesh optimization scheme to subdivision surfaces [70]. This technique was used to fit a smooth Loop surface [98] to a dense set of points produced by range scanning. A different approach was taken by Halstead et al. [62] to solve essentially the same problem for interpolating Catmull-Clark subdivision surfaces [20]. In their case, however, the number of control points is assumed to be small enough that exact interpolation is possible, and they provide a closed form solution to the problem. Given this insurance, any additional degrees of freedom are used to minimize the membrane and thin-plate energies of the surface—terms that govern its smoothness. This notion of measuring and controlling smoothness is important in surface *fairing*, for which membrane and thin-plate energy are two important fairing functionals that measure the "wiggliness" of the surface. In mathematics, the study of *variational calculus* is devoted to finding solutions to surfaces that globally minimize such functionals. Being another example of surface optimization, these techniques, which generally yield closed form solutions, have been used to construct maximally smooth continuous surfaces [109, 153, 154] and approximating meshes [83].

Similar to the simplification algorithm by Turk [146], Witkin and Heckbert [158] simulate repulsion forces between "floater" particles that sample an implicit surface. While their goal is to cover the implicit surface with a good distribution of particles rather than constructing an optimal mesh, the optimization procedure they use, which includes combinatorial optimization via fusion and fission of particles and relaxation of particle positions, could easily be generalized to meshes. Indeed, the surface modeling technique in [155] also uses repulsion to move the vertices in a triangle mesh. The connectivity in this algorithm is implicitly defined by producing a constrained Delaunay triangulation, which leads to well-shaped triangles. For fusion and fission of vertices, they use edge collapse and edge split, respectively. Markosian et al. [106] use coarse geometry to build a "skeleton," around which they evolve a smooth subdivision surface that they call the "skin." Their technique allows particles to be positioned by the user to guide the evolution, and the geometry and connectivity of the base mesh for the subdivision surface are then optimized.

3.3 Metrics for Simplification and Quality Evaluation

One of the most important characteristics of a simplification method is the *error metric* it uses. Because the simplified model is rarely identical to the original, such a metric is needed to measure how similar the two models are. As mentioned previously, most existing techniques for doing simplification rely on some measure of geometric deviation between the two surfaces. For many graphics applications, however, these models are ultimately used to produce raster images. If we are interested in the visual quality of a model, then an image metric may be more suitable. Such metrics have been developed, for example, to measure the degree of photorealism in computer generated images, search image databases, guide algorithms for image generation, and measure the quality of lossy image compression algorithms.

Formally, a metric is a real-valued, non-negative function d between a pair of objects x and y that satisfies the following properties:

1. $d(x, y) = 0 \iff x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, z) \leq d(x, y) + d(y, z)$

In practice, few "metrics" for comparing triangle meshes or images satisfy all three of these properties (property 3, called "triangle inequality," is the one most often violated). However, as most other authors, I will ignore this discrepancy and use the term "metric" to refer to any function or algorithm designed to measure similarity. Below, I will discuss previous work on geometry-based and image metrics.

3.3.1 Geometry-Based Metrics

Before delving any further into this topic, I would first like to point out that metrics for simplification are commonly used for two distinct purposes; evaluating the quality of the final model, and determining where and how to simplify a model. Metrics in the former category are more likely to fit the formal mathematical definition of a metric, and typically involve a more rigorous procedure for measuring similarity. If done correctly, a metric defined for simplification both determines the position of new vertices and the order in which coarsening operations are applied, i.e. these two decisions are made so as to minimize the error metric. However, it is sometimes difficult to express exactly what the metric is for a given simplification method. Consider, for example, a half-edge collapse algorithm that orders edges based on length and always collapses the shortest edge to the vertex with higher curvature. Such an algorithm, while simplistic, is not entirely unreasonable, but it is not clear what the metric is in this case. How does this algorithm measure and minimize the difference between the two objects?

Even though many simplification algorithms do not use a single, well-defined error metric, there are several useful and mathematically sound metrics for quality evaluation, such as the symmetric Hausdorff distance and the volume of symmetric difference, and many simplification methods are clearly inspired by these metrics. To efficiently evaluate them during simplification, however, approximate estimates of these metrics are often used. In the following section, I will discuss some of the better known global, geometry-based metrics used for off-line quality evaluation. It should be relatively easy to identify how approximate versions of these metrics are used in various simplification algorithms.

3.3.1.1 Global Metrics

The *Hausdorff distance* is probably one of the most well-known metrics for making geometric comparisons between two point sets. This metric, which is defined in terms of another metric such as the Euclidean distance, uses the shortest distance between a point x and a set of points (e.g. a surface) Y:

$$d(x, Y) = \min_{y \in Y} d(x, y) \tag{3.1}$$

where $d(x, y)$ is a metric over pairs of points.[3] This is then generalized to two sets X and Y as follows:

$$\vec{d}_\infty(X, Y) = \max_{x \in X} d(x, Y) = \max_{x \in X} \min_{y \in Y} d(x, y) \tag{3.2}$$

Note that this is an *asymmetric* distance, i.e. $\vec{d}_\infty(X, Y) \neq \vec{d}_\infty(Y, X)$ in general, so $\vec{d}_\infty$ is not a metric. The (symmetric) *Hausdorff distance*, defined as follows:

$$d_\infty(X, Y) = \max\{\vec{d}_\infty(X, Y),\, \vec{d}_\infty(Y, X)\} \tag{3.3}$$

is on the other hand a metric. The notation d_∞ comes from the similarity with the L_∞ norm (§2.2), and I will refer to it as the *maximum geometric error*. There are several simplification algorithms that are based on bounding the maximum error, such as [6, 23, 26, 57, 75, 81, 82, 126, 127]. Perhaps the most notable one is the *Simplification Envelopes* method by Cohen et al. [28], which explicitly computes and surrounds the model with two surface envelopes that are a threshold distance from the original surface.

Since a single point determines the Hausdorff error, it conveys little information about the overall similarity between the two surfaces. One possible alternative metric is based on the average deviation. It can be defined as follows:

$$\vec{d}_1(X, Y) = \frac{1}{A^X} \int_{x \in X} d(x, Y)\, dS \tag{3.4}$$

To obtain symmetry, the following convex combination of asymmetric distances can be used:

$$d_1(X, Y) = \frac{A^X}{A^X + A^Y} \vec{d}_1(X, Y) + \frac{A^Y}{A^X + A^Y} \vec{d}_1(Y, X) \tag{3.5}$$

$$= \frac{1}{A^X + A^Y} \left(\int_{x \in X} d(x, Y)\, dS + \int_{y \in Y} d(y, X)\, dS \right) \tag{3.6}$$

where A^X denotes the area of surface X. I will refer to d_1 as the *mean geometric error*. This metric is essentially an extension of the L_1 norm. These two metrics, d_∞ and d_1, are used in the mesh comparison tool *Metro* [24]. It is clearly impractical to sample every single point on both surfaces, so Metro approximates the metrics by replacing the infinite sets X and Y with two finite subsets of them, using scan conversion of the triangles, and substituting the integrals with finite summations:

$$d_1(X, Y) = \frac{1}{|X| + |Y|} \left(\sum_{x \in X} d(x, Y) + \sum_{y \in Y} d(y, X) \right) \tag{3.7}$$

Many of the quantitative results presented in later chapters were produced by Metro.

Other researchers have suggested small variations on the mean and maximum error metrics above. Hoppe et al. [72] use an asymmetric approximation $\vec{d}_2^2$ of the mean square error by sampling a large number of points on the original model. Their approach is unique in that it uses this rather elaborate, but high quality, error measure during simplification. Garland and Heckbert [50] also suggest using the mean square error for

[3] In the remainder of this thesis, I will assume that the Hausdorff metric is defined over a Euclidean metric space.

comparing simplified models, but ensure that their metric is symmetric, i.e. they use d_2^2. In more recent work, Hoppe uses the symmetric root mean square error d_2 to evaluate his results [69]. Many simplification methods are also based on some type of aggregate error, including [41, 50, 66, 69, 72, 94].

For closed models, i.e. surfaces that enclose a solid, an alternative to the mean and mean square errors would be to measure the volume of the symmetric difference between the two solids. That is, for solids X and Y, the symmetric difference is the set of points that are contained in one solid but not the other, or in set notation $(X \setminus Y) \cup (Y \setminus X)$. Unfortunately, computing this set can be very computationally costly for complex meshes. The algorithm in Chapter 4, which measures sums of squared changes in volume, is loosely based on this integral measure of error.

Rather than using Equation 3.1 to compute the minimum distance of a point to the other surface, it may be more desirable to establish a one-to-one mapping between the surfaces and measure pair-wise distances between points. The *MAPS* algorithm by Lee et al. [88] can be used to establish a homeomorphism between meshes, but it requires using their simplification algorithm. It is however possible to extend their method to other schemes for simplification.

The metric in [4] was designed for curves rather than surfaces, although it has a number of interesting features, including affine (e.g. translation, rotation, and scale) invariance, that may be useful for surfaces. This metric is based on expressing a polygonal curve as a "turning function," which essentially measures the signed changes in angle, parameterized by (normalized) arc length. It is not immediately clear how to extend such a metric to surfaces, however.

3.3.1.2 Quadric Error Metric

The metrics discussed so far are all designed for the purpose of performing an off-line comparison between two meshes (or curves). Because of their computational complexity, they cannot always be directly adapted for use in simplification without taking some shortcuts. For metrics based on maximum error, this typically means using conservative and quickly computable bounds on the error, e.g. [26, 57]. In the case of mean errors, the evaluation is often done locally on a small subset of points on the surface, e.g. [72].

There is a metric of particular interest in this thesis which was introduced by Garland and Heckbert [50], and has also been used by me [92, 94, 95, 97] and others [41, 69]. This metric, called the *quadric error metric* [50], is based on weighted sums of squared distances. It is essentially the same as the discrete version of the asymmetric $\vec{d_2^2}$ metric, where the vertices in the simplified model constitute the set of sample points. The distances are measured with respect to a collection of triangle planes associated with each vertex. In Garland and Heckbert's original scheme, these triangles are initially the vertex's set of incident triangles. As an edge is collapsed, the resulting vertex inherits the triangle planes from the merged vertices. The beauty of the quadric metric is that it can be evaluated very efficiently by representing any set of planes as a single symmetric 4×4 matrix. Beyond these features, methods based on quadric error vary in their choice of weighting scheme of plane distances (e.g. uniform or area-based), as well as in whether quadrics are accumulated [41, 50] or recomputed from scratch [69, 94]. I will provide a more detailed analysis of different quadric error metrics in §4.4.8.

3.3.1.3 Absolute versus Incremental Errors

Metrics for simplification, in contrast to metrics for quality evaluation, do not necessarily have to measure absolute errors with respect to the original model (although many of them do). Instead, errors can be measured *incrementally* as the amount of change introduced by each coarsening operation. I showed in [94] that it is possible to use an incremental error measure for high quality simplification. Even though many small errors can cascade and build up a large error, my "memoryless" method consistently yields smaller mean errors than most competing schemes that measure error against the original model. A number of other successful incremental simplification methods have been proposed. While Guéziec [57] tracks the maximum deviation from the original surface, his edge collapse order is determined by a memoryless metric, namely edge length. Cohen et al. [26] optimize the vertex from an edge collapse by minimizing the amount of change made. The curvature measure used by Schroeder and co-workers [135] is similarly memoryless. Algorithms based on deviation from the original model include [28, 50, 72]. I will address the differences between these two approaches for the case of quadric errors in §4.8.2.

3.3.2 Image Metrics

Image metrics have recently been adopted in a number of different graphics applications. Since most graphics algorithms produce images, it makes a lot of sense to drive these methods and evaluate their results using measures of image difference instead of, say, metrics based on geometry. Whereas some image metrics are rather simple and treat each image in a geometric sense as if it were any old bivariate function, there has lately been some effort to make use of the empirical research done in psychology and vision to develop computational models of human vision for perceptually motivated metrics. Because this thesis is primarily about simplification, and not image metrics, I will keep the review here short. There is a wealth of knowledge relating to visual perception, image processing, and sampling theory, and this summary barely scratches the surface in these areas. The intention of this brief review is rather to point the reader to some of the more important publications in these areas.

Before discussing perceptual metrics, I will briefly mention some simple metrics and image processing techniques, as well as applications that use such metrics. Probably the most well-known metric for comparing images is the L_p pixel-wise norm, and in particular the d_2 root mean square error. More elaborate, although not necessarily perceptually-based, metrics can be built based on filtered versions of images. Example filters include the Fourier transform [129], wavelet transforms [74], the Laplacian pyramid [17], and steerable pyramids [46]. Steerable pyramids are useful for decomposing an image into different alias-free frequency bands at varying orientations. Because the human visual system performs a similar transform [151], this decomposition has been put to use in several perceptual metrics.

Little work has been done on evaluating mesh quality using image metrics. Garland suggests a possible approach to doing this in his thesis [49] using the d_2 metric over a collection of images from different viewpoints, but does not use it to evaluate his results. This approach to evaluating visual similarity is essentially the one taken in my image-driven algorithms.

Watson and co-workers [152] have performed a user study to determine how different metrics correlate with people's ability to identify simplified models. In their study, subjects were shown images of simplified models (produced by a simple clustering method similar to [127]), such as man-made objects and animals, and were asked to as quickly as possible identify the objects. This variable, the *naming time*, was then correlated with the image differences given by the different metrics. They found that few metrics had a statistically significant correlation with naming time, but that image metrics generally performed somewhat better than the geometry-based metrics used by Metro (§3.3.1.1). It is worthwhile to point out, however, that naming time and subjective quality do not necessarily correlate, and that the low quality models used in their study may also unfairly skew the results.

In the last few years, there have been several published papers on graphics algorithms that incorporate simple image metrics. Jacobs et al. [74] use a Haar wavelet transform of a rough image sketched by the user, and compares the wavelet coefficients against those of other images. This is done to allow fast querying and retrieval of images in a large database. Marks and co-workers [107] propose using a weighted mip-map representation of an image as part of an image metric to evaluate differences in outputs (the images) resulting from variations in the inputs (the parameters of a rendering algorithm). This is used to construct a set of outputs that span the space of images obtainable by making changes to the inputs. The goal of their system is to provide the user with an atlas of possible outputs to make choosing parameters easier. Turk and Banks [147] describe an algorithm for automatic placement of streamlines for a vector field. Assuming that a nearly uniform distribution of streamline density in the output image is desirable, they measure the mean square difference between a low-pass filtered version of the streamline image and a constant gray image. This error measure is used to guide placement of and modifications to streamlines. Maciel and Shirley [105] use a Laplacian operator followed by blurring to compare images. This metric is used to determine the quality of different levels of detail from a number of viewpoints around an object. These prerecorded differences are then put to use in a view-dependent system for level of detail selection.

3.3.2.1 Perceptually Motivated Metrics

The human visual system, while not perfect, is an extraordinary imaging device, capable of processing light within broad ranges of spectral frequency, spatial resolution, and light intensity levels. In fact, these capabilities far exceed the range of outputs from conventional computer monitors and paper prints—the two most commonly used media in computer graphics. However, even within these narrow ranges, we do not perceive individual pixels as linear functions of frame buffer intensities, nor do our eyes treat an image as a simple height field of intensity values. Rather, several complex mechanisms throughout the visual cortex transform incoming light into signals that are interpreted by our brains in a manner much different from the physical light that strikes our retinas. In order to develop a metric that accurately measures how and when we perceive two images to be different, knowing how the visual system works is essential.

During the last century, research in psychology, physiology, and psychophysics has lead to remarkable advances in understanding the workings of the human visual system [151]. Drawing upon this wealth of information, researchers in computer vision have developed mathematical models that predict how we perceive

simple patterns of light, which can sometimes be generalized to more complex visual scenes. Recognizing that such models are extremely powerful tools for comparing digital images, several perceptually motivated image metrics have emerged in the computer vision and graphics fields. Currently, two of the most perceptually accurate metrics for comparing images are the *Visual Difference Predictor* by Daly [32] and what is known as the *Sarnoff Model*, due to Lubin [103]. These two metrics are built on the notion of *just noticeable differences* (JND), i.e. differences between two images that are just above the threshold of detectability. Both of these metrics attempt to emulate some of the stages in the visual system, such as spatial frequency and orientation decomposition, correction for variable sensitivity to different light and contrast levels, suppression of patterns due to visual masking, and so on. These models are motivated by the assumption that applying such transforms to the images will result in a representation that is closer to the one used in the higher level processing stages of the human visual system, and that measuring differences between these representations will more accurately reflect how we actually perceive image differences. In many respects, the models discussed here successfully achieve this goal. Without going into much further detail, I will discuss some of the recent work on perceptual image metrics for vision and computer graphics. Later, in Chapter 6, I will revisit the Sarnoff Model, which is the basis for my own image metric.

Barten [7] describes a measure of image quality that he calls the "square-root integral" (SQRI), which includes a continuous expression for estimating the *contrast sensitivity function* (CSF). The CSF is a measure of responsiveness to contrasts of different spatial frequencies (see Figure 3.3).[4] For example, the contrast threshold of detecting a sinusoid pattern is not constant over all frequencies, but peaks at around two cycles per degree, and then drops off rapidly at higher frequencies [151]. Barten's contrast sensitivity function is a key component in several recent metrics, e.g. [14, 122].

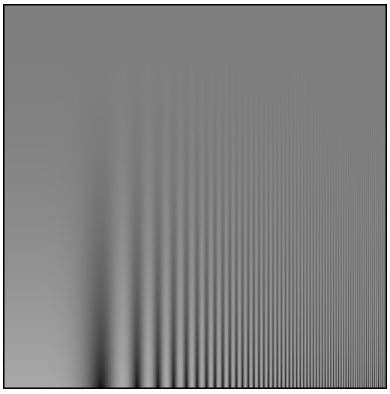
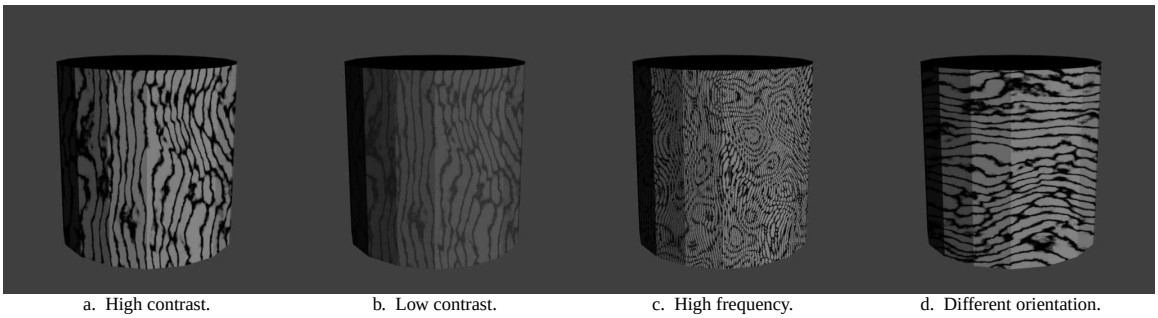


Figure 3.3: Sinusoid contrast pattern at various spatial frequencies illustrating variable contrast sensitivity. The contrast decreases linearly from bottom to top and uniformly over all frequencies, which increase from left to right. The perceived contrast, however, peaks near the middle on the horizontal axis, near two cycles per degree. This image is courtesy of Bolin and Meyer [14].

Ferwerda and co-workers [44] have developed a model for *visual masking*. This phenomenon manifests

[4]Note that the images presented here may not reproduce accurately on some display systems, leading for example to aliasing and distortion of contrast and luminance levels. In such cases, I refer the reader to the papers in which the images originally appeared.

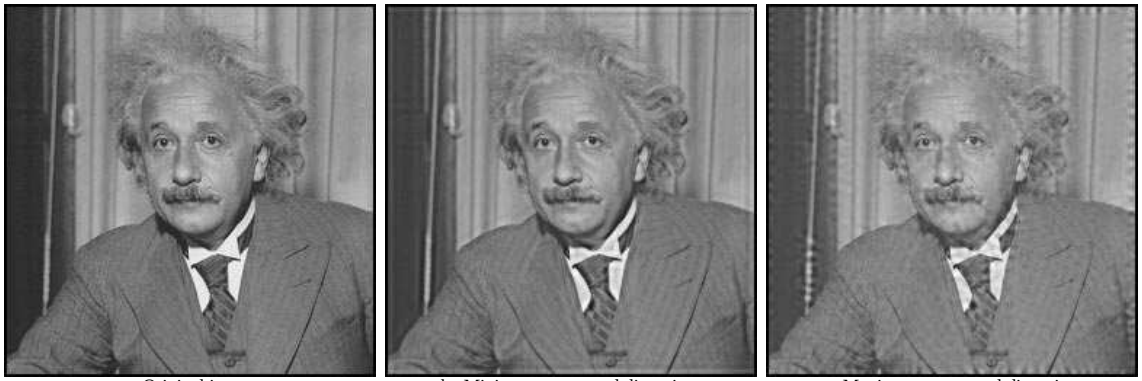
itself in a lessened ability to discriminate contrast patterns at certain frequencies in the presence of similarly oriented patterns of nearly equal frequency (see Figure 3.4). This fact is of particular significance in mesh simplification, where the faceting inherent in polygonal models can sometimes be masked when texture is applied.



a. High contrast. b. Low contrast. c. High frequency. d. Different orientation.

Figure 3.4: Illustration of visual masking. The faceting is less visibly noticeable when the texture contrast is high and the texture frequency and orientation nearly match the facet pattern. This example was adapted from [44].

Like Lubin [103] and others, Teo and Heeger [144] also use steerable pyramids for image decomposition. They show how their metric is able to detect visual artifacts that do not align well with edges and other contrast patterns in the image (Figure 3.5), and also include results for the root mean square error that illustrate its relatively poor correlation with image quality. Rushmeier and co-workers suggest several metrics for comparing real images and synthetic ones produced by global illumination algorithms [129]. They show that, using a Fourier transform of the image, frequency content is much more important than phase information in their application, and compare images after essentially throwing away all the phase information. Their metrics also make use of psychophysical data.



a. Original image. b. Minimum perceptual distortion. c. Maximum perceptual distortion.

Figure 3.5: Examples of image distortion. The distortion in the middle image is less noticeable than in the image on the right, yet the root mean square error suggests the opposite. The images are courtesy of Teo and Heeger [144].

Because many display devices have very limited resolution in intensity (typically, eight bits of intensity are available), many high contrast scenes, including scenes as simple as a furnished room illuminated by fluorescent light, cannot be reproduced very accurately. Larson et al. [85] and Tumblin and Turk [145] have developed methods for compressing the luminance of such high contrast images in a manner that preserves low contrast details. Several of the ideas used in these methods are applicable to perceptual image metrics.

Perceptual metrics have recently been put to use in graphics to guide rendering algorithms. Bolin and Meyer [14] propose a simplified metric based on the Sarnoff Model, which uses a fast Haar wavelet transform instead of the more costly steerable pyramid. Their metric is used to drive a ray-tracer. Based on the perceptible quality of the image, the metric is able to suggest where additional ray samples are needed to bring the errors in the image below the threshold of detection. A similar technique was described by Ramasubramanian et al. [122] for computing radiosity solutions. The main contribution of their work is a separation of luminance-dependent and spatially-dependent processing, allowing them to do much of the work in the physical domain directly on the intensity values, thereby avoiding having to repeatedly convert the partially completed image to the perceptual domain.

While not an image metric per se, Reddy [123] describes a metric for simplification that is based on perceptual factors. Using the contrast sensitivity function and the local scale and shape of the surface, he is able to estimate whether the change in intensity caused by a coarsening operation is perceptible. While this technique has the potential to imperceptibly reduce a dense model, it can only successfully be used until further simplification would introduce visible artifacts. Most simplification methods, however, operate above such threshold differences. In addition, the method does not address practical issues such as aliasing of polygon edges and self-occlusions which, even without simplification, can artificially create patterns of near arbitrary frequencies and contrasts. Finally, his model does not incorporate texture and other surface attributes.

Many of the metrics described here have been designed particularly to measure just noticeable differences. In simplification, we are often faced with models for which the differences are more apparent. To handle such suprathreshold differences, I have developed a new metric that borrows heavily from the work by Lubin [103] and Bolin and Meyer [14], but which is sometimes able to provide more meaningful results when the images are noticeably different. In addition, this metric is substantially faster to evaluate than the Sarnoff Model, making it a more attractive alternative for use in simplification.

Chapter 4

MEMORYLESS SIMPLIFICATION

4.1 Introduction

Automated techniques for creating polygonal models has made it very easy to construct meshes of very high complexity, and models with millions of polygons are not uncommon. As a consequence of model size, producing simplified approximations of such large models requires significant computational resources. Even though processor speed and memory sizes keep increasing at exponential rates, we have found ourselves at a point where the complexity of 3D models appears to be growing faster than computing resources. Thus, finding faster and more memory efficient algorithms for simplification has become increasingly important.

Most previous simplification methods are based on metrics that measure error against the original, dense mesh. It intuitively makes sense to use such a metric, because the ultimate goal is invariably to minimize the difference between the original and the simplified model. In order to evaluate such absolute errors, however, either a copy of the original model must be kept in memory [28, 72], or some form of history of changes made to the model must be carried along [26, 50, 57]. Regardless of which of these two approaches is taken, significant memory overhead is incurred, and evaluating these metrics is often a costly procedure.

To address these issues, I propose a *memoryless* simplification method that, instead of measuring errors with respect to the original mesh, orders a sequence of edge collapses based on the *amount of change* they incur relative to the partially simplified model. This incremental measure of error is based on the amount of volume swept out by affected triangles as they are moved during an edge collapse. To keep cascading errors to a minimum, this method ensures that the volume contained by the surface is always preserved, which serves to balance out errors in the outward and inward directions. For models with surface boundaries, the memoryless method analogously attempts to minimize the areas swept out by boundary edges. I will show how these techniques can also be generalized to minimize changes to surface properties such as color and texture.

After describing the memoryless simplification method, I will provide a thorough evaluation of it by comparing it against several well-known simplification methods, both analytically and empirically. These results show that my method is not only faster than most other methods, but it surprisingly also produces higher quality models in the mean error sense.

4.2 Computing Areas and Volumes in $\mathbb{R}^n$

Before describing the memoryless algorithm, some preliminary linear algebra is needed. The distance metric used in this new simplification algorithm is based on geometric properties of the mesh such as volumes and areas. As we shall see later, these triangle areas and tetrahedral volumes are sometimes embedded in higher dimensional spaces. In this section, I will develop the necessary machinery for representing and computing such geometric properties.

4.2.1 Determinants and Cross Products

The two fundamental geometric properties that I will make use of later are the *area* of a triangle and the *volume* of a tetrahedron. Let us first consider computing the area of a triangle $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2)$ in two dimensions. A well known expression for the signed triangle area is:

$$A = \tfrac{1}{2} \det \begin{bmatrix} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 \end{bmatrix} \tag{4.1}$$

Whether A is positive or negative depends on the relative order in which the vectors $\mathbf{x}_i$ are specified, i.e. whether the sequence of vertices is ordered clockwise or counter clockwise. This notion of signedness will become important to us later when attempting to preserve area and volume. We can extend the triangle area equation to finding the volume of a tetrahedron $(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ in $\mathbb{R}^3$:

$$V = \tfrac{1}{6} \det \begin{bmatrix} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 & \mathbf{x}_3 - \mathbf{x}_0 \end{bmatrix} \tag{4.2}$$

In general, the hypervolume, or *content* C, of an n-simplex in $\mathbb{R}^n$ is

$$C = \tfrac{1}{n!} \begin{bmatrix} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 & \cdots & \mathbf{x}_n - \mathbf{x}_0 \end{bmatrix} \tag{4.3}$$

The above equations can be used when the dimension n of the simplex (e.g. triangle, tetrahedron) equals the dimension m of the space it lies in. For an n-simplex embedded in $\mathbb{R}^m$, $m > n$, a different approach is necessary, however. It is worthwhile to note that the cross product can be used to compute the (unsigned) area of a triangle ($n = 2$) embedded in $\mathbb{R}^3$ ($m = 3$):

$$|A| = \tfrac{1}{2} \|(\mathbf{x}_1 - \mathbf{x}_0) \times (\mathbf{x}_2 - \mathbf{x}_0)\| \tag{4.4}$$

where the $\mathbf{x}_i$ are 3-vectors. For $m = n + 1$, this method can be generalized to higher dimensions using a generalized cross product of n vectors. However, the method breaks down whenever $m > n + 1$, except in two special (but less important) cases.[1] For a general solution to this problem, we first need to familiarize ourselves with the *exterior product*.

4.2.2 The Exterior Product

The exterior product, which is also called the *wedge product*, is a binary operator over pairs of *multivectors*. In $\mathbb{R}^n$, the multivectors include the scalars, which are called grade 0 elements, the vectors of dimension n

[1] It can be shown that vector-valued cross products can be defined for $(m = 7, n = 2)$ and $(m = 8, n = 3)$ [100].

(grade 1 elements), and grade k ($2 \leq k \leq n$) elements formed by applying the exterior product of lower grade elements. This exterior product is a central component in what are known as *Clifford algebras* [117], which, for example, include the *quaternions*. For an excellent introduction to Clifford algebras and the exterior product, I recommend [100].

For vectors $\mathbf{a}$ and $\mathbf{b}$, the exterior product is written $\mathbf{a} \wedge \mathbf{b} = \mathbf{c}$, where $\mathbf{c}$ is called a *bivector* (a grade 2 element). Similarly, the exterior product $\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}$ of three vectors yields a *trivector* (a grade 3 element), etc. Bivectors differ from vectors in that, instead of defining a length and a direction, they specify an area and a direction. Whereas vectors can be thought of as subsets of oriented lines, bivectors can be thought of as subsets of oriented planes. Similar to how vectors do not have unique start and end points, bivectors do not specify unique areas within the plane—neither the location nor the shape of the area enclosed is of any significance (although which side of the plane is "up" matters). Since bivectors are products of vectors, an intuitive way of visualizing them is as the parallelogram formed by the two vectors, where the order in which the vectors are multiplied determines the binary ambiguity in orientation (i.e. which side of the plane spanned by the vectors is "up"). Figure 4.1 shows examples of bivectors formed as the exterior product of vectors in $\mathbb{R}^2$. As one might guess, trivectors are simply oriented volumes in space, which can be visualized as parallelepipeds formed by three vectors. This analogy extends trivially to arbitrary dimensions, i.e. the exterior product of k vectors yields a k-dimensional oriented hypervolume.

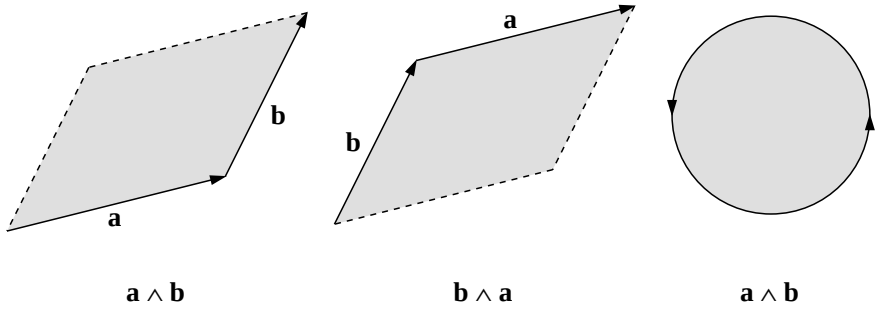


Figure 4.1: Graphical illustration of bivectors. The two examples of $\mathbf{a} \wedge \mathbf{b}$ are equivalent as they enclose the same amount of area and have the same orientation, whereas $\mathbf{b} \wedge \mathbf{a} = -(\mathbf{a} \wedge \mathbf{b})$.

Before discussing how to put the exterior product to use, let us first examine some of its properties. For vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, the following properties hold:

i. $\mathbf{a} \wedge \mathbf{a} = 0$
ii. $\mathbf{a} \wedge \mathbf{b} = -(\mathbf{b} \wedge \mathbf{a})$
iii. $r(\mathbf{a} \wedge \mathbf{b}) = (r\mathbf{a}) \wedge \mathbf{b} = \mathbf{a} \wedge (r\mathbf{b}), \quad r \in \mathbb{R}$
iv. $\mathbf{a} \wedge (\mathbf{b} + \mathbf{c}) = \mathbf{a} \wedge \mathbf{b} + \mathbf{a} \wedge \mathbf{c}$
v. $\mathbf{a} \wedge (\mathbf{b} \wedge \mathbf{c}) = (\mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{c}$

As can be seen, the exterior product bears a great deal of similarity with the cross product, but differs in several important ways. First, the exterior product is associative. Second, it operates on pairs of (multi)vectors of

arbitrary dimension. Third, the result of the exterior product lies in a space that is disjoint from the spaces associated with its domain, whereas the cross product maps $\mathbb{R}^3$ onto itself. Finally a word of caution: The anti-commutative property (ii) above holds for two *vectors*. When mixed grade multivectors are multiplied, one has to be careful. Consider, for example, the product of a vector $\mathbf{c}$ and a bivector $\mathbf{a} \wedge \mathbf{b}$:

$$-(\mathbf{c} \wedge (\mathbf{a} \wedge \mathbf{b})) = -(\mathbf{c} \wedge \mathbf{a}) \wedge \mathbf{b} = (\mathbf{a} \wedge \mathbf{c}) \wedge \mathbf{b} = \mathbf{a} \wedge \mathbf{c} \wedge \mathbf{b}$$

whereas applying property (ii) blindly would suggest $-(\mathbf{c} \wedge (\mathbf{a} \wedge \mathbf{b})) = (\mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{c} = \mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}$.

Given two vectors of dimension m, their exterior product is a bivector of dimension $m(m-1)/2$. More generally, the product of k m-vectors is a grade k element of dimension $\binom{m}{k} = \frac{m!}{k!(m-k)!}$. As a consequence, the product of m m-dimensional vectors is a 1-dimensional vector (called a *pseudoscalar*), and the product of more than m such vectors is always zero. Thus, the scalars, m-dimensional vectors, and the multivectors formed via transitive closure of the exterior product together form what is called a *graded* vector space of dimension 2^m, which is denoted $\bigwedge \mathbb{R}^m$. The natural basis for this vector space is typically expressed in terms of the exterior product. Before describing this basis, I will give some examples of applying the wedge product to vectors in $\mathbb{R}^3$, with the intent of constructing the basis for $\bigwedge \mathbb{R}^3$.

Let $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$. Using $\{\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \hat{\mathbf{e}}_3\}$ as the canonical orthonormal basis for $\mathbb{R}^3$, we can write

$$\begin{aligned}
\mathbf{a} \wedge \mathbf{b} &= (a_1\hat{\mathbf{e}}_1 + a_2\hat{\mathbf{e}}_2 + a_3\hat{\mathbf{e}}_3) \wedge (b_1\hat{\mathbf{e}}_1 + b_2\hat{\mathbf{e}}_2 + b_3\hat{\mathbf{e}}_3) \\
&= a_1b_1\hat{\mathbf{e}}_1 \wedge \hat{\mathbf{e}}_1 + a_1b_2\hat{\mathbf{e}}_1 \wedge \hat{\mathbf{e}}_2 + a_1b_3\hat{\mathbf{e}}_1 \wedge \hat{\mathbf{e}}_3 + \\
&\quad a_2b_1\hat{\mathbf{e}}_2 \wedge \hat{\mathbf{e}}_1 + a_2b_2\hat{\mathbf{e}}_2 \wedge \hat{\mathbf{e}}_2 + a_2b_3\hat{\mathbf{e}}_2 \wedge \hat{\mathbf{e}}_3 + \\
&\quad a_3b_1\hat{\mathbf{e}}_3 \wedge \hat{\mathbf{e}}_1 + a_3b_2\hat{\mathbf{e}}_3 \wedge \hat{\mathbf{e}}_2 + a_3b_3\hat{\mathbf{e}}_3 \wedge \hat{\mathbf{e}}_3 \\
&= (a_2b_3 - a_3b_2)\hat{\mathbf{e}}_2 \wedge \hat{\mathbf{e}}_3 + (a_3b_1 - a_1b_3)\hat{\mathbf{e}}_3 \wedge \hat{\mathbf{e}}_1 + (a_1b_2 - a_2b_1)\hat{\mathbf{e}}_1 \wedge \hat{\mathbf{e}}_2
\end{aligned} \tag{4.5}$$

Note how the properties $\hat{\mathbf{e}}_i \wedge \hat{\mathbf{e}}_i = 0$ and $\hat{\mathbf{e}}_i \wedge \hat{\mathbf{e}}_j = -\hat{\mathbf{e}}_j \wedge \hat{\mathbf{e}}_i$ have been used above. For convenience, I will sometimes use $\hat{\mathbf{e}}_{ij}$ to denote $\hat{\mathbf{e}}_i \wedge \hat{\mathbf{e}}_j$. As can be seen, the result of $\mathbf{a} \wedge \mathbf{b}$ is a 3-dimensional bivector whose components equal the components of $\mathbf{a} \times \mathbf{b}$, and we have $\|\mathbf{a} \times \mathbf{b}\| = \|\mathbf{a} \wedge \mathbf{b}\|$. This similarity is somewhat deceiving, however, because $\hat{\mathbf{e}}_1 \times \hat{\mathbf{e}}_2 = \hat{\mathbf{e}}_3 \neq \hat{\mathbf{e}}_1 \wedge \hat{\mathbf{e}}_2 = \hat{\mathbf{e}}_{12}$. Instead, $\hat{\mathbf{e}}_{12}$ lives in a subspace that is orthogonal to $\mathbb{R}^3$. For our purposes, it matters little how $\hat{\mathbf{e}}_{12}$ is represented (e.g. higher rank antisymmetric tensors are one possibility), as long as we treat it as having unit area and being orthogonal to each of $\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \hat{\mathbf{e}}_3$. That is, $\hat{\mathbf{e}}_i^{\mathsf{T}}\hat{\mathbf{e}}_j = \delta_{ij}$, where δ_{ij} is the Kronecker delta (Chapter 2). I will give one possible representation of $\bigwedge \mathbb{R}^3$, using vectors over an 8-dimensional basis, following the next example:

$$\begin{aligned}
\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c} &= (\mathbf{a} \wedge \mathbf{b}) \wedge \mathbf{c} \\
&= ((a_2b_3 - a_3b_2)\hat{\mathbf{e}}_{23} + (a_3b_1 - a_1b_3)\hat{\mathbf{e}}_{31} + (a_1b_2 - a_2b_1)\hat{\mathbf{e}}_{12}) \wedge (c_1\hat{\mathbf{e}}_1 + c_2\hat{\mathbf{e}}_2 + c_3\hat{\mathbf{e}}_3) \\
&= (a_2b_3c_1 - a_3b_2c_1)\hat{\mathbf{e}}_{23} \wedge \hat{\mathbf{e}}_1 + (a_3b_1c_2 - a_1b_3c_2)\hat{\mathbf{e}}_{31} \wedge \hat{\mathbf{e}}_2 + (a_1b_2c_3 - a_2b_1c_3)\hat{\mathbf{e}}_{12} \wedge \hat{\mathbf{e}}_3 \\
&= (a_2b_3c_1 - a_3b_2c_1 + a_3b_1c_2 - a_1b_3c_2 + a_1b_2c_3 - a_2b_1c_3)\hat{\mathbf{e}}_{123} \\
&= \det[\mathbf{a} \quad \mathbf{b} \quad \mathbf{c}]\,\hat{\mathbf{e}}_{123}
\end{aligned} \tag{4.6}$$

Since $\det[\mathbf{a} \quad \mathbf{b} \quad \mathbf{c}]$ is the (signed) volume of the parallelepiped formed by the three vectors, this result is consistent with the claim that the exterior product measures hypervolumes.

In this example, we have encountered yet another multivector: $\hat{\mathbf{e}}_{123} = \hat{\mathbf{e}}_1 \wedge \hat{\mathbf{e}}_2 \wedge \hat{\mathbf{e}}_3$. This trivector completes the $2^3 = 8$-dimensional orthonormal basis

$$\bigwedge \mathbb{R}^3 = \{1, \hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \hat{\mathbf{e}}_3, \hat{\mathbf{e}}_{23}, \hat{\mathbf{e}}_{31}, \hat{\mathbf{e}}_{12}, \hat{\mathbf{e}}_{123}\}$$

This basis is necessarily complete since the exterior product of $\hat{\mathbf{e}}_{123}$ with all other basis vectors yields either itself or zero. Thus, over $\mathbb{R}^3$, we can represent all scalars, vectors, bivectors, and trivectors as 8-dimensional vectors in this basis.

There is a geometric interpretation of the bivector $\mathbf{a} = \mathbf{b} \wedge \mathbf{c}$ ($\mathbf{b}, \mathbf{c} \in \mathbb{R}^3$) that I will make use of below. First, the L_2 norm of $\mathbf{a}$ equals the area of the parallelogram defined by $\mathbf{b}$ and $\mathbf{c}$. Second, the individual components of $\mathbf{a}$ correspond to the signed areas of the orthogonal projections of the parallelogram onto the corresponding planes defined by $\hat{\mathbf{e}}_i \wedge \hat{\mathbf{e}}_j$, analogous to how the components of a vector correspond to the signed lengths of the vector's projection onto $\hat{\mathbf{e}}_i$. For example, $a_1 b_2 - a_2 b_1$ is the signed area of the projection of the parallelogram onto the plane spanned by $\hat{\mathbf{e}}_1$ and $\hat{\mathbf{e}}_2$. This type of decomposition by projection onto subspaces is useful for visualizing tetrahedra embedded in $\mathbb{R}^4$ and higher dimensions, which we will encounter later when dealing with textured meshes.

Now that we know how to represent areas an volumes in arbitrary dimensions using the exterior product, I will present the memoryless simplification algorithm. I will first give a high level overview of the edge collapse algorithm, and then discuss the specifics of optimizing vertex positions and ordering the edge collapse sequence.

4.3 Iterative Edge Collapse

Similar to most recent simplification algorithms, the memoryless simplification method uses iterative edge collapse to coarsen a model. It does this by repeatedly selecting the edge with a minimum cost, collapsing this edge, and then re-evaluating the cost of edges affected by this edge collapse. Specifically, an edge collapse operation takes an edge $\bar{e} = \{v_1^{\bar{e}}, v_2^{\bar{e}}\}$ and substitutes its two vertices with a new vertex $\bar{v}$. In this process, the triangles $\lceil \bar{e} \rceil$ are collapsed to edges, and are discarded. The remaining edges and triangles incident upon $v_1^{\bar{e}}$ and $v_2^{\bar{e}}$, i.e. $\lceil \lfloor \bar{e} \rfloor \rceil \setminus \{\bar{e}\}$ and $\lceil \lceil \lfloor \bar{e} \rfloor \rceil \rceil \setminus \lceil \bar{e} \rceil$, respectively, are modified such that all occurrences of $v_1^{\bar{e}}$ and $v_2^{\bar{e}}$ are substituted with $\bar{v}$. Figure 4.2 illustrates the edge collapse operation.

The first step in the simplification process is to assign costs to all edges in the mesh, which are maintained in a priority queue. For each iteration, the edge with the lowest cost is selected and tested for candidacy; if the edge is not a valid candidate, it is removed from the queue. Most edge collapse algorithms avoid collapses that result in badly shaped meshes, such as degenerate or non-manifold topology, or geometric artifacts such as folds in the mesh [50]. Hoppe et al. [72] describe a set of topological constraints used to preserve the genus of the surface and to avoid introducing non-manifold simplices. The memoryless algorithm allows the user to specify whether such topological constraints should be used, and also allows geometrical constraints to be incorporated for preventing mesh folds, limiting edge lengths and triangle aspect ratios, etc. No constraints were used, however, for any of the simplified models presented in this chapter.

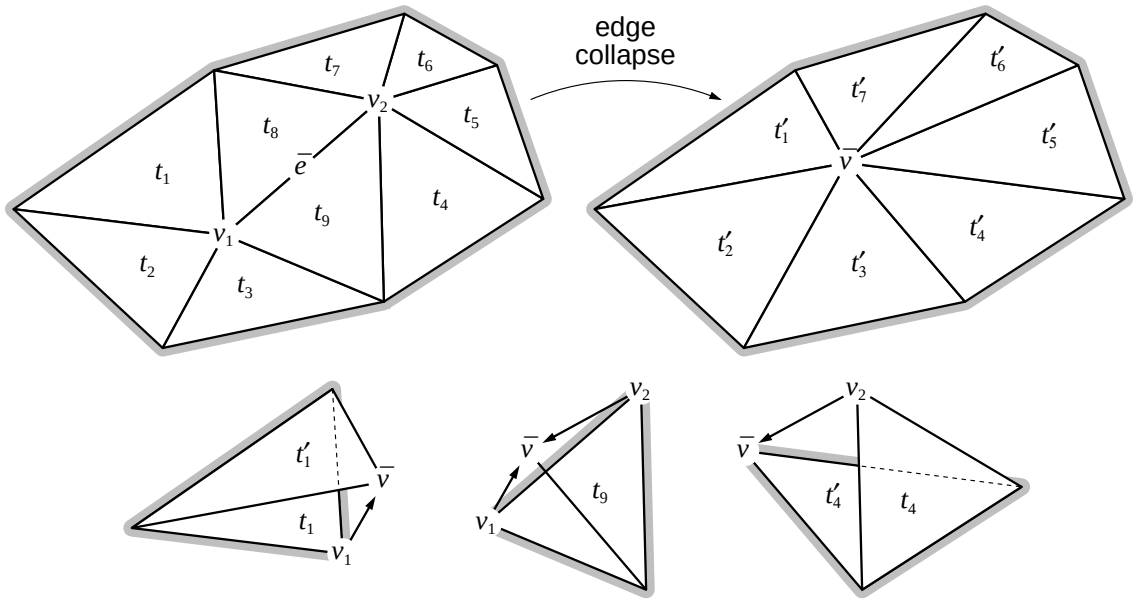


Figure 4.2: The edge collapse operation. The edge $\bar{e} = (v_1, v_2)$ is collapsed and replaced with a vertex $\bar{v}$, removing triangles t_8 and t_9. Example tetrahedral volumes associated with triangles t_1, t_4, and t_9 are shown.

Given a valid edge, the edge collapse is performed, followed by a re-evaluation of edge costs for all nearby edges affected by the collapse. If any of these edges were previously invalid candidates, they are re-inserted into the queue. As will be described below, the memoryless edge cost depends on the triangles $\lceil\lceil\lfloor\bar{e}\rfloor\rceil\rceil$ and their lower order simplices. Since all these triangles are potentially modified in an edge collapse, all other edges $\{e_i\}$ for which $\lceil\lceil\lfloor e_i\rfloor\rceil\rceil \cap \lceil\lceil\lfloor\bar{e}\rfloor\rceil\rceil \neq \varnothing$ must be updated, i.e. $\{e_i\} = \lceil\lfloor\lceil\bar{v}\rceil\rfloor\rceil$, with $|\{e_i\}| = 38$ assuming an average vertex valence of 6. Once the costs for $\{e_i\}$ have been updated, the next iteration begins, and the process is repeated until a desired number of simplices remain.

As discussed in Chapter 3, the general edge collapse method involves two decisions: choosing a measure that specifies the cost of collapsing an edge $\bar{e}$, and choosing the position $\mathbf{x}^{\bar{v}}$ of the vertex $\bar{v}$ that replaces the edge. This problem is intrinsically an optimization problem; given an error functional $Q(\mathbf{x})$ that specifies the cost of replacing $\bar{e}$ with $\bar{v}$, choose $\mathbf{x}^{\bar{v}}$ such that Q is minimized, i.e. $\mathbf{x}^{\bar{v}} = \operatorname{argmin}_{\mathbf{x}} f(\mathbf{x})$. The error functional presented here encapsulates volume and area information about a model. I will describe these geometric issues in greater detail in the following sections.

4.4 Memoryless Vertex Placement

In this and the following sections, I will focus on one of the two determining characteristics of the memoryless edge collapse algorithm, namely how to choose the optimal position $\mathbf{x}^{\bar{v}}$ of the vertex resulting from an edge collapse. I will later extend the solution to this problem to models with surface properties, allowing, for

example, high-quality simplification of textured meshes. As we shall see, the vertex placement scheme has a number of desirable properties; it is volume-preserving, it attempts to minimize the aggregate deviation, expressed in terms of tetrahedral volumes, between the two surfaces in each edge collapse, and it also explicitly accounts for changes made to surface boundaries by minimizing similar integral measures of deviation. All of these measures can be efficiently written as quadratic functionals, some which factor into the error functional, or cost of collapsing an edge. Because the error functional depends on several different factors, I have chosen to describe its different components and how to optimize each first, which should give the reader both some intuition behind the error metric and a detailed mathematical and geometrical description of its different pieces. I will then complete the description of the algorithm by describing how to incorporate and combine the different functionals into a single error functional, which is used to order the edge collapse sequence.

4.4.1 Linear Constraints

The approach taken here to choosing the position $\mathbf{x}^{\bar{v}}$ of the vertex that replaces the collapsed edge is to minimize one or more quadratic functionals $Q(\mathbf{x})$. The solution to such optimization problems is given by a linear system of equations $\hat{\mathbf{A}}\mathbf{x} = \hat{\mathbf{b}}$. In many cases, the linear system is either underconstrained, or the matrix $\hat{\mathbf{A}}$ is sufficiently ill-conditioned that a naive linear solver will not give a numerically robust solution. In Chapter 5, I will present a solution to this problem based on a *singular value decomposition* of $\hat{\mathbf{A}}$. For the memoryless simplification method, however, the solution is sometimes overconstrained because we will often want to minimize several quadratic functionals that each has its own minimum. The common approach to solving overconstrained systems is to perform a *least-squares* minimization, which, in our case, could be viewed as minimizing a linear combination of the functionals. Because some of our functionals are useful only in special cases, and are used mainly to produce a "preferable" solution when the system is otherwise underconstrained, it makes more sense to order them by importance and then minimize them one by one until a unique and robust solution is found.[2] That is, each minimization results in a set of linear constraints

$$\hat{\mathbf{a}}_i^{\mathsf{T}}\mathbf{x} = \hat{b}_i \tag{4.7}$$

and our goal is to build up a set of constraints $\mathcal{C}$ until three sufficiently independent ones are found, while performing a constrained quadratic minimization subject to the equality constraints already found. Thus, we can view the position $\mathbf{x}^{\bar{v}}$ as the intersection of three non-parallel planes in $\mathbb{R}^3$ that differ in orientation by some minimal threshold angle α.[3] Equation 4.7 gives the general form of such a plane equation. Since any given quadratic functional gives rise to at most three constraints, I will develop several functionals to ensure that enough constraints are available for a unique solution to exist. As mentioned previously, these constraints are computed and added in a pre-determined order of importance, which will be discussed in more detail in

[2] The SVD approach used in Chapter 5 could be adapted for this as well by minimizing the functionals in reverse. That is, the solution to one minimization would be projected onto the (possibly underconstrained) solution space of the next minimization. However, this would result in a slower algorithm.

[3] Not only do the plane constraints have to differ in orientation by α, but their normals $\hat{\mathbf{a}}_i$ must be sufficiently linearly independent, i.e. each $\hat{\mathbf{a}}_i$ must make a minimal angle α with the plane spanned by the other two normals.

§4.6. When three independent constraints

$$[\hat{\mathbf{a}}_1 \quad \hat{\mathbf{a}}_2 \quad \hat{\mathbf{a}}_3]^\mathsf{T}\mathbf{x} = \hat{\mathbf{A}}\mathbf{x} = \hat{\mathbf{b}} = \begin{bmatrix}\hat{b}_1 & \hat{b}_2 & \hat{b}_3\end{bmatrix}^\mathsf{T} \tag{4.8}$$

have been found, the new vertex position $\mathbf{x}^{\bar{v}}$ is computed as

$$\mathbf{x}^{\bar{v}} = \hat{\mathbf{A}}^{-1}\hat{\mathbf{b}} \tag{4.9}$$

To ensure that $\hat{\mathbf{A}}$ is well-conditioned, precautions must be taken before a new constraint is added to $\mathcal{C}$. Given $n < 3$ previous constraints $\hat{\mathbf{a}}_i^\mathsf{T}\mathbf{x} = \hat{b}_i$, $1 \le i \le n$, the new constraint $(\hat{\mathbf{a}}_{n+1}, \hat{b}_{n+1})$ is accepted according to the following rules:

$$\begin{aligned}
&n = 0: \ \hat{\mathbf{a}}_1 \neq \mathbf{0} \\
&n = 1: \ |\hat{\mathbf{a}}_1^\mathsf{T}\hat{\mathbf{a}}_2| < \|\hat{\mathbf{a}}_1\|\|\hat{\mathbf{a}}_2\| \cos\alpha \\
&n = 2: \ |[\hat{\mathbf{a}}_1, \hat{\mathbf{a}}_2, \hat{\mathbf{a}}_3]| > \max\{\|\hat{\mathbf{a}}_1\|\|\hat{\mathbf{a}}_2 \times \hat{\mathbf{a}}_3\|, \ \|\hat{\mathbf{a}}_2\|\|\hat{\mathbf{a}}_3 \times \hat{\mathbf{a}}_1\|, \ \|\hat{\mathbf{a}}_3\|\|\hat{\mathbf{a}}_1 \times \hat{\mathbf{a}}_2\|\} \sin\alpha
\end{aligned}$$

where α is the minimum permissible angle between the constraint planes.[4] These rules are more efficiently evaluated by squaring both sides, thereby eliminating the square roots associated with the vector norms. If $(\hat{\mathbf{a}}_{n+1}, \hat{b}_{n+1})$ meets these conditions, we say that it is α-compatible with the list of prior constraints. Experimental results have shown that the choice of α does not greatly influence the model quality, and that any reasonably small positive value can be used. For all results presented here, α has been set to 1°. The numerical accuracy is further improved by using double precision arithmetic throughout the calculations, and standard numerical techniques for matrix computations [54, 120] should be employed to eliminate large errors.

4.4.2 Quadratic Optimization

Several of the constraints used to position the vertex are obtained by minimizing some quadratic functional $Q(\mathbf{x})$ subject to a set of prior constraints. In fact, we can cast all subproblems related to choosing the new vertex as quadratic optimization problems. The quadratic functionals can be written in the following form:

$$\begin{aligned}
Q(\mathbf{x}) &= \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} - 2\mathbf{b}^\mathsf{T}\mathbf{x} + c \\
&= \begin{bmatrix}\mathbf{x}^\mathsf{T} & 1\end{bmatrix} \begin{bmatrix}\mathbf{A} & -\mathbf{b} \\ -\mathbf{b}^\mathsf{T} & c\end{bmatrix} \begin{bmatrix}\mathbf{x} \\ 1\end{bmatrix} \\
&= \bar{\mathbf{x}}^\mathsf{T}\mathbf{Q}\bar{\mathbf{x}}
\end{aligned} \tag{4.10}$$

with $2\mathbf{A}$ being the symmetric positive semidefinite Hessian of Q (which one can verify by differentiating Q twice), and $\mathbf{Q}$ another symmetric positive semidefinite matrix. In general, the dimensions of $\mathbf{Q}$ are one higher than the dimensions of $\mathbf{x}$. In the next few sections, $\mathbf{x} \in \mathbb{R}^3$ is the position of the vertex resulting from an edge collapse, and $\mathbf{Q}$ is a 4×4 matrix. Garland and Heckbert [50] refer to $\mathbf{Q}$ as a *quadric matrix*, and I

[4] Note that the rule for $n = 2$ has been modified slightly from [94, 95] to make it independent of the order in which constraints are introduced, and to guarantee that the three constraints are mutually α-compatible.

will adopt their terminology. Note, however, that the actual makeup of the quadric matrices presented here and theirs differ. I will expand on the connection to quadrics in §4.4.8. As we shall see below, the matrix $\mathbf{Q}$ arises in one of two ways:

$$\mathbf{Q}' = \sum_i {\mathbf{N}^i}^\mathsf{T} \sum_i \mathbf{N}^i \tag{4.11}$$

$$\mathbf{Q}'' = \sum_i {\mathbf{N}^i}^\mathsf{T} \mathbf{N}^i \tag{4.12}$$

resulting in two different quadratic functionals Q' and Q'', respectively. Here $\mathbf{N}^i$ is a $d \times 4$ matrix that encapsulates a piece of information about the local geometry of the mesh. The dimension d, the makeup of $\mathbf{N}^i$, as well as what each summation is over depend on what is being minimized. From the equations above, we see that $\mathbf{Q}'$ is an outer product of a sum, and $\mathbf{Q}''$ is a sum of outer products. For each functional, I will generally derive only the matrix $\mathbf{N}^i$, from which the two functionals Q' and Q'' can readily be constructed.

Assuming a quadratic functional Q is given, the goal of our optimization is to minimize Q subject to the set of prior linear constraints $\hat{\mathbf{A}}\mathbf{x} = \hat{\mathbf{b}}$. This is a linear problem that can be solved analytically. In the unconstrained case, we can find the minimum simply by solving

$$\tfrac{1}{2}\nabla Q = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$$

which is a set of linear equations, where $\mathbf{A}$ and $\mathbf{b}$ are the same as in Equation 4.10. When $\mathbf{x}$ is constrained, a more general approach is needed, however. Given $n < 3$ constraints $(\hat{\mathbf{a}}_i, \hat{b}_i)$, let $\hat{\mathbf{B}}$ be a $3 - n$ by 3 matrix with rows orthogonal to each other and to the vectors $\hat{\mathbf{a}}_i$.[5] Since each constraint $\hat{\mathbf{a}}_i^\mathsf{T}\mathbf{x} = \hat{b}_i$ limits $\mathbf{x}$ to a plane, the search for the minimum of Q is constrained to moving $\mathbf{x}$ parallel to each of these planes, i.e. orthogonal to each $\hat{\mathbf{a}}_i$. Thus $\mathbf{x}$ is only allowed to move in directions given by linear combinations of the rows of $\hat{\mathbf{B}}$, and the constrained minimization of Q amounts to finding its minimum in the subspace spanned by $\hat{\mathbf{B}}$. Then the additional $3 - n$ constraints are

$$\tfrac{1}{2}\hat{\mathbf{B}}\nabla Q = \hat{\mathbf{B}}(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{0} \tag{4.13}$$

That is, the constrained minimum of Q is found where the projection of its gradient onto the space of free search directions spanned by $\hat{\mathbf{B}}$ vanishes (Figure 4.3). The $3 - n$ linear constraints given by Equation 4.13 are added to $\mathcal{C}$ provided they satisfy the compatibility rules.

Throughout the remainder of this chapter, I will assume that $\bar{e}$ is the edge to be collapsed and $\bar{v}$ is the replacement vertex, positioned at $\mathbf{x}^{\bar{v}}$. I will distinguish between the final, known position $\mathbf{x}^{\bar{v}}$ of $\bar{v}$ and the general position $\mathbf{x}$ that is to be optimized. There are some important neighborhoods of simplices around $\bar{e}$ and $\bar{v}$ that I will use frequently. $T = \lceil\lceil\lfloor\bar{e}\rfloor\rceil\rceil$ are the triangles surrounding $\bar{e}$, $E = \partial\lceil\lfloor\bar{e}\rfloor\rceil$ are the boundary edges (if any) of the changing region, and $V = \lfloor\lceil\bar{v}\rceil\rfloor \setminus \{\bar{v}\}$ are the vertices adjacent to $\bar{v}$.

[5] The actual choice of basis vectors in $\hat{\mathbf{B}}$ does not matter as long as they satisfy the orthogonality requirement. In $\mathbb{R}^3$, the cross product can be used to construct a set of orthogonal vectors.

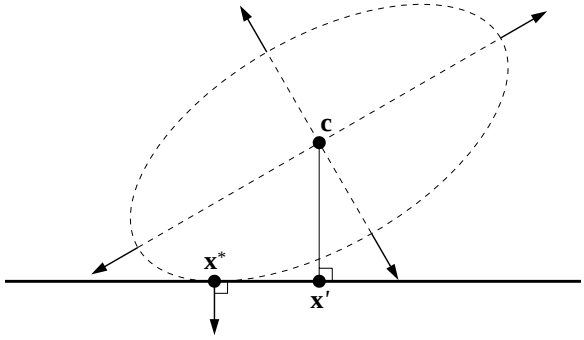


Figure 4.3: Linearly constrained quadratic optimization. The thick horizontal line corresponds to part of a linear constraint $\hat{\mathbf{a}}^{\mathsf{T}}\mathbf{x} = \hat{b}$. The dashed ellipse is an isocontour of a quadratic functional $Q(\mathbf{x})$ with minimum $\mathbf{c}$. The constrained minimum $\mathbf{x}^*$ of Q is found where the gradient ∇Q, whose direction is given by the arrows, is orthogonal to the linear subspace, i.e. where the elliptical level curve is tangent to the constraint line. The point $\mathbf{x}'$ illustrates that the orthogonal projection of $\mathbf{c}$ onto the subspace is generally a suboptimal solution, even though this is the closest point to the minimum $\mathbf{c}$ that satisfies the constraint.

4.4.3 Volume Preservation

One important geometric characteristic of a closed surface is the amount of volume it encloses. To demonstrate this, consider the 2D case of simplifying a finely tessellated circular curve to a polygon with a handful of sides. Most metrics for comparing polygonal curves (e.g. L_2 and L_∞ over some suitable homeomorphism between the curves) will report larger errors for an inscribing or circumscribing polygonal approximation than if roughly equal parts of the simplified curve are on either side of the circle, i.e. if the two shapes enclose roughly the same amount of area. This suggests that we would like the errors to be *unbiased*, i.e. the amount of error inside and outside should cancel. This analogy applies to 3D as well, for which volume preservation becomes important.[6] Intuitively, a volume-preserving simplification scheme would seem to produce more "plausible" models as the simplification process can be thought of as molding the original, smooth shape into a more faceted shape without adding or subtracting material. I will provide ample empirical evidence below to further support the importance of volume preservation.

In order to preserve the volume enclosed by the surface, first note that an edge collapse affects only a small portion of the mesh, and the global change in volume can be expressed entirely in terms of how the surface changes locally around the collapsed edge. As an edge $\bar{e} = (v_1^{\bar{e}}, v_2^{\bar{e}})$ is collapsed to a single vertex $\bar{v}$, the geometry of the triangles $T = \lceil\lceil\lfloor\bar{e}\rfloor\rceil\rceil$ surrounding $\bar{e}$ is generally affected. For a triangle $t \in T$ with vertices $(\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$, let $\mathbf{x}_1^t$ correspond to $v_1^{\bar{e}}$. Consider sliding $v_1^{\bar{e}}$ along a straight line from its current position $\mathbf{x}_1^t$ to its new position $\mathbf{x}^{\bar{v}}$. In doing so, t will sweep out a tetrahedral volume defined by the four vertices $(\mathbf{x}^{\bar{v}}, \mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$, as illustrated in Figure 4.2. If $\mathbf{x}^{\bar{v}}$ lies "outside" of the surface, i.e. on the side given by the outward pointing normal of t,[7] then t contributes a positive change in volume as $\bar{e}$ is collapsed.

[6] It is worthwhile to point out that there are applications for which volume preservation is not desirable, such as the *progressive hull* construction that creates a sequence of nested simplified surfaces [132]. However, volume preservation is useful for applications that demand geometric closeness.

[7] The orientation of the triangle normal $\hat{\mathbf{n}}^t$ is determined by the order in which its three vertices are specified. As long as this order

Similarly, if $\mathbf{x}^{\bar{v}}$ is "inside" the model, then the change in volume is negative. By ensuring that the sum of changes in volume over the triangles T vanishes, the total volume enclosed by the surface will remain the same.

For simplicity, I have here assumed that the surface is closed (or "watertight") and manifold. In practice, many surfaces are non-manifold or have boundaries, and may not enclose a volume. I have also assumed that the surface does not intersect itself—there are several possible interpretations of what the volume enclosed by such a surface is. However, as we are only concerned with *changes* in volume, it is still meaningful to think of a hypothetical volume that is locally bounded by the triangles of the mesh, and which increases or decreases as the triangles move outward or inward.

To compute the signed volume of each tetrahedron, I will make use of the exterior product (see §4.2.2). Even though the volume of a tetrahedron is a signed scalar, I will use the more general term *oriented volume* and the symbol $\vec{V}$ to refer to it, noting that, in higher dimensions, the trivector $\vec{V}$ is not a signed pseudoscalar, but a multidimensional quantity that has a direction. Let $t \in T$ be a triangle in the neighborhood around $\bar{e}$. The oriented volume $\vec{V}^t$ of $(\mathbf{x}, \mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$ can then be written as:

$$\begin{aligned}\vec{V}^t &= \tfrac{1}{6}(\mathbf{x}_1^t - \mathbf{x}) \wedge (\mathbf{x}_2^t - \mathbf{x}) \wedge (\mathbf{x}_3^t - \mathbf{x}) \\ &= \tfrac{1}{6}\left(\mathbf{x}_1^t \wedge \mathbf{x}_2^t \wedge \mathbf{x}_3^t - (\mathbf{x}_1^t \wedge \mathbf{x}_2^t + \mathbf{x}_2^t \wedge \mathbf{x}_3^t + \mathbf{x}_3^t \wedge \mathbf{x}_1^t) \wedge \mathbf{x}\right)\end{aligned} \tag{4.14}$$

In $\mathbb{R}^3$, we have

$$\begin{aligned}\vec{V}^t &= \tfrac{1}{6}\left([\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t] - [\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}] - [\mathbf{x}_2^t, \mathbf{x}_3^t, \mathbf{x}] - [\mathbf{x}_3^t, \mathbf{x}_1^t, \mathbf{x}]\right)\hat{\mathbf{e}}_{123} \\ &= \tfrac{1}{6}\left([\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t] - (\mathbf{x}_1^t \times \mathbf{x}_2^t + \mathbf{x}_2^t \times \mathbf{x}_3^t + \mathbf{x}_3^t \times \mathbf{x}_1^t)^\mathsf{T}\mathbf{x}\right)\hat{\mathbf{e}}_{123} \\ &= \tfrac{1}{6}\begin{bmatrix}-(\mathbf{x}_1^t \times \mathbf{x}_2^t + \mathbf{x}_2^t \times \mathbf{x}_3^t + \mathbf{x}_3^t \times \mathbf{x}_1^t)^\mathsf{T} & [\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t]\end{bmatrix}\begin{bmatrix}\mathbf{x} \\ 1\end{bmatrix}\hat{\mathbf{e}}_{123}\end{aligned} \tag{4.15}$$

Now define

$$\mathbf{N}_V^t = \frac{1}{6}\begin{bmatrix}-(\mathbf{x}_1^t \times \mathbf{x}_2^t + \mathbf{x}_2^t \times \mathbf{x}_3^t + \mathbf{x}_3^t \times \mathbf{x}_1^t)^\mathsf{T} & [\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t]\end{bmatrix} \tag{4.16}$$

$\mathbf{N}_V^t$ is a 1×4 matrix determined entirely by the vertices of the triangle t. The total change in volume $\vec{V}$ is then the sum of the tetrahedral volumes associated with the triangles T, which can be written as

$$\vec{V} = \sum_t \vec{V}^t = \sum_t \mathbf{N}_V^t \bar{\mathbf{x}}\,\hat{\mathbf{e}}_{123} \tag{4.17}$$

To preserve the volume, we require $\vec{V} = 0$, which, since $\vec{V}$ is linear in $\mathbf{x}$, can be written in the form $\hat{\mathbf{a}}^\mathsf{T}\mathbf{x} = \hat{b}$, i.e. it is a single linear constraint. In other words, there exists an entire plane of solutions $\mathbf{x}$ that allow the volume to be preserved, and we may place $\mathbf{x}$ anywhere in this plane. However, some choices of $\mathbf{x}$ are likely to be preferable over others, which I will address in the following two subsections. Before doing so, I will propose an alternative expression for $\mathbf{x}$ that leads to the same volume-preserving linear constraint. The

is determined in a consistent manner, it does not matter which side of the mesh is labeled inside or outside. The convention used here is, for example, opposite that of [94, 95].

general idea is that solving a linear equation $ax = b$ for x is equivalent to finding the x that minimizes $(ax - b)^2$. In the context of volume preservation, this simply means that instead of solving $\vec{V} = 0$, we would like to minimize $\|\vec{V}\|^2$:

$$Q'_V(\mathbf{x}) = \vec{V}^\mathsf{T}\vec{V} = \left(\sum_t \mathbf{N}_V^t \bar{\mathbf{x}}\right)^\mathsf{T} \left(\sum_t \mathbf{N}_V^t \bar{\mathbf{x}}\right) = \bar{\mathbf{x}}^\mathsf{T} \left(\sum_t {\mathbf{N}_V^t}^\mathsf{T} \sum_t \mathbf{N}_V^t\right) \bar{\mathbf{x}} = \bar{\mathbf{x}}^\mathsf{T} \mathbf{Q}'_V \bar{\mathbf{x}} \tag{4.18}$$

This is simply a quadratic functional of the same form as Equation 4.10, whose minimum is given by Equation 4.13. Since $\mathbf{Q}'_V$ is an outer product of two vectors, it is a rank 1 matrix, and the submatrix $\mathbf{A}$ associated with it (Equation 4.10) also has rank 1. Consequently, $\mathbf{Ax} = \mathbf{b}$ (Equation 4.13) does not have a unique solution and yields at most one constraint since the remaining ones are equivalent (and, thus, not α-compatible; see §4.4.1), i.e. this is the same constraint (or some scalar multiple of it) as the one given by $\vec{V} = 0$.

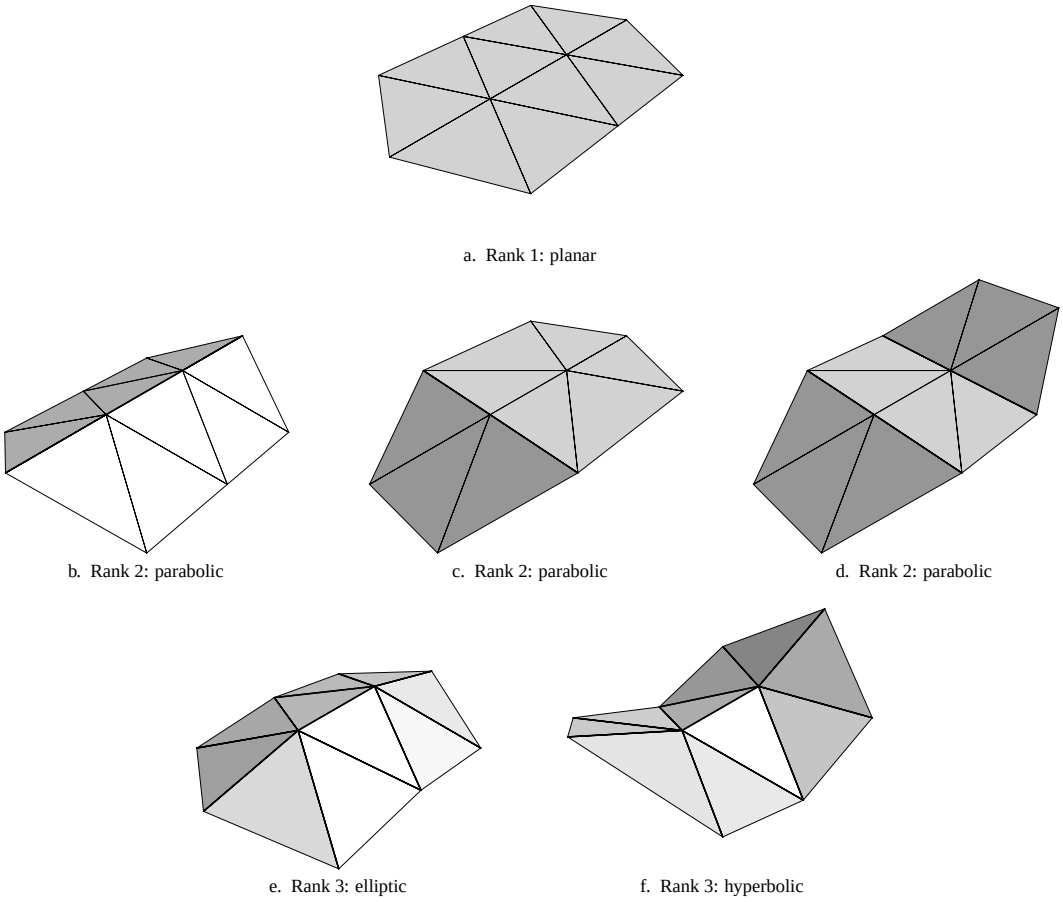
It may seem that I have unnecessarily complicated matters by introducing a quadratic functional and a number of additional and redundant steps to obtain a linear equation that was already given to us. My reasons for doing so are two-fold: First, by associating a quadratic functional with volume preservation, I will maintain symmetry with boundary preservation (described below). Later on, I will also discuss how to preserve surface properties, in which case volume preservation must be written in terms of minimization of a functional. Secondly, by using the functional notation, we are able to treat all subproblems of vertex positioning as minimizing quadratic functionals—one may even want to combine them into a single functional as a weighted sum, which is indeed the approach taken below for defining the error functional associated with each edge collapse. In essence, the quadratic functional notation is a more general form of setting up the problem, for which we can use a single unified framework to produce solutions.

4.4.4 Volume Optimization

One way of thinking about the volumetric tetrahedral elements discussed in the previous section is as an integral measure of error, i.e. each tetrahedron corresponds to the aggregate displacement of points between a pair of corresponding triangles. One of the reasons why volume preservation is desirable is that it keeps cascading errors to a minimum. If the vertex placement were not volume-preserving, we may inadvertently end up accumulating errors along a single direction, for example by repeatedly inflating the model. Thus, volume preservation ensures that there is no bias in the error introduced by a single edge collapse, and that any inflation is always offset by an opposite deflation. Since we still have some freedom in choosing the position $\mathbf{x}$ of the replacement vertex, we would like to not only eliminate any bias in the error, but to also minimize the error itself, i.e. we want to minimize the size of each tetrahedral volume. The most natural approach to doing this is to introduce yet another quadratic functional: the sum of squared volumetric changes. We can write this as:

$$Q''_V(\mathbf{x}) = \sum_t \vec{V}^{t\mathsf{T}} \vec{V}^t = \bar{\mathbf{x}}^\mathsf{T} \left(\sum_t {\mathbf{N}_V^t}^\mathsf{T} \mathbf{N}_V^t\right) \bar{\mathbf{x}} = \bar{\mathbf{x}}^\mathsf{T} \mathbf{Q}''_V \bar{\mathbf{x}} \tag{4.19}$$

Note the subtle difference between Q''_V—the sum of squared changes in volume—and Q'_V (Equation 4.18)—the squared sum of changes in volume. Since $\mathbf{Q}''_V$ is a sum of outer products, its top left 3×3 submatrix $\mathbf{A}$



a. Rank 1: planar

b. Rank 2: parabolic

c. Rank 2: parabolic

d. Rank 2: parabolic

e. Rank 3: elliptic

f. Rank 3: hyperbolic

Figure 4.4: Examples of submeshes with different rank Hessians for Q''_V. The edges drawn as thick lines make non-zero dihedral angles, i.e. their two incident triangles are not coplanar. Note that whereas the geometry can be preserved in 4.4b and 4.4c by collapsing the interior centermost edge to a point somewhere on the thick lines where two planar regions meet, this is not possible in 4.4d. The terms "parabolic," "elliptic," and "hyperbolic" are borrowed from differential geometry, where they are used to characterize points as having zero, positive, and negative Gaussian curvature, respectively.

(one half the Hessian of Q''_V) is generally a full-rank matrix that gives rise to three constraints. Specifically, the rank of $\mathbf{A}$ depends on whether the surface is locally planar (rank 1), parabolic (rank 2), or elliptic or hyperbolic (rank 3), as illustrated in Figure 4.4. These constraints are considered one at a time, in no particular order, for addition to the set of constraints $\mathcal{C}$ by testing them for α-compatibility.

4.4.5 Triangle Shape Optimization

If the surface is locally planar around the edge, then Q_V'' does not have a unique minimum and yields a single constraint that is equivalent to the constraint given by volume preservation (i.e. the plane defined by the local geometry). Even if the surface is not exactly planar, but close enough to planar that no two constraints are α-compatible, or if it is parabolic (e.g. cylindrical), then the solution is underconstrained, and we can use another functional to produce a desirable solution. When the surface geometry can be preserved exactly, such as when the region is planar, a sensible goal would be to position $\mathbf{x}^{\bar{v}}$ such that the resulting triangulation is as "well-shaped" as possible, i.e. to optimize the aspect ratios of the triangles and make them as near equilateral as possible. Thin sliver triangles are undesirable, for example, in finite element methods, and may introduce shading artifacts and rendering overhead for graphics applications.[8] The fact that many geometric algorithms rely on the *Delaunay triangulation* of a set of points to maximize triangle aspect ratios also suggests that well-shaped triangulations are important. Based on these observations, I will use the following expression from [57] as a measure of triangle quality:

$$q^t = \frac{4\sqrt{3}\,A^t}{L(e_1^t)^2 + L(e_2^t)^2 + L(e_3^t)^2}$$

where A^t is the unsigned area of triangle t, and $L(e_j^t)$ is the length of the j^{th} edge of t. For symmetry with volume and boundary optimization, I will sometimes write $\vec{L}^v$ as the vector associated with a directed edge $e = (\bar{v}, v)$. Using the expression above, an equilateral triangle has a quality of one—the highest possible. For a set T of triangles, I will use the following quality measure[9] for the entire set

$$q^T = \frac{4\sqrt{3}\sum_{t\in T} A^t}{\sum_{t\in T}\sum_{j=1}^{3} L(e_j^t)^2} \tag{4.20}$$

Since we have assumed that the neighborhood of the edge $\bar{e}$ is such that collapsing it will not affect the geometry of the surface, the total area of the triangles surrounding $\bar{v}$ is independent of $\mathbf{x}$ as long as $\mathbf{x}$ is placed in the *kernel* [26] of the polygon described by the boundary of $\bar{v}$'s incident triangles.[10] While it is possible to construct polygons with empty kernels as well as underconstrained situations in which the geometry does change, those cases are fairly rare in practice. Thus, assuming the numerator in Equation 4.20 is constant, the quality of the neighborhood is determined only by the denominator, which we would then want to minimize in order to maximize q^T.

For any triangle t incident upon $\bar{v}$, only the lengths of two of its edges depend on the position of $\bar{v}$, i.e. the edges $\{\bar{v}, v\}$ connecting the surrounding vertices $V = \lfloor\lceil\bar{v}\rceil\rfloor \setminus \{\bar{v}\}$. Therefore, the sum of squared lengths of

[8] In regions of high curvature, however, sliver triangles are sometimes needed to accurately represent the underlying surface.

[9] It is not difficult to come up with other aggregate quality measures involving q^t that penalize degenerate and ill-shaped triangles more heavily. However, the measure used here has a special structure that makes it easy to work with.

[10] A similar argument can be made when the surface is not planar, such as when $\bar{e}$ lies along the intersection of two non-parallel planes.

these edges is minimized:

$$\begin{aligned} Q''_S(\mathbf{x}) &= \sum_v \vec{L}^{v\mathsf{T}} \vec{L}^v \\ &= \sum_v (\mathbf{x}^v - \mathbf{x})^\mathsf{T} (\mathbf{x}^v - \mathbf{x}) \\ &= \sum_v \left(\begin{bmatrix} -\mathbf{I} & \mathbf{x}^v \end{bmatrix} \bar{\mathbf{x}} \right)^\mathsf{T} \left(\begin{bmatrix} -\mathbf{I} & \mathbf{x}^v \end{bmatrix} \bar{\mathbf{x}} \right) \\ &= \bar{\mathbf{x}}^\mathsf{T} \left(\sum_v \begin{bmatrix} -\mathbf{I} & \mathbf{x}^v \end{bmatrix}^\mathsf{T} \begin{bmatrix} -\mathbf{I} & \mathbf{x}^v \end{bmatrix} \right) \bar{\mathbf{x}} \end{aligned} \tag{4.21}$$

By defining

$$\mathbf{N}_S^v = \begin{bmatrix} -\mathbf{I} & \mathbf{x}^v \end{bmatrix} \tag{4.22}$$

the triangle shape functional can be written in the form of Equation 4.12. Carrying out the sum in Equation 4.21, it is easy to show that the unconstrained minimum of Q''_S is given by the solution to

$$|V|\mathbf{I}\mathbf{x} = \sum_v \mathbf{x}^v$$

That is, the optimum is simply the centroid of the vertices surrounding $\bar{v}$. This result makes sense intuitively as moving $\mathbf{x}$ away from the centroid would tend to stretch the mesh and make some or all of its incident triangles skinnier. As a physical interpretation, Q''_S corresponds to the total energy of a set of springs with zero rest length attached between $\bar{v}$ and its neighbors, similar to the E_{spring} term minimized in [72]. Note that shape optimization will unconditionally yield three orthogonal, and thus mutually α-compatible, constraints. Therefore, regardless of the local geometry, this minimization is guaranteed to yield a unique solution, and no additional functionals are necessary.

4.4.6 Boundary Preservation

It is quite common for polygonal surfaces to have *boundaries*, either by design or due to topological degeneracies. Figure 4.9a is an example of a range scanned model with surface boundaries. The physical object has two circular holes in the base of the model, and the computerized model has two additional jagged holes which are due to missing samples from the scanning device that captured the model. Other examples of surfaces with boundaries include isolated surface patches such as triangulated height fields, which don't necessarily enclose a volume. Surface boundaries may also arise from topology-modifying mesh processing algorithms, e.g. by teasing non-manifold edges apart, thereby creating a topologically disjoint surface. Whether intentional or not, these boundaries are often visually important because they form sharp discontinuities in rendered images, just like manifold silhouette edges do. Boundaries may also be geometrically important, e.g. they may be pieces of a larger surface that have to be stitched together, or they may have some intrinsic semantic importance. Finally, the boundaries do not necessarily have to be topological in nature, but

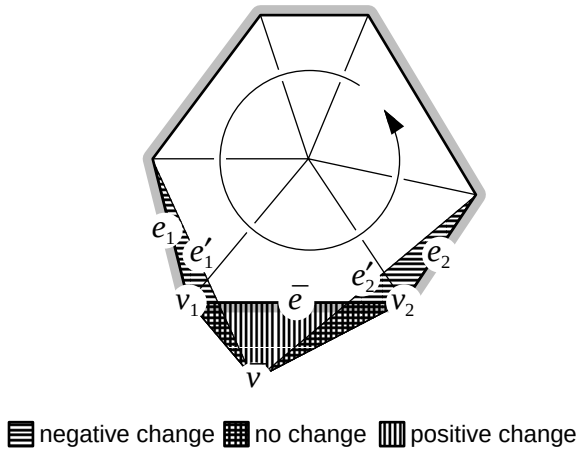


Figure 4.5: Area swept out by collapsing a boundary edge $\bar{e}$ to a vertex $\bar{v}$. The sum of signed areas of the hatched triangles is zero. The circular arc indicates the orientation of the boundary edges E.

may, for example, delineate material boundaries in a mesh. For these reasons, most surface simplification algorithms attempt to preserve the shape of boundaries and discontinuity curves on the mesh, e.g. [28, 50, 66]. Since boundary loops are one-dimensional, whereas surfaces are two-dimensional, preserving the shape of boundaries generally requires a different error metric and a separate algorithm. In this section, I will present an algorithm for boundary preservation that is based on the volume-preserving method described above, followed by a method for boundary optimization that uses the same principles as volume optimization. As it turns out, the term "boundary preservation" is somewhat of a misnomer in that we cannot always guarantee that the area enclosed by the boundary (for some meaningful definition of area) is preserved. However, for planar boundaries, the analogy is accurate, and the equations are set up in a similar manner. Hence, I will stick with the term "boundary preservation".

First, let us consider a closed planar boundary curve. A planar curve is the 2D analogue to the 3D triangle mesh. When an edge on a boundary is collapsed to a point in the same plane, the nearby affected boundary edges sweep out signed triangular areas, similar to how triangles on the surface of a mesh sweep out tetrahedra (Figure 4.5). Consequently, we can use the same techniques as above to exactly preserve the area enclosed by the curve, as well as minimize the sum of squared error regions swept out.

In practice, planar boundaries are rare, and we need a more general approach to preserving the boundary shape. I introduced a generalized method to this problem based on cross products as oriented areas in [94, 95], but provided little justification and intuition for why this method works. Armed with a more mathematically powerful tool—the exterior product—I will here give a more rigorous treatment of this problem, which should convince the reader of the soundness of this technique. I will first describe the mathematics of the algorithm in terms of the exterior product, and then discuss how to interpret the solution it produces.

In the case of planar curves, we must first acknowledge that the boundary is embedded in $\mathbb{R}^3$, so the problem must first be transformed (projected) to $\mathbb{R}^2$. One could simply determine the plane of the curve

and construct the appropriate rotation that brings it into some canonical plane, say the xy-plane, and then use the 2D equivalent of volume preservation to preserve the boundary area. Alternatively, one can take the approach in [94] and work in $\mathbb{R}^3$ using cross products. The idea is to establish a "positive" direction normal to the boundary plane. The signed area of a triangle in the boundary plane is then measured by computing the cross product of two of the edge vectors of the triangle, resulting in a vector parallel to the plane normal. The projection of the cross product onto the (unit) normal then gives twice the signed area. A perhaps more intuitive method is to utilize the exterior product in place of the cross product, which, by definition, yields an element of oriented area. The idea is to use sums of exterior products to accumulate changes in (oriented) area whether the triangular areas are coplanar or not.

Let $\bar{e} = (v_1^{\bar{e}}, v_2^{\bar{e}})$ be the collapsed edge, and let $\mathbf{x}$ be the position of the replacement vertex $\bar{v}$. Furthermore, let $e \in \partial\lceil\lfloor\bar{e}\rfloor\rceil$ with vertices $(\mathbf{x}_1^e, \mathbf{x}_2^e)$ be a directed boundary edge incident upon $v_1^{\bar{e}}$ or $v_2^{\bar{e}}$. When $\bar{e}$ is collapsed, e sweeps out an oriented area described by

$$\begin{aligned}\vec{A}^e &= \tfrac{1}{2}(\mathbf{x}_1^e - \mathbf{x}) \wedge (\mathbf{x}_2^e - \mathbf{x}) \\ &= \tfrac{1}{2}(\mathbf{x}_1^e \wedge \mathbf{x}_2^e - (\mathbf{x}_1^e - \mathbf{x}_2^e) \wedge \mathbf{x})\end{aligned} \tag{4.23}$$

Notice the resemblance of this equation to Equation 4.14. Using the fact that the exterior products above are over vectors in $\mathbb{R}^3$, we could work out a lengthy expression for $\vec{A}^e$ involving determinants of the vector components over the basis $\{\hat{\mathbf{e}}_{23}, \hat{\mathbf{e}}_{31}, \hat{\mathbf{e}}_{12}\}$. Instead, I will make use of an isomorphic mapping called the *Hodge dual* [100], which in $\mathbb{R}^3$ maps vectors to bivectors and vice versa, and which allows us to exchange exterior products and cross products:

$$\begin{aligned}\star(\mathbf{a} \wedge \mathbf{b}) &= \mathbf{a} \times \mathbf{b} \\ \star(\mathbf{a} \times \mathbf{b}) &= \mathbf{a} \wedge \mathbf{b}\end{aligned}$$

I will be using the Hodge dual for notational convenience only, as it allows me to use the more familiar cross product and remain in $\mathbb{R}^3$.[11] Thus, we can write

$$\begin{aligned}\vec{A}^e &= \tfrac{1}{2}(\mathbf{x}_1^e \wedge \mathbf{x}_2^e - (\mathbf{x}_1^e - \mathbf{x}_2^e) \wedge \mathbf{x}) \\ &= \star\tfrac{1}{2}(\mathbf{x}_1^e \times \mathbf{x}_2^e - (\mathbf{x}_1^e - \mathbf{x}_2^e) \times \mathbf{x}) \\ &= \star\tfrac{1}{2}\left[-[(\mathbf{x}_1^e - \mathbf{x}_2^e)\times] \quad \mathbf{x}_1^e \times \mathbf{x}_2^e\right]\bar{\mathbf{x}}\end{aligned} \tag{4.24}$$

where $[\mathbf{a}\times]$ is the 3×3 matrix such that $[\mathbf{a}\times]\mathbf{b} = \mathbf{a} \times \mathbf{b}$ (see Chapter 2). To preserve the oriented area, we would like to minimize the square $\vec{A}^{\mathsf{T}}\vec{A}$ of the residual[12] $\vec{A} = \sum_e \vec{A}^e$, which can be expressed in terms of the 3×4 matrix

$$\mathbf{N}_B^e = \tfrac{1}{2}\left[-[(\mathbf{x}_1^e - \mathbf{x}_2^e)\times] \quad \mathbf{x}_1^e \times \mathbf{x}_2^e\right] \tag{4.25}$$

[11] In particular, the cross product can be written as a linear transformation $[\cdot\times]$ in $\mathbb{R}^3$, whereas the linear transformation associated with the exterior product requires all $2^3 = 8$ dimensions.

[12] In least-squares fitting, the term "residual" is used both to refer to the deviation of each discrete sample point from the associated line or plane, as well as the sum of squared deviations. I will use "residual" to mean the sum of all *signed* deviations, or sum of changes in signed area or volume.

This matrix and Equation 4.12 are then used to define the quadratic functional Q'_B for boundary preservation. Since $\|\mathbf{a}\| = \|\star\mathbf{a}\|$, the $\star$ operator can be excluded in the definition of $\mathbf{N}^e_B$. The matrix $[\mathbf{a}\times]$ has a one-dimensional null space (parallel to $\mathbf{a}$), so it is a rank 2 matrix. Consequently, boundary preservation results in at most two linear constraints.

4.4.6.1 Interpretation

Whereas in the case of volume preservation the functional is guaranteed to be zero-valued over an entire plane of points, thus ensuring that the volume can be preserved, no such guarantees can be made for boundary "preservation". The only case in which we know with certainty that Q'_B is zero somewhere is if the boundary edges E all lie in the same plane. This automatically begs the question: If the boundary is non-planar, what is it that we are attempting to preserve, and what is it that we are minimizing? Specifically, what does the bivector $\vec{A} = \sum_e \vec{A}^e$ represent? I will answer these questions shortly. First, I will take a step back and consider a more fundamental question: What should our goals be for preserving the boundary shape, and what is a suitable error metric for measuring similarity?

For computer graphics applications, a reasonable measure of similarity for two boundary curves is if they *appear* similar. (I will here neglect issues such as occlusion by other parts of the mesh and assume that we are only concerned with the boundary curve itself.) Since such a measure of visual similarity depends on the vantage point of the viewer, one approach would be to identify a "most important" direction from which the curve is likely to be viewed. This approach is motivated by the fact that viewing a near planar curve edge-on conveys very little visual information, whereas the curve shape can be better appreciated when viewed from a direction closer to perpendicular to the curve. Based on this, one could project the curve onto the plane orthogonal to some "maximally important" direction, thereby transforming the 3D problem into two dimensions. From the discussion above, we know how to preserve area in two dimensions. To find a good projection plane, one could, for example, fit a least-squares plane to the boundary edges or vertices, or one could choose a plane such that its projection does not self-intersect [26]. For highly non-planar and more complex boundaries, however, there is often no single plane that captures all of the geometry of the curve. Take for instance the saddle-shaped closed curve $\begin{bmatrix}\cos\theta & \sin\theta & \cos^2\theta - \sin^2\theta\end{bmatrix}$ which has large area projections in most directions. For a more general solution, one could project the curve onto *all possible planes*, evaluate the chosen error metric in each plane, and integrate the errors over the Gauss sphere, thus accounting for all possible directions. In the context of area preservation, the error would simply be the unsigned residual area in the plane left over after summing up the signed changes, which we can ensure is zero for any fixed plane, but which generally cannot be guaranteed for all planes. An ideal simplification method would then minimize the aggregate error, measured, for example, as the L_p norm of the residuals. I will show below how the boundary preservation technique presented above does the best job possible preserving the projected boundary area when integrated over all possible directions.

Let us now return to the bivector $\vec{A}^e = A^e_{23}\hat{\mathbf{e}}_{23} + A^e_{31}\hat{\mathbf{e}}_{31} + A^e_{12}\hat{\mathbf{e}}_{12}$. This quantity corresponds to the change in boundary area in the plane defined by the boundary edge e and the new vertex $\bar{v}$. As suggested in §4.2.2, the three components of $\vec{A}^e$ are the orthogonal projections of the swept out area onto the three planes

defined by $\{\hat{\mathbf{e}}_{23}, \hat{\mathbf{e}}_{31}, \hat{\mathbf{e}}_{12}\}$. In fact, these three unit bivectors constitute an orthonormal basis for representing (subsets of) planes of all possible orientations. Note that only the orientation part, and not the translation part, of the plane is relevant since we are performing orthogonal projections. Thus, we can project $\vec{A}^e$ onto any plane $\vec{P} = P_{23}\hat{\mathbf{e}}_{23} + P_{31}\hat{\mathbf{e}}_{31} + P_{12}\hat{\mathbf{e}}_{12}$ using an inner product

$$\vec{P}^{\mathsf{T}}\vec{A}^e = P_{23}A^e_{23} + P_{31}A^e_{31} + P_{12}A^e_{12}$$

Summing over all triangular areas, we have

$$\sum_e \vec{P}^{\mathsf{T}}\vec{A}^e = \vec{P}^{\mathsf{T}}\sum_e \vec{A}^e = \vec{P}^{\mathsf{T}}\vec{A} = \|\vec{P}\|\|\vec{A}\|\cos\varphi = \|\vec{A}\|\cos\varphi$$

where φ is the angle between the bivectors $\vec{P}$ and $\vec{A}$. Since we are not interested in the magnitude of $\vec{P}$, I have chosen it to be of unit area. We can compute the L_p norm[13] of the projected residual area by integrating over all possible projection planes $\vec{P}$. In polar coordinates, this becomes an integration over the Gauss sphere:

$$\begin{aligned}
\left(\int_S |\|\vec{A}\|\cos\varphi|^p \, dS\right)^{1/p} &= \left(\int_0^{\pi}\int_0^{2\pi} \|\vec{A}\|^p |\cos\varphi|^p \cos(\tfrac{\pi}{2} - \varphi)\; d\theta\, d\varphi\right)^{1/p} \\
&= \|\vec{A}\| \left(\int_0^{\pi}\int_0^{2\pi} |\cos\varphi|^p \sin\varphi \; d\theta\, d\varphi\right)^{1/p} \\
&= \|\vec{A}\| \left(\frac{4\pi}{p+1}\right)^{1/p}
\end{aligned}$$

As we can see, regardless of what p-norm we choose, the error is proportional to $\|\vec{A}\|$. This implies that, by minimizing $\|\vec{A}\|^2 = \vec{A}^{\mathsf{T}}\vec{A}$, we end up minimizing the change in area over all possible directions, which is what we set out to show.

To summarize, the boundary preservation procedure above can simply be viewed as the "best possible" attempt to preserve the 2D boundary area simultaneously in the three basis planes $\{\hat{\mathbf{e}}_{23}, \hat{\mathbf{e}}_{31}, \hat{\mathbf{e}}_{12}\}$, which is equivalent to minimizing the change in projected area over all possible directions.

4.4.7 Boundary Optimization

As we saw above, volume preservation and optimization are closely related. Both functionals are constructed from the same set of matrices $\{\mathbf{N}_V^t\}$; one is a product of sums, the other a sum of products. Analogous to this duality, we would like to have a method for minimizing the individual changes made to the boundary, in addition to limiting their net effect on the boundary area. Since we have already derived the matrix $\mathbf{N}_B^e$ used in computing the area of a triangle (Equation 4.25), we have done all the work required to set up this functional. That is, using Equations 4.12 and 4.25 together, we can define Q_B'', which measures the sum of squared areas $\sum_e \vec{A}^{e\mathsf{T}}\vec{A}^e$ of the swept out triangles. The associated Hessian matrix is a sum of outer products of rank 2 matrices, which yields three constraints as long as the boundary vertices near the collapsed edge are not collinear. While the methods for boundary and volume preservation are not completely analogous, boundary

[13] The L_1, L_2, and L_∞ norms are commonly used in metrics.

optimization is the 2D equivalent of the 3D volume optimization since it deals only with the magnitudes of the errors, and requires no representation of orientation of the errors.

4.4.8 Connection with Error Quadrics

As mentioned in §4.4.2, the quadratic functionals have the same structure as the error quadrics introduced by Garland and Heckbert [50]. In particular, and as pointed out in [49, 69, 95], the functional associated with volume optimization has a special connection with Garland and Heckbert's original quadrics. Their error metric measures the sum of squared distances from the replacement vertex $\bar{v}$ to the (infinite) planes associated with a set of triangles. Specifically, let $\mathbf{x}$ be the position of the replacement vertex and let t be a triangle with vertices $(\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$. Furthermore, define the following normal vectors of the triangle:

$$\begin{aligned} \mathbf{n}^t &= \mathbf{x}_1^t \times \mathbf{x}_2^t + \mathbf{x}_2^t \times \mathbf{x}_3^t + \mathbf{x}_3^t \times \mathbf{x}_1^t \\ \hat{\mathbf{n}}^t &= \frac{\mathbf{n}^t}{\|\mathbf{n}^t\|} = \frac{\mathbf{n}^t}{2A^t} \end{aligned}$$

where A^t is the unsigned area of the triangle and $\hat{\mathbf{n}}^t$ is a unit normal. Then the squared distance $Q^t(\mathbf{x})$ from $\mathbf{x}$ to t can be written as

$$\begin{aligned} Q^t(\mathbf{x}) &= \left(\hat{\mathbf{n}}^{t\mathsf{T}}(\mathbf{x}_1^t - \mathbf{x})\right)^2 \\ &= \left(\hat{\mathbf{n}}^{t\mathsf{T}}\mathbf{x}_1^t - \hat{\mathbf{n}}^{t\mathsf{T}}\mathbf{x}\right)^2 \\ &= \left(\frac{1}{2A^t}[\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t] - \frac{1}{2A^t}\mathbf{n}^{t\mathsf{T}}\mathbf{x}\right)^2 \\ &= \left(\frac{1}{2A^t}\left[-\mathbf{n}^{t\mathsf{T}} \quad [\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t]\right]\bar{\mathbf{x}}\right)^2 \\ &= \left(\frac{3}{A^t}\mathbf{N}_V^t\bar{\mathbf{x}}\right)^2 \\ &= \left(\frac{3}{A^t}\right)^2 \bar{\mathbf{x}}^{\mathsf{T}}\left(\mathbf{N}_V^{t\mathsf{T}}\mathbf{N}_V^t\right)\bar{\mathbf{x}} \end{aligned}$$

using the definition of $\mathbf{N}_V^t$ from Equation 4.16. This result should not be surprising as the volume of a tetrahedron is simply one third the area of one of its triangles times the orthogonal distance to the opposite vertex. Therefore, we can view the error metric in [50] as an instance of volume optimization, for which each tetrahedral volume is weighted by the inverse triangle area. Viewed the other way around, Garland and Heckbert's original scheme uses uniform weighting of the squared orthogonal distances, whereas the memoryless method uses weights proportional to the squared triangle area. Based on these similarities, we can construct a more general *weighted* quadric error metric:

$$Q^t(\mathbf{x}) = w^t\left(\left[-\hat{\mathbf{n}}^{t\mathsf{T}} \quad \frac{[\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t]}{2A^t}\right]\bar{\mathbf{x}}\right)^2$$

and we have $w^t = 1$ for Garland and Heckbert's original scheme, and $w^t = \frac{1}{9}{A^t}^2$ for volume optimization in the memoryless method.

In his Ph.D. thesis [49], Garland discusses yet another weighting scheme using the unsquared triangle area ($w^t = A^t$), which is also used by Hoppe [69] and Erikson and Manocha [41]. He shows how using these weights for simplification leads to triangle aspect ratios in the simplified model that, in the limit, approach the optimum (with respect to the geometry of the original surface) as the triangle area approaches zero [49, pp. 75–79].[14] In his derivation, he uses a Taylor series expansion for the inverse area, which ultimately leads to higher order terms that, in the limit, approach zero rapidly and can be discarded. Using the squared area weighting scheme presented here, no Taylor series expansion is necessary, and these higher order terms never appear. Consequently, one can show that the volume optimization functional results in the same optimal triangle shapes.

In addition to this difference in weighting, the methods differ in two other important ways. First, the quadrics used in the memoryless method are computed with respect to the *partially simplified* surface, whereas Garland and Heckbert compute quadrics for the original model and then accumulate them when vertices are merged. Thus, their quadrics, which I will call "memorizing quadrics," always measure the distance to the *original* model. I will emphasize this difference and illustrate the advantage of memoryless quadrics in §4.8.2. Second, the memoryless method uses volume preservation, which consistently leads to smaller errors, even when combined with other vertex placement schemes. I will discuss this second difference in §4.8.3.

4.5 Edge Priorities

Many edge collapse methods order the sequence of collapses to minimize the deviation between the simplified model and the *original* surface. In the memoryless algorithm, no such distance measure is available. However, I have already discussed functionals for volume and boundary optimization that are directly related to the relative error introduced in any particular edge collapse, which measure the deviation of the surfaces and boundaries between two *successive* edge collapse iterations. Based on these error measures, the best edge collapse in each iteration is the one that yields the smallest amount of change. Formally, the edge cost Q_C is written as

$$Q_C(\mathbf{x}^{\bar{v}}) = \lambda Q''_V(\mathbf{x}^{\bar{v}}) + (1-\lambda)L(\bar{e})^2 Q''_B(\mathbf{x}^{\bar{v}}) \tag{4.26}$$

The parameter $\lambda \in (0, 1]$ provides the user with explicit control over the tradeoff between surface and boundary fidelity. In general, $\lambda = \frac{1}{2}$ tends to work well. Later on, I will demonstrate the effects of varying λ for a model with boundaries. Note that for boundaryless surfaces, Q_C reduces to Q''_V—the sum of squared changes in volume.

As can be seen, the error functional Q_C used to prioritize edges is a convex combination of the volume and boundary functionals discussed in §4.4.4 and §4.4.7. The boundary term Q''_B is weighted by the square

[14] This property does not imply that the entire simplified mesh is globally optimal, but that the position $\mathbf{x}^{\bar{v}}$ of each replacement vertex $\bar{v}$ is chosen such that the neighborhood around $\bar{v}$ is optimally shaped when the remaining vertices are fixed.

of the length of the edge $\bar{e}$ to ensure that the terms involved have the same units, thereby guaranteeing scale invariance. There is no fundamental reason for taking this particular approach to achieve scale invariance, and other alternatives, such as prescaling the geometry to fit within the unit cube [72], are possible. I have found the method above to work well in practice, however, and it is straightforward to implement.

4.6 Summary of Vertex Placement

The last few sections on vertex placement and edge priorities are rather lengthy and contain plenty of technical material and long derivations. The purpose of this section is to summarize the results of the previous sections and to provide the final details of how they fit together.

First, I will list all of the functionals that we have encountered:

volume preservation
$$Q'_V(\mathbf{x}) = \sum_t \vec{V}^{t\mathsf{T}} \sum_t \vec{V}^t = \bar{\mathbf{x}}^\mathsf{T} \left(\sum_t \mathbf{N}_V^{t\mathsf{T}} \sum_t \mathbf{N}_V^t \right) \bar{\mathbf{x}}$$

volume optimization
$$Q''_V(\mathbf{x}) = \sum_t \vec{V}^{t\mathsf{T}} \vec{V}^t = \bar{\mathbf{x}}^\mathsf{T} \left(\sum_t \mathbf{N}_V^{t\mathsf{T}} \mathbf{N}_V^t \right) \bar{\mathbf{x}}$$

boundary preservation
$$Q'_B(\mathbf{x}) = \sum_e \vec{A}^{e\mathsf{T}} \sum_e \vec{A}^e = \bar{\mathbf{x}}^\mathsf{T} \left(\sum_e \mathbf{N}_B^{e\mathsf{T}} \sum_e \mathbf{N}_B^e \right) \bar{\mathbf{x}}$$

boundary optimization
$$Q''_B(\mathbf{x}) = \sum_e \vec{A}^{e\mathsf{T}} \vec{A}^e = \bar{\mathbf{x}}^\mathsf{T} \left(\sum_e \mathbf{N}_B^{e\mathsf{T}} \mathbf{N}_B^e \right) \bar{\mathbf{x}}$$

triangle shape optimization
$$Q''_S(\mathbf{x}) = \sum_v \vec{L}^{v\mathsf{T}} \vec{L}^v = \bar{\mathbf{x}}^\mathsf{T} \left(\sum_v \mathbf{N}_S^{v\mathsf{T}} \mathbf{N}_S^v \right) \bar{\mathbf{x}}$$

edge cost
$$\begin{aligned} Q_C(\mathbf{x}) &= \lambda \sum_t \vec{V}^{t\mathsf{T}} \vec{V}^t + (1-\lambda) L(\bar{e})^2 \sum_e \vec{A}^{e\mathsf{T}} \vec{A}^e \\ &= \bar{\mathbf{x}}^\mathsf{T} \left(\lambda \sum_t \mathbf{N}_V^{t\mathsf{T}} \mathbf{N}_V^t + (1-\lambda) L(\bar{e})^2 \sum_e \mathbf{N}_B^{e\mathsf{T}} \mathbf{N}_B^e \right) \bar{\mathbf{x}} \end{aligned}$$

Here the matrices $\mathbf{N}^i$ are defined as:

volume matrix
$$\mathbf{N}_V^t = \tfrac{1}{6} \begin{bmatrix} -(\mathbf{x}_1^t \times \mathbf{x}_2^t + \mathbf{x}_2^t \times \mathbf{x}_3^t + \mathbf{x}_3^t \times \mathbf{x}_1^t)^\mathsf{T} & [\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t] \end{bmatrix}$$

boundary matrix
$$\mathbf{N}_B^e = \tfrac{1}{2} \begin{bmatrix} -[(\mathbf{x}_1^e - \mathbf{x}_2^e)\times] & \mathbf{x}_1^e \times \mathbf{x}_2^e \end{bmatrix}$$

triangle shape matrix
$$\mathbf{N}_S^v = \begin{bmatrix} -\mathbf{I} & \mathbf{x}^v \end{bmatrix}$$

The indices t, e, and v are over the sets $T = \lceil \lfloor \bar{e} \rfloor \rceil$, $E = \partial \lceil \lfloor \bar{e} \rfloor \rceil$, and $V = \lfloor \lceil \bar{v} \rceil \rfloor \setminus \{\bar{v}\}$, respectively, where $\bar{e}$ is the edge being collapsed to vertex $\bar{v}$. The functionals are then minimized in the following order:

method	*functional*	*constraints*
1. volume/boundary preservation	Q'_V, Q'_B	$\leq 1+2$
2. edge cost	Q_C	≤ 3
3. triangle shape optimization	Q''_S	3

Since volume and boundary preservation together yield at most three constraints, the order in which they are performed does not matter. The functionals above are minimized one by one until three mutually compatible constraints have been found, at which point the solution $\mathbf{x}^{\bar{v}}$ can be computed using Equation 4.9.

4.7 Preservation of Surface Properties

In addition to geometry, many meshes have surface attributes attached to them. Examples include continuous attributes such as normals, colors, and texture coordinates, as well as discrete attributes such as material and texture indices. Typically, the continuous attributes are associated with the vertices of the model, and are linearly interpolated over the triangles. This is the type of attributes that I will consider in this section.

When a model is simplified, new surface properties must be computed for the new geometry. Specifically, after an edge $\bar{e}$ is collapsed, surface attributes must be assigned to the new vertex $\bar{v}$. This is a problem that has recently been given attention in the simplification community, and several methods have been proposed, including [6, 26, 27, 41, 51, 66, 69]. This problem is related to parameterizing the simplified surface and establishing a mapping with the original. For these tasks, the algorithms in [37, 88, 91] can be used. Several of these simplification methods optimize the geometry and surface attributes simultaneously, e.g. by treating the surface attributes as if they were coordinate axes in a higher dimensional geometric space, and then generalizing the error metric to higher dimensions. This typically necessitates the use of conversion factors to make geometry and attributes "compatible."

Rather than adopting one of the methods cited above, I will describe an algorithm that is based upon and generalizes the memoryless selection of vertex coordinates using the notion of volume preservation and optimization. This method, too, extends the vertex representation to higher dimensions, but allows the geometry to be decoupled from the surface properties. A nice benefit of this approach is that the algorithm for computing new surface properties is independent of the vertex placement method. In addition, because it is memoryless, the algorithm can be treated as a black box and can very easily be integrated with any edge collapse method. As a consequence, the error metric used to order edge collapses in my memoryless method is independent of attribute error. For simplicity, I will present the algorithm as it applies to texture coordinates (or any scalar attributes in $\mathbb{R}^2$), but the method can easily be extended to an arbitrary number of surface attributes.

4.7.1 Assignment of Texture Coordinates

Similar in spirit to [51, 69], I will extend the representation of vertices from $\mathbb{R}^3$ to $\mathbb{R}^5$, letting the additional two (orthogonal) coordinate axes represent texture space. As in [51, 69], the space is made scale-invariant, that is, a linear, uniform scaling κ is applied to the texture coordinates to make them "compatible" with the geometric coordinates. However, instead of scaling the model geometry to the unit cube, the scale factor κ is computed as the ratio of the average geometric edge length to the average texture space edge length. For models with multiple textures (or multiple parameterizations), a different κ is computed for each texture patch.

I have chosen to treat the texture coordinate computation as a black box that takes as input the position $[x_1^{\bar{v}} \quad x_2^{\bar{v}} \quad x_3^{\bar{v}}]^\mathsf{T}$ of the replacement vertex $\bar{v}$ and the geometry and texture coordinates of the triangles $T = \lceil\lceil\lfloor\bar{e}\rfloor\rceil\rceil$ surrounding the collapsed edge, and which outputs the (scaled) texture coordinates $[x_4^{\bar{v}} \quad x_5^{\bar{v}}]^\mathsf{T}$ for $\bar{v}$. Thus, in contrast to [51, 69], the position of $\bar{v}$ is not influenced by the texture coordinates. Rather, the "best" set of texture coordinates is chosen for a given vertex position. It is possible that optimizing both geometry and surface properties simultaneously would result in more optimal meshes, and extending the simplification algorithm in this manner would be reasonably straightforward.

My goal in this section is to formulate a method that generalizes the volume-preserving and per-triangle volume-minimizing properties of the vertex placement described in §4.4.3 and §4.4.4 to five dimensions. Consequently, we need to measure the tetrahedral volumes—embedded in a higher dimensional space $\mathbb{R}^5$—swept out by the triangles T. This problem is similar and in many ways analogous to preserving the area of surface boundaries in $\mathbb{R}^3$ by accounting for the orientation of each error area. In §4.4.6.1 I justified why this method works. I should be clear that this technique extends to n-simplices embedded in any dimensional space $\mathbb{R}^m$ ($m \geq n$), and I will show here how it applies to the case $n = 3, m = 5$ by using the exterior product to encode the volumetric changes as oriented trivectors.

From Equation 4.14, the oriented volume $\vec{V}^t$ of a tetrahedron $(\mathbf{x}, \mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$ defined by the new vertex $\bar{v}$ and the three vertices of a triangle $t \in T$ is

$$\begin{aligned}\vec{V}^t &= \tfrac{1}{6}(\mathbf{x}_1^t - \mathbf{x}) \wedge (\mathbf{x}_2^t - \mathbf{x}) \wedge (\mathbf{x}_3^t - \mathbf{x}) \\ &= \tfrac{1}{6}\left(\mathbf{x}_1^t \wedge \mathbf{x}_2^t \wedge \mathbf{x}_3^t - (\mathbf{x}_1^t \wedge \mathbf{x}_2^t + \mathbf{x}_2^t \wedge \mathbf{x}_3^t + \mathbf{x}_3^t \wedge \mathbf{x}_1^t) \wedge \mathbf{x}\right)\end{aligned}$$

The result is a $\binom{5}{3} = 10$-dimensional trivector $\vec{V}^t$ which is linear in the two unknowns x_4 and x_5 (the texture coordinates of $\mathbf{x}$). We can use the following basis for $\vec{V}^t$:

$$\{\hat{\mathbf{e}}_{123}, \hat{\mathbf{e}}_{124}, \hat{\mathbf{e}}_{125}, \hat{\mathbf{e}}_{134}, \hat{\mathbf{e}}_{135}, \hat{\mathbf{e}}_{145}, \hat{\mathbf{e}}_{234}, \hat{\mathbf{e}}_{235}, \hat{\mathbf{e}}_{245}, \hat{\mathbf{e}}_{345}\}$$

Thus the $\hat{\mathbf{e}}_{ijk}$ component of $\vec{V}^t$ is

$$V_{ijk}^t = \frac{1}{6}\left(\begin{vmatrix} x_{1i}^t & x_{1j}^t & x_{1k}^t \\ x_{2i}^t & x_{2j}^t & x_{2k}^t \\ x_{3i}^t & x_{3j}^t & x_{3k}^t \end{vmatrix} - \begin{vmatrix} x_{1i}^t & x_{1j}^t & x_{1k}^t \\ x_{2i}^t & x_{2j}^t & x_{2k}^t \\ x_i & x_j & x_k \end{vmatrix} - \begin{vmatrix} x_{2i}^t & x_{2j}^t & x_{2k}^t \\ x_{3i}^t & x_{3j}^t & x_{3k}^t \\ x_i & x_j & x_k \end{vmatrix} - \begin{vmatrix} x_{3i}^t & x_{3j}^t & x_{3k}^t \\ x_{1i}^t & x_{1j}^t & x_{1k}^t \\ x_i & x_j & x_k \end{vmatrix}\right) \tag{4.27}$$

where x_{ij}^t is the j^{th} component of the i^{th} vertex of triangle t. The texture coordinates can then be factored out as follows:

$$V_{ijk}^t = \begin{bmatrix} V_{ijk,1}^t & V_{ijk,2}^t & V_{ijk,3}^t \end{bmatrix} \begin{bmatrix} x_4 \\ x_5 \\ 1 \end{bmatrix} \tag{4.28}$$

where the coefficients in the row vector are expressions of known constants. As an example, the factorization

of V^t_{345} is

$$
\begin{aligned}
V^t_{345,1} &= \frac{1}{6}\left(\begin{vmatrix} x^t_{13} & x^t_{15} \\ x^t_{23} & x^t_{25} \end{vmatrix} + \begin{vmatrix} x^t_{23} & x^t_{25} \\ x^t_{33} & x^t_{35} \end{vmatrix} + \begin{vmatrix} x^t_{33} & x^t_{35} \\ x^t_{13} & x^t_{15} \end{vmatrix}\right) \\
V^t_{345,2} &= -\frac{1}{6}\left(\begin{vmatrix} x^t_{13} & x^t_{14} \\ x^t_{23} & x^t_{24} \end{vmatrix} + \begin{vmatrix} x^t_{23} & x^t_{24} \\ x^t_{33} & x^t_{34} \end{vmatrix} + \begin{vmatrix} x^t_{33} & x^t_{34} \\ x^t_{13} & x^t_{14} \end{vmatrix}\right) \\
V^t_{345,3} &= -\frac{1}{6}\left(\begin{vmatrix} x^t_{14} & x^t_{15} \\ x^t_{24} & x^t_{25} \end{vmatrix} + \begin{vmatrix} x^t_{24} & x^t_{25} \\ x^t_{34} & x^t_{35} \end{vmatrix} + \begin{vmatrix} x^t_{34} & x^t_{35} \\ x^t_{14} & x^t_{15} \end{vmatrix}\right) x_3 + \frac{1}{6}\begin{vmatrix} x^t_{13} & x^t_{14} & x^t_{15} \\ x^t_{23} & x^t_{24} & x^t_{25} \\ x^t_{33} & x^t_{33} & x^t_{33} \end{vmatrix}
\end{aligned}
$$

Thus, the trivector $\vec{V}^t$ can be written as

$$
\vec{V}^t = \begin{bmatrix} V^t_{123} \\ V^t_{124} \\ \vdots \\ V^t_{345} \end{bmatrix} = \begin{bmatrix} V^t_{123,1} & V^t_{123,2} & V^t_{123,3} \\ V^t_{124,1} & V^t_{124,2} & V^t_{124,3} \\ \vdots & \vdots & \vdots \\ V^t_{345,1} & V^t_{345,2} & V^t_{345,3} \end{bmatrix} \begin{bmatrix} x_4 \\ x_5 \\ 1 \end{bmatrix} = \mathbf{N}^t_T \begin{bmatrix} x_4 \\ x_5 \\ 1 \end{bmatrix} \tag{4.29}
$$

where $\mathbf{N}^t_T$ is a 10×3 matrix. We have here arrived at a matrix form for the oriented volume, where the unknown texture coordinates have been factored out. We can then use our general framework for defining quadratic functionals for volume-based texture coordinate preservation (Q'_T) and optimization (Q''_T):

$$
Q'_T(x_4, x_5) = \begin{bmatrix} x_4 & x_5 & 1 \end{bmatrix} \left(\sum_t {\mathbf{N}^t_T}^\mathsf{T} \sum_t \mathbf{N}^t_T\right) \begin{bmatrix} x_4 \\ x_5 \\ 1 \end{bmatrix} \tag{4.30}
$$

$$
Q''_T(x_4, x_5) = \begin{bmatrix} x_4 & x_5 & 1 \end{bmatrix} \left(\sum_t {\mathbf{N}^t_T}^\mathsf{T} \mathbf{N}^t_T\right) \begin{bmatrix} x_4 \\ x_5 \\ 1 \end{bmatrix} \tag{4.31}
$$

and employ the quadratic minimization procedure described in §4.4.2 to obtain the optimal texture coordinates. Here Q'_T signifies the square of the residual oriented volume (if any), and Q''_T is the sum of squared changes in volume. As in the case of vertex placement, the new texture coordinates are computed by minimizing Q'_T first. If the solution is underconstrained, Q''_T is then minimized.

Many surface attributes are valid only for certain ranges. Colors, for example, usually have lower and upper intensity bounds, and surface normals are typically constrained to be of unit length. Rather than attempting to perform a non-linear constrained optimization of these attributes, or constantly forcing them to lie within their bounds, such constraints are handled using clamping and re-normalization during a post-processing pass, after the model has been simplified.

The method presented here is for the specific case of a pair of scalar attributes, i.e. $m = 5$. For $l = m - 3$ surface attributes, the trivector $\vec{V}^t$ has dimension $d = \binom{l+3}{3} = \frac{1}{6}(l+1)(l+2)(l+3) = O(l^3)$, and the $d \times (l+1)$ matrix $\mathbf{N}^t_T$ generates an $(l+1) \times (l+1)$ quadric matrix. Consequently, deriving the surface attributes for a vertex has time complexity $O(l^4)$ (the time needed to compute the quadric matrix), and can

become fairly expensive computationally when the mesh is augmented with more than a few different surface attributes. As a comparison, the methods described in [51] and [69] both require solving a system of $l+3$ linear equations, which generally takes $O(l^3)$ time. More recently, Hoppe and Marschner [73] show that the special structure of the matrix in [69] allows a more efficient $O(l)$ algorithm for computing the surface attributes. Even though the algorithm presented here is theoretically more computationally expensive than either of these other two methods, the matrix $\mathbf{N}_T^t$ is computed from scratch for each (potential) edge collapse, whereas Garland and Heckbert's method requires storing $\frac{1}{2}(l+4)(l+5)$ matrix coefficients with each vertex, making their method much less memory efficient. For textured models ($l=2$), empirical results have shown the memoryless method to be at least as fast as the QSlim implementation of [51]. These results as well as qualitative comparisons between my own attribute preserving method and the one in [51] appear in §7.3.

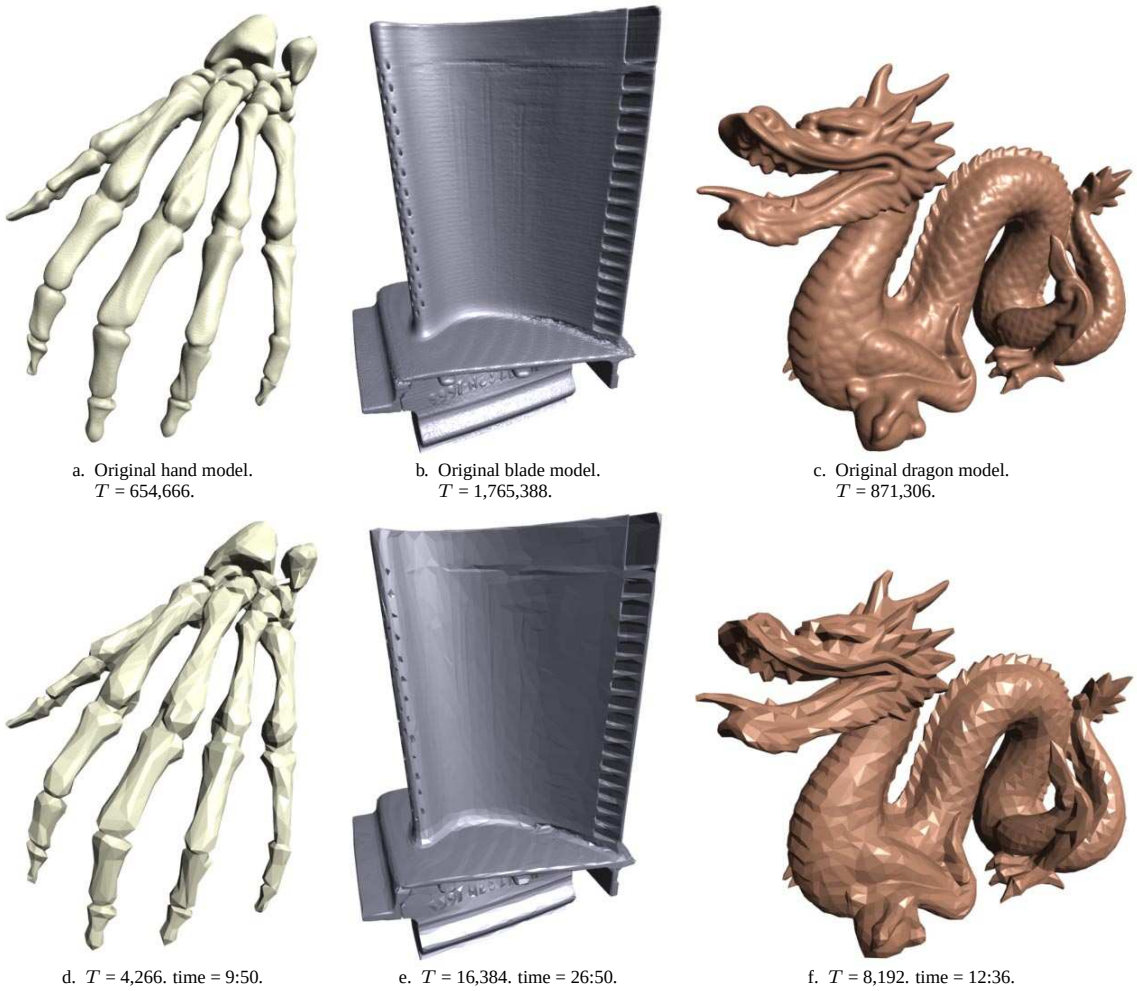
4.8 Results

In this section, I will demonstrate the effectiveness of the memoryless algorithm by presenting images of simplified models, error graphs, and timing results. In addition, I will make several qualitative and quantitative comparisons of the method against implementations of other well-known simplification algorithms, followed by a discussion of some of the reasons why the memoryless method performs as well as it does. In §4.8.2, I will compare the use of memoryless error quadrics with Garland and Heckbert's method which computes the quadrics once for the original model and then accumulates them. Then, in §4.8.3, I will show the importance of volume preservation by integrating it with several different vertex placement schemes, which consistently results in lower errors. Because the memoryless method is so effective, it makes for a good benchmark for future comparisons with new simplification methods. Indeed, I will include several additional results of memoryless simplified models in Chapters 5, 7, and 8.

Before comparing the results of the different methods, I will briefly highlight some of the results of the memoryless algorithm. Figures 4.6 and 4.7 illustrate the results of memoryless simplification of four different—and rather large—models. Figure 4.6a shows a range scan of a Buddha model that consists of over one million triangles. This model was simplified to 20,000, 5,000, and 1,000 triangles (compare with Figure 10 in [51]). Each model took less than 20 minutes to simplify, translating into a simplification speed of roughly 1,000 triangles eliminated per second. Even at 20,000 triangles—a reduction of over 98%—the simplified model retains virtually all of the features in the original model, although at this resolution the model is visibly faceted when rendered using per-face normals and flat shading. Figure 4.7 illustrates three other models of comparable complexity, which were reduced to a few thousand polygons at a rate of $\sim 1,100$ triangles per second. Again, even at these drastic simplification ratios, the simplified models are fair approximations of the originals (compare, for example, Figure 4.7e with Figure 9 in [134]). Several of these test models will be used again in the next few chapters to allow comparisons.

a. Original model. T = 1,087,716. b. T = 20,000. time = 19:32. c. T = 5,000. time = 19:33. d. T = 1,000. time = 19:09.

Figure 4.6: Memoryless simplifications of Buddha model.



a. Original hand model. T = 654,666.
b. Original blade model. T = 1,765,388.
c. Original dragon model. T = 871,306.
d. T = 4,266. time = 9:50.
e. T = 16,384. time = 26:50.
f. T = 8,192. time = 12:36.

Figure 4.7: Memoryless simplifications of hand, blade, and dragon models.

4.8.1 Comparison with Previous Methods

In order to determine the relative performance of the memoryless method, I will compare it against a number of published methods:

- Mesh Optimization [72].
- Progressive Meshes [66].
- Simplification Envelopes v1.1 [28].
- JADE v2.1 [23].
- QSlim v2.0 [50].

These algorithms are generally considered to be among the state-of-the-art in simplification. Public domain implementations made by the authors of all these algorithms were obtained, except for Progressive Meshes for which Hugues Hoppe generously offered to provide simplified models. All of the models included in this section were simplified on a single processor of a a four-processor, 195 MHz R10000 Silicon Graphics Onyx^2 machine with 1 GB of main memory.[15]

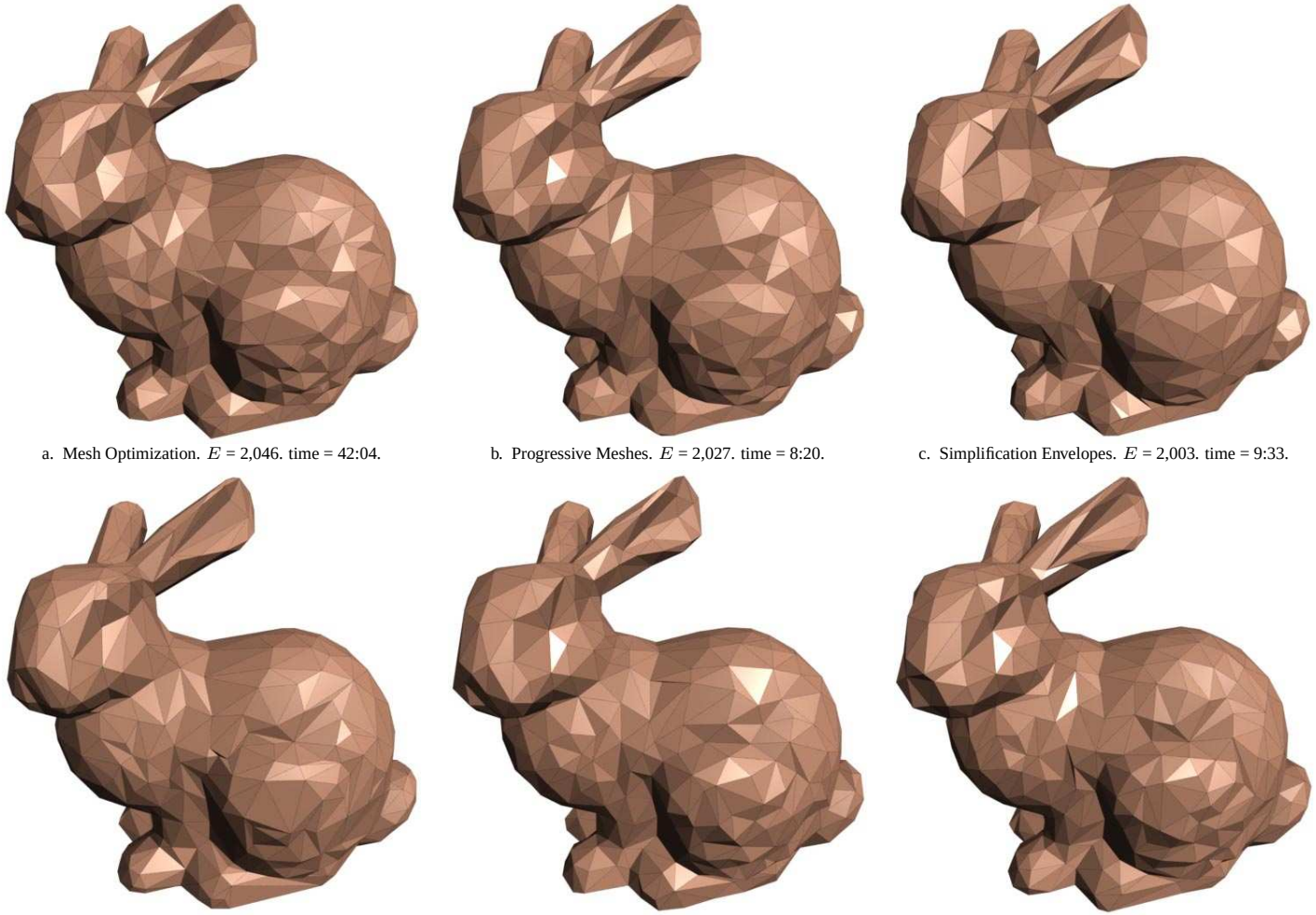
4.8.1.1 Qualitative Assessment

Four test objects were used to evaluate the speed and quality of the different methods: a bunny and a horse model, which both were created using laser range scanning, and two synthetic objects constructed from dense samplings of a sphere and a spherical segment (a subset of a sphere that has boundaries). Six levels of detail were created for the bunny and horse model using each method while making sure that the complexity of the simplified models was roughly the same among the different methods. For models without boundaries, the number of triangles was used as the measure of complexity. For models that do have boundaries, the number of edges is a more meaningful measure of complexity as it accounts for both the surface and boundary complexity. Therefore, the edge count was kept the same for these models.

Simplified versions of the bunny and horse model are shown at select levels of detail in Figures 4.8, 4.9, and 4.10. The visual quality of the bunnies in Figure 4.8 does not vary greatly between the different simplification methods. However, one can easily observe a qualitative difference between the models produced by edge collapse methods (Figures 4.8a, 4.8b, 4.8e, and 4.8f) and the ones resulting from vertex removal (Figures 4.8c and 4.8d). Notice how Simplification Envelopes and JADE produce models that appear smoother, as there is less variation in shading over these two surfaces, whereas the edge collapse methods produce surfaces of a distinctly different character. The horses in Figure 4.10 reveal another difference; the edge collapse methods generally produce more uniform triangle shapes, whereas the vertex removal methods often produce sliver triangles, which can be seen, for example, on the head of the horse in Figure 4.10e and on the farthest ear of the bunny in Figure 4.8c. These results are due to the fact that the edge collapse methods allow the vertex locations to be optimized, resulting in more desirable triangulations and allowing high curvature features to be preserved more accurately. Such wrinkled features are ironed out by the vertex removal methods since the vertices are not allowed to move. We will see later how optimizing the vertices consistently leads to lower average errors.

Figure 4.9 shows the undersides of the bunny models. Notice the four dark "holes" in the mesh, marked by the red boundary edges, where there is no surface information. The purpose of this figure is to illustrate each method's ability to preserve the shape of surface boundaries. The difference in quality between several of the methods is readily apparent. In particular, Mesh Optimization caused significant distortion of the boundaries, whereas JADE seemingly over-emphasized their importance. Progressive Meshes and the memoryless method ($\lambda = 1/2$) produced models with many fewer boundary edges by completely eliminating the small sawtooth-like features. Still, their boundary curves convey the same basic shape as the models produced by

[15]The *Progressive Mesh* models were simplified on a one-processor, 195 MHz R10000 Silicon Graphics Octane with 256 MB of main memory. For the purpose of simplifying the relatively small models used in the comparison, these two machines are comparable in performance.



a. Mesh Optimization. E = 2,046. time = 42:04.

b. Progressive Meshes. E = 2,027. time = 8:20.

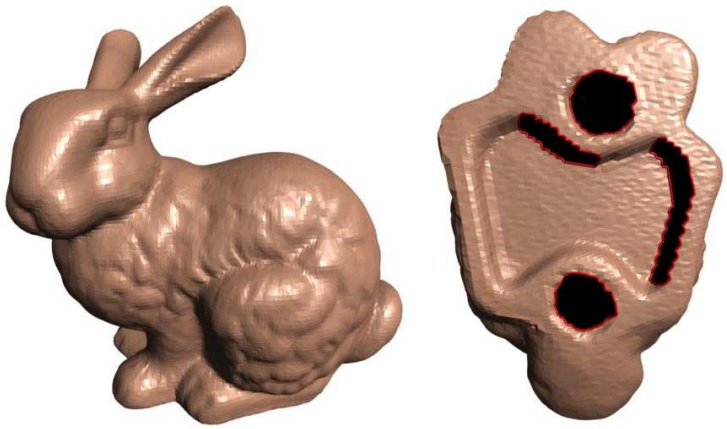
c. Simplification Envelopes. E = 2,003. time = 9:33.

d. JADE. E = 1,983. time = 4:55.

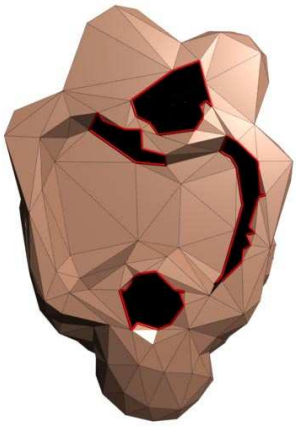
e. QSlim. E = 2,000. time = 0:22.

f. Memoryless Simplification. E = 2,025. time = 1:05.

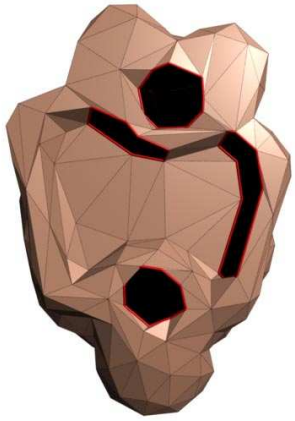
Figure 4.8: Side view of bunny model. See Figure 4.9 for original model.



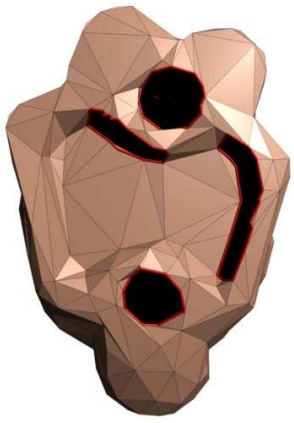
a. Original model. E = 104,288. E_∂ = 223.



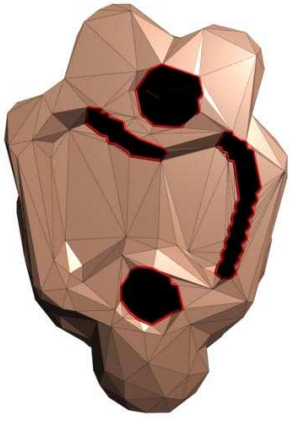
b. Mesh Optimization. E_∂ = 66.



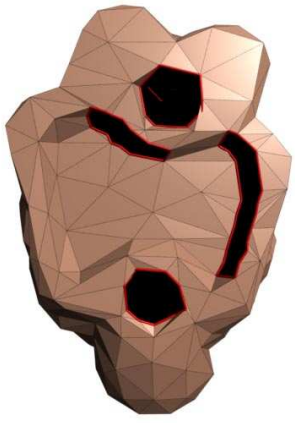
c. Progressive Meshes. E_∂ = 40.



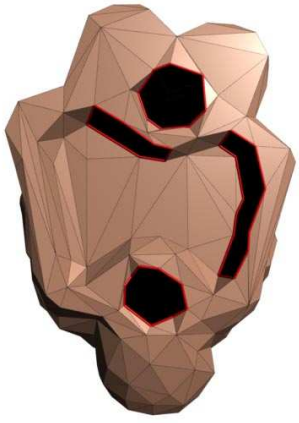
d. Simplification Envelopes. E_∂ = 64.



e. JADE. E_∂ = 99.

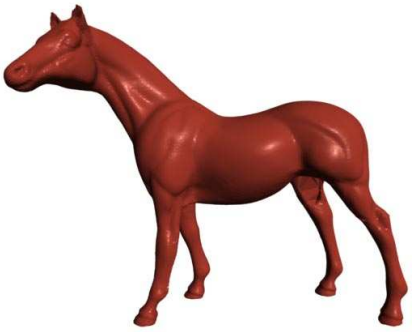


f. QSlim. E_∂ = 61.

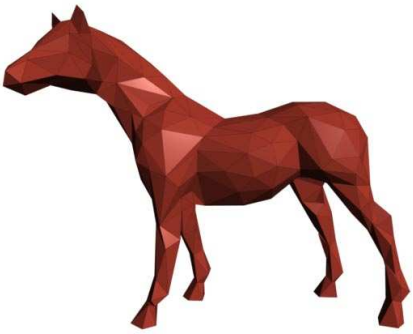


g. Memoryless Simplification. E_∂ = 42.

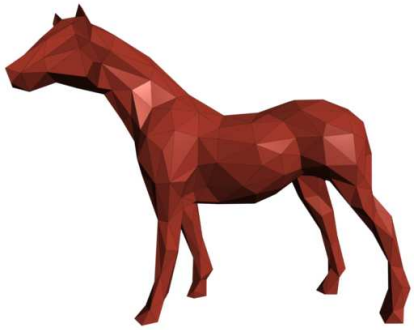
Figure 4.9: Bottom view of bunny model. The red edges correspond to the boundary edges of the model. The number of boundary edges, E_∂, is indicated for each model.



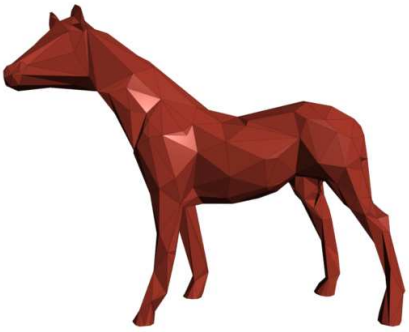
a. Original model. T = 96,966.



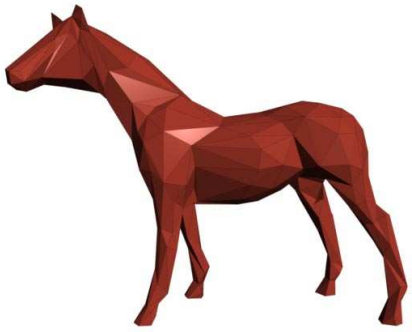
b. Mesh Optimization. T = 598. time = 1:07:27.



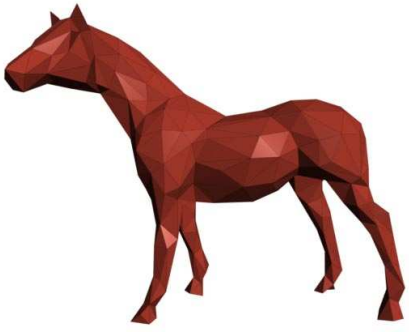
c. Progressive Meshes. T = 596. time = 19:41.



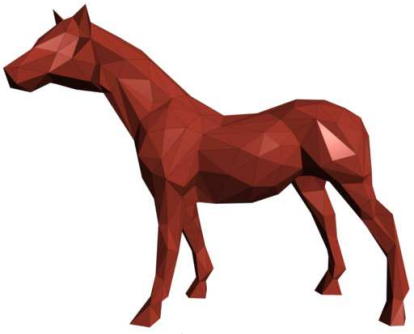
d. Simplification Envelopes. T = 810. time = 13:41.



e. JADE. T = 592. time = 8:04.



f. QSlim. T=596. time = 0:11.



g. Memoryless Simplification. T = 596. time = 1:36.

Figure 4.10: Horse model.

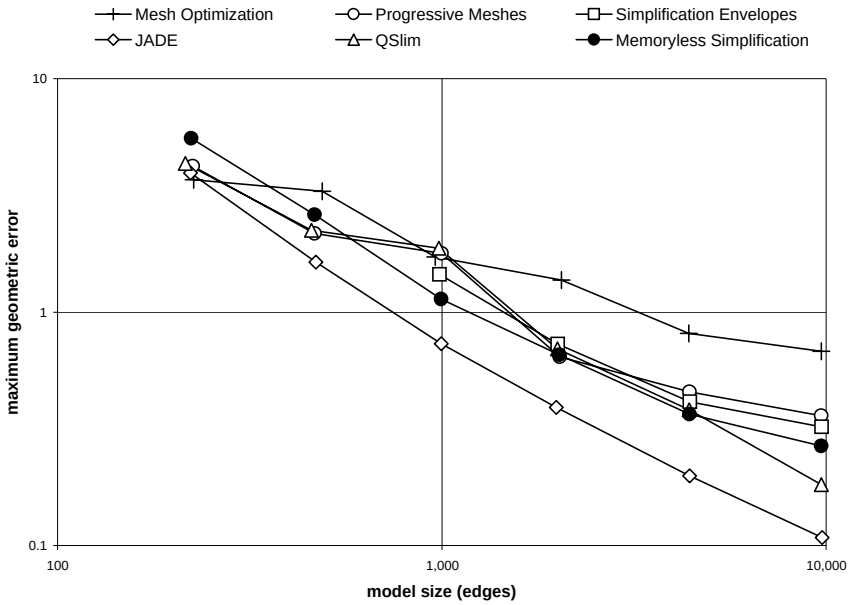


Figure 4.11: Maximum geometric errors for the bunny model.

Simplification Envelopes and QSlim, which allocated roughly 50% more edges to the boundaries. While the number of boundary edges varies greatly between these methods, keep in mind that most of them have user-selectable parameters for determining the relative importance between surface and boundary fidelity (such as the parameter λ in §4.5), and the default settings used to simplify the models are probably not tuned for the set of models used in this comparison.

4.8.1.2 Geometric Error

The visual quality evaluation above, while important, is somewhat subjective, and covers only a small set of models. A more objective comparison can be made by measuring the geometric error between the simplified models and the original. While several error measures exist (see §3.3), I have chosen the maximum (d_∞) and mean (d_1) errors as reported by the *Metro* public domain software [24],[16] Metro samples a large number of points on one surface and measures the smallest distance of each point to the other surface, but optionally allows this error to be measured symmetrically by sampling both surfaces. Thus, the maximum error reported by Metro is equivalent to the Hausdorff error for sufficiently dense samplings. Unfortunately, Metro does not report the deviation between boundary curves (if present). I have developed a similar tool for this purpose which uses the same error measure as Metro, but which only samples points on the boundaries and measures the distance to the closest boundary edge.

[16]Metro version 2.5 was used with the options `-s` for symmetric error evaluation and `-t` for textual output. The errors reported here were normalized by dividing by the length of the longest diagonal of the smallest axis-aligned bounding box of the *original* model. By default, Metro reports percentage errors with respect to the simplified model, which often leads to slightly different results.

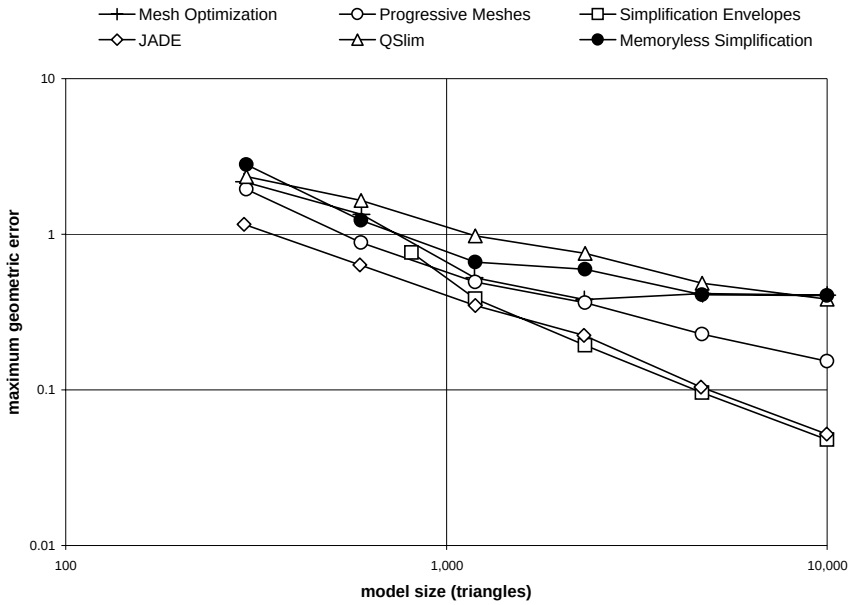
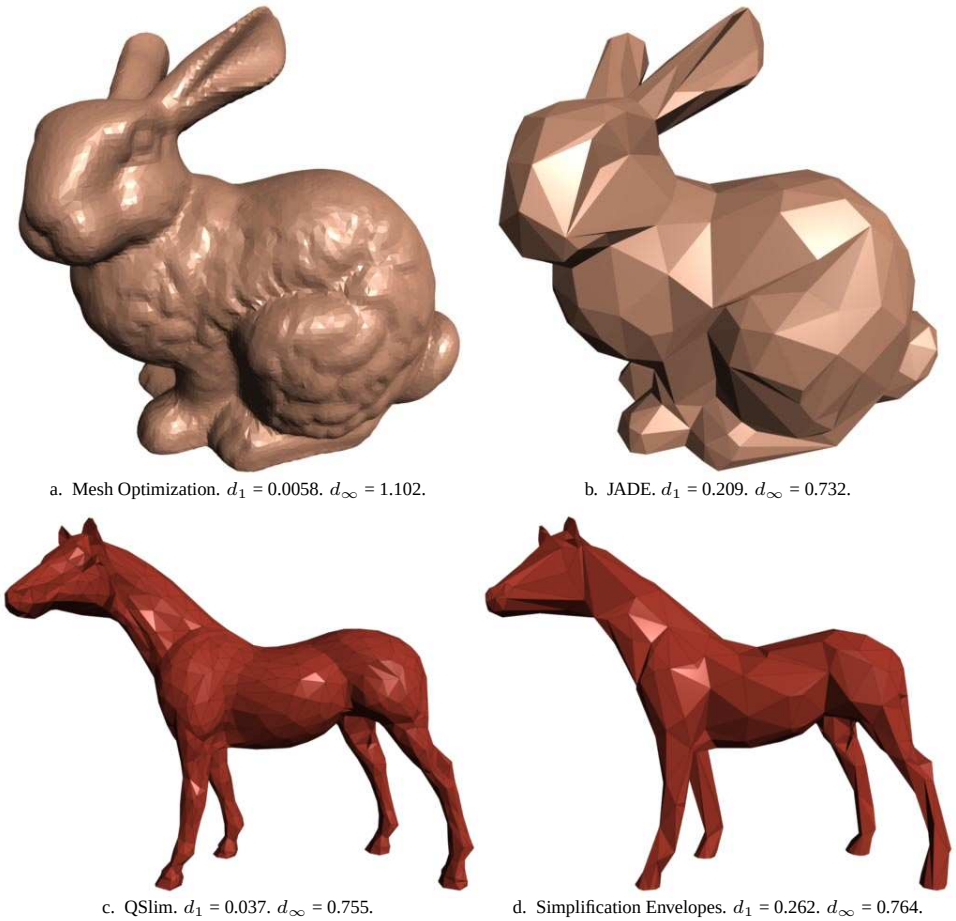


Figure 4.12: Maximum geometric errors for the horse model.

Figure 4.11 shows a log-log graph of the maximum geometric error for six levels of detail of the bunny model, which were created using the six different simplification methods. These models were simplified to have nearly an equal number of edges among the different methods, and the complexity of consecutive simplified models is roughly doubled from left to right. While there is a general trend in the data—the error increases as the number of edges decreases—the coherence in the relative performance of the different methods is poor as the error curves often cross, and the errors do not even change monotonically for a single method, as evidenced by Figure 4.12, which illustrates the maximum errors for the horse model. Not surprisingly, Simplification Envelopes and JADE display the largest degree of coherence, since both of them are driven by the goal of meeting a maximum error tolerance. Because this error measure is so sensitive to outliers, i.e. a single point among infinitely many on the two surfaces determines the error, it makes for a rather poor predictor of overall geometric and visual quality.[17] In fact, among the large number of simplified models I have produced, Simplification Envelopes and JADE generally produce models of lower visual quality than the other methods (in particular Mesh Optimization), while the maximum error graphs would suggest the opposite. As an example, the maximum errors for each pair of models in Figure 4.13 are roughly the same, even though the visual quality and geometric complexity are drastically different. For these reasons, I have chosen not to pay any particular attention to the maximum errors, and will instead use the

[17] Somewhat surprisingly, Watson et al. [152] found that the naming time, i.e. the time required to recognize an object, correlates better with the maximum error than the mean error in simplified models. However, naming time and aesthetic quality are two different measures that do not necessarily correlate. Their results may also be biased by limiting their test objects to models that were simplified using a variant of Rossignac and Borrel's clustering scheme [127]. Not only does this method produce very poor quality approximations, but is driven by bounding the maximum error, without any concern for average errors.



a. Mesh Optimization. d_1 = 0.0058. d_∞ = 1.102.

b. JADE. d_1 = 0.209. d_∞ = 0.732.

c. QSlim. d_1 = 0.037. d_∞ = 0.755.

d. Simplification Envelopes. d_1 = 0.262. d_∞ = 0.764.

Figure 4.13: Models with nearly equal maximum errors (d_∞). See Figures 4.9 and 4.10 for original models. This figure illustrates the inadequacy of the maximum error as a predictor of visual quality. The mean errors (d_1) are included for comparison.

mean error as the prime measure of geometric quality throughout the remainder of this thesis.

Figures 4.14 and 4.15 are mean error graphs of the bunny and horse models. Figure 4.16 is a less cluttered version of Figure 4.15, where the curves have been normalized relative to a curve that nearly matches the data (see caption for details). Notice how the memoryless method performs better than most other methods, and nearly as well as Mesh Optimization, even though the memoryless method is 40 times faster. In contrast to the maximum error graphs, these exhibit more coherence as each method yields a nearly linear (in log-log space) curve, and the ordering between the methods remains the same for all levels of detail. In fact, the correlation coefficient between $\log(E)$ and $\log(d_1)$ for the memoryless method in these two graphs is -0.9993 and -0.9994, respectively. This suggests that the mean error is inversely proportional to the complexity of the model. This empirical result is consistent with the theoretical relationship derived by King and Rossignac in [80]. They found that the maximum error d_∞ of a (hypothetical) optimal triangulation of an approximating

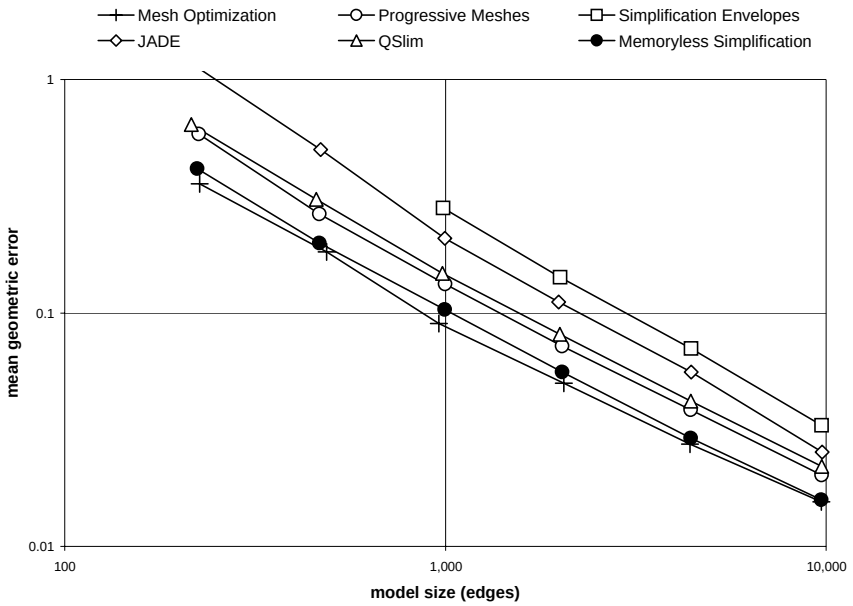


Figure 4.14: Mean geometric errors for the bunny model.

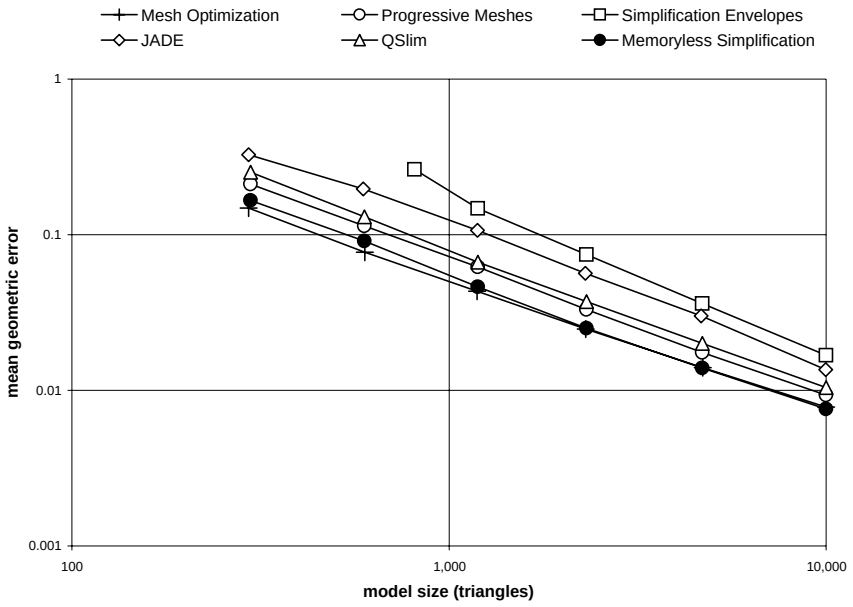


Figure 4.15: Mean geometric errors for the horse model.

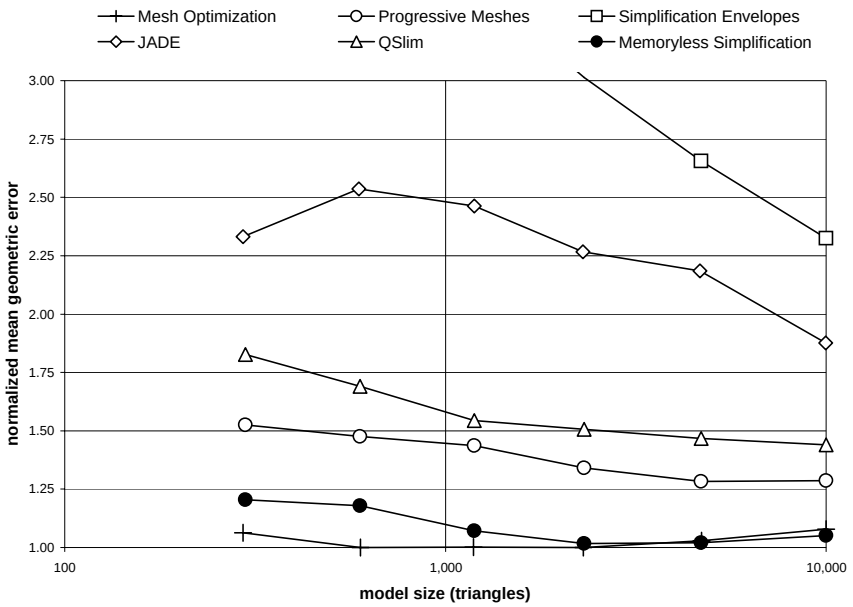


Figure 4.16: Normalized mean geometric errors for the horse model. The error values have been divided by $e^{c_0 - c_1 T}$, where c_0 and c_1 are fixed positive constants and T is the number of triangles in the simplified model.

surface with V vertices is given by

$$d_\infty = \frac{k}{V}$$

where k depends on the *shape complexity* of the original model; essentially an integral measure of curvature. Taking the logarithm of each side, we get

$$\log(d_\infty) = \log\left(\frac{k}{V}\right) = \log(k) - \log(V)$$

which is in close agreement with the mean error graphs presented here. While their derivation is for the *maximum* error, they have the luxury of assuming an *optimal, uniform* triangulation (whether one exists or not; even an object as simples as a sphere only has a handful of uniform approximating triangulations) with near uniform error. However, under such circumstances, the mean error (which is less sensitive to optimality and uniformity of the triangulation) generally follows the same basic behavior. It is therefore reasonable to apply their analysis to the mean error graphs.

Unlike complex models such as the bunny and the horse, we actually know the "best" simplified model of a sphere for certain numbers of faces. In particular, with a budget of 20 faces, the best possible simplification of a sphere is a regular icosahedron. We are still left with one parameter, namely the diameter of the icosahedron, and the choice of error metric will determine the optimal choice of this scaling parameter. Assuming this parameter can be found, I will use the sphere and its optimal approximation to evaluate each of the six simplification methods.

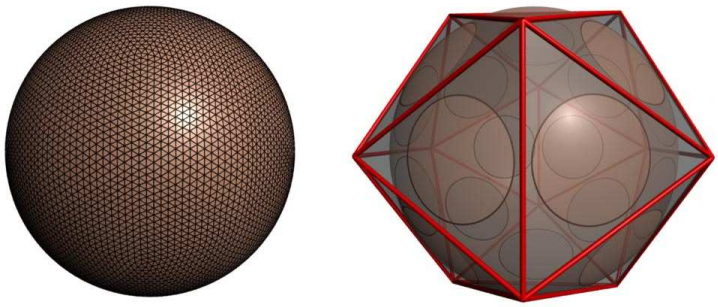
Figure 4.17a shows the original sphere model. This model was created from a regular icosahedron by repeatedly subdividing each triangle into four smaller triangles by bisecting its edges and projecting the new vertices onto the surface of the unit sphere. Starting with this 20,480 triangle sphere model, a single simplified model from each of the simplification methods was created, and each method was forced to produce a model with exactly 20 triangles. Figure 4.17 shows images of these simplified models. A translucent version of the original sphere has been superimposed, and the faces of each model are also translucent so that the back edges can be seen. The "ideal" result, an icosahedron, is shown in Figure 4.17b. The radius of this regular polyhedron was chosen empirically to be the value for which Metro reported a minimum mean error. Notice how roughly equal parts of this optimal model are on each side of the sphere, a fact I used to justify the use of volume preservation in §4.4.3.[18] In contrast, the vertex removal methods produced inscribed polyhedra, as their vertices were never allowed to move.

Figure 4.18 contains the numerical results of simplifying the sphere model. Once again, Mesh Optimization outperformed all other methods, followed by memoryless simplification.[19] In fact, the relative order of the different methods according to mean error follows exactly the performances for the more complex bunny and horse models, as well as the hand model in [94]. This result suggests that there may be a handful of simple models, such as the sphere, that are good predictors of how a simplification method behaves over a wide class of models. By extending the analysis in [80] to mean errors, it may be possible come up with a set of benchmark models of known shape complexity k, and then produce an objective rating R of a method by simplifying these models, measuring the error d_1, and computing $R = d_1 V/k$. Here $R = 1$ corresponds to the (hypothetical) optimal model with V vertices. Such a rating may of course vary slightly between different models and levels of detail, but it would nevertheless be a useful measure of how close a simplification method is to optimal.

Given that the sphere above appears to be a fair test model for evaluating the performance of simplification methods, one might ask if there are equally simple models that can be used to determine how well a method preserves boundary curves. A possible candidate would be to cap off part of the sphere model, thus introducing one or more boundary curves. The model in Figure 4.19a was created by including only the part of the sphere between latitudes 30° and 60°. It consists of 16,384 triangles that connect 256×33 vertices positioned uniformly in polar angles. This model was simplified using each of the six methods to either 87 or 88 edges. The results shown in Figure 4.19 emphasize that boundary preservation is a tug-of-war between the number vertices on the boundary of a model and the number of interior (manifold) vertices. There was considerable variation in the number of vertices that each method placed on the boundaries. Mesh Optimization used the fewest boundary vertices, whereas QSlim chose the opposite extreme and placed the most vertices on the boundary. Changing the parameter λ of the memoryless method spans this variation, which can be seen in Figures 4.19g, 4.19h, and 4.19i. Setting $\lambda = \frac{1}{32}$ produces a model much like that of QSlim, with few vertices that are not on the boundary, while $\lambda = \frac{31}{32}$ results in a model that is much like that of Mesh

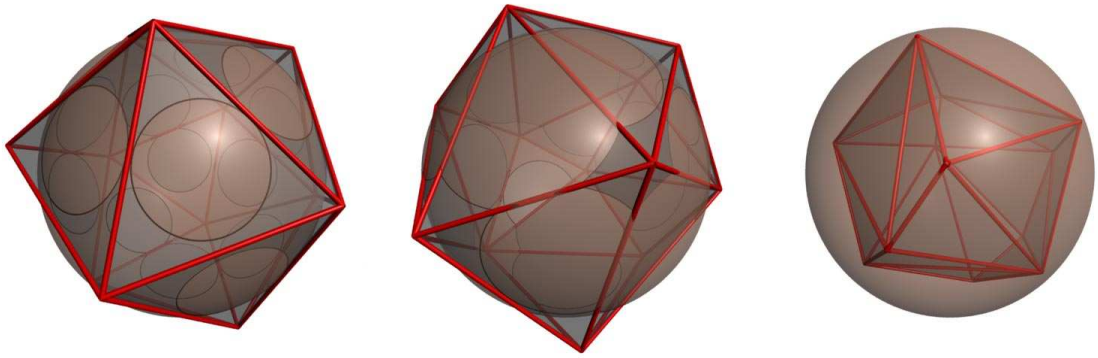
[18] The radius of the d_1-optimal model was found to be $r = 1.191$, while a volume-preserving regular icosahedron has $r = 1.189$. The original unit sphere, of course, has a radius of one.

[19] Indeed, as evidenced by Figure 4.17, Mesh Optimization nearly matched the optimal icosahedron. For such simple models as the sphere, this simplification/optimization method is likely to converge to the global optimum.



a. Original model.
T = 20,480.

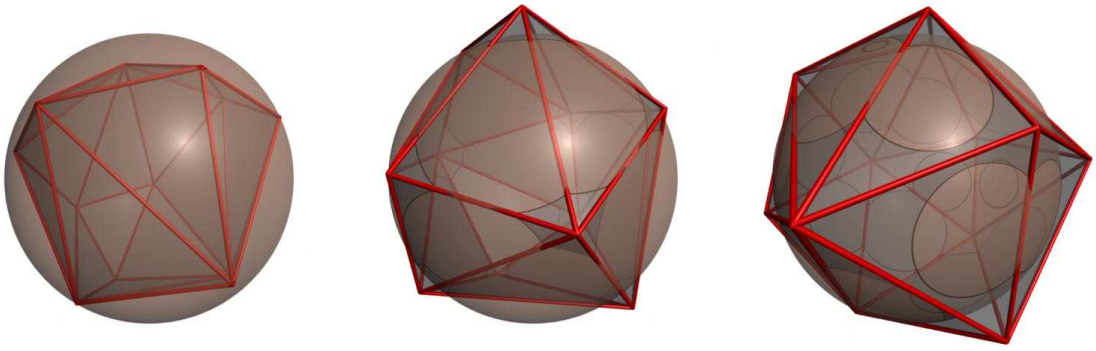
b. Optimal icosahedron.
T = 20.



c. Mesh Optimization.
T = 20. time = 13:43.

d. Progressive Meshes.
T = 20. time = 2:50.

e. Simplification Envelopes.
T = 20. time = 15:22.



f. JADE.
T = 20. time = 2:06.

g. QSlim.
T = 20. time = 0:03.

h. Memoryless Simplification.
T = 20. time = 0:18.

Figure 4.17: Sphere model.

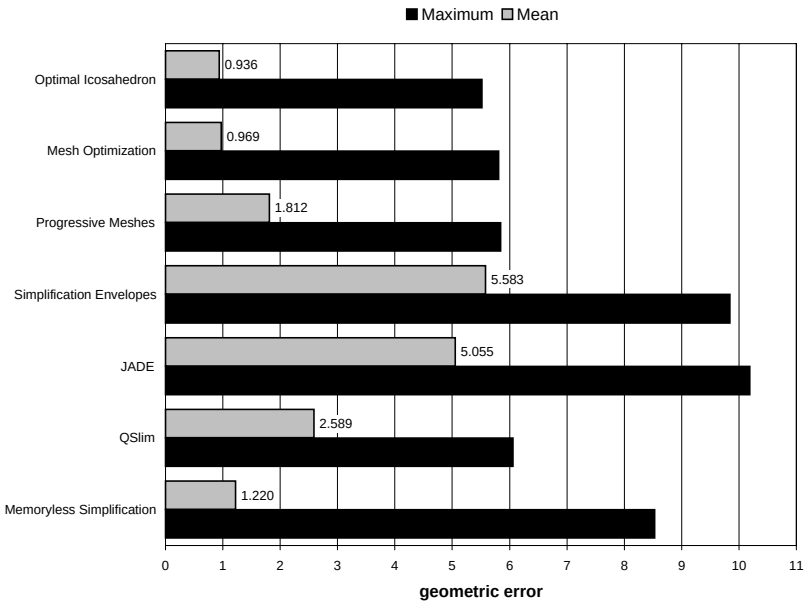
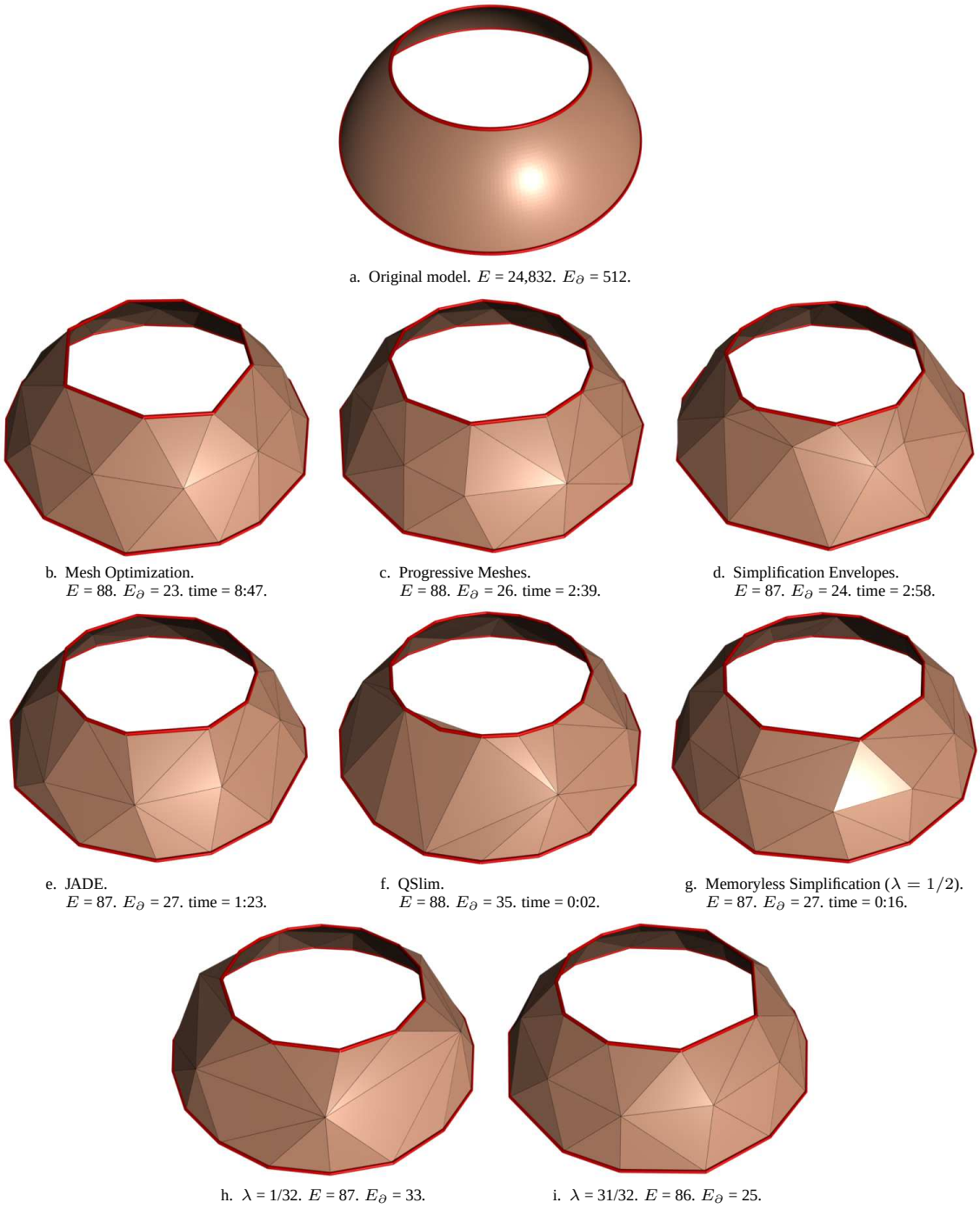


Figure 4.18: Geometric errors for the sphere model.

Optimization, with relatively fewer vertices on the boundary. Using $\lambda = \frac{1}{2}$ produces a model that is between these two extremes.

Figure 4.20 is a numerical comparison of the surface errors between the different simplification methods for the spherical segment. This figure graphs the mean and maximum error for a range of λ values for memoryless simplification to highlight the tradeoff between manifold and boundary fidelity. The figure shows that Progressive Meshes and Mesh Optimization give the least mean geometric error, and that for large values of λ these results are also matched by memoryless simplification. There is a clear relationship between high values of λ (greater emphasis on volume optimization) and smaller mean geometric error. With lower λ values, the edge cost emphasizes the boundary cost Q''_B more heavily, resulting in better boundary detail but more error over the surface of the model. As noted above, these geometric errors do not take the boundary deviation into account. A specialized tool for measuring boundary errors was used to produce the numerical results in Table 4.1, which includes boundary errors for both the spherical segment and the bunny model. Since the number of edges devoted to the boundaries varies greatly among the methods, it is somewhat difficult to judge their relative performance. A more fair comparison can be made by computing the product of the mean boundary error with the number of boundary edges. This assumes that the boundary error is inversely proportional to the number of boundary edges—an assumption that was shown earlier to be valid for surface errors. This adjusted error measure is arguably still not entirely fair, but it undoubtedly allows a more meaningful comparison than the "raw" errors. Again, it is possible to specify the boundary importance for all these methods to obtain a better balance in quality, and one needs to consider both the surface error and boundary error when comparing models.



a. Original model. E = 24,832. E_∂ = 512.

b. Mesh Optimization.
E = 88. E_∂ = 23. time = 8:47.

c. Progressive Meshes.
E = 88. E_∂ = 26. time = 2:39.

d. Simplification Envelopes.
E = 87. E_∂ = 24. time = 2:58.

e. JADE.
E = 87. E_∂ = 27. time = 1:23.

f. QSlim.
E = 88. E_∂ = 35. time = 0:02.

g. Memoryless Simplification ($\lambda = 1/2$).
E = 87. E_∂ = 27. time = 0:16.

h. λ = 1/32. E = 87. E_∂ = 33.

i. λ = 31/32. E = 86. E_∂ = 25.

Figure 4.19: Spherical segment model. Figures 4.19h and 4.19i show the results of varying the boundary fidelity parameter λ in the memoryless method.

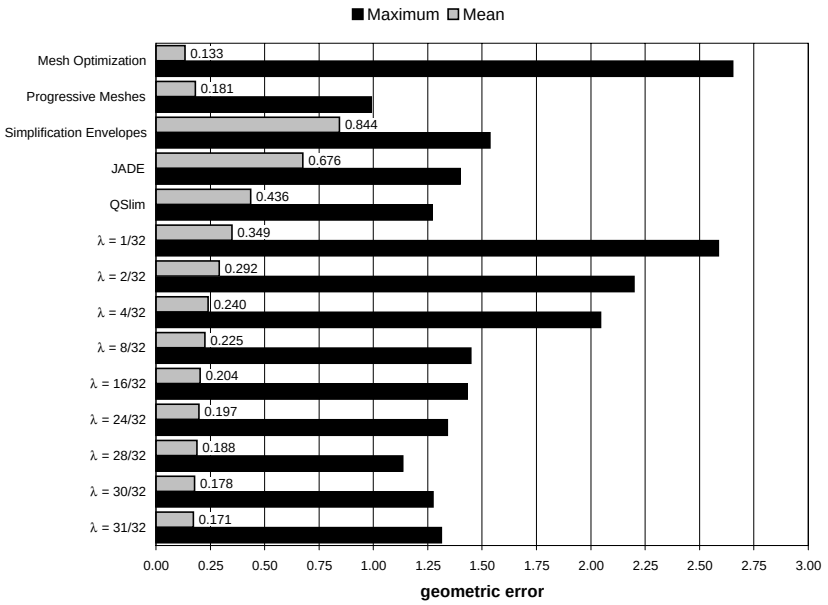


Figure 4.20: Geometric errors for the spherical segment model.

method	*bunny*			*spherical segment*		
	E_∂	d_1^∂	$E_\partial d_1^\partial$	E_∂	d_1^∂	$E_\partial d_1^\partial$
MO [72]	66	0.802	52.9	23	2.521	58.0
PM [66]	40	0.268	**10.7**	26	1.076	28.0
SE [28]	64	0.257	16.5	24	2.092	50.2
JADE [23]	99	**0.159**	15.7	27	1.653	44.6
QSlim [50]	61	0.356	21.7	35	0.726	25.4
MS ($\lambda = 1/2$)	42	0.290	12.2	27	**0.666**	**18.0**
MS ($\lambda = 1/32$)	76	0.142	10.8	33	0.438	14.4
MS ($\lambda = 31/32$)	30	0.479	14.4	25	0.764	19.1

Table 4.1: Mean geometric boundary errors for the models in Figures 4.9 and 4.19. d_1^∂ is expressed as the ratio of the mean deviation to the radius of the smallest enclosing bounding sphere of the original model. E_∂ is the number of boundary edges. The product $E_\partial d_1^\partial$ is included as an error measure to allow a fair comparison among the methods when the number of boundary edges differs greatly. The lowest errors are highlighted in boldface.

4.8.1.3 Simplification Time

Simplification speed is yet another important performance measure worth comparing, in particular since the methods being evaluated here vary greatly in algorithmic complexity. Timing statistics for all six algorithms were gathered during the process of simplifying the horse model. These times are shown in Figure 4.21. As can be seen in this figure, QSlim was the fastest of the algorithms tested, followed in order by memoryless simplification, JADE, Simplification Envelopes and Mesh Optimization. Only one data point was produced

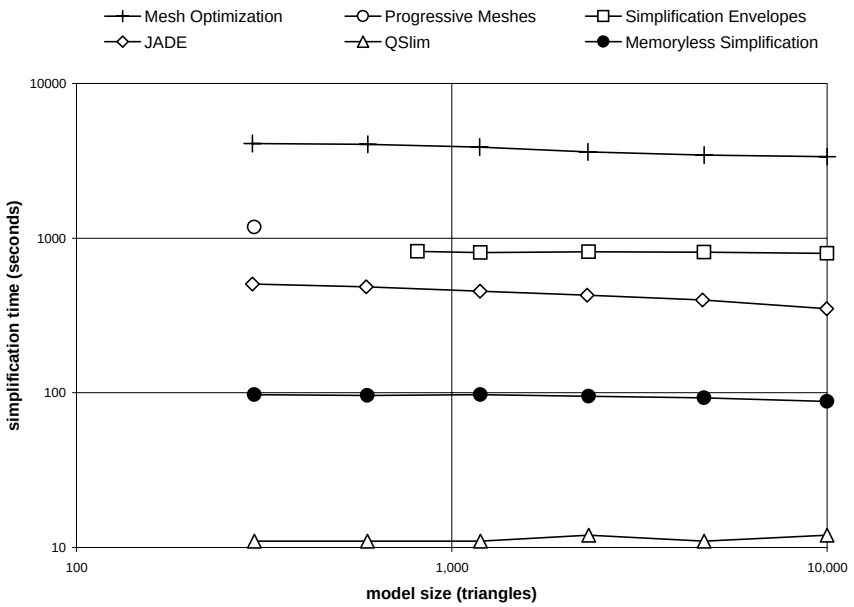


Figure 4.21: Simplification time for the horse model.

for Progressive Meshes, and it indicates that its speed is close to that of Simplification Envelopes.

One striking feature of Figure 4.21 is that the methods have nearly constant running times, regardless of the degree of simplification. This is in part due to the fact that the simplified models are close in complexity relative to the original model, which has roughly 100,000 triangles. Another reason is that the initialization time for many of these methods is relatively large and independent of the degree of simplification. The timing results presented here also include the I/O time needed to input and output the models, which becomes significant for some of the faster methods, such as QSlim. Indeed, QSlim took less time to produce the coarser few models than the most detailed one due to shorter output time.

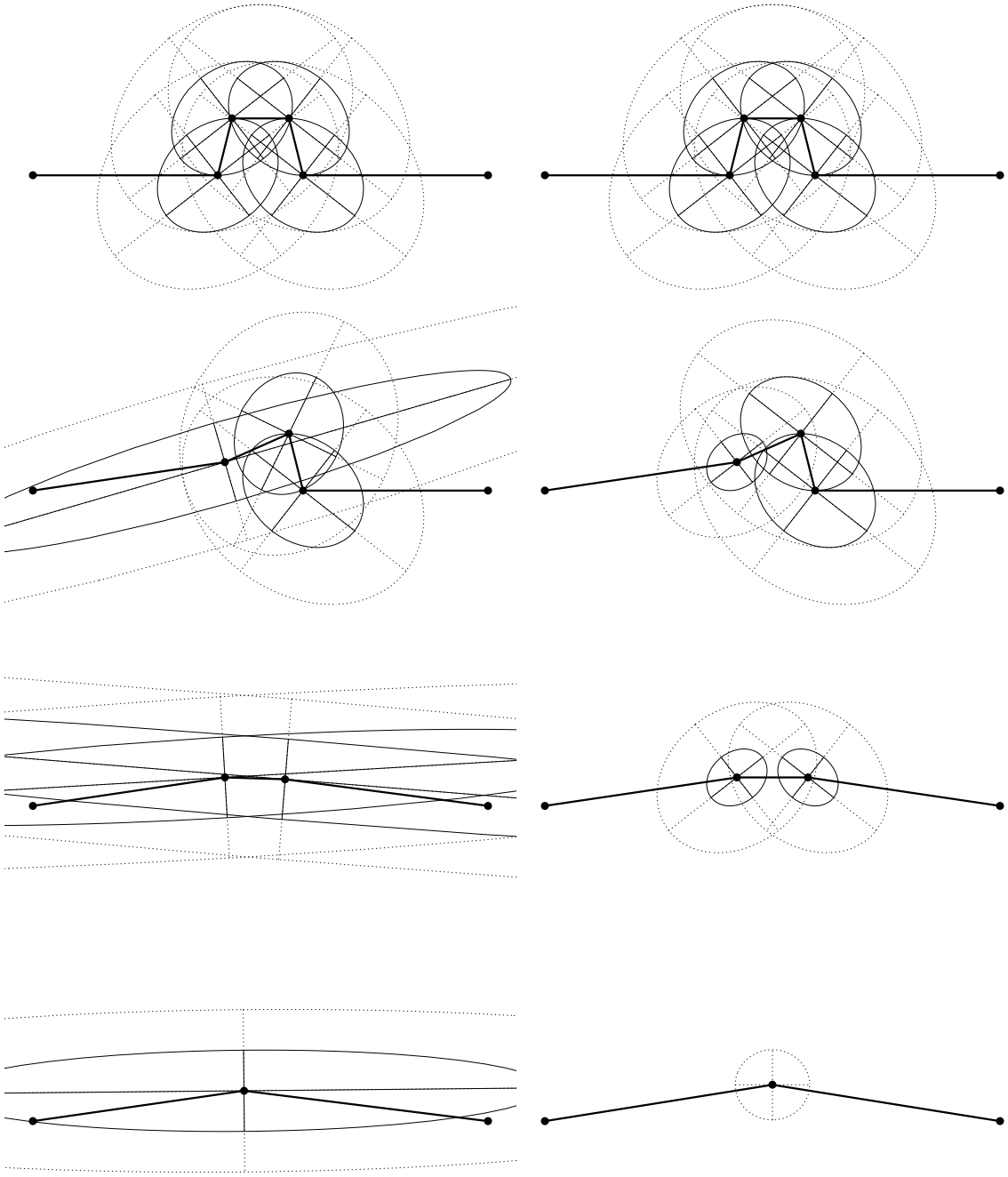
It is important to recognize that this comparison of running times of different methods is highly dependent on the particular *implementations* of these algorithms. It is entirely possible that each of these methods may have faster and more efficient implementations. However, the results presented in this section provide both a reasonable estimate of how the algorithms compare amongst each other, as well as what their individual characteristic running times are with respect to model size. Memory usage is another interesting and important aspect of algorithmic efficiency, and is also one of the strengths of the memoryless method. While memory usage is even more implementation dependent, due to the nature of the memoryless algorithm, there should be little doubt that it has the potential to be more memory efficient than the other methods discussed here. For further discussion on this topic and a comparison between QSlim and the memoryless method, see §5.3.

4.8.2 Effect of Memoryless Quadrics

We have seen above that the memoryless method consistently yields smaller mean errors than most of the other methods, which all retain information about the original model. In particular, memoryless simplification performs better than QSlim, which uses memorizing quadrics to allow fast evaluation of the distance to the original surface. At first, this result seems entirely at odds with our intuition; by maintaining information about the original model, one should in theory be better equipped to reduce the distance to it than if no such information is available. What is it about memoryless quadrics that leads to this surprising result? Hoppe provides an answer to this question in [69], and part of the analysis here is based on his observations. I will, however, give a somewhat more detailed analysis, including real graphical and numerical results of using different types of quadrics and weighting schemes. These results demonstrate that the use of memorizing—as opposed to memoryless—quadrics often leads to suboptimal sequences of edge collapses and choices of replacement vertices.

To illustrate the difference between memoryless and memorizing quadrics, I will use a simple 2D example of simplifying a piecewise linear curve. The 3D mesh analogue is based on the same principles, but is more difficult to visualize. Figures 4.22, 4.23, and 4.24 illustrate the effect of collapsing three edges on this curve using memoryless and memorizing quadrics, and using the three different quadric weighting schemes (generalized to 2D) mentioned in §4.4.8; uniform, edge length, and squared edge length. The three figures yield similar results, so let us focus on Figure 4.23. The top left figure shows five edges of a polygonal curve (the solid thick line segments). The "bump" formed by the three middle edges can be thought of as a fine detail on the curve that will be eliminated via three edge collapses. For each edge in the memoryless example, the error quadric Q''_B (§4.4.7) was computed (in $\mathbb{R}^2$), weighted appropriately, and added to the incident vertices. The resulting *vertex quadric* measures the error, relative to the partially simplified model, introduced by moving the vertex from its current position to some other position. For each of the four middle vertices, the ellipses in this figure correspond to points in 2D where this error level is constant, analogous to the ellipsoidal level surfaces in [49, 50]. The solid curves correspond to a quadric error $Q = 1$, while the dashed curves are for $Q = 4$. The axes of the ellipses define the principal directions of the associated error quadric's Hessian, and I will use the term *quadric* to refer both to the error functionals and the graphs of their level curves.

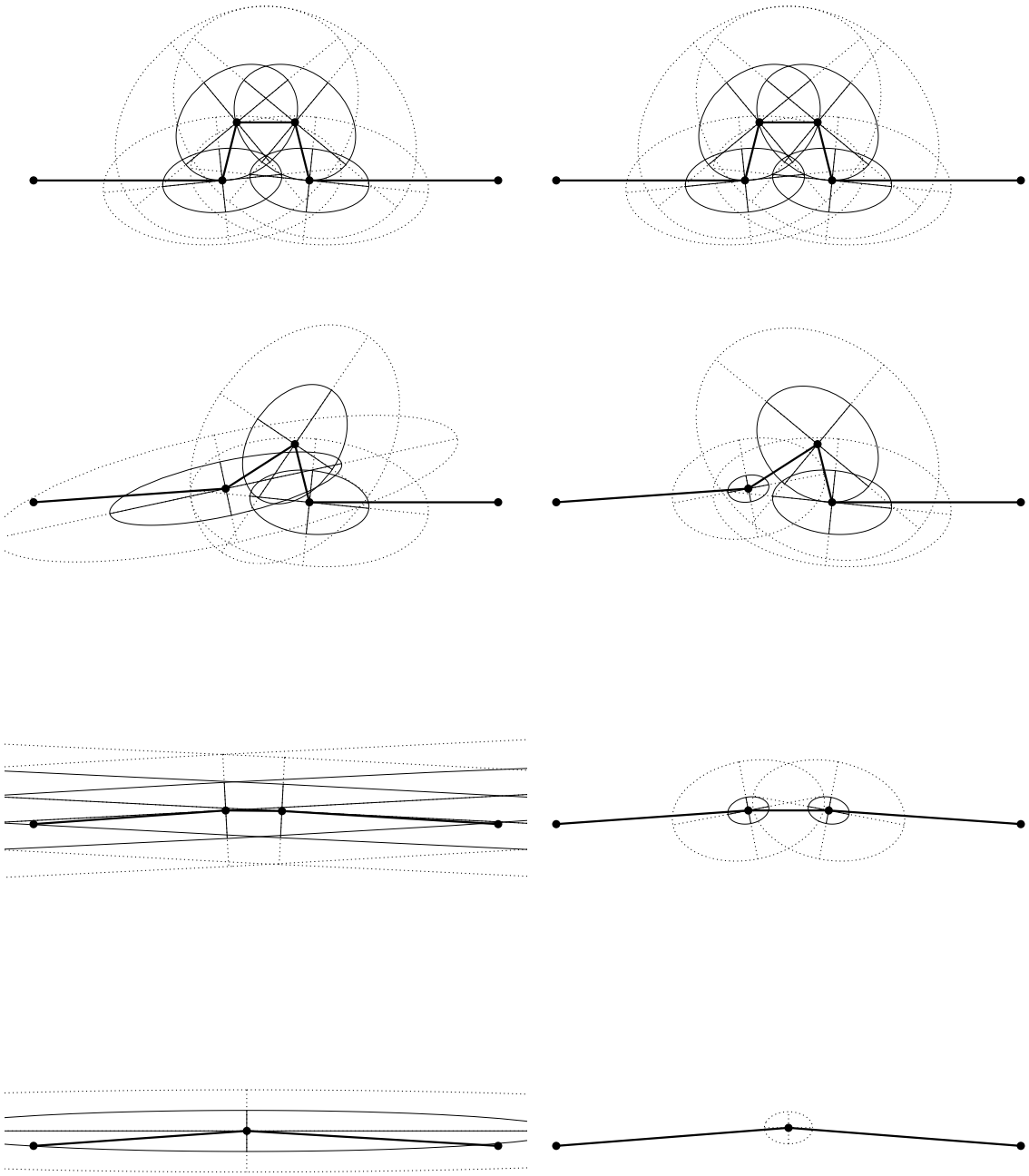
The curve second from the top in Figure 4.23a is the result after the leftmost near vertical edge is collapsed. The position of the replacement vertex was computed by minimizing the sum of quadric errors for the three edges incident upon the two merged vertices. Thus, the 2D vertex placement algorithm here is analogous to using only the volume optimization term in the 3D memoryless method (modulo the different weighting schemes), i.e. no area preservation constraint has been used here. Since this method is memoryless, the affected quadrics have been recomputed, resulting in an elongated shape of the leftmost new quadric, consistent with the supporting edges which are now closer to collinear. Collapsing the rightmost vertical edge results in quadrics that are extremely stretched and almost parallel to the underlying curve, suggesting that future errors are determined almost entirely by movement perpendicular to the curve. Finally, the bottommost curve is the result of collapsing the middle edge. Again, the center vertex's quadric approximates the near



a. Memoryless quadrics.

b. Memorizing quadrics.

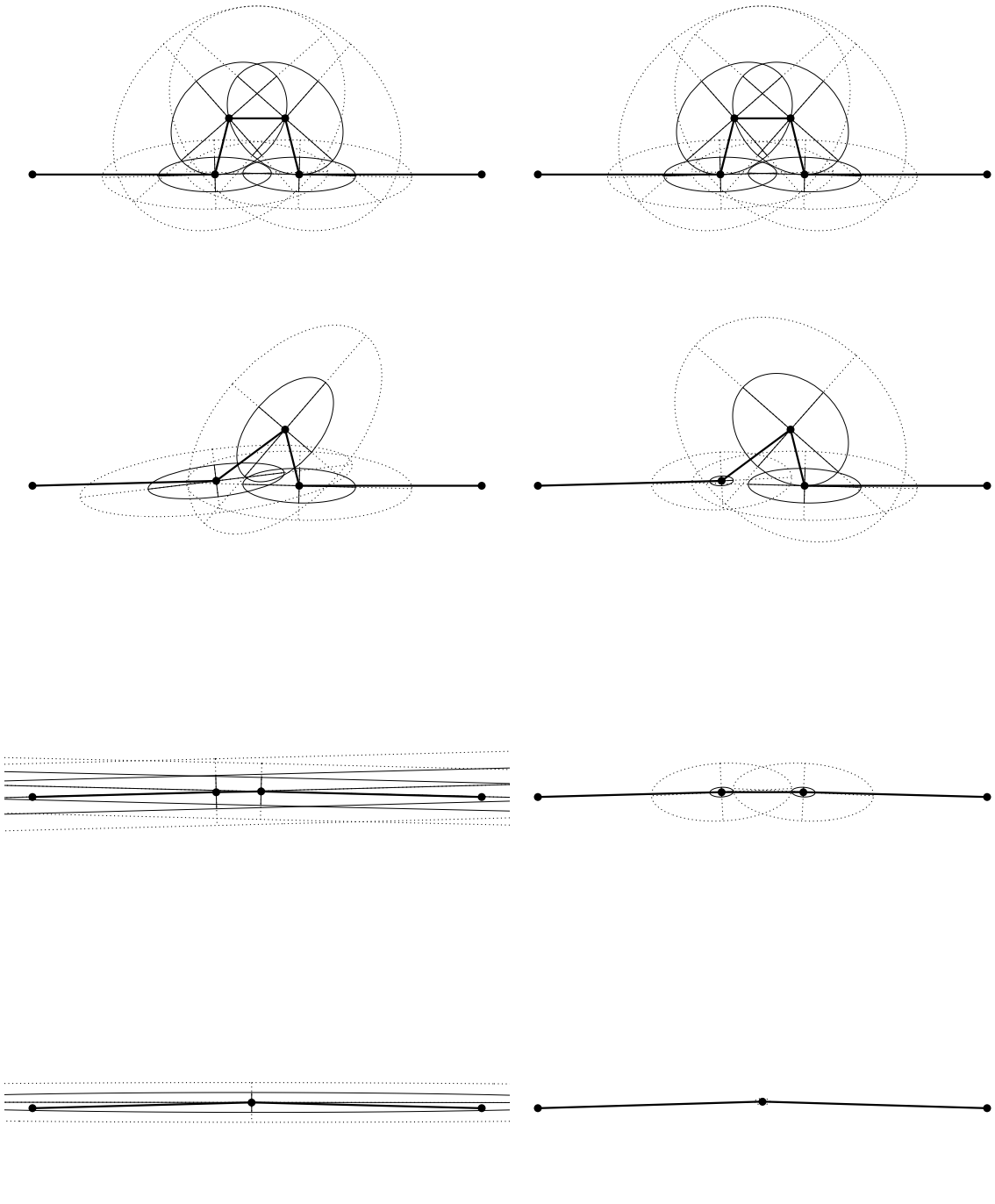
Figure 4.22: Uniformly weighted quadrics. Elliptical level curves for the quadric errors Q are shown at isolevels $Q = 1$ (solid) and $Q = 4$ (dashed).



a. Memoryless quadrics.

b. Memorizing quadrics.

Figure 4.23: Quadrics weighted by edge length.



a. Memoryless quadrics.

b. Memorizing quadrics.

Figure 4.24: Quadrics weighted by squared edge length.

	memoryless			*memorizing*		
	1	L	L^2	1	L	L^2
edge collapse	*edge cost*					
#1	0.500	0.765	0.914	0.500	0.765	0.914
#2	0.467	0.733	1.043	0.500	0.765	0.914
#3	$3 \cdot 10^{-3}$	$8 \cdot 10^{-4}$	$3 \cdot 10^{-5}$	2.389	3.332	3.926
level of detail	*mean geometric error*					
#1	0.789	0.570	0.448	0.789	0.570	0.448
#2	1.586	1.221	1.047	1.638	1.251	1.056
#3	1.538	1.208	1.045	1.660	1.262	1.059

Table 4.2: Results of simplifying the piecewise linear curves in Figures 4.22, 4.23, and 4.24 using memoryless and memorizing quadrics, and with three different edge weighting schemes; uniform (1), edge length (L), and squared edge length (L^2). The edge collapse numbers and levels of detail correspond to the illustrations in Figures 4.22, 4.23, and 4.24, ordered from top to bottom.

linear shape of the curve. Table 4.2 lists the cost of collapsing the three edges. Since the third edge collapse did not significantly alter the shape of the curve, its cost is very low compared to the other two.

Let us now focus on Figure 4.23b. The same curve was simplified using the 2D analogue of Garland and Heckbert's QSlim method with quadrics weighted by edge length. Note how these quadrics are initially the same as in the memoryless method. After one edge collapse, however, the new vertex is assigned the sum of quadrics from the vertices of the collapsed edge, instead of being recomputed from the simplified geometry. Consequently, the areas of the elliptical level curves are smaller. Notice also how the quadrics for the other vertices remain the same in their method. After another edge is collapsed, we are left with two small elliptical quadrics. Notice that whereas the simplified curves are in this case virtually indistinguishable between the two methods, the error quadrics are dramatically different. The memorizing quadrics are not nearly as elongated, and are also much smaller. Another significant difference is the cost of collapsing edges; the memoryless cost of collapsing the final edge is roughly one thousandth of the previous two collapses, whereas the corresponding collapse for the memorizing method is more than four times as expensive as the other two. The change in geometric shape due to this last edge collapse in Figure 4.23b is, however, negligible. Indeed, the difference in geometric error between the original and the last two curves, listed in Table 4.2, is very small, compared to the relatively larger errors introduced in each of the previous two edge collapses. Thus, the memorizing method greatly *overestimates* the cost of collapsing the last edge, and often suggests edge costs that are inconsistent with the associated geometric error. The memoryless edge cost, on the other hand, much better correlates with the actual change in geometric error, as evidenced by Table 4.2.

The reason for the high edge cost for the memorizing method in this example is the nearly perpendicular distance from the final vertex to the two near vertical edges in the original model. Meanwhile, the actual movement of the merged vertices in the final edge collapse is nearly tangential to the simplified surface, resulting in a very small increase in the actual error. It is this property of attempting to preserve fine features that have already been simplified away that can interfere with the memorizing method's decision of choosing the best edge to collapse. As quadrics are accumulated at a vertex, the costs of collapsing incident edges increase. Thus the cost of an edge collapse is not paid just once, but it factors into the cost of future edge

collapses as well, and must be paid over and over at increasingly inflated prices. Furthermore, the near circular quadric of the final center vertex in Figure 4.23b may result in less than optimal vertex positioning in future edge collapses involving this vertex, since the quadric penalizes movement tangential to the simplified curve nearly as much as perpendicular movement. Since the fine detail has already been eliminated, it makes little sense to constrain the center vertex horizontally to this location. The memoryless method, on the other hand, allows much more freedom in tangential movement.

Finally, let me point out that the results for the other two quadric weighting schemes—uniform and squared edge length—are consistent with the findings here. These figures are useful for getting an intuitive feel for the impact of the different weighting schemes on the shape of the error quadrics and the resulting vertex positions.

4.8.3 Effect of Volume Preservation

We have seen above how the memoryless quadrics perform better than Garland and Heckbert's memorizing quadrics. In every other aspect, the two methods are similar, except for one significant difference: memoryless simplification imposes a volume preservation constraint on the placement of vertices. It is then natural to ask whether volume preservation plays a role in reducing the approximation error. To answer this question, I will examine the impact of using volume preservation with several different, and rather simple, vertex placement schemes. Like the original memoryless method for vertex placement, these other methods rely on no geometric history, but can all be derived from the partially simplified geometry. As noted above, vertex placement is only half of an edge collapse algorithm—we also need to evaluate the cost of a potential edge collapse in order to prioritize the list of edges. Some of the vertex placement schemes included here do not induce any error functionals, and so there is no canonical edge cost associated with them. To simplify matters, I have chosen to use the same edge cost with each of the eight vertex placement methods, namely the memoryless edge cost Q_C from §4.5.

A number of vertex placement algorithms have been proposed in the simplification literature. Using the eight methods discussed below, I have attempted to cover a range of possible history-free vertex placement approaches. Selecting one of the original vertices of an edge is one possibility. Which vertex should be selected? I will evaluate two possibilities: (1) randomly select one of the vertices, and (2) select the vertex that minimizes the edge cost (the "best" vertex). Note that using an original edge vertex forces an edge collapse operation to be a special kind of vertex removal operation. In the simplification literature, this type of edge collapse is often called *half-edge collapse* [82, 88] (Figure 2.2). Another logical position for a new vertex is the midpoint of the edge to be collapsed. This is the third vertex placement method examined. Note that the volume optimization functional Q''_V for a particular edge generally has a unique minimum. The position that minimizes Q''_V is used for the fourth vertex placement method.

In each of these four methods, we can optionally impose a linear volume preservation constraint. By computing this plane constraint, the position given by each of the methods just described can simply be projected onto the volume preserving plane, resulting in four slightly different vertex placement schemes, one

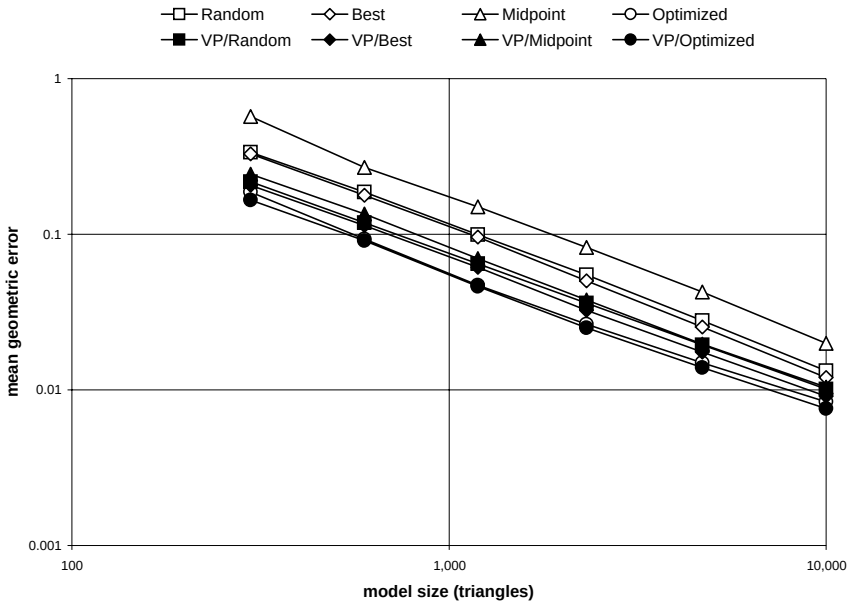


Figure 4.25: Mean geometric errors for the horse model. The different vertex placement methods correspond to randomly choosing one of the edge's vertices, the vertex with the smaller error (the "best" vertex of the two), the edge midpoint, and the optimal position with respect to the error functional. All four methods are improved by adding volume preservation (VP) as a constraint.

of them being the original memoryless method.[20] My intent is to evaluate the effect that volume preservation has on each of these methods, which are summarized in this list:

1. Random vertex.
2. Best vertex.
3. Edge midpoint.
4. Volume optimization.
5. Volume preservation, random vertex.
6. Volume preservation, best vertex.
7. Volume preservation, edge midpoint.
8. Volume preservation, volume optimization.

Using the horse as a test model, each of the methods above was used to produce six levels of detail. Figures 4.25 and 4.26 show the mean geometric errors for these simplified models. Several results are worth noting. First, the two edge midpoint selection methods (3 and 7) are out-performed by the methods that use

[20]For the two methods that use volume optimization, I have also included triangle shape optimization whenever the system is under-constrained, i.e when the surface is locally planar or parabolic. The horse model used in this evaluation does not have boundaries. Thus the edge cost and functionals used for vertex placement are free of boundary terms.

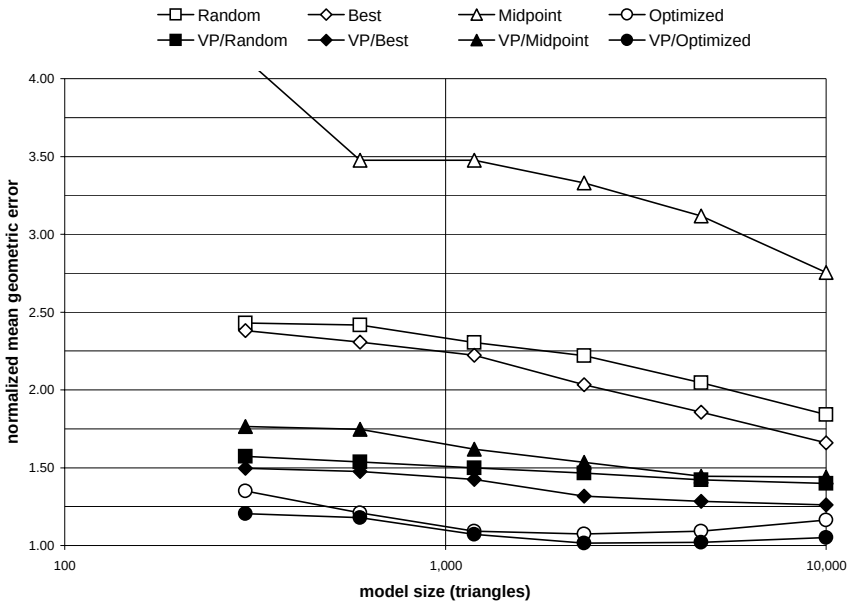


Figure 4.26: Normalized mean geometric errors for the horse model. See Figure 4.16 for details.

one of the original vertices of the edge. This somewhat surprising result indicates that half-edge collapses are to be preferred over using the more symmetric edge midpoint. Second, the best vertex methods (2 and 6) do slightly better than their respective random vertex methods (1 and 5). Third, volume optimization performs better than either using an original vertex or the midpoint. Finally, adding the constraint for volume preservation (methods 5, 6, 7, and 8) improves all of the methods (1, 2, 3, and 4). Method 8, the vertex placement method described in §4.4, yields the smallest mean error over all of the different levels of detail. Notice that for the horse model, the mean errors for these last four methods are as low or even lower than the ones produced by most of the simplification methods discussed in §4.8.1, which suggests that volume preservation alone is an important and useful property for model simplification. Further support for these findings is provided by Hoppe [69], who concluded that both volume preservation and memoryless quadrics consistently improve model quality.

Chapter 5

OUT-OF-CORE SIMPLIFICATION

5.1 Introduction

Model simplification has traditionally been a battle between speed and quality; as in most computing disciplines, faster generally means lower quality and vice versa. Quite elaborate algorithms have been proposed, such as [28, 72], which are slow but produce models of high quality (with respect to their error metrics). On the other side of the spectrum are fast but lower quality methods, e.g. [127, 135], and most recent research has been centered around simultaneously improving both speed and quality. Nearly all of these methods work under the assumption that memory is an infinite resource, which until recently has been a reasonably fair assumption, and relatively little effort has been spent on designing memory efficient algorithms. In the last few years, however, there has been an explosion in model size, in part due to improvements in resolution and accuracy of data acquisition devices, such as laser range and CT/MRI scanners. Indeed, submillimeter resolution data sets such as the *Visible Human* [1], which consists of well over 10 billion voxels, and the range scans of Michelangelo's sculptures made independently by research groups at IBM [9] and Stanford University [90] contain up to two billion triangles. These enormous data sets pose great challenges not only for mesh processing tools such as rendering, editing, compression, and surface analysis, but paradoxically also for simplification methods that seek to alleviate these problems. In addition to their large memory consumption, previous algorithms also suffer from insufficient simplification speed to be practically useful for simplifying very large meshes. As an example, an efficient implementation of the memoryless simplification method described in Chapter 4—one of the fastest and most memory efficient algorithms available—requires $160n$ bytes of internal storage to represent an n-vertex model and the necessary edge collapse priority queue. Simplifying a one billion vertex model to a few million triangles using this algorithm would require 160 gigabytes of RAM and, disregarding memory thrashing, would take weeks to complete on a high end workstation!

One might argue that high resolution data sets such as the ones described above are greatly oversampled, and that this problem should be solved more directly during the data acquisition or synthesis stage, e.g. by using adaptive sampling and tessellation during range scanning and isosurface extraction. At best, this simply shifts the problem to an earlier stage of the modeling pipeline, and results not only in a need for specialized tools for each acquisition method, but often raises a number of practical issues. In particular, it places an additional burden on the data acquisitor in terms of deciding how to sample the model and dealing with the

difficult issues of registering and integrating different resolution surface patches. In some cases, such a head-on approach is not even practical; one might not know in advance what parts of a surface should be sampled densely, or one might simply wish to retain the model at its full resolution and allow the end-user to resample the model in a manner that suits the given application.

Currently, few algorithms exist for performing high quality out-of-core simplification. One reason for this is that existing in-core methods are difficult to adapt to perform out-of-core simplification, because the majority of them are based on performing simple local operations that rely on having direct access to the connectivity of the mesh. For example, the quality measures associated with the *vertex removal* and *edge collapse* operations typically depend on the triangles surrounding the vertex or edge. Consequently, such methods use large in-core data structures to allow efficient queries of the local connectivity for any given mesh vertex. As mentioned above, such data structures may require hundreds of bytes per vertex, which might even be too large to off-load to disk. Instead, developers of out-of-core simplification algorithms are faced with two alternatives: segmenting the model into multiple pieces and simplifying them individually, or simplifying models using only limited connectivity information, which is the approach taken here.

In this chapter, I will present an efficient and easy to implement surface simplification algorithm that accepts models of arbitrary complexity and outputs a model that is small enough for in-core mesh processing tools to handle and store internally. The algorithm is inspired by Rossignac and Borrel's vertex clustering algorithm [127], and is enhanced by a novel use of error quadrics, which were originally developed for edge collapse methods (cf. [50, 94]). Its design inherently allows models of arbitrary complexity to be simplified—a feature supported by few previous simplification methods. As will be shown later, the method is also fast and produces models that are superior in geometric quality to those obtained using Rossignac and Borrel's original clustering method.

The out-of-core method presented here was first published as a short paper in [92]. I will provide a more detailed description of it in this chapter and present some new results to further support its strengths.

5.2 Algorithm Description

The simplification algorithm presented here is a hybrid of several schemes, including [50, 94, 127]. At a high level, it resembles Rossignac and Borrel's vertex clustering algorithm, but is improved both in execution time and quality by using a variant of the quadric error metric introduced by Garland and Heckbert, discussed in §4.4.8, for positioning vertices. The particular quadric weighting scheme used in this new out-of-core simplification (OoCS) algorithm is the same as in the memoryless simplification method from Chapter 4. See §4.8.2 for a comparison between the two weighting schemes. The main idea behind the method is to process the original mesh one triangle at a time (in no particular order). Given a triangle, its quadric matrix is computed and added to the clusters that its vertices belong to. After the entire model has been read, representative vertices for the clusters are computed by minimizing the quadric error, and the resulting simplified model is output.

Because the OoCS algorithm must be able to process extremely large models, an important design goal

was to make it run more efficiently than previous methods. Most greedy simplification methods, such as edge collapse and vertex removal algorithms, require maintaining a priority queue, and consequently need at least $O(n \log n)$ time to simplify a model [29], where n is the number of mesh simplices. The OoCS algorithm, on the other hand, does not use a priority queue, and runs in linear time $O(n)$. In addition, its memory requirements depend only on the size of the *simplified* model. Such an algorithm is said to be *output sensitive*. In particular, the OoCS algorithm improves upon [127] by requiring only a single pass over the input model, compared to two or more, and does not use any disk space beyond the input mesh, whereas their algorithm requires an importance value to be stored with each vertex of the input model. As a result, OoCS is not only much more accurate than Rossignac and Borrel's algorithm, but it is also considerably faster.

For a fair characterization, I will also point out some of the weaknesses of my method. Because the simplification is based on vertex clustering, the method allows modification of the surface topology, such as removal of small holes and noise in the data. On the other hand, this topology modification can potentially introduce non-manifold simplices, which may be undesirable for some applications. Because no connectivity information is used, the method does not account for surface boundaries, although I will in §9.2.2 discuss promising extensions that would allow such information to be recorded and used at minimal additional cost. As in [127], the uniform cluster grid leads to a uniformly sampled simplified model, and further simplification in regions of low curvature is often possible. Such a post-processing step could be performed using an in-core simplification method, or using the hierarchical approach to vertex clustering discussed in §9.2.2.

In the following sections, I will describe the details of the algorithm. Because vertex clustering and quadric error metrics are described at length in Chapters 3 and 4, I will assume that the reader is familiar with these techniques, and I will focus on the novel aspects of the OoCS algorithm. I will first discuss briefly what assumptions are made on the external representation of the input model, and then describe how the quadrics and representative vertices are computed, followed by a description of the actual simplification algorithm.

5.2.1 Model Representation

For performance reasons, it is important that the external mesh representation is conducive to the types of mesh queries needed for the given simplification operator. Fortunately, the combination of vertex clustering and quadrics allows commonly used off-line data structures to represent the mesh, such as an *indexed mesh* in which each triangle is a triplet of indices associated with an ordered list of vertex coordinates. By storing the mesh in binary form as fixed-length records, the vertices of a triangle can be fetched from disk indirectly via random access. While such a format is compact, the OoCS algorithm requires no incidence information, and is thus able to operate on a *triangle soup* in which each triangle is represented directly as a triplet of vertex coordinates. The triangle soup representation requires roughly twice as much disk space as the indexed mesh, but typically increases the simplification speed by a factor of 15–20 as cache and seek friendly sequential reads can be made instead of random seeks. The triangle soup has a number of other benefits: it allows non-fixed-length representations, such as text files, and because the algorithm makes a single pass over the mesh triangles, the triangle soup can be compressed externally and then uncompressed on-the-fly during simplification. The model can even be split up into several files if, for example, it is too large to store

on a single disk. The triangle soup representation was used for the results presented in §5.3.

Similar to Rossignac and Borrel's original clustering algorithm, the OoCS algorithm also uses a bounding box for the model, which is divided into a user-specified number of rectilinear grid cells. If a bounding box has not been precomputed for the model, which most data acquisition methods would otherwise be able to supply, an arbitrary origin can be chosen for the grid, leaving the user to specify the desired grid resolution in each dimension, as well as upper bounds on the number of anticipated grid cells along each of the three axes. This information is needed so that a unique finite-length identifier can be associated with each occupied grid cell.[1]

5.2.2 Error Quadrics

In order to integrate quadrics with the general vertex clustering scheme, I will use the observation made in [50] that vertex clustering is a special case of *vertex pair contraction*—a generalization of edge collapse to arbitrary pairs of vertices. That is, merging n vertices within a cluster cell is equivalent to performing any sequence of $n-1$ contractions of pairs of vertices within the cluster until a single vertex remains. As a consequence, Garland and Heckbert's original scheme can be extended from individual vertex contractions to a predefined sequence of such operations. In fact, their algorithm and the one described here are in theory equivalent, with the exception that the sequence of contractions in OoCS is determined by the cluster grid rather than by the mesh geometry. To improve quality and performance, I will use the quadrics from Chapter 4, instead of the uniformly weighted quadrics from [50].

Using the derivations from §4.4.4, the volume-optimizing quadric matrix $\mathbf{Q}^t$ for each triangle $t = (\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t)$ can be written as:

$$\mathbf{Q}^t = {\mathbf{N}^t}^{\mathsf{T}} \mathbf{N}^t \tag{5.1}$$

$$\mathbf{N}^t = \left[-\left(\mathbf{x}_1^t \times \mathbf{x}_2^t + \mathbf{x}_2^t \times \mathbf{x}_3^t + \mathbf{x}_3^t \times \mathbf{x}_1^t\right)^{\mathsf{T}} \quad [\mathbf{x}_1^t, \mathbf{x}_2^t, \mathbf{x}_3^t] \right] \tag{5.2}$$

where $\mathbf{N}^t$ is a 1×4-matrix made up of the area-weighted triangle normal and the scalar triple product of its three vertices. For each vertex v of t, v is mapped to a cluster of vertices, V, which is ultimately collapsed to a single vertex $\bar{v}$. $\mathbf{Q}^t$ is distributed to each of the three clusters by adding it to their quadric matrices $\mathbf{Q}^{\bar{v}}$. That is, for a given cluster V, its quadric matrix is defined as

$$\mathbf{Q}^{\bar{v}} = \sum_{v \in V} \sum_{t \in \lceil\lceil v \rceil\rceil} \mathbf{Q}^t \tag{5.3}$$

Since $\mathbf{Q}^{\bar{v}}$ is symmetric, and since the scalar c is not used, only 9 scalar values need to be stored with each cluster. After adding up the quadrics of all the triangles in a cluster, the following block decomposition of $\mathbf{Q}^{\bar{v}}$ is used

$$\mathbf{Q}^{\bar{v}} = \begin{bmatrix} \mathbf{A} & -\mathbf{b} \\ -\mathbf{b}^{\mathsf{T}} & c \end{bmatrix} \tag{5.4}$$

[1] The computation of grid cell identifiers can be done by concatenating the three integer coordinates of the grid cell. To ensure uniqueness, an upper bound n must be chosen for each axis such that the space occupied by the model along the axis does not exceed n cells. For an arbitrarily chosen origin, modulo n arithmetic may be needed to shift the indices into range.

and the linear system $\mathbf{Ax} = \mathbf{b}$ is solved for the optimal representative vertex position $\mathbf{x}$. This general strategy is the same as in §4.4 and [50], except for (1) the set of triangles used in constructing $\mathbf{Q}^{\bar{v}}$,[2] and (2) the procedure used for solving the linear system of equations, which is described below.

If a cell contains two nearly parallel surface sheets, the quadrics will sometimes suggest a solution $\mathbf{x}$ that is close to the intersection of the extension of these two surfaces. The solution may in such cases lie arbitrarily far outside the cell itself, which may be undesirable. One possible way of handling such cases would be to introduce artificial quadrics defined by the six cell walls. Such an approach would also solve the problem of underdetermined systems, i.e. when $\mathbf{A}$ is not of full rank. On the other hand, this approach would always bias vertices towards the cell center, leading to larger geometric errors than necessary. Instead, I have chosen to restrict the position $\mathbf{x}$ to the cell itself or to a small neighborhood around it. In the current implementation, this is accomplished by limiting the distance of $\mathbf{x}$ from the cell center to a constant times the length of the cell diagonal (this constant is currently set to $\frac{1}{2}$). That is, if $\mathbf{x}$ is outside this spherical region, it is projected (radially) onto the sphere. Alternative methods, such as clamping the individual coordinates of $\mathbf{x}$, or performing a constrained minimization of the quadric error on the cell boundary, are also possible.

5.2.2.1 Numerical Robustness

In the discussion above, I assumed that the matrix $\mathbf{A}$ is invertible and well-conditioned. In practice, this is often not the case, e.g. if the surface is locally flat or has zero Gaussian curvature (§4.4.4). In §4.4, I proposed a partial solution to this problem by ensuring that the system is overdetermined, and then successively combining linear constraints that yield a sufficiently large value for the determinant of $\mathbf{A}$. In the case here, however, the quadrics yield at most three constraints, and I will use a slightly different approach that is able to both diagnose potential problems and also robustly produce the "best" vertex in the sense that $\mathbf{x}$ is chosen such that its distance to the cell center is minimized. Stated differently, $\mathbf{x}$ is the orthogonal projection of the cell center $\hat{\mathbf{x}}$ onto the space of all solutions to $\mathbf{Ax} = \mathbf{b}$. This is accomplished by performing a *singular value decomposition* $\mathbf{A} = \mathbf{U\Sigma V}^{\mathsf{T}}$, which for a real symmetric positive semidefinite matrix $\mathbf{A}$ is equivalent to doing an eigenvalue decomposition. This can be done quickly using a small number of Jacobi rotations [54, 120]. The matrices $\mathbf{U}$ and $\mathbf{V}$ above are orthogonal while $\mathbf{\Sigma}$ is diagonal. The basic approach to solving $\mathbf{Ax} = \mathbf{b}$ is to compute what is called the *pseudo-inverse* $\mathbf{A}^{+} = \mathbf{V\Sigma}^{+}\mathbf{U}^{\mathsf{T}}$, where $\mathbf{\Sigma}^{+} = \mathrm{diag}(\sigma_1^{+}, \sigma_2^{+}, \sigma_3^{+})$ is a diagonal matrix, defined below, that roughly speaking is the inverse of $\mathbf{\Sigma}$. As is common in underdetermined least squares minimization, a lower limit is imposed on the singular values for numerical robustness, and the ones that are negligible are discarded [120, pp. 63–64]. Thus, a robust pseudo-inverse can be computed using

$$\sigma_i^{+} = \begin{cases} 1/\sigma_i & \text{if } \sigma_i/\sigma_1 > \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

where σ_1 is the largest singular value of $\mathbf{A}$ and ϵ is a threshold parameter currently set to 10^{-3}. Then $\mathbf{Ax} = \mathbf{b}$ can be solved using $\mathbf{x} = \mathbf{A}^{+}\mathbf{b}$. For underdetermined systems, one can show that this is the solution vector

[2] As in [50], a triangle t's quadric matrix is possibly counted more than once for a given cluster, depending on how many of t's vertices are contained in the cluster cell.

with the smallest norm [54]. We can use this fact to compute the position $\mathbf{x}$ that both satisfies the system of equations and is closest to the cell center $\hat{\mathbf{x}}$:

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{A}(\mathbf{x} - \hat{\mathbf{x}} + \hat{\mathbf{x}}) = \mathbf{b} \\ \mathbf{A}(\mathbf{x} - \hat{\mathbf{x}}) &= \mathbf{b} - \mathbf{A}\hat{\mathbf{x}} \\ \mathbf{x} - \hat{\mathbf{x}} &= \mathbf{A}^{+}(\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}) \\ \mathbf{x} &= \hat{\mathbf{x}} + \mathbf{A}^{+}(\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}) = \hat{\mathbf{x}} + \mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{\mathsf{T}}(\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}) \end{aligned} \tag{5.6}$$

Clearly, $\|\mathbf{x} - \hat{\mathbf{x}}\|$ is minimized since it is the minimum norm least squares solution to the second equation. If $\mathbf{A}$ is well-conditioned, then $\mathbf{A}^{+} = \mathbf{A}^{-1}$ and $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. If $\mathbf{A}$ is ill-conditioned, then this procedure yields a numerically robust solution which minimizes the distance to the cell center.

As an aside, I pointed out in §4.4.1 that the SVD can be used for vertex placement in the memoryless simplification algorithm. This would be done by first letting the "default" position $\hat{\mathbf{x}}$ be the unique position that minimizes the triangle shape functional (§4.4.5). Using the procedure just described, $\hat{\mathbf{x}}$ would then be projected onto the solution space of the edge cost functional, which in turn would be the default position for minimizing the volume and boundary preservation functionals. Thus, the functionals are minimized in reverse order, and each step results in a solution that minimizes the next higher priority functional Q_{i+1} and is closest to the minimum $\hat{\mathbf{x}} = \operatorname{argmin}_{\mathbf{x}} Q_i(\mathbf{x})$ of the next lower priority functional Q_i.[3]

5.2.3 Vertex Clustering

The actual simplification of the input model occurs in a single pass, as summarized by the pseudo-code in Table 5.1. These steps are explained one by one in this section. Once the cluster grid has been determined for the input model, the algorithm proceeds by reading the mesh one triangle at a time (the order in which the triangles are processed is irrelevant.) As these triangles are visited, an in-core representation of the simplified mesh is constructed on-the-fly. It is generally fair to assume that enough memory exists for this simplified mesh since our goal is to produce a mesh coarse enough for in-core tools to process it. Given a triangle $t \in T_{in}$ from the original mesh, its vertex coordinates are fetched and the quadric matrix $\mathbf{Q}^t$ for t is computed. For each vertex v of t, a hash key is constructed from the grid cell (i, j, k) that v falls in (e.g. by concatenating the bit patterns of i, j, and k), and a hash table lookup is made. This dynamic hash table maps grid cells, or clusters, to vertices $\bar{v}$ in the simplified mesh. If this cell has not previously been visited, i.e. if the hash lookup fails, then a new vertex identifier $\bar{v}$ is created (e.g. using consecutive integers) and inserted into the hash table, and the quadric matrix $\mathbf{Q}^{\bar{v}}$ associated with $\bar{v}$ is initialized to zero. In either case, $\mathbf{Q}^{\bar{v}}$ is then updated by adding $\mathbf{Q}^t$ to it, and we proceed with the next vertex of t. If two or more of t's vertices belong to the same cluster, then t reduces to an edge or a point and is discarded. Otherwise, it is added, as a triplet of indices into V_{out}, to the set of simplified triangles T_{out}.

[3] This does not necessarily mean that the best possible solution $\mathbf{x}$ is found, since the projection of the gradient ∇Q_i onto the space of minima for Q_{i+1} does not necessarily vanish at $\mathbf{x}$ (the projection of $\hat{\mathbf{x}}$). Figure 4.3 illustrates this graphically. $\mathbf{x}$ is, however, the closest solution to $\hat{\mathbf{x}}$, and is therefore a good heuristic.

```
1  set V_out and T_out to ∅
2  for each triangle t ∈ T_in
3        fetch vertex coordinates x_1^t, x_2^t, x_3^t
4        compute quadric matrix Q^t
5        for each vertex v of t
6              map v to cluster representative v̄
7              if v̄ ∉ V_out
8                    then add v̄ to V_out and set Q^v̄ to 0
9              add Q^t to Q^v̄
10       if v_1^t, v_2^t, v_3^t belong to distinct clusters
11             then add t to T_out
12 for each vertex v̄ ∈ V_out
13       compute x^v̄ = argmin_x x̄^T Q^v̄ x̄
14 output (V_out, T_out)
```

Table 5.1: Pseudo-code for the OoCS algorithm.

After the input has been exhausted, we are left with a list of quadrics and a list of triangles. Each quadric corresponds to a cluster of vertices and triangles that share a grid cell, and the coordinates for the cluster's representative vertex $\bar{v}$ is computed from the quadric matrix $\mathbf{Q}^{\bar{v}}$ using the procedure described above. The simplification then ends by outputting the simplified mesh (V_{out}, T_{out}) in an appropriate format.

5.3 Results

To evaluate the performance of the OoCS algorithm, I will use four large polygonal data sets: the Buddha, dragon, and turbine blade models from Chapter 4, and a model of Michelangelo's St. Matthew statue created by researchers at Stanford using a range scanner. These models were simplified on one CPU of a 195 MHz R10000 SGI Origin with 4 GB of RAM and a standard SCSI disk drive. Because few, if any, implementations of other out-of-core simplification algorithms are publically available, I will make comparisons only with Garland and Heckbert's QSlim software [50], my own implementation of Memoryless Simplification [94], as well as several variations of the OoCS algorithm, including Rossignac and Borrel's original clustering algorithm [127]. I will first discuss the visual and geometric quality of these simplified models, and then conclude with a numerical comparison of simplification time and memory usage for the different methods.

5.3.1 Geometric and Visual Quality

Before comparing the different methods, I will first give a few examples of out-of-core simplifications and comment on their quality. Figure 5.1 shows the original buddha model and two out-of-core simplified models. Notice how the models in Figures 5.1a and 5.1b are virtually indistinguishable, while some blocking artifacts appear in Figure 5.1c, yet most details are still present.

Figure 5.2 shows opaque and translucent renderings of the turbine blade model. This 28 million triangle model was constructed from the model in Figure 4.7b by applying two levels of Loop subdivision [98],

a. Original model. T = 1,087,716

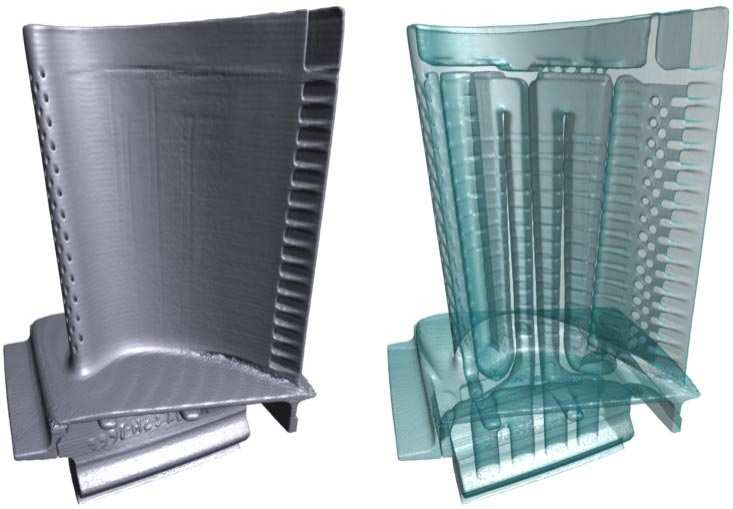
b. T = 204,750. time = 0:17. $105 \times 256 \times 105$ cells.

c. T = 62,354. time = 0:12. $53 \times 128 \times 53$ cells.

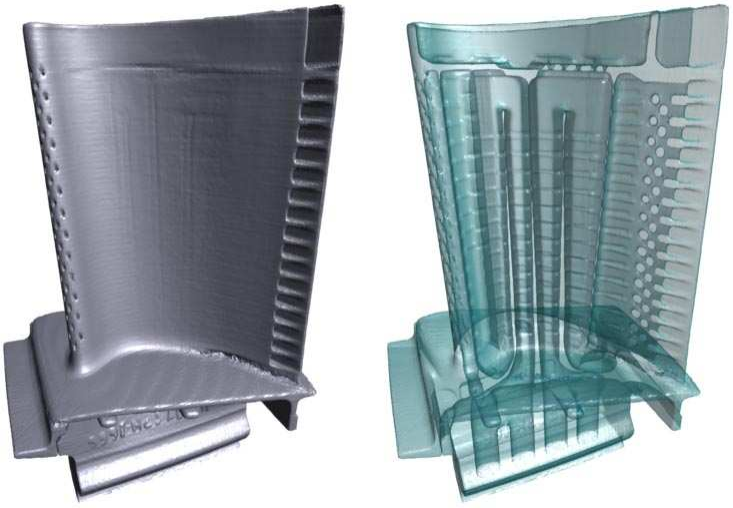
Figure 5.1: Out-of-core simplifications of Buddha model. The number of triangles T, the simplification time (minutes:seconds), and the number of cluster grid cells are indicated.

thereby increasing the triangle count by a factor of 16 and making it more challenging to simplify. This model was too large to simplify using QSlim. Again, the simplified model is difficult to distinguish from the original one. However, this is to be expected since the triangle count of the original model before subdivision is only about 3.5 times that of the simplified model, and the subdivision step did not greatly affect the geometry of the model. The purpose of including this model was mainly to illustrate the dramatic difference in speed and memory usage between OoCS and the memoryless method, which I will discuss further below.

Figures 5.3a and 5.3b show close-ups of the face of the St. Matthew statue, covering less than 15% of its overall height. This complex model consists of nearly 400 million triangles, and could only be simplified using the out-of-core method. Even after a reduction by a factor of 100, many fine details, such as the chisel marks, are still preserved. This model was too large to store as a single file and had to be broken up into several pieces, which were then compressed using the Unix `gzip` general compression tool. During simplification, these different parts of the polygon soup were uncompressed and concatenated on-the-fly.



a. Original model. T = 28,246,208.

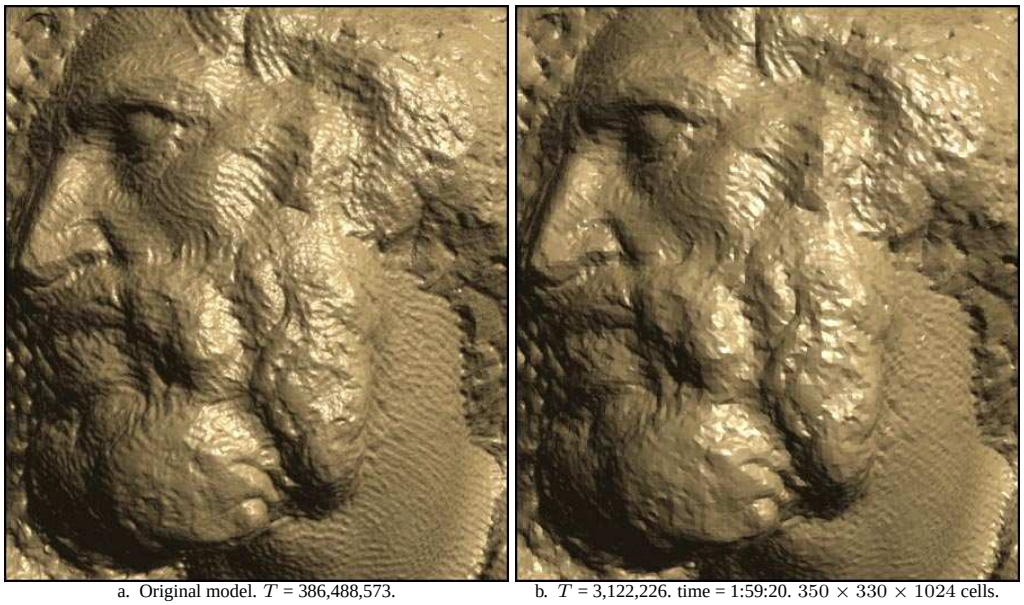


b. T = 507,104. time = 5:02. $151 \times 256 \times 117$ cells.

Figure 5.2: Out-of-core simplification of turbine blade model.

Finally, Figure 5.4 shows several simplifications of the dragon model. I will here compare out-of-core simplification using quadrics not only against QSlim and Memoryless Simplification, but also against other vertex placement strategies using the general OoCS framework. The additional placement methods for a given cluster are: (1) the cluster cell center, (2) Rossignac and Borrel's vertex "grading" method, improved by the importance measure from [102], which chooses the most "important" vertex within a cluster, and (3) the mean of a cluster's vertices. Since Rossignac and Borrel's method requires augmenting the mesh representation with vertex importance values, the OoCS algorithm was modified to accommodate this difference, and a separate pre-processing tool was built for rating the vertices.

Not surprisingly, we can see that the using the cell center, in essence truncating the vertex coordinates,



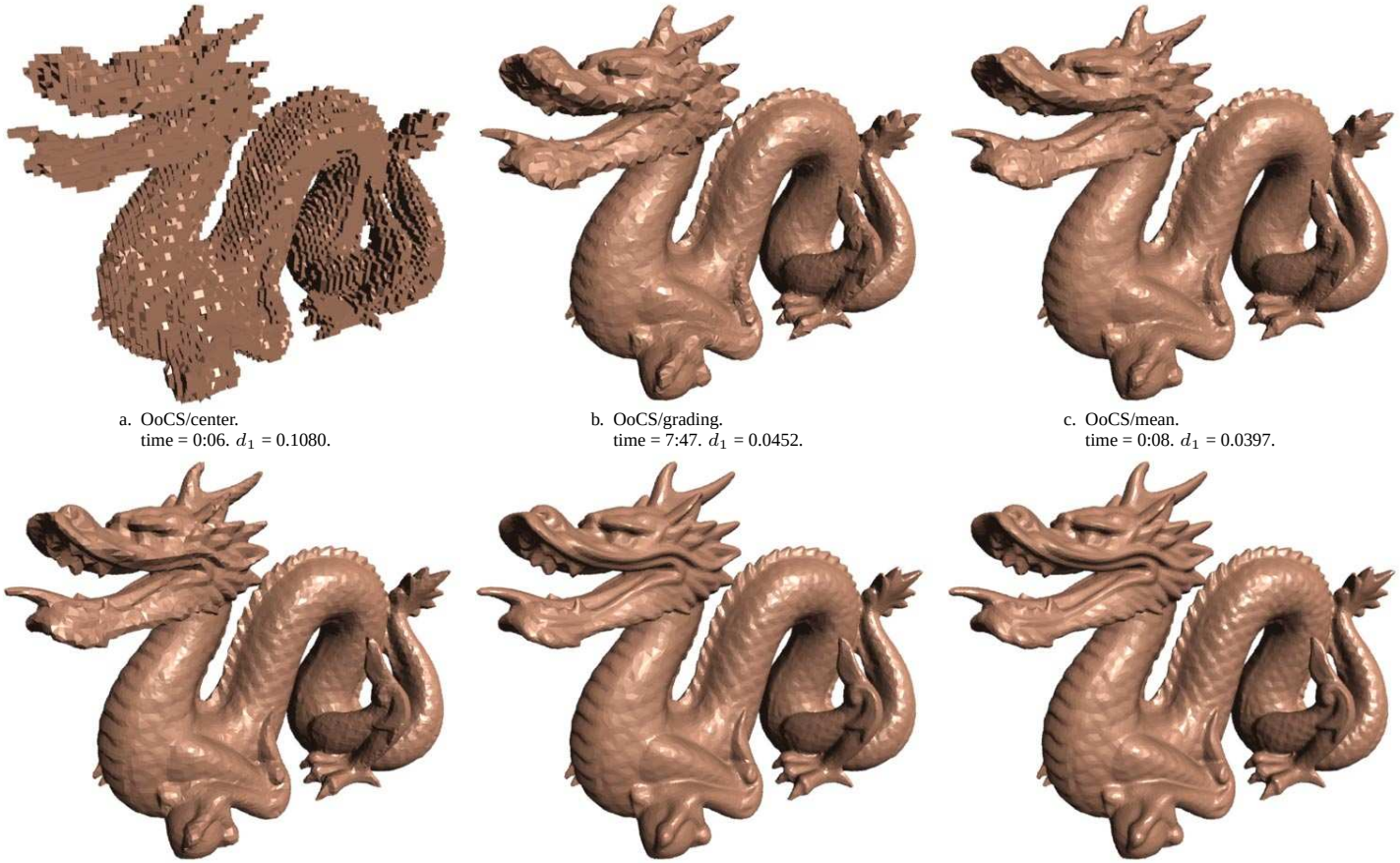
a. Original model. T = 386,488,573. b. T = 3,122,226. time = 1:59:20. $350 \times 330 \times 1024$ cells.

Figure 5.3: Out-of-core simplification of St. Matthew model.

is a very poor choice. The quality is improved greatly using vertex grading, even though there are still some quantization artifacts visible, especially in the face. A much smoother model is produced by the vertex averaging scheme, although fine details near the jaws, neck, and hind leg are washed out, and the ridge along the back has lost its sharpness. The best OoCS model is obtained by using error quadrics. While it is not as good as the two models produced by the in-core methods (Figures 5.4e and 5.4f), it is a clear improvement over the other vertex placement schemes. These subjective observations are also supported empirically by measuring the mean geometric errors of the simplified models (see §4.8 for details). Figure 5.5 shows the errors for four different levels of detail of the dragon model. As expected, both vertex grading and averaging perform much better than using the cell center, while using quadrics results in an additional factor of two reduction in the error. Memoryless simplification performs the best, and while it is significantly better than OoCS with quadrics, the performance gap between OoCS and QSlim is actually smaller than the difference between QSlim and the memoryless method! Thus, OoCS does not only guarantee a maximum error tolerance (the length of the grid cell diagonal), but also produces models with mean errors that are nearly comparable to those of state-of-the-art in-core methods.

While the quality of the OoCS method is high in comparison with other vertex clustering schemes, it does not perform adaptive sampling of the model, and often produces models that can be further coarsened in areas of low curvature with little loss in quality. For applications that require extreme reduction and very high visual quality, the OoCS algorithm can be used as a fast preprocessing step that produces a model with a few hundred thousand triangles, which can then be further simplified by a slower in-core simplification algorithm.

Since the OoCS algorithm does not rely on any connectivity information, it has no way of detecting boundary edges (or non-manifold edges for that matter). As a consequence, surface boundaries are generally



a. OoCS/center.
time = 0:06. d_1 = 0.1080.

b. OoCS/grading.
time = 7:47. d_1 = 0.0452.

c. OoCS/mean.
time = 0:08. d_1 = 0.0397.

d. OoCS/quadrics.
time = 0:10. d_1 = 0.0186.

e. QSlim.
time = 6:14. d_1 = 0.0120.

f. Memoryless simplification.
time = 15:13. d_1 = 0.0054.

Figure 5.4: Simplifications of dragon model. Each model was simplified from 871,306 to 47,236 triangles. $100 \times 70 \times 45$ cells were used in the OoCS method.

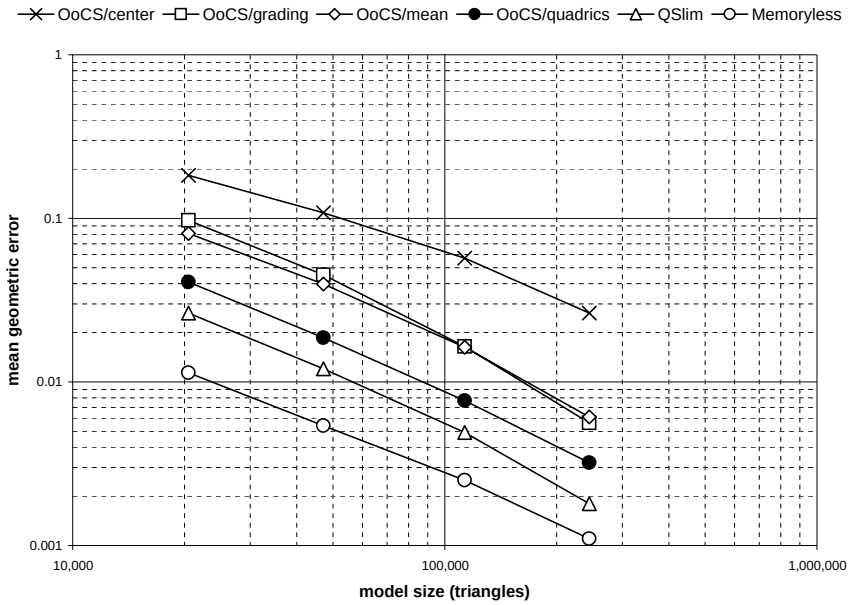


Figure 5.5: Mean geometric errors for the dragon model.

not preserved well. Like all other algorithms based on clustering and pair contraction, manifold surfaces will not generally remain manifold after simplification, and the clustering operation occasionally produces double-sided faces in areas where the mesh folds onto itself.[4] In fact, some of the meshes presented above contain several duplicate triangles which have not been removed and are included in the triangle counts reported. For rendering applications, this is generally not an issue since back-face culling can be used to eliminate undesirable faces [159]. For applications that demand manifold surfaces, methods such as [58, 110, 128] can be used to turn a non-manifold surface into a manifold one. For surfaces with very complex topology, however, the ability of closing small holes and merging disconnected components via clustering operations often outweighs the downsides of non-manifold meshes [50, 104, 111].

5.3.2 Memory Utilization and Simplification Speed

When simplifying very large models, speed and memory usage become crucial factors to consider. In some cases, these computer resources are scant enough that out-of-core simplification is a necessity, as is the case for the St. Matthew statue. In others, both in-core and out-of-core methods are possible, and in order to decide on the best method, speed must be weighed against quality. In either case, one may consider the hybrid approach suggested above by performing a fast out-of-core pre-simplification of a large model, followed by a slower, but more accurate, in-core simplification. I will conclude this chapter by comparing memory usage and speed for OoCS and the in-core methods.

[4] A double-sided face is one that is made up of two triangles with the same set of vertices but with opposite orientation.

model	$\|T_{in}\|$	$\|T_{out}\|$	*RAM (MB)*			*time (h:m:s)*		
			QSlim	*MS*	*OoCS*	*QSlim*	*MS*	*OoCS*
dragon	871,306	244,562	213	134	28	5:31	11:59	0:16
dragon	871,306	113,090	214	134	11	5:55	14:12	0:12
dragon	871,306	47,236	214	134	7	6:06	15:21	0:10
buddha	1,087,716	204,750	250	166	26	7:13	16:58	0:17
buddha	1,087,716	62,354	251	166	8	7:35	19:19	0:12
blade	28,246,208	507,104	-	3,185	63	-	12:37:25	5:02
statue	386,488,573	3,122,226	-	-	366	-	-	1:59:20

Table 5.2: Simplification results of running QSlim [50], Memoryless Simplification [94] (MS), and the out-of-core method (OoCS).

Table 5.2 includes the triangles counts, memory usage, and timing results of simplifying the four test models using OoCS (with quadrics), QSlim, and Memoryless Simplification. The memory usage reported here was obtained from the `top` Unix command, while the simplification time corresponds to wall clock time, including model input and output time. As can be seen from this table, while being much more memory efficient than both in-core methods, OoCS is also orders of magnitude faster, reaching a simplification speed of up to 100,000 triangles per second. On average, this translates into 30 times faster than QSlim and 100 times faster than Memoryless Simplification. Using [121] as a representative case of out-of-core simplification based on model segmentation and stitching (see §3.1.5), OoCS is about 100 times faster and, by extrapolating their results, an order of magnitude more memory efficient. Note that the reported memory usage in Table 5.2 is consistently higher than the current implementation's theoretical usage of 63 to 72 $|T_{out}|$ bytes,[5] as the former includes freed memory not reclaimed by the operating system. The variation in the theoretical usage is due to the varying size and load of the dynamic hash table. The hash load in the current implementation varies between $\frac{1}{3}$ and $\frac{2}{3}$. As a comparison, my rather efficient implementation of memoryless simplification requires 80 or 92 $|T_{in}|$ bytes, depending on whether (32-bit) single or (64-bit) double precision floating point is used. Using the same data structures, Garland and Heckbert's method requires an additional 20 or 40 $|T_{in}|$ bytes for storing the quadric matrices. Note the important difference of input vs. output sensitivity; the memory usage of the OoCS algorithm does not depend on the size of the input model, whereas the in-core methods are entirely limited by its size. Since the input model is generally vastly larger than the output model, and since the complexity of models that need simplification is increasing rapidly, I anticipate that the OoCS algorithm and similar input insensitive algorithms will become increasingly important in the future.

[5] This usage assumes 64-bit double precision for the quadrics.

Chapter 6

IMAGE METRICS FOR SIMPLIFICATION AND OPTIMIZATION

By far the most common use of simplification is to produce a model that is *visually similar* to the original, yet existing methods for simplification rely on geometric closeness as an indirect indicator of visual quality. The geometric shape of a model is unquestionably integral to its appearance, but it is by no means the only contributing factor. Color and texture, for example, can dramatically influence the appearance, as evidenced by Figure 3.1. Other techniques often used with polygonal meshes, such as bump, normal, and environment mapping, often add a great degree of realism to a model. The choice of rendering style, such as whether flat, Gouraud, or Phong interpolation is used to shade the model, as well as material properties, such as translucency, specularity, and emitted light, also define how the surface of the model reflects and transmits light. There are also other geometric considerations beyond the deviation between the two surfaces, such as whether a part of the model is likely to be on a silhouette, or whether it is even visible at all. In addition, geometric artifacts such as mesh folds and interpenetrations can be visually distracting, but are difficult to detect robustly and efficiently using geometry-based metrics.

While some previous simplification methods incorporate basic support for preserving properties such as color and texture coordinates using separate metrics for color and parametric error, it is very difficult to design a single metric based on a set of such heuristics that robustly accounts for the complex interactions between geometry, color, shading, etc. Such heuristics alone cannot reliably be used, for example, to determine whether a one percent error in position has a greater impact on appearance than a one percent error in the texture parameterization, or whether these two errors multiply or even offset each other. Instead, to accurately measure the visual similarity between an original model and a simplified version, different inputs to the metric are needed. In the following three chapters, I will describe how *rendered images* of objects and *image metrics* can be used to measure their visual similarity. Based on this framework, I will later develop algorithms that use image metrics to guide simplification and optimization of polygonal models.

It is worthwhile to consider what applications might benefit from an image-based measure of visual similarity. A few applications, such as CAD/CAM, collision detection, and mesh signal processing, are likely to require a close geometric match to the original model. Visual similarity may be entirely inconsequential for such applications. More common, however, are applications that require the two models to appear similar, including vehicle simulations, architectural walk-throughs, virtual reality, acceleration of off-line rendering,

video games, movie special effects, and web-based e-commerce applications that make use of 3D graphics. For this set of applications, visual similarity is imperative, and an image-driven simplification method is appropriate.

In this chapter, I will lay the groundwork for using image metrics to evaluate the visual quality of a simplified model. I will first describe how to perform the off-line quality evaluation, and later present specialized data structures and algorithms for quickly evaluating the image metric and updating the images after a local change to the mesh, such as en edge collapse, has been made. I will use the term "run-time evaluation" to refer to computing the metric during simplification or optimization. These applications demand continuous evaluation of the metric, possibly over a long period of time. This use of "run-time" should not be confused with evaluating a metric to manage and display level of detail representations in interactive graphics systems. I will conclude this chapter by describing a new perceptually motivated metric that can be used to perform an off-line quality evaluation. This metric has been designed with computational efficiency in mind, which also makes it a good candidate for driving simplification.

6.1 Off-Line Evaluation of Model Quality

In this part of the chapter, I will discuss how to use multiple images, taken from a sphere of camera positions, to compare polygonal models. This form of comparison will later provide the error measure used in my image-driven simplification and optimization methods. Before describing the procedure itself, I will give a brief introduction to image metrics.

An *image metric* is a function over pairs of images that gives a non-negative measure of the distance between the two images. Roughly speaking, the larger the distance, the more dissimilar the images appear, while a zero difference implies that the two images are identical. As pointed out in §3.3, such a function has to satisfy a list of properties to formally constitute a metric, although I will here relax these requirements by leaving the triangle inequality property out, since it is of little practical value for our purposes.

While several image metrics have been proposed in the literature, some which are quite elaborate, I have for simplicity chosen to use a less complex metric in this chapter when describing the procedure for the quality evaluation. Without loss of generality, I will use the pixel-wise mean square error (MSE) metric d_2^2. This metric was chosen in part because of its efficiency and high locality. As it turns out, it also happens to be reasonably well suited for simplification and optimization, and most of the results presented in later chapters were produced using the MSE metric. Note, however, that the framework presented here is general enough that any image metric can be used, albeit at potentially lower computational efficiency.

Even though many colored models have separate red, green, and blue components, the metrics discussed here all use a single luminance channel Y for each image as input, and measure only differences in luminance. This has worked well for all test models I have used. Of course, there is nothing inherent in this approach that precludes taking differences in hue into account, yet such a metric would require additional storage and computation, and would add little other than the ability to distinguish variations in color between equiluminant regions—a feature that I have not found to be of any greater significance. To compute Y, the

standardized coefficients from Recommendation 709 [118] (which pertains to contemporary monitors) are used to weight the RGB channels:

$$Y = 0.2126R + 0.7152G + 0.0722B$$

Based on this, I will define the mean square difference d_2^2 between two luminance images Y and Y' of dimensions $m \times n$ pixels as

$$d_2^2(Y, Y') = \frac{1}{mn} \sum_{j=1}^{m} \sum_{i=1}^{n} \left(y_{ij} - y'_{ij}\right)^2 \tag{6.1}$$

I will now cover the general procedure for comparing models.

6.1.1 Comparing Models using Multiple Views

We cannot hope to capture the entire appearance of an object in a single image. Ideally, we wish to capture the set of all radiance samples that emanate from the surface of an object under all possible lighting conditions. This is obviously not possible in practice. To capture a large collection of radiance samples of the object, I render images from a number of different camera positions around the object (Figure 6.1), and the image metric is applied to this entire set of images. These per image differences must then be combined into a single measure of error. The image metric itself will often suggest how to do this, e.g. for d_2^2 it is natural to sum the differences over all pixels in all images. Formally, given two sets of l images $\mathcal{Y} = \{Y_k\}$ and $\mathcal{Y}' = \{Y'_k\}$ of dimensions $m \times n$ pixels, the mean square difference between two sets of images is computed as

$$d_2^2(\mathcal{Y}, \mathcal{Y}') = \frac{1}{lmn} \sum_{k=1}^{l} \sum_{j=1}^{m} \sum_{i=1}^{n} \left(y_{ijk} - y'_{ijk}\right)^2 \tag{6.2}$$

The root mean square error (RMSE) d_2, which will be used for quality evaluation in later chapters, is defined similarly by taking the square root of the result. For more complex image metrics, such as the one described in §6.3, the mean or max norm can be used to consolidate the individual image differences.

6.1.1.1 Rendering Parameters

There are a number of choices that need to be made to produce the images used in the quality evaluation, including image dimensions and camera settings for each image, the choice of background to render the objects on, light source information, and shading parameters. For most of these parameters, there is no obvious best choice. While seemingly discouraging at first, this may instead be desirable as it gives the user freedom to choose an environment that is suitable for the application. Note that geometry-based metrics would not have an option to incorporate such parameters. In either case, it is not exceedingly difficult to come up with a set of reasonably neutral choices for these variables if so desired.

In this section, I will discuss the rendering parameters used in my own implementation. Most of these were chosen with the goal of capturing a large amount of information about the models, while simultaneously selecting a set of parameters that are representative of actual graphics applications.

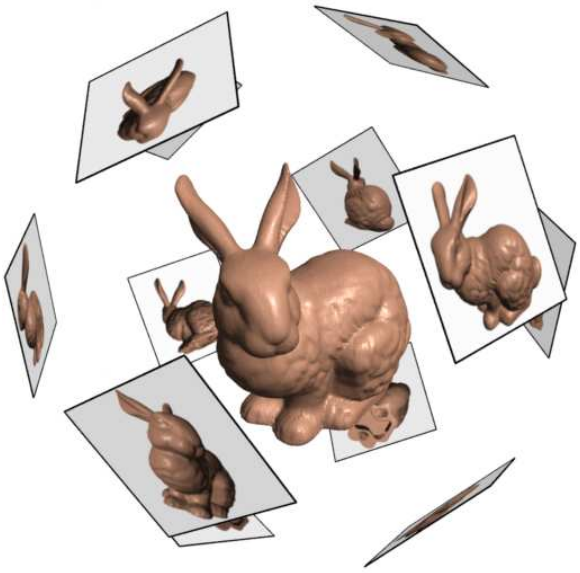


Figure 6.1: Example of 12 camera views surrounding an object. The viewpoints and view orientations are distributed uniformly.

Camera and Image Parameters While the number of views required and the "optimal" placement of viewpoints vary between objects, I have chosen to limit myself to a fixed set of viewpoints, and require that they are distributed (roughly) uniformly in direction. For the many models I have used, a fixed set of camera positions has proven adequate, even for the less uniformly shaped objects, although it is possible that these viewpoints do not yield optimal or even total coverage of more complex surfaces. This issue has been addressed, for example, in surface reconstruction to ensure total coverage of a model using a minimum number of overlapping range scans [115]. Other promising methods are also currently being developed to address this optimization problem [139].

To ensure even coverage of image samples, the viewpoints are arranged so as to approximate a sphere of camera positions surrounding an object, and the viewpoints are chosen to be (near) equidistant from each other. In practice, only five possible configurations—the vertices of the Platonic solids—exist for which the viewpoints are uniformly distributed. Other reasonable configurations can be obtained using subdivision of these polyhedra, or by using the vertices of the semiregular Archimedean solids. For off-line quality evaluation of a model, I use the 24 vertices of the *small rhombicuboctahedron* (Figure 6.2). For simplification and optimization, I use fewer viewpoints to make the metric more efficient to evaluate, and I have experimented with the 4, 6, 8, 12, and 20 viewpoints associated with the Platonic solids. For the majority of results presented in this thesis, 20 viewpoints from the vertices of a regular dodecahedron were used during simplification. Another reason to use different camera positions for these two cases is to avoid too much bias towards models that were created using an image-driven method when comparing the results of different simplification methods. Also to avoid bias, I use Pixar's *PhotoRealistic RenderMan* [150] to produce high-quality

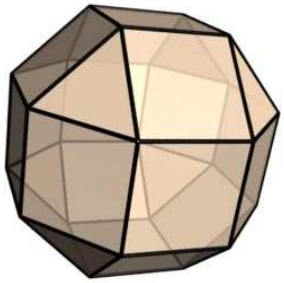


Figure 6.2: The small rhombicuboctahedron. The 24 vertices of this uniform polyhedron are used as the viewpoints surrounding an object in the off-line quality evaluation. See also Figure 6.3.

512×512 images for off-line quality evaluation, while fast scan conversion without antialiasing is used to generate 256×256 images for run-time evaluation.

In order to ensure that the model is entirely contained in each image, the minimum bounding sphere of its vertices is first computed. This sphere can be computed quickly in expected linear time using the algorithm in [156]. Assuming the images are square and that perspective projection is used, the radius r of the bounding sphere and a user-specified field of view φ are used to determine the shortest distance r' from which the object is guaranteed to be seen in each image, i.e.

$$r' = \frac{r}{\sin \frac{\varphi}{2}}$$

For a 60° field of view—the default setting—we have $r' = 2r$. The camera positions are then constrained to lie on this larger sphere with radius r'.

There is no compelling reason to use perspective projection; an orthographic projection would work equally well, and may additionally reveal a larger part of the surface in each view. Since perspective projection is a more popular choice for graphics, however, it has the potential to produce images that better represent actual scenes in which the two objects are used.

Given a set of fixed camera positions, the camera at each viewpoint is directed towards the center of the bounding sphere, and the camera up-vectors are varied roughly uniformly to avoid any bias in orientation. To avoid registration errors, the same set of viewpoints is used for both of the two objects being compared, with one object acting as the reference mesh from which the bounding sphere is computed.[1]

Scene Background Without any specific information about the environment in which the models will appear, a solid gray (50% intensity) background is used. Even though it is possible for parts of the surface silhouette to blend with the background, leading to an apparent loss in visual importance, it is highly unlikely that the same surface patch would be "camouflaged" in several views. In addition, silhouettes often approach tangency to the view direction, leading to a sharp intensity gradient and causing the silhouettes to appear

[1] While this approach does not guarantee that parts of the simplified mesh do not extend outside the images, such geometric differences will not likely go unnoticed in all views, and I have not observed this to be a problem.

dark, as relatively little light is reflected back to the viewer. Based on this observation, it may be possible to choose different background intensities and patterns to either emphasize or downweight the importance of silhouettes.

Lighting Rather than using a single configuration of light sources that are fixed in relation to the object, I have chosen to use one light source per view that is slightly offset from the viewpoint. Consequently, the front of the model (as seen from each viewpoint) is always illuminated. This is done to maximize contrast and bring out as much detail in the model as possible in each view.

Shading Unless otherwise specified, flat (per triangle) shading is used, in part because it avoids construction of inherently ill-defined vertex normals, and also because the most common alternative, Gouraud shading, often results in meshes of poor visual quality at very low polygon counts. The loss in detail incurred by Gouraud shading[2] and its tendency to smear dark shading samples, resulting from normals facing away from the light source, across the surface near silhouettes makes it less useful for coarse models. If no material properties accompany the models, a default white (100% intensity) color is used with a nearly Lambertian surface (a mostly diffuse surface with small ambient reflectance and no specular reflectance). Without the benefit of (per pixel) Phong shading, specular materials and flat shading of large facets do not mix well, as very small changes in geometry can lead to dramatic differences in reflected light.

Using the default parameters discussed here, Figure 6.3 shows a uniform distribution of views of the Stanford bunny model.

6.1.1.2 Comparison to Light Fields

The rendering community has recently devoted attention to the idea of capturing a large portion of the *light field* or *plenoptic function* of a given scene. A light field is a 5D function that represents the radiance at a given 3D point in a given direction. Several groups of researchers have used sampled representations of a light field for image-based rendering [18, 55, 89, 108, 138]. Recognizing that in free space the radiance does not change along a ray, the dimension of the light field around an object can be reduced to 4D. One such representation of the light field, used by [55, 89], is a collection of images taken from a 2D array of camera positions on a flat surface. The sphere of camera positions that I use for comparing models is also a 4D representation of the radiance of an object. Two of the dimensions are from the pixel position in an image; the other two are from the camera position on the sphere surrounding the object, similar to the parameterizations used in [18]. At the limit, an infinitely dense set of camera positions on the sphere and infinite image resolution would give a complete representation of the free-space light field for an object. This assumes that the camera viewing direction and the field of view are set so that no part of the model lies outside the image, as is the case for the images used in my model comparison framework.

[2]This is due to the fact that in an average mesh there are twice as many triangles as there are vertices, and consequently twice as many shading samples in flat shading.

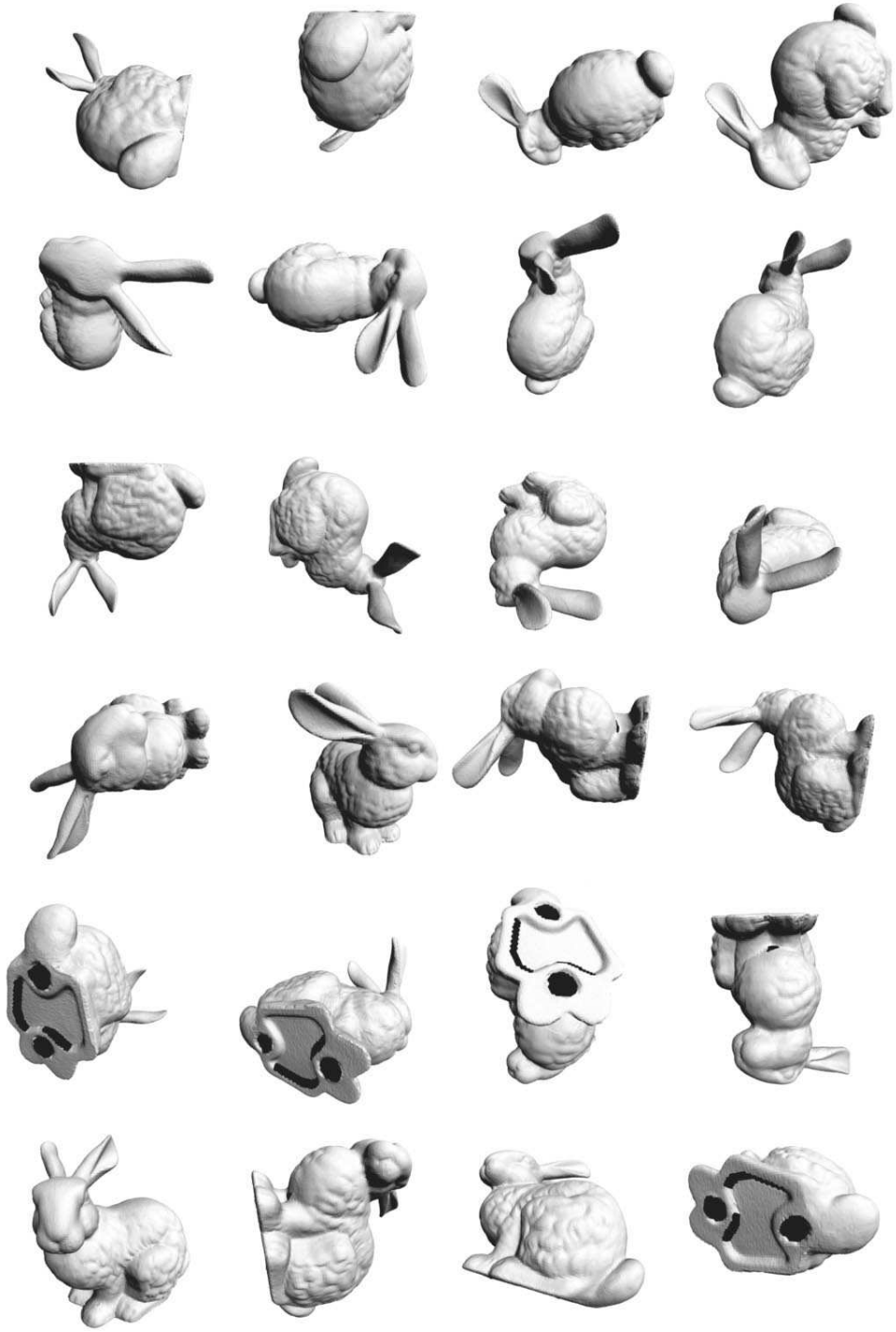


Figure 6.3: The twenty-four different views of an object used during off-line quality evaluation. The viewpoint distribution corresponds to the vertices of the small rhombicuboctahedron.

One difference between my approach and most image-based rendering (IBR) techniques based on light fields is the lighting of the object. Most IBR methods assume fixed lighting for a given object or scene, whereas I have chosen to position the light differently for each camera position, fixed to the *viewer*, in order to bring out a large amount of detail in each image. The space of radiance samples under all possible lighting conditions is an infinite dimensional space, therefore compromises must be made in order to sample this space using finite computational resources and memory.

6.2 Run-Time Evaluation of Model Quality

The image comparison framework discussed in the previous section is a very natural approach to measuring the visual quality of simplified models. We are not only interested in evaluating the results of different simplification algorithms, however, but would also prefer to put image metrics to use directly in simplification and optimization methods. For example, we would like to use an image metric to measure the visual impact of collapsing an edge, and use this error measure to order the sequence of edge collapses. The method in §6.1, as stated, is unfortunately far too inefficient to be of practical use in such applications. In the most naive approach, the cost of collapsing an edge would have to be evaluated by attempting the collapse, rendering the resulting mesh from scratch from several viewpoints, and then evaluating the image metric over all pixels in these images. Even using high-end graphics hardware to render the images and using the fast MSE image metric to compare them, it may take as much as a second per edge cost evaluation to clear the frame buffer, render the views, read back the pixels, and evaluate the metric. Clearly, such a method would be prohibitively slow.

A key observation will allow us to greatly improve the running time of these proposed methods: the operations needed to perform simplification and optimization are local in nature—each operation affects only a very small portion of the mesh. For simplification, the edge collapse operator moves only a few triangles. Similarly, the optimization method described in Chapter 8 improves the mesh by using local connectivity operations and making small perturbations to just a handful of nearby vertices. Consequently, these operations exhibit high spatial locality and "temporal" coherence in the sense that consecutive meshes are nearly identical. We can take advantage of this fact in the following two ways. First, assuming that we have already produced images of the model before the operation is made, updating these images requires only rerendering a small part of the mesh. Second, since only a fraction of pixels are affected by the operation, the image difference can be evaluated more efficiently knowing that the remaining pixels will not change. In this section, I will describe how to perform these two steps—quickly updating the images and evaluating the image metric incrementally—using efficient data structures and algorithms. In §6.2.2.1, I will discuss a solution that uses software-based rendering and a multiple-layer frame buffer. Then, a more efficient method, using hardware graphics acceleration, is covered in §6.2.2.2. First, I will briefly revisit the image comparison method that is the basis for the error metric used in my image-driven simplification and optimization methods. By making a few modifications, I will show how this metric can be evaluated more efficiently.

6.2.1 Fast Evaluation of Metric

My image-driven simplification and optimization algorithms both make use of local operations to change the mesh. These operations, whether they involve changes to the connectivity or the geometry, can be described in terms of replacing a set of triangles T with another set T'. (These sets may be topologically equivalent, but their geometric extents may differ whenever their supporting vertices are moved.) As an example, consider collapsing an edge $\bar{e}$ to a vertex $\bar{v}$. For this operation, $T = \lceil\lceil\lfloor\bar{e}\rfloor\rceil\rceil$ is replaced by $T' = \lceil\lceil\bar{v}\rceil\rceil$, while the remaining triangles in the mesh are unaffected. In this section, I will consider the general case of substituting a set of triangles, regardless of whether the operation is for simplification or optimization. I will refer to such an operation as a *move*. Later, in Chapter 8, I will distinguish between connectivity moves, e.g. edge collapse, swap, and split, and geometry moves, e.g. changing the position of a vertex. In both simplification and optimization, we need to consider the original or target model $\widehat{M}$ that we wish to replicate, the current model M to which the triangles T belong, and the model M' obtained by replacing T with T'. For each of these models, we have associated sets of images $\widehat{\mathcal{Y}}$, $\mathcal{Y}$, and $\mathcal{Y}'$.

As in Chapter 4, I will define the error metric as a relative quantity. That is, the cost Δd associated with a move is defined to be the change in image difference:

$$\Delta d(\mathcal{Y}, \mathcal{Y}', \widehat{\mathcal{Y}}) = d(\mathcal{Y}', \widehat{\mathcal{Y}}) - d(\mathcal{Y}, \widehat{\mathcal{Y}}) \tag{6.3}$$

In simplification, Δd is typically positive, although occasional negative Δd are possible whenever an edge collapse *improves* the mesh quality, such as when straightening out a visually distracting fold in the mesh. For optimization, we are generally interested only in moves that reduce the total error, i.e. $\Delta d < 0$. Even though the metric d inherently measures global differences, the main reason for using incremental instead of absolute errors (§3.3.1.3) is that it allows the cost of a potential move to be evaluated asynchronously from other moves, and ranked more or less independent of whether changes in other areas of the mesh are made.[3] To see why this is important, consider using the absolute error $d(\mathcal{Y}', \widehat{\mathcal{Y}})$ as the cost measure for ordering a sequence of edge collapses. After an edge has been collapsed, the error d associated with any nearby affected edges would have to be re-evaluated. This update occurs asynchronously from the evaluation of all remaining edges, however. Because d becomes more and more inaccurate over time as changes are made to the mesh, the edge collapses will inevitably occur out of order eventually unless an expensive forced synchronization of all edge costs is done periodically. Clearly, this is undesirable.

Given the general definition for Δd from Equation 6.3, I will now narrow my focus by considering a specific metric: the mean square error d_2^2. I have purposely chosen this simple metric because it makes the discussion below easy to follow, but it should be relatively straightforward to extend these principles to more complex metrics. We can define the incremental mean square error as

$$\Delta d_2^2(\mathcal{Y}, \mathcal{Y}', \widehat{\mathcal{Y}}) = \frac{1}{lmn} \sum_{k=1}^{l} \sum_{j=1}^{m} \sum_{i=1}^{n} \left[\left(y'_{ijk} - \widehat{y}_{ijk} \right)^2 - \left(y_{ijk} - \widehat{y}_{ijk} \right)^2 \right] \tag{6.4}$$

[3] This may not be the case for very complex metrics that involve inter-dependencies between large portions of the images, such as metrics based on hierarchical decompositions. However, even in such cases, incremental errors are still likely to be less volatile than absolute errors.

Note that any pixel satisfying $y_{ijk} = y'_{ijk}$ makes no contribution to Δd_2^2. In fact, this holds for the majority of pixels due to the spatial locality of the moves—the only pixels that can differ between the images $\mathcal{Y}$ and $\mathcal{Y}'$ are the ones covered by the triangles $T \cup T'$. By computing a screen space axis-aligned bounding box $R_k = I_k \times J_k$ for each view k around these triangles, we obtain a conservative estimate of the affected pixels. Visiting only this smaller set of pixels results in an expression for Δd_2^2 that is faster to evaluate:

$$\Delta d_2^2(\mathcal{Y}, \mathcal{Y}', \widehat{\mathcal{Y}}) = \frac{1}{lmn} \sum_{k=1}^{l} \sum_{j \in J_k} \sum_{i \in I_k} \left[\left(y'_{ijk} - \widehat{y}_{ijk}\right)^2 - \left(y_{ijk} - \widehat{y}_{ijk}\right)^2 \right] \tag{6.5}$$

For some metrics, pixel-wise cancellation of differences may not be possible, and $d(\mathcal{Y}', \widehat{\mathcal{Y}})$ must be computed explicitly. A simple example is the root mean square error, for which the square root prevents the cancellation. However, the computation of $d(\mathcal{Y}', \widehat{\mathcal{Y}})$ can often be accelerated by reusing partial information from the previous evaluation of $d(\mathcal{Y}, \widehat{\mathcal{Y}})$.

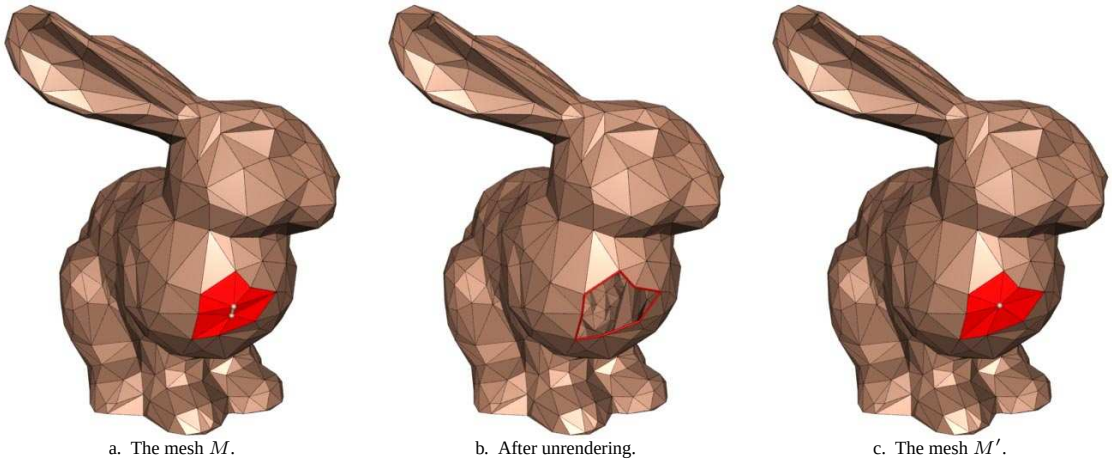
6.2.2 Fast Image Updates

So far, I have described how to evaluate Δd given sets of images $\mathcal{Y}$, $\mathcal{Y}'$, and $\widehat{\mathcal{Y}}$. I will now explain how these images are generated. The images $\widehat{\mathcal{Y}}$ of the original model $\widehat{M}$ need of course be rendered only once, at startup, after which $\widehat{M}$ is no longer needed. As for $\mathcal{Y}$ and $\mathcal{Y}'$, the basic approach in both of my image-driven methods is to maintain images $\mathcal{Y}$ of the "current" model, i.e. the partially simplified model or the most optimal mesh found so far, and, for each move considered, make incremental updates to these images to produce $\mathcal{Y}'$. After a move is accepted, $\mathcal{Y}$ is replaced with $\mathcal{Y}'$. As pointed out in the previous section, only parts of $\mathcal{Y}'$ need to be generated in general, and I will in this section describe data structures and algorithms for efficiently querying what portions of the mesh to render in order to produce the necessary subimages.

The goal of the process described here is to replace a small set of triangles T in the current model with T', without having to rerender the entire mesh from scratch. On average, these sets each contain less than a dozen triangles, which is small compared to the thousands or even millions of triangles in the original model $\widehat{M}$. Therefore, we can gain significant time savings by reusing the images $\mathcal{Y}$ that have already been rendered, and doing the minimal amount of rendering to swap out T with T'. Conceptually, the substitution of triangles can be done by "unrendering" T, revealing any obscured parts of the surface, and then rendering the replacement triangles T' (Figure 6.4). Unfortunately, few rendering systems, whether implemented in software or in hardware, support such an unrendering operation. In the following two subsections, I will describe how to efficiently implement unrendering in both software and hardware.

6.2.2.1 Software-Based Method: "Unrendering" using an Object Buffer

When unrendering a set of triangles T, we must ensure that parts of the surface previously occluded by these triangles reappear. Frame buffers in conventional rendering systems record only the color and depth of the top, visible layer of a model, and are therefore inadequate for this task. In the technique described here, a "multi-layer" frame buffer is used that maintains all layers of a surface. Atherton [5] use the term "object



a. The mesh M. b. After unrendering. c. The mesh M'.

Figure 6.4: The "unrendering" operation for collapsing an edge. The triangles T, highlighted in red in 6.4a, are unrendered, leaving a hole in the model that exposes the model interior (6.4b), which is then covered by rendering the replacement triangles T', as seen in 6.4c.

buffer" to refer to this data structure. Similar data structures, called "layered depth images," have been used recently for image-based rendering [137]. In the object buffer, each pixel is represented as a depth-sorted list of all surface fragments that project into it. Along with the color and depth information stored with each entry in the pixel list is a primitive identifier that uniquely references the triangle that produced the pixel entry. The unrender operation scan converts a triangle and removes from the pixel lists visited those entries that match the triangle's primitive identifier, while pixel entries for surfaces previously occluded by the triangle are promoted towards the front of the frame buffer.

For models with relatively low depth complexity, the computational overhead of maintaining multiple layers instead of one is negligible, and the added memory requirements are not prohibitively large (10–20 MB in our case). However, the low fill-rate afforded by a software-based renderer greatly limits the speed of updating the images. Below, I will describe how hardware rendering can be used to accelerate image generation. Compared to the software-based approach, I have found that using graphics hardware can speed up simplification by as much as a factor of fifty.

6.2.2.2 Hardware-Based Method: Spatial Subdivision of Images

In this section, I will describe a fast method for updating images that uses conventional graphics hardware to do the rendering. My implementation is based on *OpenGL* [159] and uses *pixel buffers* [77] for hardware-assisted off-screen rendering.[4] Pixel buffers are used in order to avoid occupying the entire screen. For our purposes, they otherwise behave like a standard displayed frame buffer.

Even though unrendering is not supported in hardware, it is possible to limit the number of triangles

[4]The pixel buffer was originally a vendor specific extension to OpenGL, but has recently been incorporated into the standard set of features.

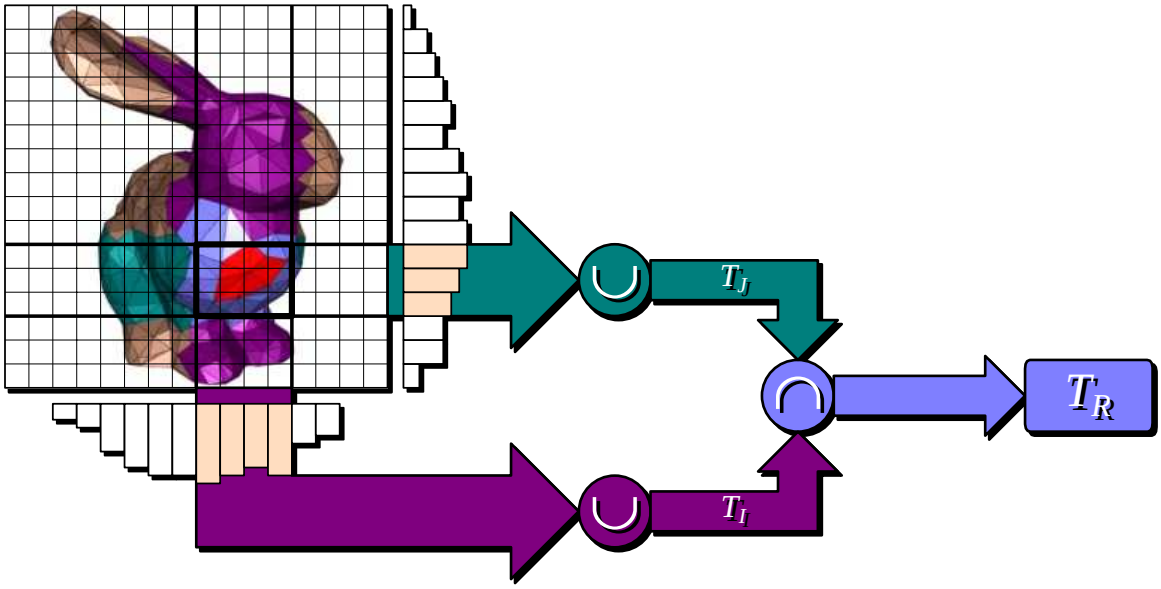


Figure 6.5: The triangle bucket data structure. The triangles of the model are projected onto the two image axes and are maintained in hash tables for all pixel rows and columns. This data structure allows for all triangles T_R (shown in violet) that intersect the rectangular region R surrounding the triangles T' (red) to be accessed quickly. T_R is found by computing the intersection of the triangles T_I (magenta) and the triangles T_J (blue-green).

that have to be rendered to produce $\mathcal{Y}'$. Using the rectangular bounds $\{R_k\}$ on affected pixels from §6.2.1, we can for each view k determine which triangles of M' would be needed to rerender the region R_k from scratch. That is, we can produce Y'_k from Y_k by clearing the subimage R_k, querying what triangles T_{R_k} of M' intersect this region, rendering these triangles, and reading back the pixels contained in R_k. All of these steps are trivial, except for the two-dimensional range query that returns the triangles T_{R_k} in R_k.

There are many well-known data structures for performing fast range queries, including quadtrees [131] and k-d trees [8]. The approach taken here uses a somewhat different subdivision of the search space (the images) and is a bit more fine-grained. In my implementation, a triangle "bucket" is maintained for each row and column of pixels (Figure 6.5). The buckets are implemented as hash tables that are indexed by the triangle identifiers. This subdivision allows us to efficiently cull away most triangles that do not intersect the rectangular region $R_k = I_k \times J_k$ in each view k, leaving the set T_{R_k}. Since the procedure is the same for all views k, I will omit the subscript k in the next paragraph for the sake of readability.

The range query is accomplished by computing the union of the vertical buckets $T_I = \cup_{i \in I} T_i$ and the horizontal buckets $T_J = \cup_{j \in J} T_j$ spanned by R, and then letting $T_R = T_I \cap T_J$ be a conservative (but generally tight) estimate of the set of triangles contained in R. Because the triangle buckets extend across the image, each hash table T_i can get quite large. To avoid having to merge several large hash tables, the computation of unions is accelerated by maintaining an additional set of tables $\Delta T_i = T_i \setminus T_{i-1}$ that are the set differences between consecutive pixel columns (and similarly for the horizontal buckets). That is, ΔT_i

contains the triangles whose left-most vertex is in column i. In general, the hash tables ΔT_i are considerably smaller than the tables T_i. We can then rewrite T_I as $T_I = T_{\min I} \cup \Delta T_{(\min I)+1} \cup \cdots \cup \Delta T_{\max I}$, with the property that the sets in this union are pair-wise disjoint. Given T_I and T_J, the intersection T_R can be computed in linear time by associating a "time stamp" with each triangle. Prior to computing T_R, a unique time stamp is chosen for this range query. While building the set T_I, all triangles encountered are marked with the new time stamp. As T_J is traversed, only the triangles with the given time stamp are added to T_R.

Even though the sets T_I and T_J may be much larger than their intersection, this simple image partitioning scheme has proven very efficient in practice. Compared to an implementation that uses a k-d tree partitioning of each image, the bucket-based implementation described in this section runs two to three times faster.

To summarize, the unrendering step can be implemented using common graphics hardware as follows. To replace T with T', each region R_k in which these sets of triangles are contained is cleared. The triangles $T_{R_k} \setminus T$ are then rendered, i.e. all triangles (both visible and occluded) in R_k except those that we wish to unrender. The operation is completed by rendering the set of replacement triangles T', which results in the set of requested images $\mathcal{Y}'$.

6.3 A Perceptually Motivated Image Metric

The mean square image metric discussed above is a fair indicator of visual similarity for many geometric models. In particular, as we shall see in later chapters, this metric is far superior to relying on geometric deviation alone. Nevertheless, there are models, and images in general, for which the mean square error d_2^2 does not reflect well how differences between two images are perceived. One example of this is shown in Figure 6.15, for which the mean square error suggests a uniform error distribution over the image, while the perceived differences vary greatly between different regions of the image. The reason for this poor correlation is that the human visual system (HVS) does not process images as though they were collections of independent pairs of linear pixel intensities, which the d_2^2 metric does. Rather, the images that strike the retinas undergo some fairly complex processing before they are interpreted by higher level systems in the visual cortex [151]. These processes can lead to suppression of some image differences, and magnification of others. There are in fact many phenomena in the HVS that are rather well understood, and for which computational models have been developed that accurately predict when and how differences are perceived. Some of these phenomena, which will be explained below, include visual masking, variable contrast sensitivity for different spatial frequencies and luminance levels, low sensitivity to absolute spatial position, and edge detection at various orientations and spatial frequencies. Using such computational models, it is possible to more accurately predict how differences are perceived, which can be exploited in simplification in two ways. First, it allows *imperceptible simplification* to be performed, for which model differences that are below the visual threshold of detection cannot be perceived. Second, it provides a more accurate way of determining how to measure differences at suprathreshold levels. While imperceptible simplification is important in some applications, it generally allows only a modest reduction in complexity before differences are noticeable. To be truly effective, this approach requires view-dependent simplification, so that factors like distance, relative

orientation, and lighting can be accounted for. Because I am interested primarily in improving the off-line stage of simplification, I will concern myself mostly with near and above threshold differences, and allow more aggressive simplification to be performed. In this section, I propose a new image metric that is more accurate than the mean square error for certain types of models. This metric draws heavily upon previous work in vision, but has been designed to be very efficient and particularly suitable for comparing 3D models.

Before describing my perceptually motivated metric, it is important to consider what knowledge about the visual system can be put to use in our application. Put another way, what are some of the things that the mean square metric does not handle well? The following is a list of such issues that my metric was designed to address, which the mean square error does not handle:

- **Visual masking of faceting**. For textured models that are flat shaded, the texture can sometimes hide the presence of faceting (i.e. visible edges between two adjacent faces, where the surface should be smooth). Such visual masking may occur if the texture image contains high contrast patterns at spatial frequencies near the resolution of the mesh tessellation (see, for example, Figure 3.4).

- **Contrast sensitivity and edge detection**. At high simplification ratios, we can expect the simplified model to appear faceted (if flat shaded). However, some of the edges between facets are more visually distracting than others, depending on whether the model is otherwise smooth near the edge, and what the intensity levels are on either side of the discontinuity. The visual system generally does not measure absolute intensities, but adapts to the ambient light level and is sensitive to the context in which a feature appears (e.g. whether the feature appears on a white or a black background—see Figure 6.6). In addition, many components in the visual system are wired to detect edges in the image, and the HVS has mechanisms tuned to different edge lengths and orientations. Consequently, instead of computing differences in absolute intensity, like the mean square metric does, we should be more concerned with whether edges are present or not, and whether they correlate in the two images.

- **De-emphasis of small phase differences**. Because we are not particularly sensitive to the absolute position of a feature such as an edge, but rather to its location relative to other features, small differences between the silhouettes of two objects should not be penalized heavily if the two silhouettes otherwise have similar shapes. Similarly, a translational mismatch between high contrast texture content may go unnoticed if the texture parameterization is not greatly distorted. These observations lead Rushmeier et al. [129] to propose image metrics for which the phase information is factored out and discarded altogether.

In addition to these perceptual issues, the mean square metric, as defined above, operates on frame buffer intensities, which generally do not correspond directly to either the physical luminance output by the CRT or to its perceived brightness. My metric, on the other hand, models the CRT transfer function and accounts for ambient light reflected off the monitor.

In §3.3.2, I discussed previous work in vision and perceptual image metrics. One may ask if any of these metrics are suitable for our task. The answer to this is that many of them are applicable to off-line evaluations, but are too computationally intensive to be practical for driving simplification. Considered two of the most

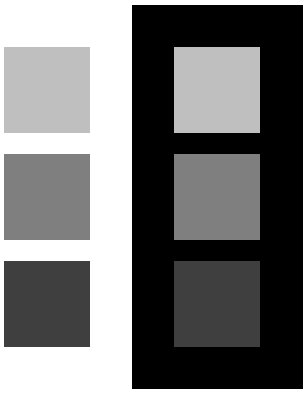


Figure 6.6: The *simultaneous contrast* effect. The squares on the black background appear brighter than the corresponding ones on the white background, even though the physical intensity levels are the same. This figure was adapted from [53].

accurate metrics, the *Sarnoff Model* [103] and the *Visible Difference Predictor* [32] are both very computationally demanding. The metrics by Rushmeier et al. [129] require computing the 2D Fourier transform of each image, which is likewise an expensive operation. The recent publications by Bolin and Meyer [14] and Ramasubramanian et al. [122] are attempts to drastically improve the computational complexity of previous perceptual metrics, in order to drive rendering applications such as ray-tracing and radiosity. In fact, I have implemented the metric by Bolin and Meyer, which essentially is a fast version of the Sarnoff Model, but found it to be of little use in simplification. Somewhat surprisingly, when used to drive simplification, their metric often leads to less pleasing results than the mean square error. Two potential reasons for this are that the use of the Haar wavelet transform, instead of the steerable filters used in the Sarnoff Model, to perform a multiscale image decomposition, leads to significant aliasing in all frequency bands [141]. This may cause undesirable leakage of frequency information, resulting in large image differences even for very small changes to the model, such as shifting a feature a single pixel. Bolin and Meyer also base their metric on the *contrast sensitivity function* (CSF), which is a function of spatial frequency, measured in cycles per degree of visual angle. Because it is very difficult to obtain an accurate estimate on the spatial frequency, which depends on the geometry of the viewer in relation to the displayed image as well as the object within the image—variables which are rarely ever fixed—and because of the aliasing of the wavelet transform, the CSF values cannot reliably be used in a metric for simplification. The metric by Ramasubramanian and co-workers is interesting in that part of it operates in the physical domain—as opposed to the perceptual domain—and can be used to compute a *threshold map* for an image. These thresholds provide a tolerance on the intensity of each pixel. As long as the pixel values in the other image being compared respect these tolerances, the two images are guaranteed to appear identical. While ideal for imperceptible simplification, it is not obvious how to extend the threshold map to dealing with the suprathreshold differences that dominate in our application.

Because the metrics mentioned above are not directly applicable to simplification, I have designed my own image metric. This metric was inspired almost entirely by the Sarnoff Model, but is more than a hundred

times as fast to evaluate. My metric also makes limited use of domain specific knowledge by assuming that the comparison is between two 3D models. Whereas my metric is largely based on published psychophysical data, there are some components of it that are not based on perception, and nothing has been done to calibrate the metric to agree with the observations of human subjects. Therefore, I do not suggest that my metric is in any sense more perceptually accurate than previous metrics. On the contrary, because of the many shortcuts taken and the lack of calibration, it is likely to be less accurate than many competing approaches. I do claim, however, that my metric correlates significantly better with perceived differences between images than the mean square error d_2^2 does, regardless of whether the images are of polygonal models or not. Meanwhile, my metric is nearly as computationally efficient as d_2^2.

In the following section, I will describe the different components of the new metric, and later conclude this chapter by comparing it to several other metrics, both on a set of static images and as an error metric for simplification.

6.3.1 Computational Model

My perceptual metric, which I will denote by d_p, can be described as a sequence of transformations made to the input images to arrive at a visual representation that I call a *cortex pyramid*, followed by a differencing step. The steps involved in this process are summarized in Figure 6.7, and will be explained one by one below. I will use the two images in Figures 6.15a and 6.15b as a running example for illustrating each step.

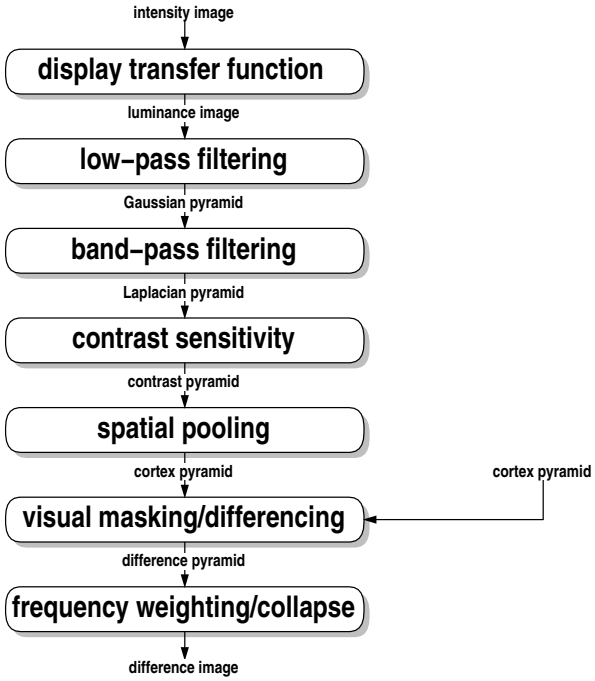


Figure 6.7: Diagram highlighting the major steps in the perceptual metric. Both intensity images undergo the same process, resulting in two *cortex pyramids*, between which the perceptual difference is computed.

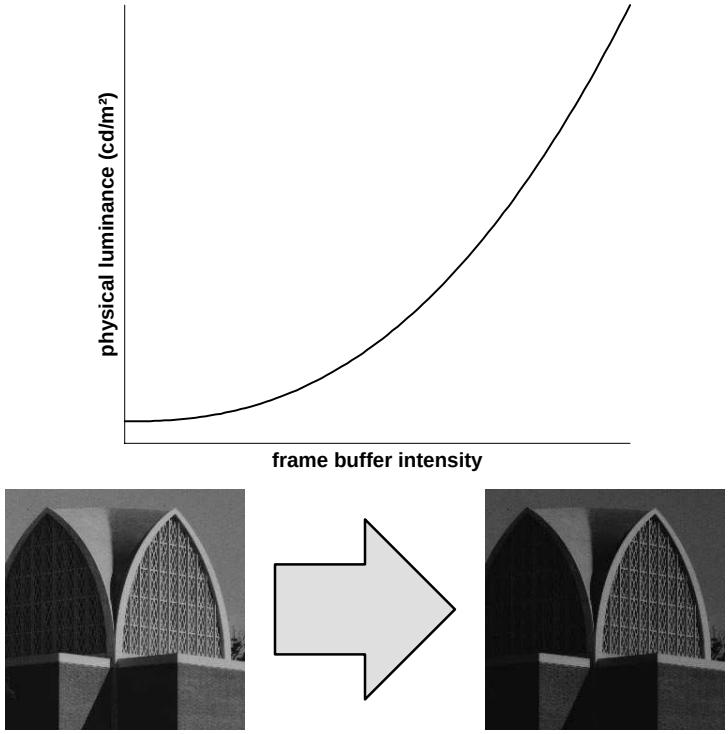


Figure 6.8: Top: Non-linear transfer function between linear frame buffer intensity and CRT output luminance. Ambient light and the black level setting of the CRT cause zero intensity to map to a positive display luminance. The function otherwise follows the exponential relationship $L \propto Y^{\gamma}$, where γ is the gamma value of the CRT (typically $\gamma \simeq 2.5$). Bottom: Luminance image (right) resulting from applying $\gamma = 2.4$ to the intensity image (left).

Display Transfer Function The first step in the metric is to convert frame buffer intensities, which are assumed to be linear, to physical luminance using the transfer function of the display device. This is needed because the published psychophysical data that I will make use of is measured in real physical units, such as candelas per square meter. The majority of CRTs have a non-linear transfer function that relates input intensity Y to output luminance L using a power law $L = L_{max}Y^{\gamma}$ [118] (Figure 6.8). The exponent, called the *gamma* of the CRT, is roughly 2.5 for conventional monitors. The constant L_{max}—the *white reference* or maximum display luminance of the CRT—converts normalized intensity in the range $[0, 1]$ to luminance. Typically, $L_{max} = 100$ cd/m^2. To compensate for the non-linearity of the display, many computers apply what is called *gamma correction* to the intensities before they reach the CRT, typically using a lookup table, by raising them to a power $1/\gamma'$. The gamma correction may not entirely cancel the display gamma, e.g. $\gamma' < \gamma$ in general, resulting in a reduced, but still measurable, non-linearity. As an example, SGI monitors have a gamma of $\gamma = 2.4$, and use a default gamma correction of $\gamma' = 1.7$, resulting in a transfer function of $L = L_{max}Y^{\gamma/\gamma'} = L_{max}Y^{2.4/1.7} \simeq L_{max}Y^{1.41}$.

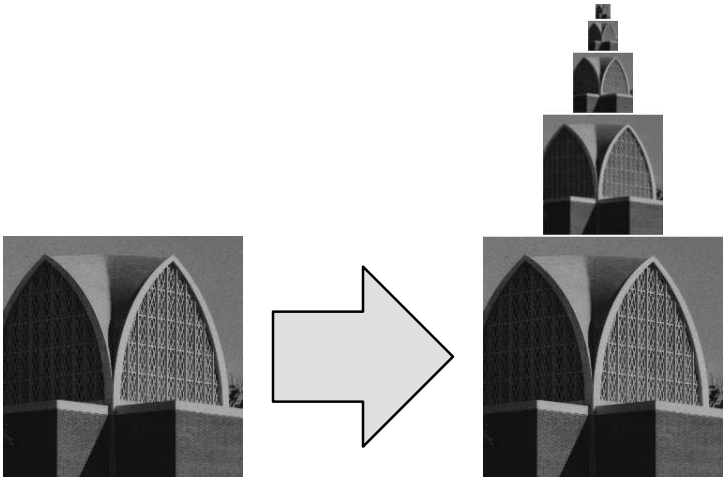


Figure 6.9: The five bottommost levels in the Gaussian pyramid.

Unfortunately, there are at least two other factors that influence the actual transfer function: the *black level* setting of the monitor, and ambient light reflected off the screen. The black level, often set by a "brightness" control, adds a bias Y_{min} to the display voltage, resulting in an effective transfer function $L = L_{max}(Y^{1/\gamma'} + Y_{min})^{\gamma}$. Taking the ambient luminance L_{min} into account, we have

$$L = L_{min} + L_{max}\left(Y^{1/\gamma'} + Y_{min}\right)^{\gamma} \tag{6.6}$$

Because we do not know what the black level is set to, we might as well assume that $Y_{min} = 0$, resulting in a final transfer function

$$L = L_{min} + L_{max}Y^{\gamma/\gamma'} \tag{6.7}$$

For the results presented below, I used the following settings:

$$L_{min} = 5 \text{ cd/m}^2 \quad L_{max} = 100 \text{ cd/m}^2 \quad \gamma = 2.4 \quad \gamma' = 1.7$$

Low-Pass Filtering As in most other perceptual metrics, I will use a multiscale decomposition of the image into a set of spatial frequency bands. There is evidence that the human visual system performs a similar transform [151]. Rather than using the alias prone Haar wavelet transform [14], or the computationally expensive steerable pyramid [46, 103], I have chosen to use the *Laplacian pyramid* from [17]. This decomposition has much better aliasing properties than Haar wavelets, and is essentially the orientation insensitive part of the transform used by Lubin. The term "pyramid" refers to the successive halving in image dimensions on consecutive levels (see Figure 6.9).

The first step in the Laplacian pyramid transform is the construction of a *Gaussian pyramid*. This can be thought of as a recursive low-pass filtering step, which creates frequency bands with successively lower cutoff frequencies. In conjunction with the Laplacian pyramid, the Gaussian pyramid will later be used to compute contrast values.

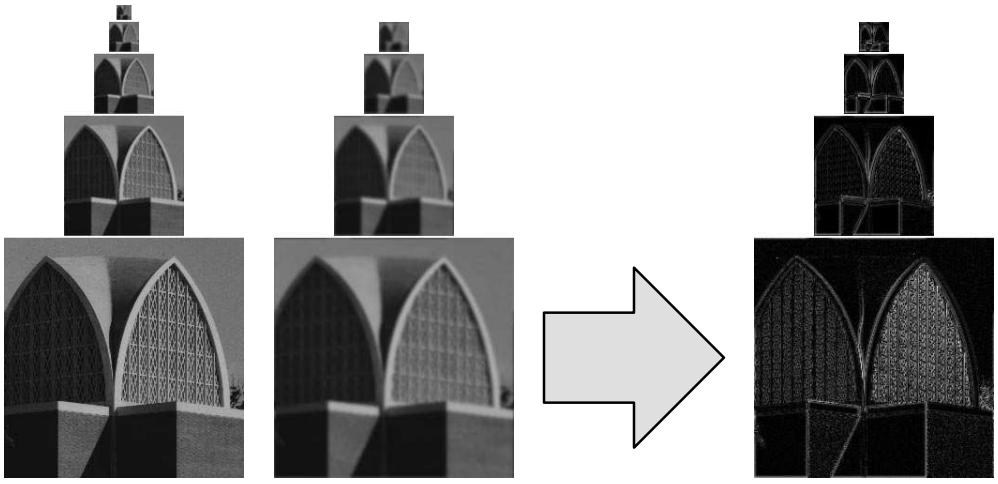


Figure 6.10: Laplacian pyramid (right), computed as the difference between each level in the Gaussian pyramid (left) and an upsampled version of the level immediately above (middle).

Band-Pass Filtering Applying the smoothing operator previously used to construct the Gaussian pyramid, an upsampling step is performed on each pyramid level. By subtracting these upsampled images from the levels immediately below, a Laplacian pyramid is obtained [17] (Figure 6.10). In essence, the Laplacian operator is an approximation to a difference of Gaussians (DOG), which performs multiscale edge detection. As a result of this construction, each level on the Laplacian pyramid corresponds to a given spatial frequency band, which can be processed independently.

Contrast Sensitivity Similar to the Sarnoff Model, a *contrast pyramid* is computed from the Gaussian and Laplacian pyramids. This is done slightly differently in my metric, however. The perceived contrast of a signal depends on the amplitude of the contrast pattern as well as the background luminance. Based on data from the psychophysics literature, it is possible to determine whether a test pattern that differs by ΔL in luminance over a uniform background at luminance L can be detected. Ward Larson et al. [85] give a mathematical expression that relates L and ΔL, which is graphed in Figure 6.11. Points along this curve correspond to contrasts that are just noticeable. The values in my contrast pyramid are normalized to this curve, by dividing the Laplacian values by the contrast thresholds associated with the local background luminance. The background luminance is estimated using the corresponding Gaussian pyramid values one level up.[5] That is, the contrast c_{ij}^k on level k is computed as

$$c_{ij}^k = \frac{|l_{ij}^k|}{f(g_{ij}^{k+1})} = \frac{|g_{ij}^k - g_{ij}^{k+1}|}{f(g_{ij}^{k+1})} \tag{6.8}$$

[5]This assumes that the eye can adapt to the average luminance in a neighborhood as small as three pixels wide on the lowest level. While somewhat unrealistic, this is a compromise that obviates having to search several levels up in the Gaussian pyramid when the level immediately above is directly available from the Laplacian pyramid computation.

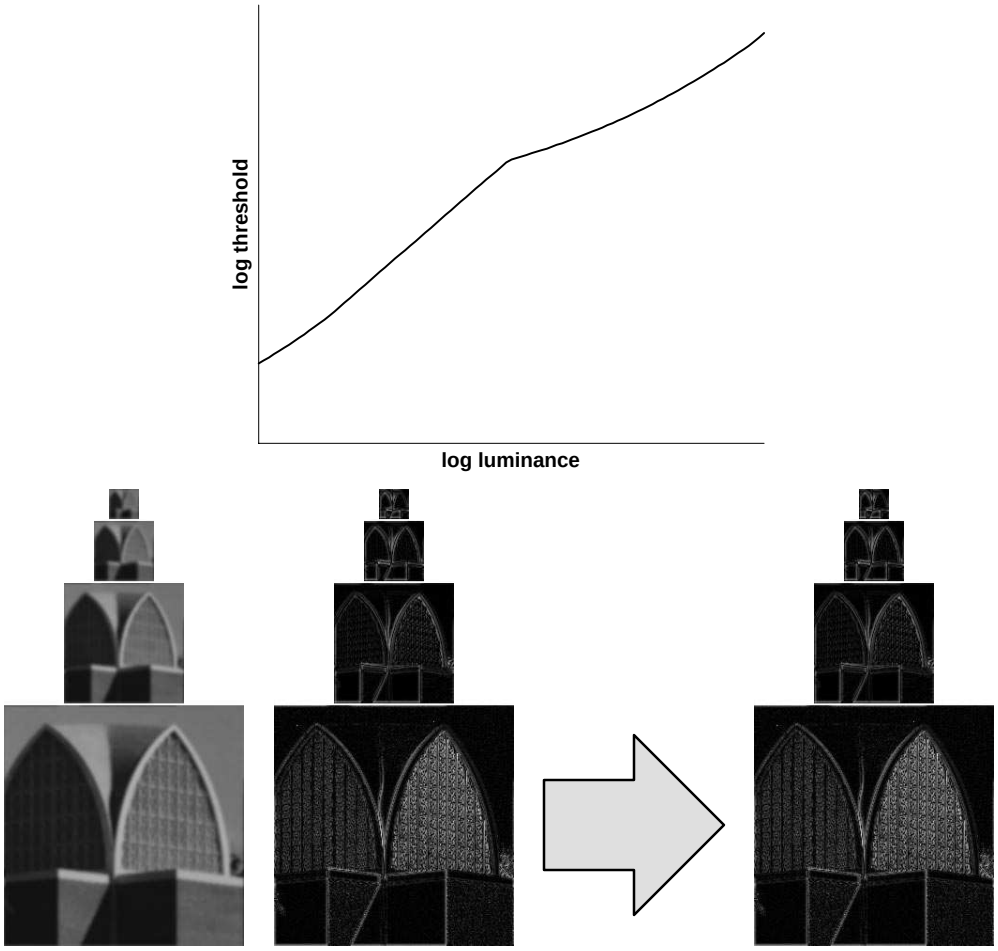


Figure 6.11: Top: Log-log graph of threshold vs. luminance. This graph shows the envelope of the responsiveness of cones (top right portion of the curve) and rods (bottom left portion) to contrasts for different levels of background luminance. As the background luminance increases, so does the threshold for detecting a contrast pattern. This function was adapted from [85]. Bottom: The contrast pyramid (right) is computed using the background luminance from the (upsampled) Gaussian pyramid (left) and the contrast from the Laplacian pyramid (middle).

where g_{ij}^k is the luminance value in the Gaussian pyramid for pixel ij on level k, l_{ij}^k is the corresponding Laplacian pyramid value, and f is the threshold versus luminance function from Figure 6.11.

Spatial Pooling In order to de-emphasize small phase differences, the contrast pyramid is blurred using the same 3×3 kernel used to construct the Gaussian pyramid (Figure 6.12). This step serves the same purpose as the spatial pooling steps employed in [14, 103], although using a smaller filter kernel.

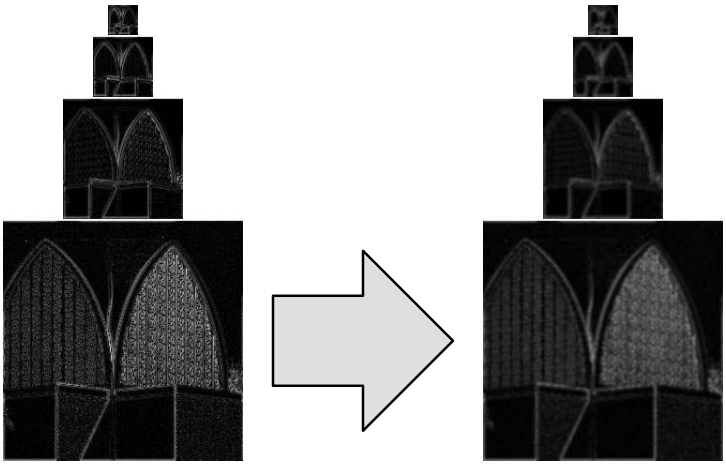


Figure 6.12: Spatial pooling. The contrast pyramid (left) is convolved with a Gaussian kernel, resulting in a cortex pyramid (right).

Visual Masking and Differencing When two contrast patterns are present, having similar spatial frequencies and orientation, one may *mask* the other (see Figure 3.4), depending on the relative contrast levels of the patterns. For a strong contrast pattern c and a weaker pattern Δc, $c + \Delta c$ cannot be distinguished from c alone, even though Δc is clearly visible when c is absent. In this case, we say that c masks Δc. Lubin [103] and Bolin and Meyer [14] account for this behavior by passing each of the two contrast signals through a sigmoid *masking transducer*:

$$T(c) = \frac{2c^{\alpha}}{1 + c^{\beta}}$$

In [14], $\alpha = 2.25$ and $\beta = 2.05$.[6] This S-shaped transfer function compresses the contrast response at high levels, bringing c and $c + \Delta c$ closer when c is large, while contrasts below the detection threshold are driven towards zero, essentially eliciting a binary response to contrast.

I, too, account for visual masking, but integrate this step directly in the differencing function. The difference d between contrast signals c and c' is measured as

$$d(c, c') = \frac{(c - c')^2}{1 + (c + c')^2} \tag{6.9}$$

Clearly, when $c \simeq c'$, the difference is small. Also, when $c, c' \gg 1$ (i.e. far above the contrast threshold of detection), then d approaches $\left(\frac{c-c'}{c+c'}\right)^2$, which leads to a suppressed difference. When $c, c' \ll 1$, d approaches $(c - c')^2$; also a small number. However, when c and c' straddle the threshold, then the difference becomes relatively larger. That is, if an edge is clearly present in one image, but not in the other, then the difference is large. Otherwise, the strength of the contrast patterns is of less importance, which is consistent with the visual

[6] The exponents in [14, 103] are halved, canceling out the squaring of the contrast in previous steps. The contrast c here is on the other hand unsquared.

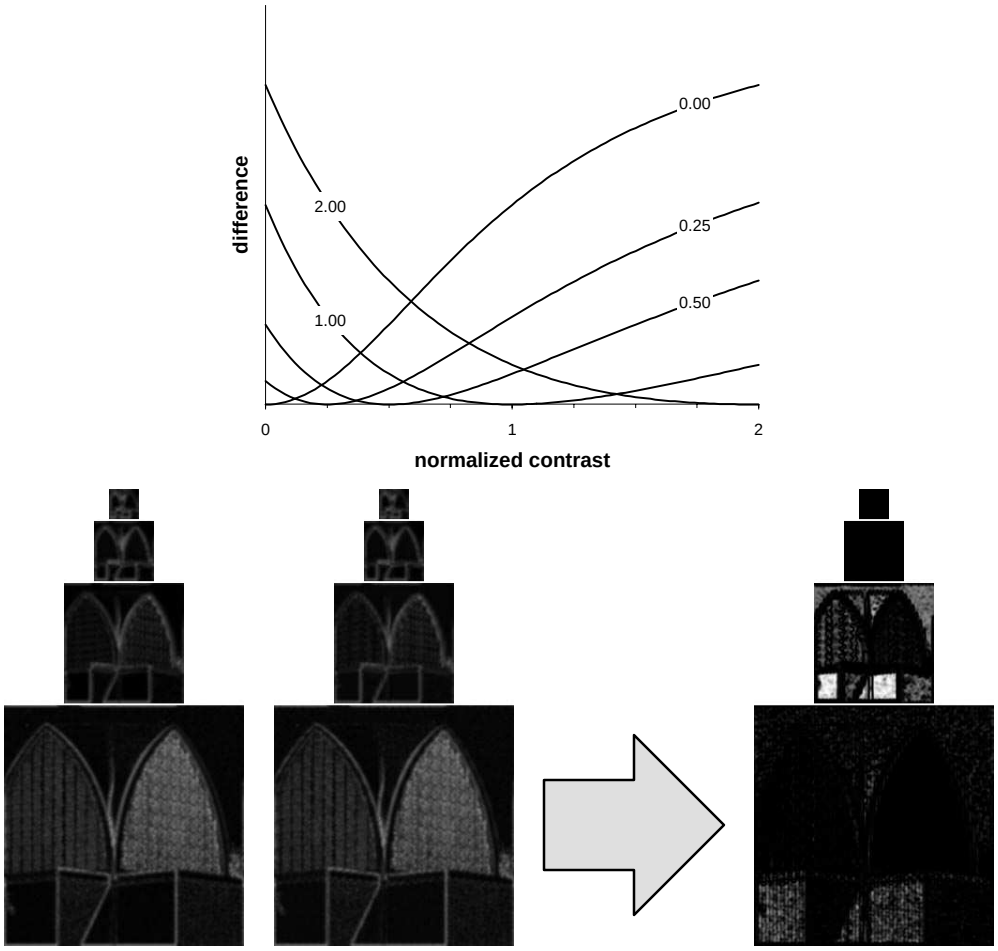


Figure 6.13: Top: Slices of bivariate difference function. The horizontal axis corresponds to the normalized contrast level, measured in units of just noticeable contrast, for one of the two input signals. Five graphs are shown corresponding to different contrast values for the other input signal. The vertical axis measures the perceived difference between these two signals. Bottom: The two contrast pyramids (left) and the difference pyramid (right).

masking phenomenon. $d(c, c')$ is graphed for a continuous c and a few discrete levels of c' in Figure 6.13. Notice the sigmoid shape of d for $c' = 0$.

Frequency Weighting and Pyramid Collapse Once the difference pyramid has been constructed, it needs to be turned into a flat difference image. This is done by first weighting each pyramid level, and then collapsing the pyramid. For comparing models, we are interested mainly in large-scale differences. Being off in position by a pixel or two should not matter much. Therefore, I have chosen to weight the levels non-uniformly, using a geometric sequence of weights $\{w^k\}$. Specifically, $w^k = \sqrt{2}\, w^{k-1}$ for all levels k, and

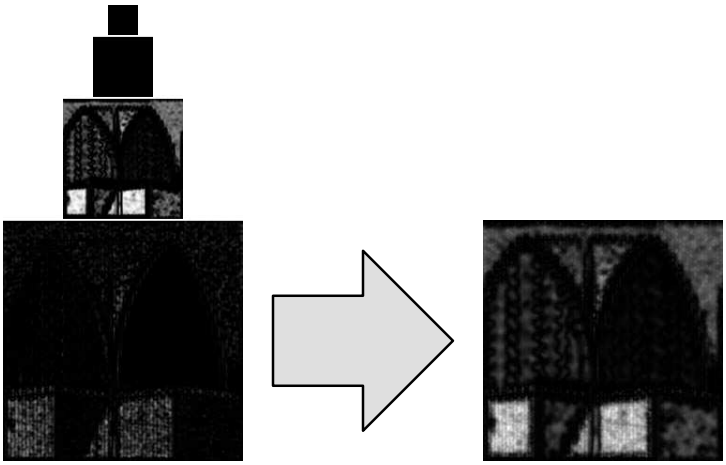


Figure 6.14: Frequency weighting. The difference pyramid (left) is collapsed to a difference image (right) using non-uniform weighting of the pyramid levels.

$\sum_k w^k = 1$. Thus the top levels are weighted more heavily than the bottom ones. This set of weights was selected after trying different weighting schemes on a number of test images. After frequency weighting, the pyramid is collapsed recursively from top to bottom. For each level, the Gaussian upsampling operation is applied, and the result is added to the level below. These combined layers are in turn upsampled further, and the recursive summation eventually results in a difference image (Figure 6.14). By summing the pixel values in this difference image, a numerical difference is obtained, which represents the perceptual difference $d_p(Y, Y')$ between the two input images.

6.3.2 Efficient Implementation of Metric

In order to use my perceptual metric for simplification and optimization, we must ensure that it can be evaluated efficiently. Compared to the mean square metric, for which incremental evaluation is straightforward, my own metric is considerably more elaborate, and consists of more than a handful of non-trivial steps. One can engineer a fast solution by examining what parts of the evaluation are affected by making a change to a rectangle of pixels in one of the images (the type of image update performed in §6.2.2). Even though my metric uses a hierarchical image decomposition and convolution within each level, the filter kernels are very small and consequently leave most of the pyramid pixels intact, except within and immediately surrounding the affected region. Thus, by maintaining copies of the intermediate pyramids (i.e. Gaussian, Laplacian, contrast, etc.) for the previous image, and only building up partial pyramids from the rectangular area of new pixels, one can select pixels from the previous or the partially constructed pyramid based on simple range checks. Because we do not need a difference image for simplification, but instead only a numerical measure of similarity, the difference computation can be done pixel-wise on the difference pyramid without collapsing it. Again, because the error metric defined in §6.2 measures the change between consecutive images, most differences cancel between the difference pyramids, allowing only portions of them to be traversed.

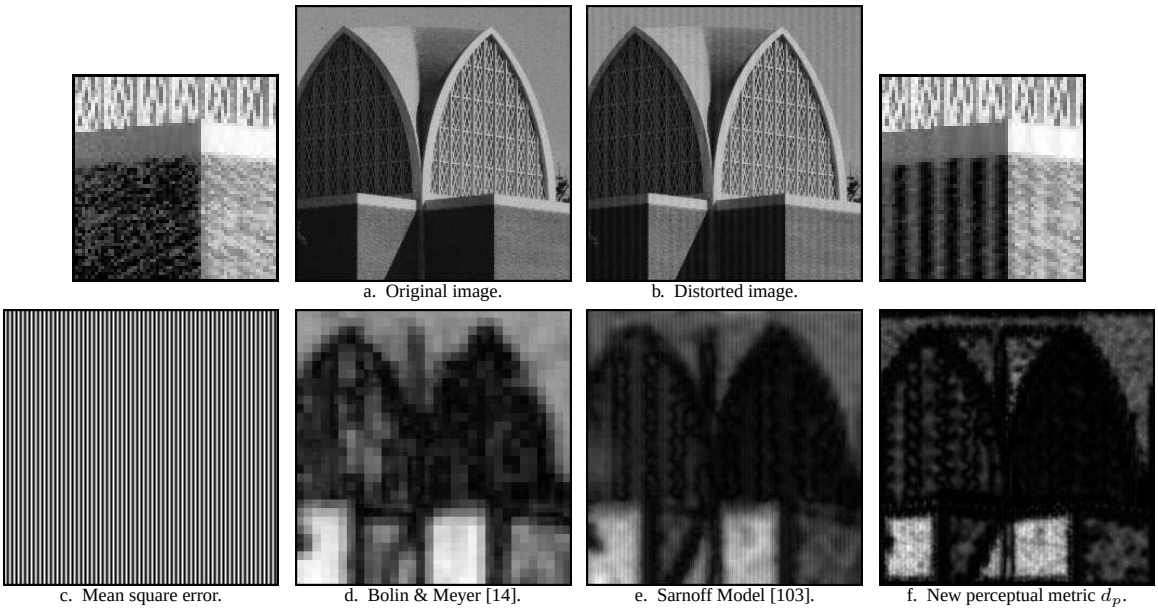
Another savings in time and memory can be obtained by using 16-bit integer arithmetic instead of floating point calculations. Assuming the input images are 8 bits deep, we can make use of 16-bit lookup tables for the display transfer function and the contrast sensitivity function. It is possible to also encode the 2D masking and difference function using a $2^8 \times 2^8$-entry lookup table by first quantizing the 16-bit entries (non-linearly) in the cortex pyramids. I have chosen instead to use floating point for this final step in the difference computation, as it is nearly as fast as the two-way indirection needed for lookup and yields more accurate results.

6.3.3 Results

Figure 6.15 shows how my metric, the mean square error, and the metrics in [14, 103] perform on a pair of test images. A sinusoid grating pattern of constant amplitude was superimposed over the image in Figure 6.15a, resulting in Figure 6.15b. This contrast pattern is recovered by the mean square metric, as seen in Figure 6.15c. Because of variable contrast sensitivity and visual masking, the sinusoid pattern is much less visible in the brighter regions and in the ornate windows and other high contrast regions. This is well reflected in the difference images produced by the three perceptual metrics. Whereas the difference image produced by Bolin and Meyer's metric is aliased and poorly resolved, my metric picks up higher resolution differences and exhibits less aliasing. There are, however, two noticeable differences between my metric and the more accurate Sarnoff Model. First, near the edges of the image, the difference drops to close to zero in my metric, resulting in a dark frame around the image. This is a direct result of the manner in which the multiple convolution steps are performed. Whereas my metric assumes that the image is a periodic signal, and therefore wraps the convolution kernel over to the opposite edge, the implementations of the metrics in [14, 103] handle this boundary case differently by reflecting the kernels. By wrapping the images, my metric picks up the sharp edges induced at the boundaries, which mask the faint contrast pattern. Second, there is a dark horizontal band near the top of the difference image in Figure 6.15f. This is because my metric does not perform orientation dependent processing, and therefore mistakingly assumes that the horizontal edge formed along the top of the structure masks the vertical test pattern. Recall that visual masking occurs mostly when the patterns have similar orientation. As we shall see, this limitation is seldom a liability in our target application of comparing 3D models.

To see how my metric balances silhouette differences, varying contrast levels, and visual masking, I created a set of textured cylinders, similar to the ones used by Ferwerda et al. [44]. These are shown in Figure 6.16. From the difference images, it is clear that the mean square error is overly sensitive to small phase differences, such as a mismatch in silhouettes or texture. The only appreciable difference between the top row and the one below it is the faceting of the cylinders in the second row. My metric correctly predicts this and the visual masking occurring at high contrasts and mid-range spatial frequencies. Again, it does not handle the transverse texture orientation correctly, although the other two perceptual metrics did not appear to do significantly better in this case. In summary, these images show a significant increase in correlation with our visual expectations by using my metric instead of the mean square error.

The final example illustrates how my metric can successfully be put to use in simplification (see Chapter 7 for a description of the simplification algorithm). For this example, a textured sphere with a partially smooth



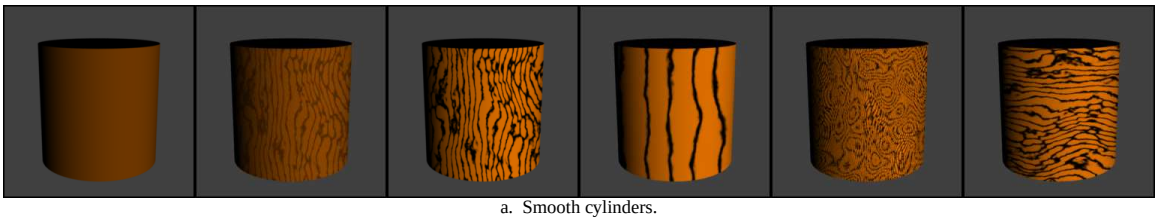
a. Original image. b. Distorted image.

c. Mean square error. d. Bolin & Meyer [14]. e. Sarnoff Model [103]. f. New perceptual metric d_p.

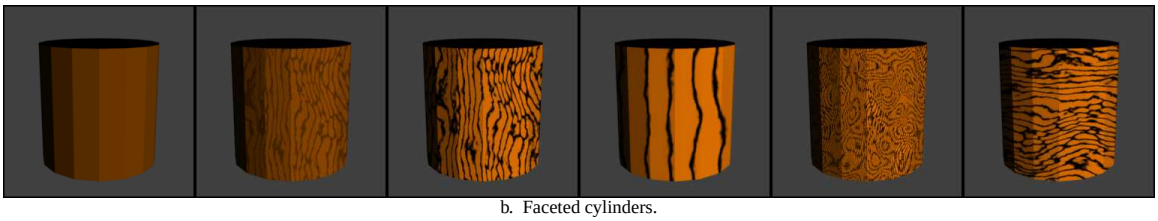
Figure 6.15: Comparison of image metrics. The distorted image was created by superimposing a sinusoid grating over the original image. This pattern is more apparent in the enhanced close-up on the right. The difference images in the bottom row were produced by applying various image metrics to the images in 6.15a and 6.15b. Here black signifies no difference, while white corresponds to maximum difference. The source images were taken from [14].

surface and a very noisy opposite half was used (Figure 6.17b). This model was simplified both using d_2^2 and my perceptual metric d_p. Because the noisy part of the texture masks most of the faceting, more triangles should be spent preserving the smooth gradients on the left side. However, the mean square metric, which penalizes even minute shifts in the high contrast part of the texture, was forced to spend most of the triangles on the right side of the sphere. My metric, on the other hand, correctly places most vertices on the left side, creating the illusion of a smoother sphere. In the difference images shown (which were produced by d_p), it is also evident how some of the sharper edges and vertices on the left side of the sphere were considered more distracting by my metric, which agrees well with our subjective impressions.

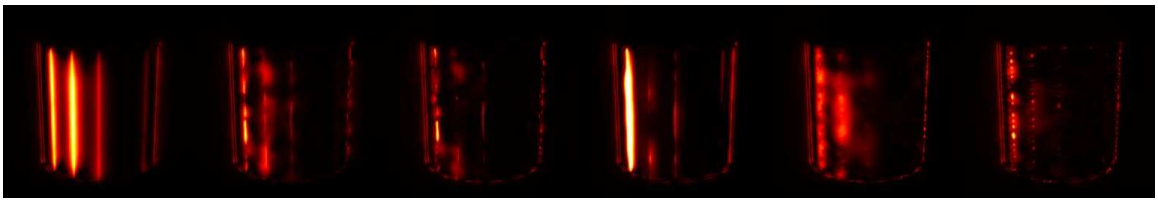
I have so far given examples of how my perceptual metric can produce more perceptually accurate results than the mean square metric by better accounting for visual masking, relative contrast, and small phase differences. In later chapters, I will use this metric to highlight differences between simplified models. These difference images provide more results and opportunities for evaluating the quality of the metric. Before concluding this chapter, let me comment on the computational efficiency of some of the metrics used here. Disregarding the initial overhead of precomputing lookup tables, my metric takes roughly 120 milliseconds to compare all pixels between two 256×256 pixel images, which is about six times slower than the mean square metric. Using incremental updates, more overhead is involved in my metric for the range checks needed to multiplex between the pyramids, resulting in a ten to one speed ratio between the two metrics. Compared



a. Smooth cylinders.

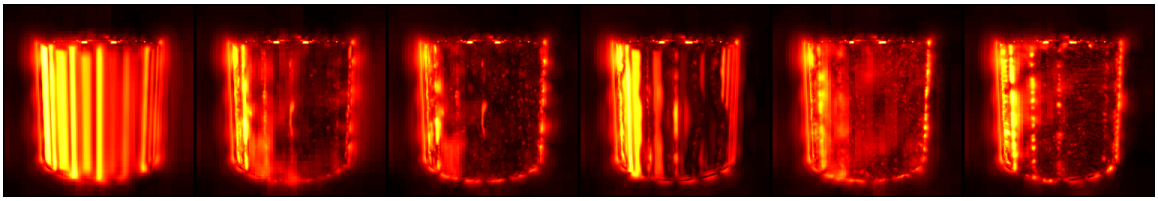


b. Faceted cylinders.



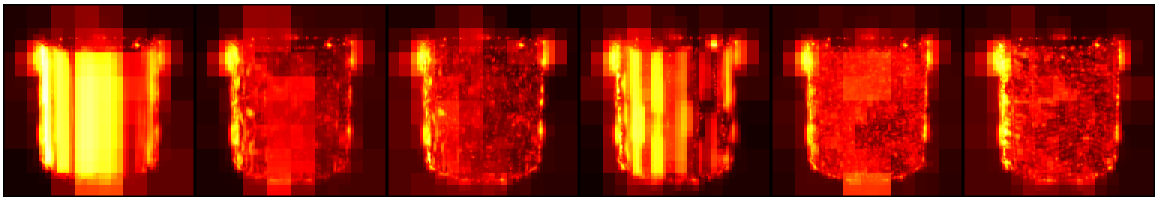
1.00. 0.45. 0.30. 0.69. 0.59. 0.34.

c. New perceptual metric d_p.



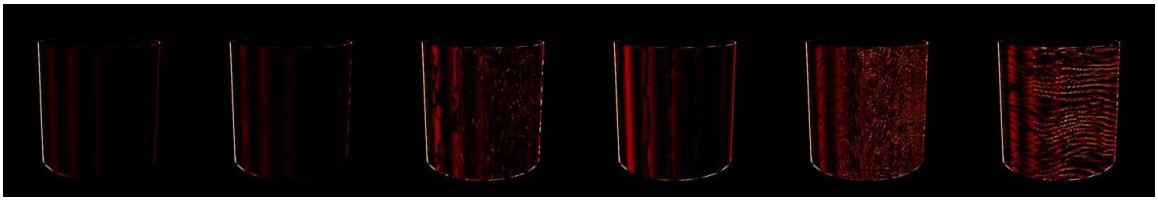
1.00. 0.56. 0.44. 0.58. 0.55. 0.53.

d. Sarnoff Model [103].



1.00. 0.47. 0.46. 0.67. 0.53. 0.48.

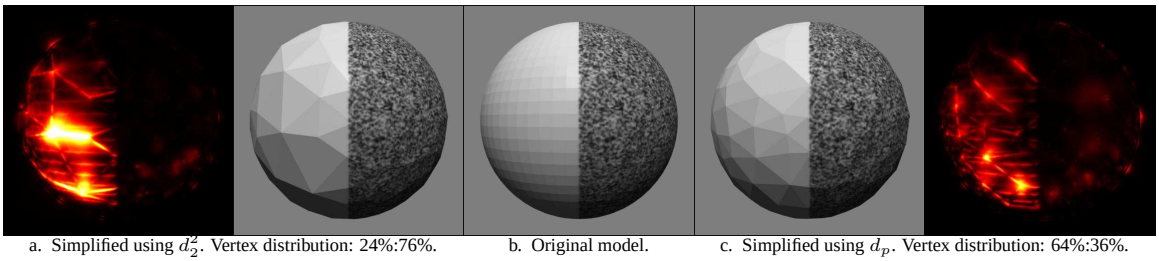
e. Bolin & Meyer [14].



0.24. 0.27. 0.57. 0.50. 0.94. 1.00.

f. Mean square error.

Figure 6.16: Example showing how the contrast, frequency, and orientation of a pattern affect visual masking. The faceting in 6.16b is less visible when the texture pattern has high contrast and its frequency and orientation nearly match the tessellation. Difference values are included for each metric and image pair.



a. Simplified using d_2^2. Vertex distribution: 24%:76%. b. Original model. c. Simplified using d_p. Vertex distribution: 64%:36%.

Figure 6.17: Textured sphere simplified using the mean square error and the perceptual metric. The noisy texture masks the faceting on the right side of the sphere, which the perceptual metric correctly accounts for by allocating nearly two thirds of the vertices to the left side. Using d_2^2, most vertices are devoted to the right side in order to match the noisy texture pixel by pixel.

to an implementation of the Sarnoff Model, which admittedly has not been optimized fully, my perceptual metric is nearly four hundred times as fast as the Sarnoff metric, which takes three quarters of a minute to evaluate on the same set of images. Meanwhile, I have not found the Sarnoff Model to be considerably more accurate for comparing 3D models, which suggests that the perceptually motivated metric described in this section is a reasonable substitute for such evaluations, while also being a practical alternative to the mean square error for driving simplification and optimization.

Chapter 7

IMAGE-DRIVEN SIMPLIFICATION

7.1 Introduction

Equipped with the tools developed in Chapter 6 for using image metrics to evaluate the visual quality of a model, I will now describe an *image-driven simplification* (IDS) method that uses visual similarity as the main criterion for decimating a mesh. This method was first published in [97]. I have already given plenty of motivation in previous chapters (see Chapters 1, 3, and 6) for why such a method is important in graphics, so I will refrain from repeating myself here. Instead, I will briefly explain the main approach of the algorithm in this section.

Similar to my memoryless method (Chapter 4), IDS uses edge collapse to coarsen a model. In fact, the two methods are identical with one exception—the choice of error metric used to order the sequence of edge collapses. Whereas the memoryless method uses a geometry-based metric, my image-driven method relies on rendered images and an image metric to measure the cost of collapsing an edge. This metric is based on the framework developed in §6.2. Using this image metric, edges are ordered such that the one that yields the smallest *visual difference* is the next edge collapsed.

In addition to ordering edge collapses, most error metrics implicitly provide for each edge collapse the optimal position of the replacement vertex, i.e. the position at which the error is minimized. Contrary to the memoryless method, for which a closed form expression exists for the optimum, it is virtually impossible to directly find the new vertex position that minimizes the image difference. In addition to the intricacies of the image metric itself, there are simply too many factors and complex interactions between them that influence the pixel values, including lighting, shading, surface properties, visibility, and so on. Therefore, the simple heuristic introduced in Chapter 4 is used to position vertices in IDS. This heuristic has proven successful both in conjunction with the image-based edge cost and in the original geometry-based scheme.

After describing the simplification algorithm in more detail, I will in §7.3 present numerical results and images of simplified models to show how the image-driven method does not only produce high quality simplifications for simple organic models like the Stanford bunny, but also performs better than most geometry-based methods by taking into account what parts of a model are visible, the type of shading model used, the content and parameterization of one or more textures, and whether artifacts such as cracks and interpenetrations are visible. In particular, I will show that IDS consistently produces high quality results for models of very low complexity, for which traditional simplification methods often have difficulties.

7.2 Simplification Algorithm

As in any simplification method based on edge collapse, my image-driven method requires two fundamental decisions to be made: where to place the new vertex following an edge collapse, and choosing the order in which the edges are collapsed. These two components will be described in the following two sections. In addition to the acceleration techniques described in §6.2 for comparing images, I will make use of *lazy evaluation* of edge costs and fast pre-simplification to further improve the overall speed of the algorithm. These features will be described in §7.2.2.1 and §7.2.3. Because the image-driven method is identical in all other respects to my memoryless simplification method, I refer the reader to Chapter 4 for a more detailed explanation of the general edge collapse scheme. I will conclude this section with a discussion of different image metrics that I have used in my simplification method.

7.2.1 Vertex Placement

As pointed out in Chapter 4, vertex placement is implicitly an optimization problem; given an edge $\bar{e}$ to collapse, the goal is to choose the position $\mathbf{x}^{\bar{v}}$ of the new vertex $\bar{v}$ such that the associated error functional is minimized. In IDS, the error functional is expressed in terms of rendered images of an object, and cannot as such be written as a simple function of $\mathbf{x}^{\bar{v}}$. Several global optimization methods exist for solving such problems, and we will see in Chapter 8 how such an approach can be used to produce a near optimal mesh from an already simplified model. Instead of employing an expensive optimization procedure for each edge collapse, however, I have opted to use the fast memoryless method (Chapter 4) for placing new vertices. While any vertex placement method may be used in conjunction with IDS, I use the memoryless method because it makes reasonable placement choices and is fast and memory efficient. In addition, it is easy to extend the memoryless method to handle surface attributes (§4.7) without having to maintain any information about the original model. As we shall see in §7.3, another benefit of using the memoryless vertex placement method is that it also leads to models of high geometric quality.

7.2.2 Edge Collapse Priorities

The most novel aspect of this simplification algorithm is the use of images to measure visual similarity. In Chapter 6, I developed a framework for making such comparisons, based on collections of images surrounding the model (Figure 6.1). Using images $\widehat{\mathcal{Y}}$, $\mathcal{Y}$, and $\mathcal{Y}'_e$ of the original model, the partially simplified model, and the model after an additional edge e is collapsed, respectively, I earlier defined an error metric $\Delta d(\mathcal{Y}, \mathcal{Y}'_e, \widehat{\mathcal{Y}})$ (Equation 6.3). This metric determines the cost $C(e)$ of collapsing an edge e. In each iteration of the algorithm, the goal is to find the edge $\bar{e}$ that minimizes the loss in visual quality, and then collapse it, i.e.

$$\bar{e} = \operatorname*{argmin}_e C(e) = \operatorname*{argmin}_e \Delta d(\mathcal{Y}, \mathcal{Y}'_e, \widehat{\mathcal{Y}})$$

To produce $\mathcal{Y}'_e$, the triangles $T = \lceil\lceil\lfloor e\rfloor\rceil\rceil$ are unrendered and replaced with $T' = \lceil\lceil v\rceil\rceil$, where v is the substitute vertex, using the procedure in §6.2.2. For a mesh with an average vertex valence of six, we can

expect $|T| = 10$ and $|T'| = 8$. Because of the small number of triangles involved in this process, relatively little work is needed to update the images, as compared to rerendering the entire mesh from scratch. Once the images $\mathcal{Y}'_e$ have been rendered, the metric Δd is evaluated incrementally using the method described in §6.2.1.

7.2.2.1 Evaluation of Edge Cost

In theory, the edge cost $C(e)$ would have to be evaluated in each iteration for the entire set of remaining edges. Contrary to most geometry-based edge collapse methods, for which the edge cost does not change over time, except for a small set of edges near the previously collapsed edge, an edge collapse in IDS could in principle affect the cost of any remaining edge. Even geometrically and topologically distant edges may be arbitrarily near in some views. However, since only a fraction of pixels generally change between iterations, collapsing an edge typically does not alter the edge costs for most of the other edges. While the number of affected edge costs depends on the locality of the image metric, it is generally very small for most metrics. For a pixel-wise metric like d_2^2, we can say with certainty that Δd_2^2 does not change for an edge e between iterations as long as all changes to the images take place outside e's associated bounding box in each view (see Equation 6.5), which eliminates the need to constantly update each edge cost. Therefore, instead of attempting to identify all affected edges, it often suffices to update a small set of edges around a collapsed edge. By default, I use the same set of edges $\lceil \lfloor \lceil \bar{v} \rceil \rfloor \rceil$ as in my memoryless method. These are generally the only edges whose cost terms change in more than one view. To validate this assumption, I recorded the actual rank of the collapsed edges among all valid candidates over a large number of iterations. For the Stanford bunny model, the cost of the collapsed edge was on average in the lowest 0.05% using d_2^2, versus 0.14% for my perceptual metric.

As in most edge collapse algorithms, collapsing an edge generally leads to an increase in the cost of collapsing any of the nearby affected edges. This suggests that the algorithm is amenable to *lazy evaluation*, that is, instead of updating the cost of an affected edge, the edge is marked as "dirty" and the evaluation is deferred until the edge reaches the front of the queue [26]. It is also possible to mix direct and lazy evaluation, e.g. by updating a smaller set of edges, say $\lceil \bar{v} \rceil$, and marking a larger surrounding set of edges, say $\lceil \lfloor \lceil \bar{v} \rceil \rfloor \rceil \setminus \lceil \bar{v} \rceil$, as dirty. When a dirty edge is removed from the queue, its edge cost is re-evaluated, and the edge is re-inserted into the queue. Without lazy evaluation, each edge generally has its cost updated several times before it reaches the front of the queue. By employing lazy evaluation, many of these extraneous evaluations are eliminated. Lazy evaluation generally reduces the number of edge cost evaluations by a factor of five (depending on the degree of simplification), without significantly compromising model quality. For the same test case as above using d_2^2, lazy evaluation degraded the average percentile of the collapsed edges from 0.05% to 5%. This lazy evaluation scheme, for which the edges $\lceil \lfloor \lceil \bar{v} \rceil \rfloor \rceil$ were marked as dirty, was used in my image-driven algorithm for all results presented below, except for the frog model which used mixed evaluation.

7.2.3 A Hybrid Method

I mentioned in §1.1 that most geometry-based simplification methods perform well up until a certain point. After all the redundant geometry and fine details have been removed, the model quality tends to decrease exponentially. Until that point is reached, however, geometry-based simplification can often produce models of high visual quality. In fact, the quality difference between geometry- and image-driven simplification is not very noticeable until the final stages of simplification, when only a small percentage of triangles remain. Meanwhile, my geometry-based memoryless algorithm is roughly fifty times faster than the image-driven method. Therefore, one can save much time by using a fast geometry-based method to pre-simplify the model, and then switch over to image-driven simplification in the final stage to "fine-tune" the model. In addition to speeding up the effective edge collapse rate, this strategy also cuts the overhead of evaluating the image metric for every single edge in the model at startup, since the image-driven method would be started with a much coarser model. This hybrid approach to simplification was used exclusively to produce the results below, for which memoryless simplification was used until 4–8 times the target complexity was reached.

Somewhat surprising, the hybrid method often performed slightly better than image-driven simplification alone. A possible reason for this is that edges in the original model are so short that collapsing them has no visual impact unless the image resolution is sufficiently high. Therefore, the image-driven method initially produces a near random, suboptimal sequence of edge collapses that has long-term adverse consequences.

7.2.4 Image Metrics

Perhaps the most important component of IDS is the choice of image metric. Ideally, the metric should be a good indicator of visual similarity, but should also be efficient to evaluate. Not surprising, these two goals are generally in conflict with each other, so a compromise is needed. Because the image-driven method is rather slow to begin with, we are somewhat limited in our choice of image metric. The complex Sarnoff Model [103], for example, may take several seconds per image pair to evaluate, and so is not a viable alternative. This issue of efficiency is of course not limited to simplification, and recent work has been done to improve the speed of these complex metrics for other graphics applications, e.g. [14, 122]. Even so, most of these metrics are not easy to implement efficiently, and the need for fast incremental evaluation following an image update adds complexity to the implementation. Therefore, I have so far considered only a few different metrics.

During my work on image-driven simplification, I have incorporated a handful of different image metrics with my simplification method, including the mean square error d_2^2, my own perceptual metric from §6.3, and an implementation of Bolin and Meyer's metric [14]. These metrics have been used to simplify a wide variety of models, with and without surface properties. I have found that Bolin and Meyer's metric, while based on rigorous perceptual models of human vision, rarely performs better qualitatively than the mean square error metric. On the contrary, it often produces less appealing simplified models than d_2^2. This conclusion is, of course, based on my own subjective preferences. However, even those models simplified using [14] generally

yield lower quality results as determined by an off-line evaluation using the very same metric. More research is needed to identify the reason for this surprising result.

My own perceptually motivated metric, on the other hand, typically results in models at least as good as those produced by d_2^2, and sometimes noticeably outperforms d_2^2 on textured models. Note that perceptual metrics are most useful for exploiting differences that are near the threshold of detection. While there are also important perceptual factors at play at suprathreshold levels, these are significantly more difficult to account for in a computational model. Factors such as illusions, symmetry, and context sensitive semantics, such as the importance of the face of a model compared to its feet, are associated with higher level vision processing. Short of emulating the brain neuron by neuron, it is exceedingly difficult to model such high level behavior.

One might ask why a simple image metric like d_2^2 performs as well as it does for simplification—it is quite easy to construct pathological cases for which such pixel-wise metrics fail (see, for example, [144]), and many authors make it a point to illustrate how inadequate d_2^2 is for estimating visual similarity. Many of these shortcomings are, however, due to situations that in all but a few cases do not arise in simplification. Examples include image distortion (e.g. noise, superimposed intensity patterns, compression artifacts, etc.) and affine transformations (e.g. translations, rotations, and scaling of the image contents). Because the two objects are registered in 3D, so are the resulting images, and since the images are synthetic, there is no potential for noise or similar artifacts.

As a potential problem, one might suspect that the mean square metric d_2^2 would overemphasize image differences due to minor shifts in the texture parameterization of a model, particularly if the texture is rich in high frequency content. However, as shown in §7.3, using the this metric to guide simplification leads to results that are substantial improvements over those obtained without an image metric. The mean square error also has the virtue of being fast to compute, which is important for guiding simplification. Finally, as mentioned before, my image-driven simplification framework easily allows other image metrics to be used, should d_2^2 prove inadequate.

7.3 Results

In this section I present several models that have been simplified using my image-driven technique. All of these models were simplified on a one-processor 250 MHz R10000 Silicon Graphics Octane workstation with 256 MB of RAM and Maximum Impact graphics. As mentioned in §7.2.3, the models were pre-simplified using the memoryless method from Chapter 4, and the image-based cost measure was used for the remainder of the simplification. Fast image updates were achieved using the graphics hardware-based approach described in §6.2.2.2. Unless otherwise stated, the mean square error metric d_2^2 was used in the image-driven simplification method.

Table 7.1 contains the number of triangles eliminated and the time spent in this pre-simplification step. This table also provides data for the image-driven step, as well as total simplification time and effective

model	*options*	*triangles*			*simplification time (h:m:s)*			*tri/s*
		original	*intermediate*	*final*	*MS*	*IDS*	*total*	
bunny		69,451	30,254	2,899	0:47	28:56	29:43	37
			14,535	1,333	0:57	11:18	12:15	93
			6,444	654	1:01	4:38	5:39	203
			2,918	304	1:04	2:20	3:24	339
			1,336	143	1:05	1:28	2:33	453
turbine blade	20/20	1,765,388	32,767	8,191	26:45	3:28:10	3:54:55	125
	10/20		32,767	8,191	26:45	1:06:49	1:33:34	313
	20/20		8,191	2,046	27:06	23:25	50:31	582
dragon	flat	871,306	32,768	8,191	12:15	38:24	50:39	284
	smooth		32,768	8,191	12:15	1:14:12	1:26:27	166
buddha	flat	1,087,716	65,536	16,384	15:19	1:57:49	2:13:08	134
	smooth		65,536	16,384	15:19	2:13:27	2:28:46	120
salamander	orange	71,312	16,384	2,047	1:21	15:52	17:13	67
	green		16,384	2,047	1:21	16:09	17:30	66
frog		47,028	4,096	1,024	0:45	3:17:01	3:17:46	4
zebra		58,503	4,095	511	1:14	2:53	4:07	235
			2,047	384	1:16	1:24	2:40	363
			2,047	256	1:16	1:29	2:45	353

Table 7.1: Simplification results for the image-driven method. For the turbine blade model, the "options" column indicates the ratio of the number of exterior views to the total number of views used during simplification. For the dragon and buddha models, the options indicate the choice of shading interpolation, while the choice of texture used during simplification is given for the salamander model. The "intermediate" column indicates the number of triangles after pre-simplification. The simplification time is given in number of wall clock hours, minutes, and seconds, and is shown separately for the memoryless pre-simplification (MS) and the final image-driven phase (IDS). The last column gives the effective number of triangles eliminated per second, i.e. the difference in number of triangles in the original and final model over the total simplification time.

triangle reduction rates. While the image-driven algorithm is slow in comparison with the fastest geometry-based simplification methods currently available, it still produces acceptable reduction rates[1] (roughly in the range of 100 to 300 triangles per second) when pre-simplification is used. Since the edge collapse algorithm can produce a progressive mesh [66], allowing any intermediate level of detail to be extracted, the expense of simplification needs only be paid once.

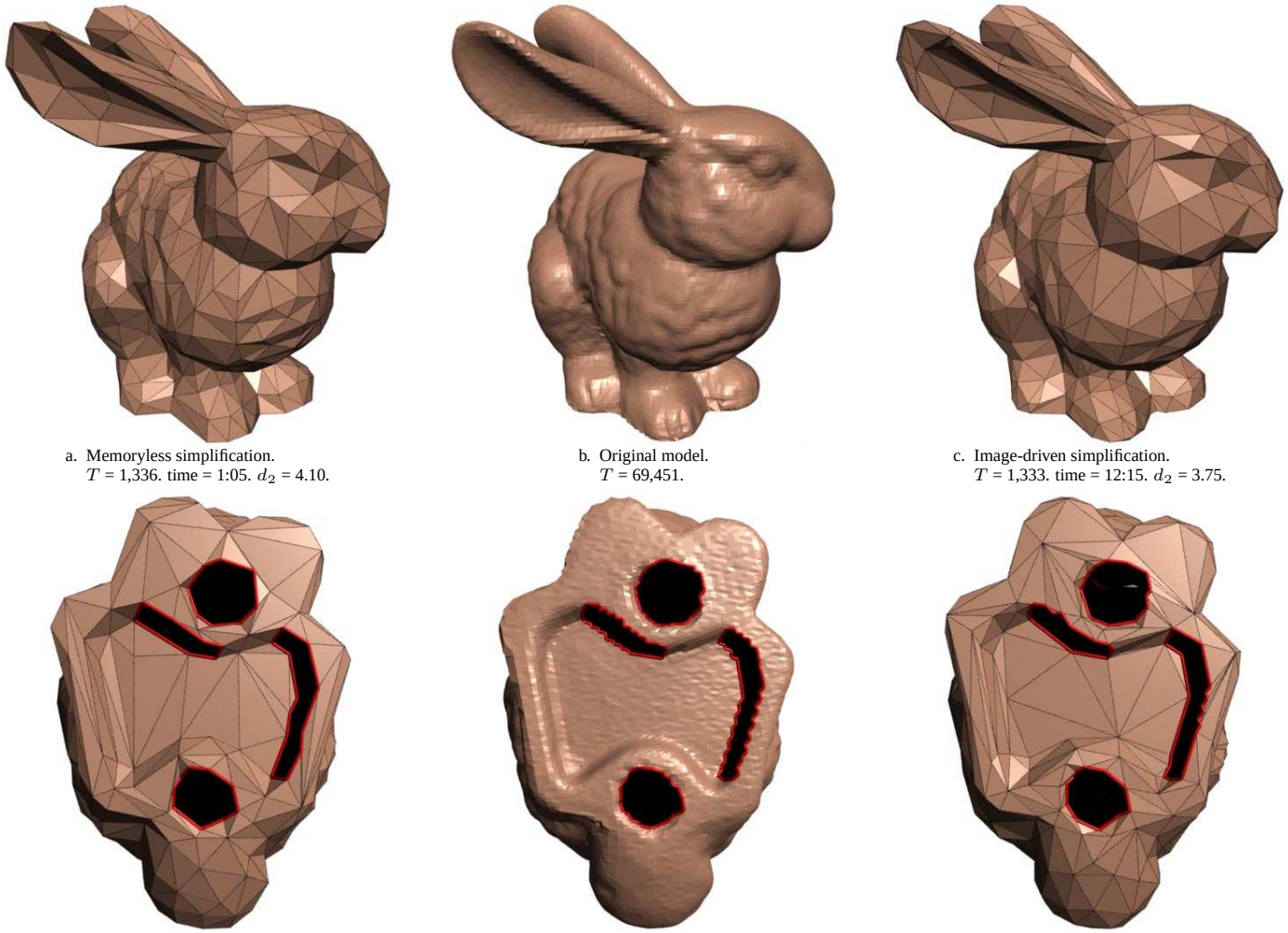
7.3.1 Image- versus Geometry-Driven Simplification

Because the idea of using images rather than geometry to guide simplification is such a large departure from mainstream simplification methods, I will first compare objects created by this new method against those of two other simplification techniques, namely my memoryless method and Garland and Heckbert's QSlim software [50, 51].[2] I used these three methods to create several levels of detail of the Stanford bunny.

Figure 7.1 shows two views of the original bunny model, a memoryless simplification model, and an

[1] As described below, different options were used for the frog model, resulting in a much slower simplification.

[2] For untextured models, QSlim v2.0 was used with the options `-B 10`, while PropSlim v1.2 was used for the textured models.



a. Memoryless simplification. T = 1,336. time = 1:05. d_2 = 4.10.

b. Original model. T = 69,451.

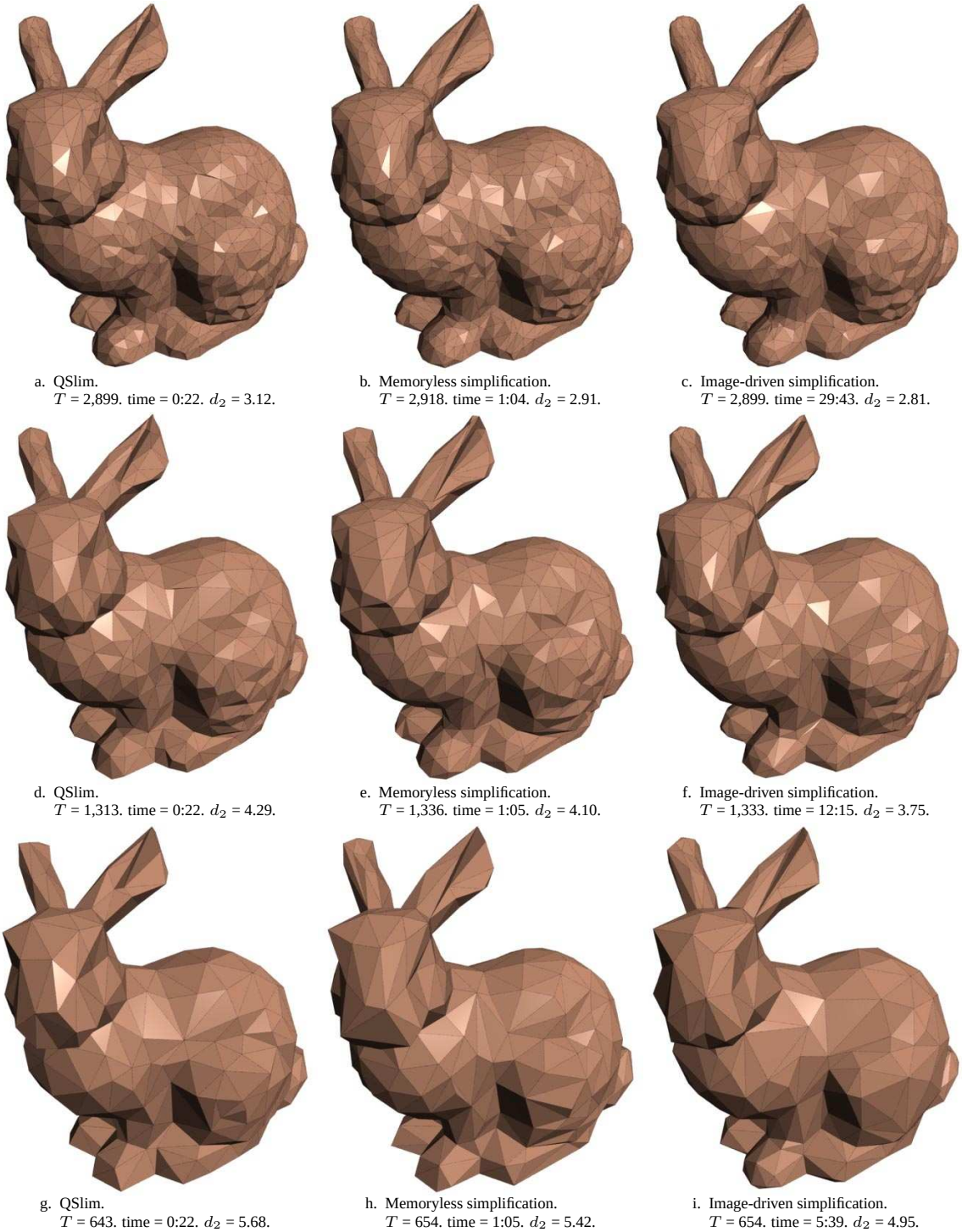
c. Image-driven simplification. T = 1,333. time = 12:15. d_2 = 3.75.

d. Memoryless simplification. 42 boundary edges.

e. Original model. 223 boundary edges.

f. Image-driven simplification. 59 boundary edges.

Figure 7.1: Views showing the front and underside of the Stanford bunny model.



a. QSlim.
T = 2,899. time = 0:22. d_2 = 3.12.

b. Memoryless simplification.
T = 2,918. time = 1:04. d_2 = 2.91.

c. Image-driven simplification.
T = 2,899. time = 29:43. d_2 = 2.81.

d. QSlim.
T = 1,313. time = 0:22. d_2 = 4.29.

e. Memoryless simplification.
T = 1,336. time = 1:05. d_2 = 4.10.

f. Image-driven simplification.
T = 1,333. time = 12:15. d_2 = 3.75.

g. QSlim.
T = 643. time = 0:22. d_2 = 5.68.

h. Memoryless simplification.
T = 654. time = 1:05. d_2 = 5.42.

i. Image-driven simplification.
T = 654. time = 5:39. d_2 = 4.95.

Figure 7.2: Three levels of detail of the bunny model.

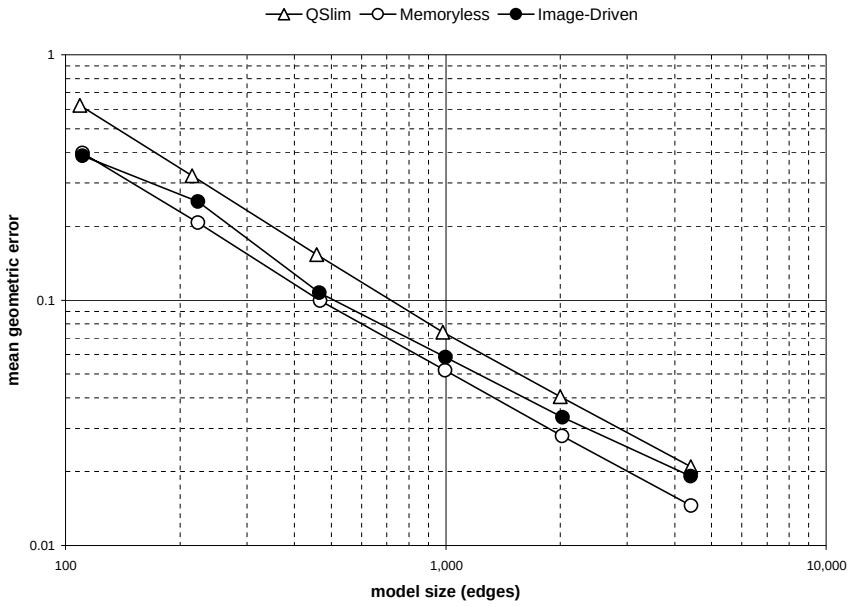


Figure 7.3: Mean geometric errors for the bunny model.

image-driven model. The bunny models produced by these two methods are comparable in terms of visual quality. There are, however, some subtle differences between the results of these two methods that I have observed for a variety of flat shaded models. First, the image-driven method generally pays more attention to preserving the shape of silhouettes and sharp edges (see, for example, the shape of the bunny ears and the rounding along and across the edges of the feet). Second, soft interior edges with small intensity gradients, such as small bumps and furrows, are typically smoothed out and replaced with larger triangles by the image-driven method (e.g. the chest and the back of the bunny). As a rule of thumb, the image-driven method spends relatively more detail preserving the sharp intensity boundaries in an image, which convey most of the large-scale shape of an object. This is typically the relevant information one seeks to retain in a highly simplified model. Third, the image-driven method better balances the ratio of boundary edges to interior edges of a model by using a single unified error metric, as evidenced by Figures 7.1d, 7.1e, and 7.1f. Most geometry-based algorithms have difficulties choosing an appropriate level of boundary fidelity without explicit user intervention, e.g. via a parameter that allows boundary and interior edges to be weighted differently. The image-driven method, on the other hand, accounts for the visual impact due to boundary changes directly. Finally, the image-driven method often produces better shaped triangulations that more closely follow the intrinsic geometry of a surface, with few sliver triangles and other artifacts such as folds in the mesh. Such degeneracies are visually disturbing and are easily caught by the image metric. Figure 7.2, which shows several different levels of detail of the bunny, lends further support to these claims.

Two quantitative measurements were made of the levels of detail that were produced. First, the geometric tool Metro was used to calculate the mean geometric error between each of the simplified models and the

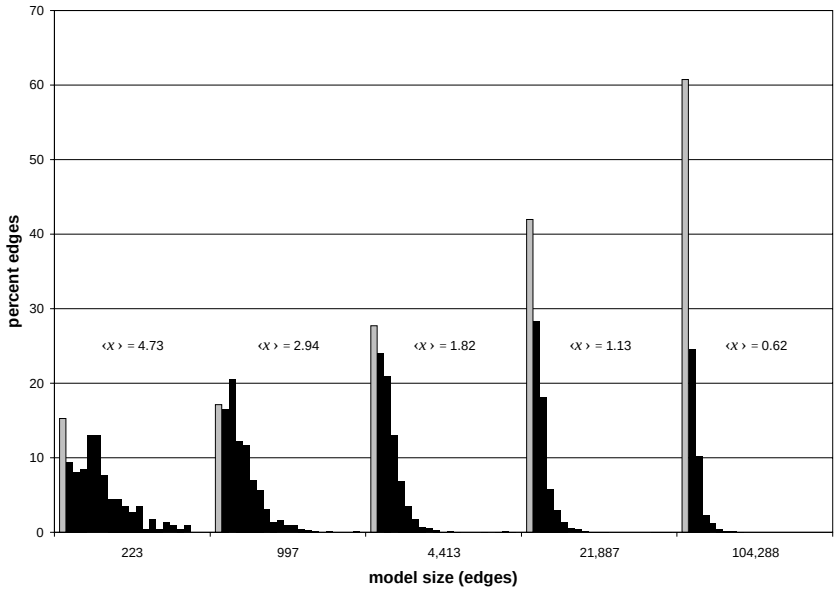


Figure 7.4: Silhouette edge histograms for the bunny model. For each of the five levels of detail, each histogram column corresponds to a certain number of views between zero (leftmost, shaded column) and twenty. The height of the i^{th} column gives the percentage of edges that are silhouette edges in exactly i views. Thus the shaded columns correspond to the percentage of edges that are not on the silhouette in any view. For each level of detail, the expectation value $\langle x \rangle$ gives the expected number of views in which an edge is a silhouette edge.

original [24].[3] Figure 7.3 is a graph of the results, where the errors are expressed in units of $1/50^{th}$ of the bounding box diagonal of the original model. The memoryless method produces models that have the least geometric error, followed closely by the image-driven method, and then QSlim. Using the image-based cost produces models that are surprisingly close to the original models geometrically. I attribute this result to three factors: (1) the use of the high quality memoryless algorithm to pre-simplify the models and as the method of positioning vertices in the image-driven method, (2) the indirect effects of preserving the shading of a model, which depends directly on the model's geometric shape, and, more important, (3) the sensitivity of the mean square image metric to silhouette discrepancies. For progressively coarser models, the likelihood of an edge being on a silhouette increases (see Figure 7.4). Thus, given a large number of views, we expect the majority of edges to be silhouette edges from several viewpoints, which limits the amount of geometric degradation.

The second form of evaluation performed used an image metric. Each model was rendered from 24 camera positions, as described in §6.1.1, and the RMS error d_2 between the original and simplified model's images was calculated. These errors are the same as the d_2 values provided in the captions. Figure 7.5 shows the results of this evaluation. As expected, the image-driven method consistently produced models of higher quality than the two other methods with respect to this measure. Note that there are at least three ways

[3] Metro v2.5 was used with the options `-s -t`.

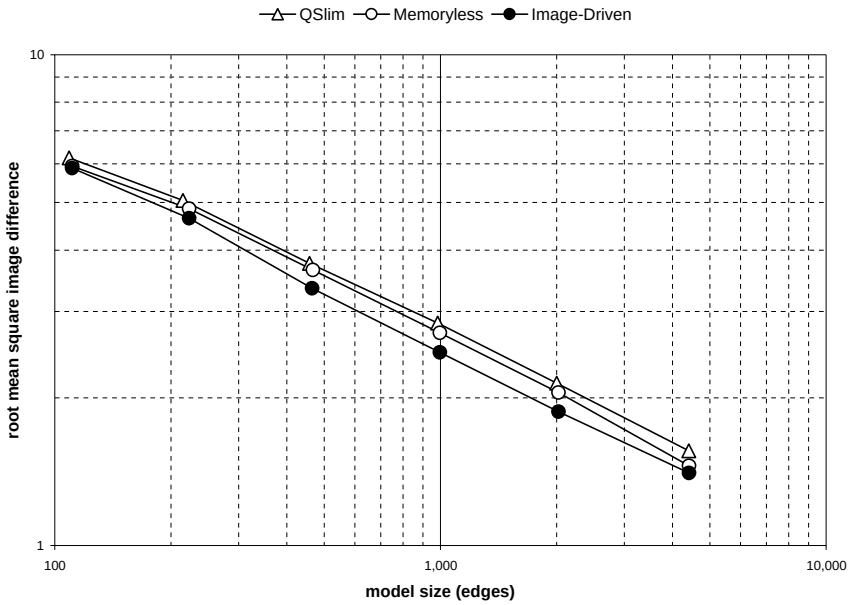


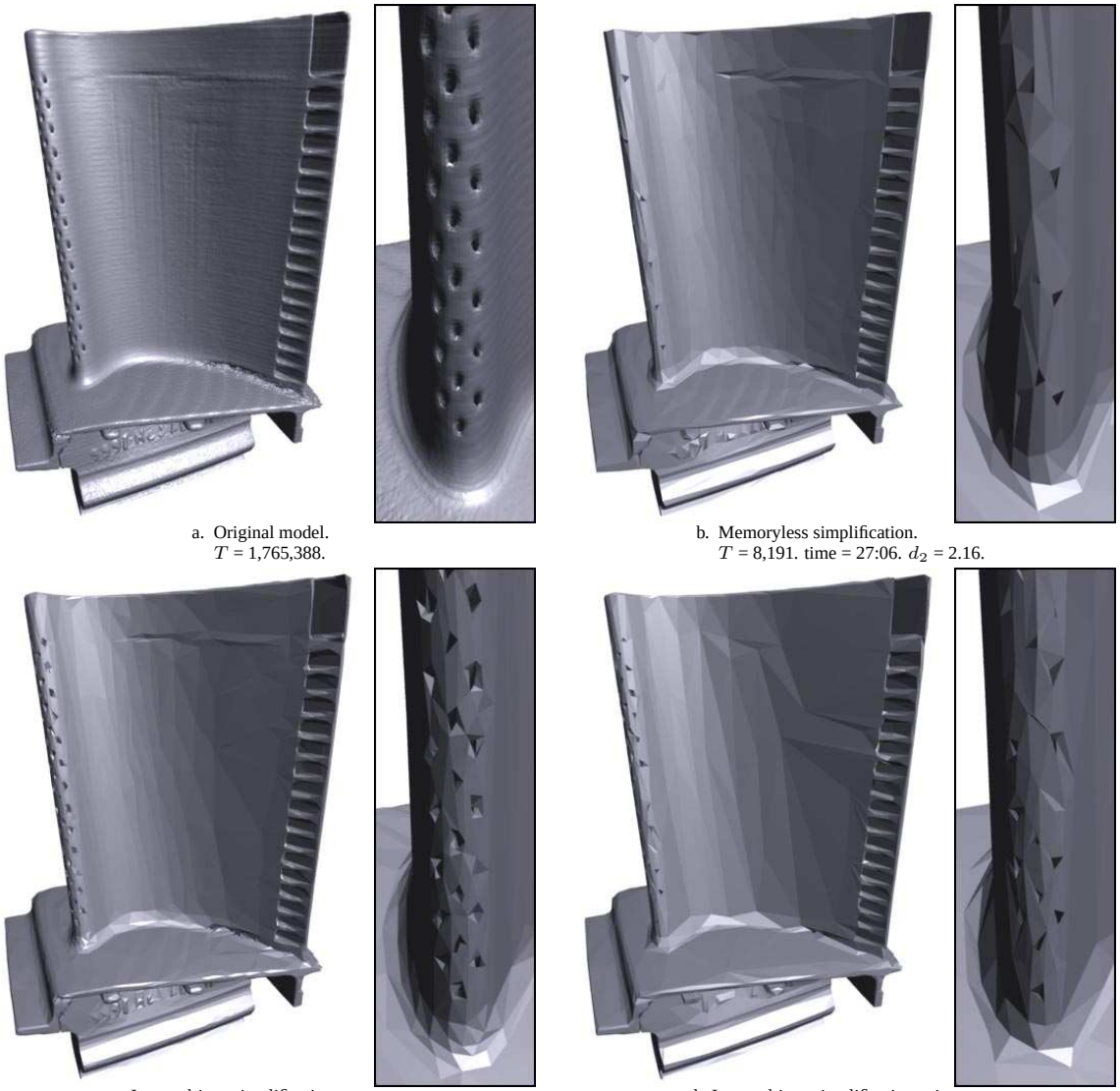
Figure 7.5: Root mean square image differences for the bunny model. Due to the severe degree of simplification, the coarsest two levels of detail for all three methods are of very poor visual quality, which explains the convergence of image differences among the methods.

in which this image metric differs in detail from the actual metric used during the simplification process. First, the simplification method relies on rendering hardware without anti-aliasing to perform polygon scan conversion. In contrast, the images that were used in the off-line evaluation (Figure 7.5) were created by RenderMan and were anti-aliased. Second, the dimensions of the images used during simplification were 256×256 pixels, while the final images have dimensions 512×512. Finally, and most importantly, the distribution and number of camera positions on the sphere were different during simplification than evaluation (20 versus 24).

Is it a tautology that a model produced by image-driven simplification is a close match to the original model when evaluated using an image metric? No more so, I argue, than using geometric proximity to guide simplification and then measuring geometric differences. More important is the decision of what result is desired—geometric similarity or visual similarity. Because visual similarity is often the goal of simplification, I believe that we should try to achieve this goal in a direct manner, using images.

7.3.2 Simplification of Hidden Interiors

Image-driven simplification removes detail that has little effect on rendered images of a model. Consequently, if a portion of a model is completely invisible in all views, it will be radically simplified. Figures 7.6 and 7.7 provide an example of this. The original model (7.6a) is a turbine blade that has a complex arrangement of internal cavities that accounts for much of the model's 1,765,388 polygon complexity. Figure 7.7a shows



a. Original model.
T = 1,765,388.

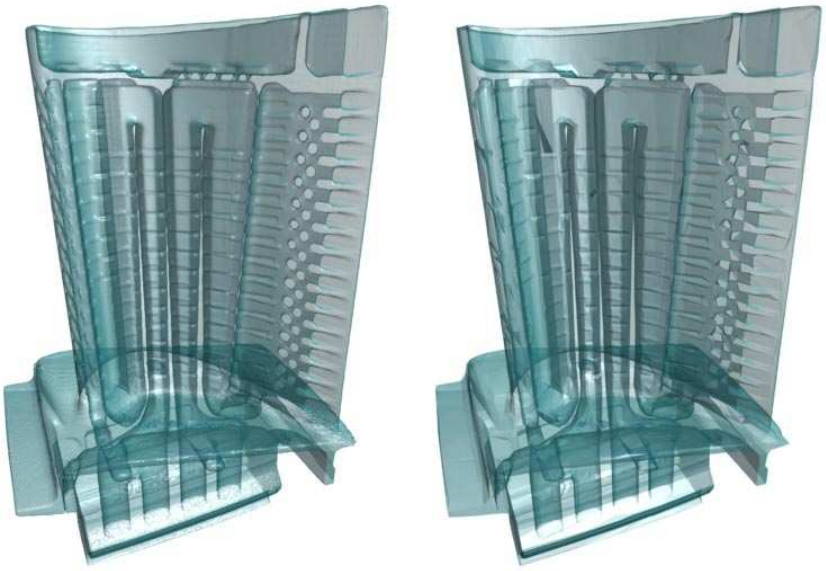
b. Memoryless simplification.
T = 8,191. time = 27:06. d_2 = 2.16.

c. Image-driven simplification.
T = 8,191. time = 3:54:55. d_2 = 1.57.

d. Image-driven simplification using
front-face culling in 10 of 20 views.
T = 8,191. time = 1:33:34. d_2 = 2.10.

Figure 7.6: Exterior views of turbine blade model.

the same model for which the surface has been made semi-translucent to reveal this interior detail. When simplified using a geometry-driven method all of this interior detail is retained, yet none of it contributes to opaque images of the model (Figures 7.6b and 7.7b). The image-driven method, on the other hand, produces a model that has a more detailed exterior and a greatly simplified interior (Figures 7.6c and 7.7c). Removing all of the internal detail allowed a substantially higher polygon budget to represent the visible portions of the model, such as the numerous holes on the left side of the blade, as illustrated by the close-ups in Figure 7.6.



a. Original model.

b. Memoryless simplification.

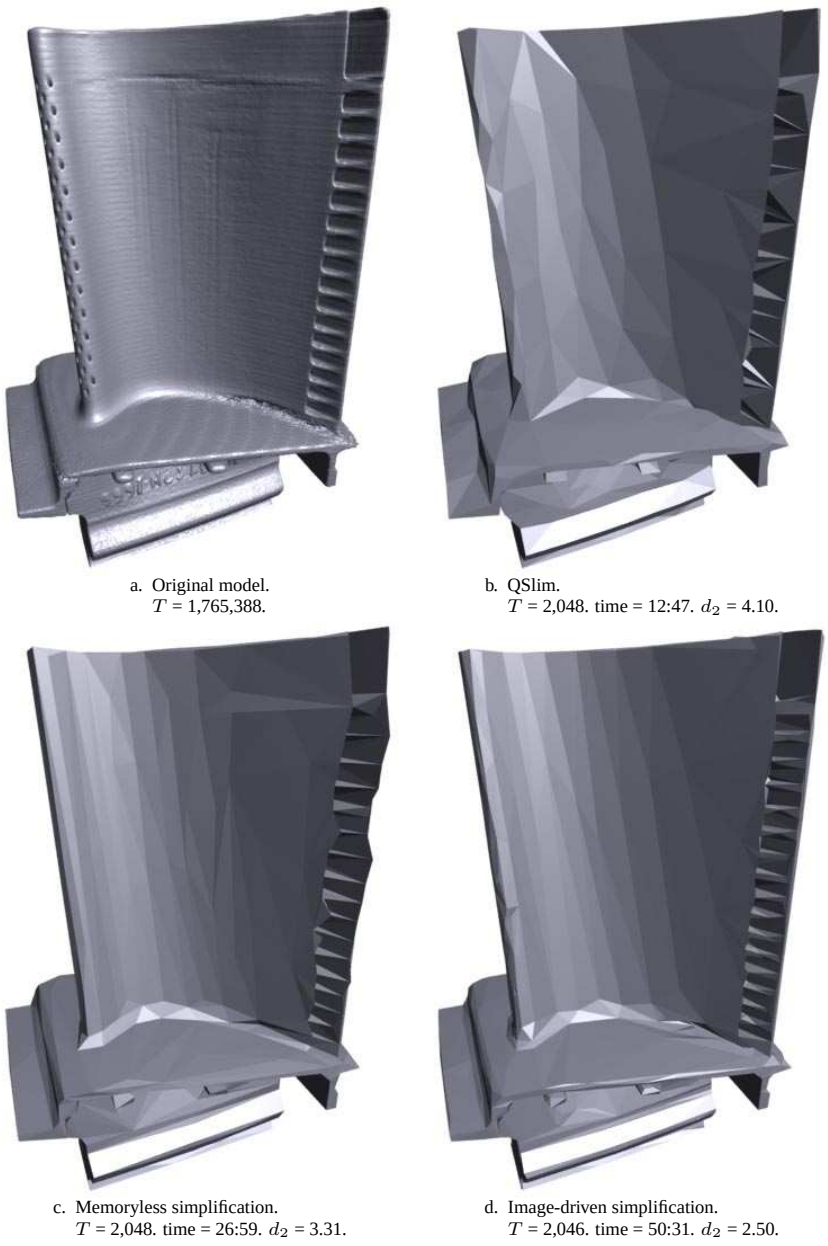
c. Image-driven simplification.

d. Image-driven simplification using front-face culling.

Figure 7.7: Interior views of turbine blade model.

Due to the topological complexity of this model, both methods were allowed to simplify the surface topology by placing no restrictions on what edges could be collapsed. This is the reason why many of the smaller holes were closed by the geometry-based method.

Figure 7.8 shows a coarser version of the same model, simplified using the three different methods. These images clearly demonstrate the advantage of not being constrained to preserve the interior, invisible parts of



a. Original model.
T = 1,765,388.

b. QSlim.
T = 2,048. time = 12:47. d_2 = 4.10.

c. Memoryless simplification.
T = 2,048. time = 26:59. d_2 = 3.31.

d. Image-driven simplification.
T = 2,046. time = 50:31. d_2 = 2.50.

Figure 7.8: Coarse levels of detail of turbine blade model.

the model. As can be seen, both QSlim and the memoryless method had great difficulties preserving the shape of the exterior.

The improvement in exterior appearance provided by the image-driven method clearly comes at the expense of drastically eliminating interior detail. This tradeoff is desirable if the model is only to be viewed from the outside. There are applications, however, that require more control over the tradeoff between interior

and exterior quality such that hidden parts, while being greatly simplified, at least retain their basic shape. For models with low depth complexity, I have devised a partial solution by allowing the exterior surface to be removed in a subset of the views during simplification. This is achieved by enabling front-face culling in OpenGL, which comes at no additional computational expense. Figures 7.6d and 7.7d show the same blade model for which the interior was revealed in half of the 20 views used during simplification. Consequently, the interior of this model is much better retained, while the exterior quality suffered slightly. In this manner, the user is given smooth control over the balance between interior and exterior detail by choosing the number of interior and exterior views appropriately. While acceptable for this particular model, I acknowledge that this solution is not universal. There are models for which additional steps may have to be taken to preserve fully or partially hidden details. Possible alternative approaches include the use of translucency, cutting planes, model segmentation into independent parts, and optimal view coverage (such as camera positions inside the model). I see this issue is secondary, however, and I will return my focus to the goal of producing high quality simplifications of the visible parts of a model.

7.3.3 Flat versus Gouraud Shading

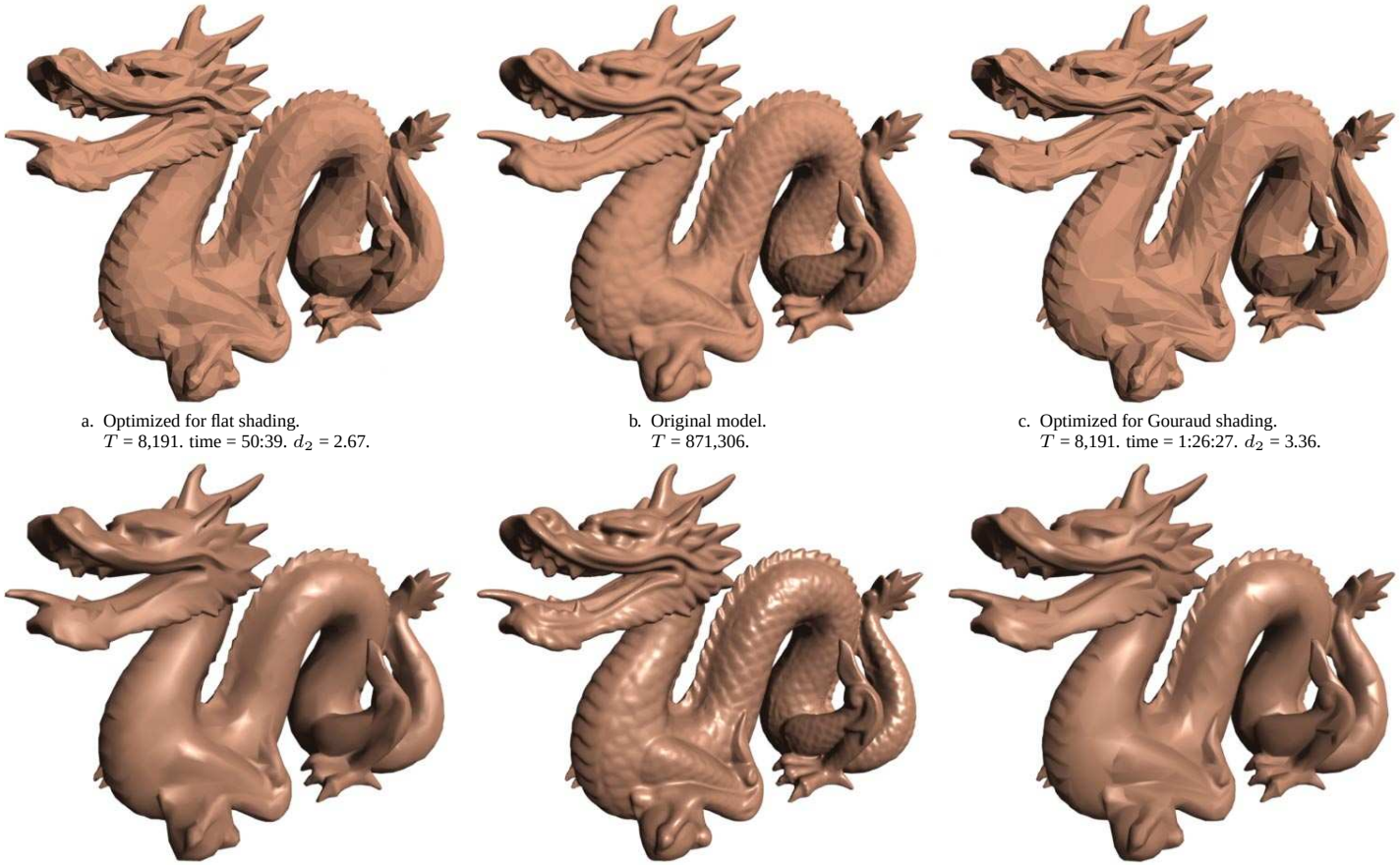
Geometric simplification methods do not take into account the type of shading interpolation that will be used when a model is rendered. None of the published methods make a distinction between objects that will be flat shaded (constant shading over each face) and those that will be rendered with Gouraud interpolation (linear interpolation of the vertex intensities). Image-driven simplification, on the other hand, can account for these differences during simplification.

I chose to compare flat and Gouraud shading since these are the two shading models most commonly supported in both software and hardware. While Phong interpolation usually yields higher quality shading, it is seldom available in graphics hardware, and is rarely applied to coarse polygonal objects as its high computational expense would offset any performance gains attained by the use of simplified models.

Figure 7.9 shows a dragon model that was simplified twice with the image-driven method. The first simplified model was guided by images in which the intermediate models were rendered using diffuse flat shading of the faces (Figures 7.9a and 7.9d). The second simplification was guided by images for which specular reflection and Gouraud interpolation of the intensities calculated at the polygon vertices were used (Figures 7.9c and 7.9f). For the Gouraud shaded models, the vertex normals were computed as the area-weighted sum of incident face normals.[4]

Figure 7.9 shows each of these two simplified models and the original model rendered with flat and with Gouraud interpolation. The model produced with flat shading is the better of the two when viewed with flat shading, and the Gouraud-image-driven model retains much more of the detail in the Gouraud-rendered image. The lower-right dragon retains better details in many places, including the front scales, the face, the feet, the hind leg and the ridge along the back. The reason this detail is kept is because the simplification process takes into account the effect of all the various surface attributes during the simplification process.

[4]Note that since the normals for the vertices $\lfloor\lceil v\rceil\rfloor$ depend on the position of v, so does the shading of the triangles $\lceil\lceil\lfloor\lceil v\rceil\rfloor\rceil\rceil$. Thus, for Gouraud shading, the sets T and T' must be expanded accordingly (see §7.2.2).



a. Optimized for flat shading. T = 8,191. time = 50:39. d_2 = 2.67.

b. Original model. T = 871,306.

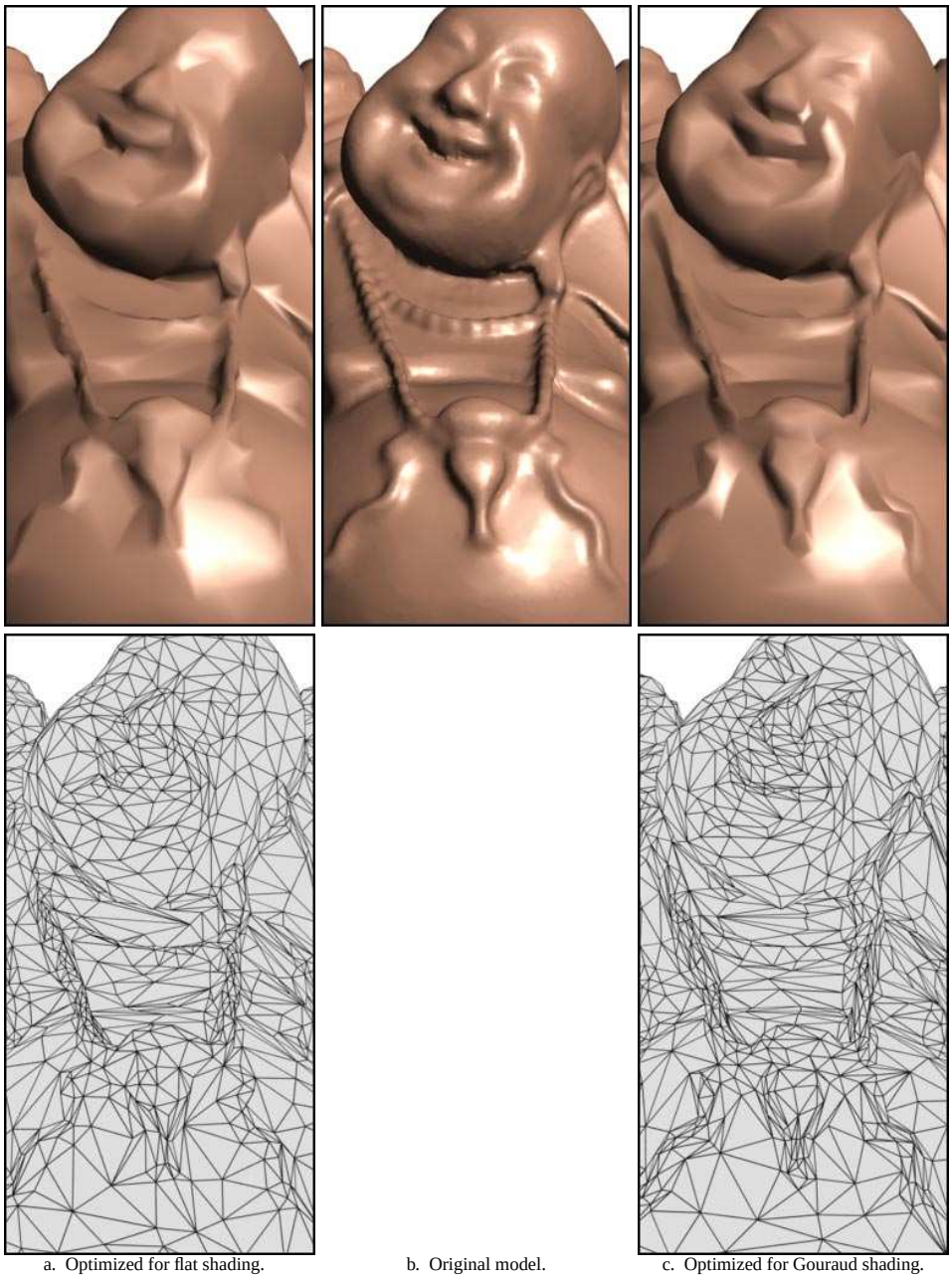
c. Optimized for Gouraud shading. T = 8,191. time = 1:26:27. d_2 = 3.36.

d. Optimized for flat shading. d_2 = 3.46.

e. Original model.

f. Optimized for Gouraud shading. d_2 = 2.86.

Figure 7.9: Dragon model rendered using flat (top) and Gouraud (bottom) shading. The results reflect the use of flat (left) and Gouraud (right) shading during image-driven simplification.



a. Optimized for flat shading.
T = 16,384. time = 2:13:39.
d_2 = 2.26.

b. Original model.
T = 1,087,716.

c. Optimized for Gouraud shading.
T = 16,384. time = 2:29:17.
d_2 = 1.75.

Figure 7.10: Close-ups of Gouraud shaded Buddha model. The wireframe renderings show how in 7.10c relatively more polygons were devoted to features such as the eyelids, the mouth, and the necklace. In particular, many skinny triangles were used along the boundaries of these features to sharpen creases and localize changes in vertex normal orientation, whereas the normal field is more diffuse in 7.10a.

The quality tradeoff between polygon size, vertex positions, and variation in surface normals across triangles are all handled by the image metric. Figure 7.10 further demonstrates how different shading models can successfully be accounted for.

If the rendering parameters are not known beforehand, one might suspect that the models produced by the image-driven method would be greatly suboptimal when displayed using a different set of rendering parameters. In practice, however, I have found that the image-driven method usually outperforms the geometry-driven methods even when the rendering parameters do not match. To optimize the model for a variety of rendering conditions, one could alternatively vary the parameters between views during simplification (cf. §7.3.2), and thus produce an "average" model that is not tailored to a particular set of rendering or surface parameters.

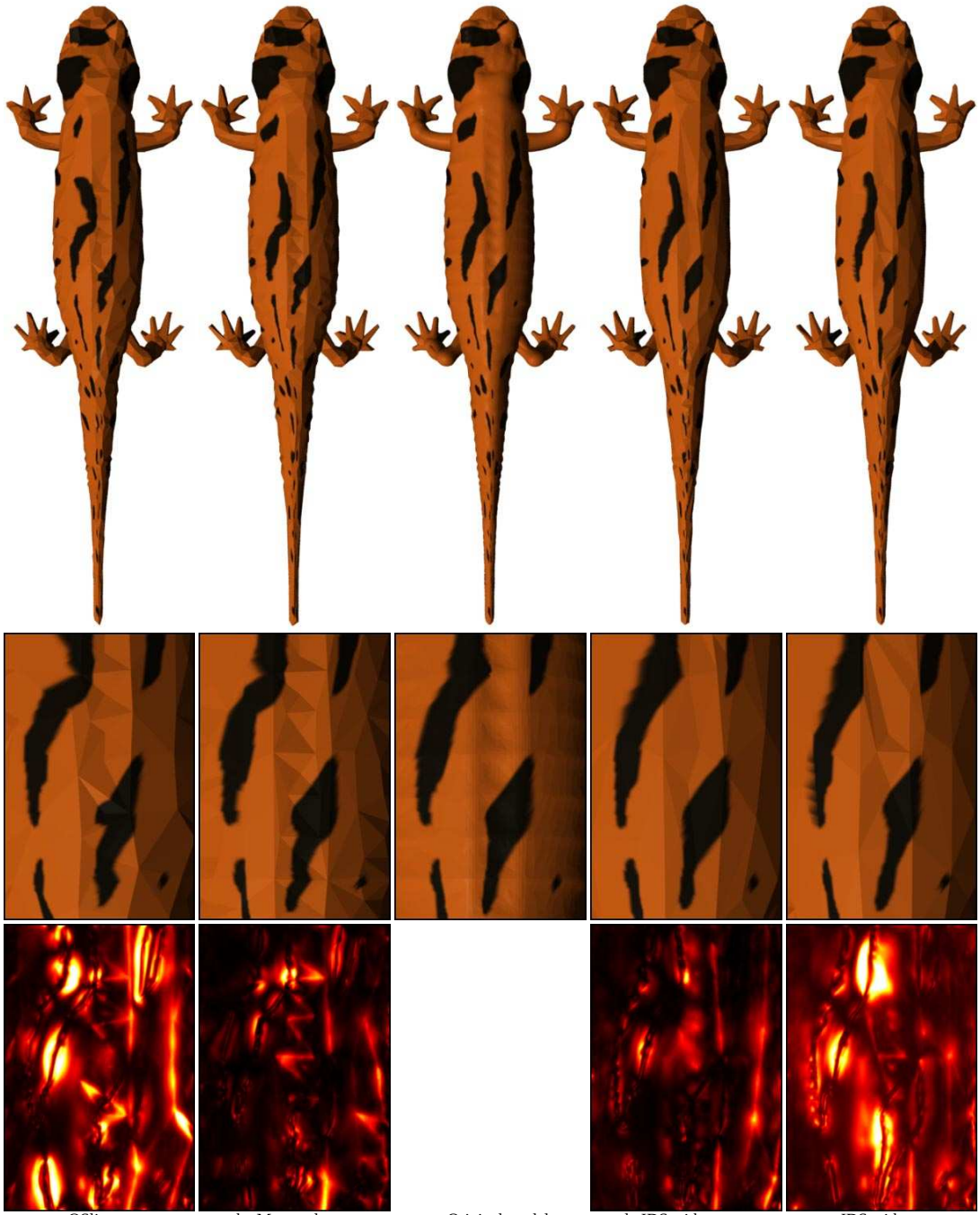
7.3.4 Texture-Sensitive Simplification

Published methods for simplifying textured models have concentrated on avoiding shifts in the *texture parameterization* across the entire surface of the model. Many textures, however, vary in the amount of color variation over different portions of a model. Texture of a human face is nearly uniform over much of the skin, but varies greatly near the eyes, and a map of the Earth is detailed on the continents, but is uniform over the oceans. These differences in color detail provide an opportunity for heavier simplification in the regions of nearly uniform color.

The image-driven simplification method easily recognizes and exploits the opportunity of varying texture detail. Figures 7.11c and 7.12c show two different views of two textured versions of the same geometric salamander model, one with thin black spots on orange and another with large black spots on green. The underlying polygon model and the texture coordinates are the same for both original models. This model was simplified once with QSlim and the memoryless geometric method, which both ignore the actual texture image used, and twice with the image-driven method, once with each texture. When the image-driven models are rendered with the same texture used during simplification, they closely match the originals. When the textures are swapped, however, the spots shift and change shape. This can be seen in the close-ups and difference images with respect to the original model that are given for each simplified salamander. This indicates that the simplified models devoted only those polygons necessary to maintain the texture detail, and did not expend additional polygons in the uniformly colored regions of the texture. The geometrically simplified salamander has texture distortions with either texture. In particular, both of the geometry-driven algorithms had great difficulties preserving the texture along the underside of the salamander, where the texture wraps around and meets itself along a jagged seam (see Figure 7.12).[5] The image-driven method, on the other hand, had no difficulties accounting for distortion along the seam, and consequently produced more visually pleasing results.

Just as geometric fidelity of a simplified model depends on the methods used for placing new vertices and ordering edge collapses, the visual quality of a texture parameterization over a simplified surface is

[5] In contrast to my geometry- and image-driven algorithms, QSlim does not support multiple texture coordinate pairs per vertex, and so had additional problems dealing with the texture seams.



a. QSlim. T = 2,048. time = 0:28. d_2 = 1.29.

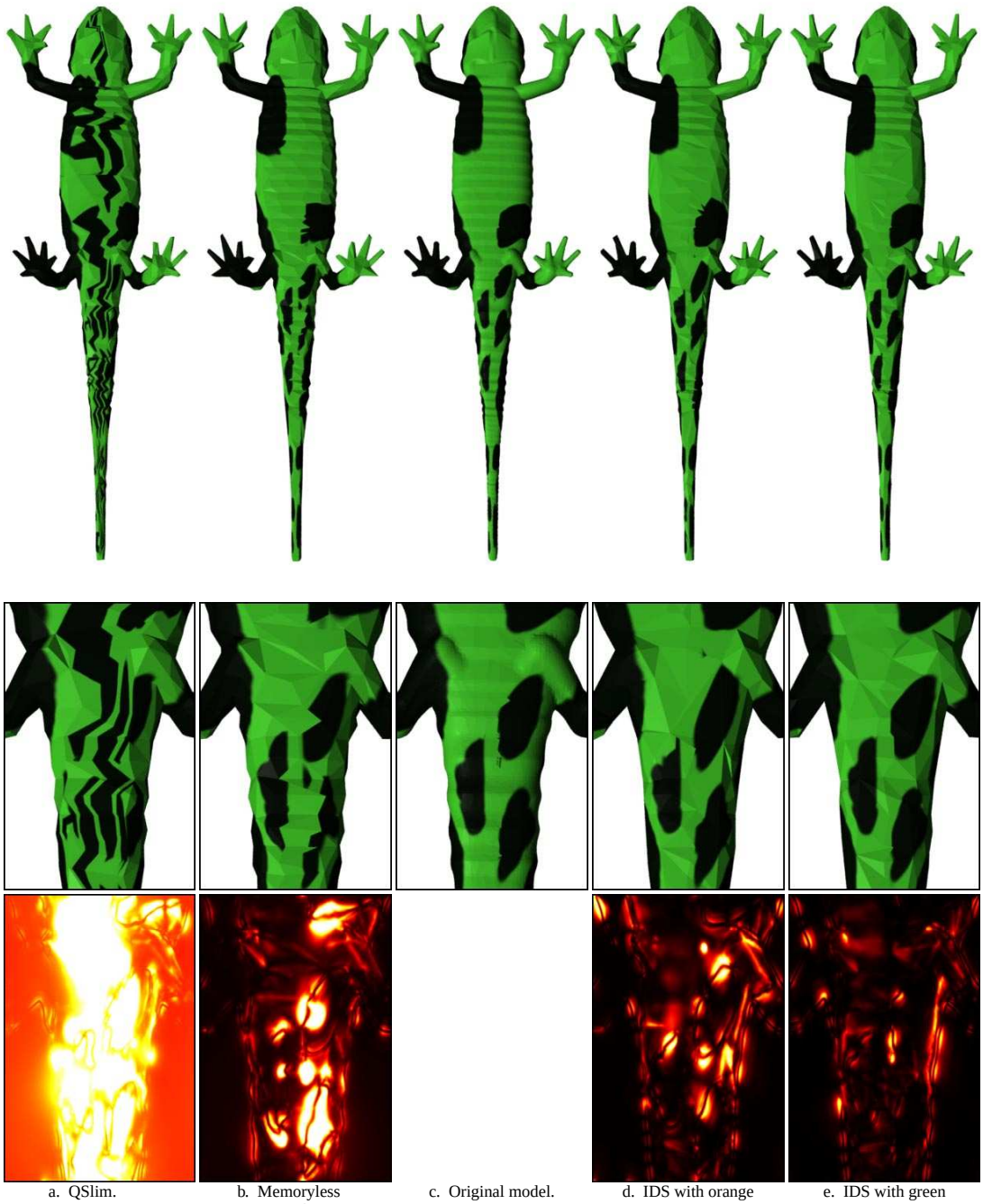
b. Memoryless simplification. T = 2,047. time = 1:21. d_2 = 0.96.

c. Original model. T = 71,312.

d. IDS with orange texture. T = 2,047. time = 17:13. d_2 = 0.92.

e. IDS with green texture. T = 2,047. time = 17:30. d_2 = 1.00.

Figure 7.11: Top view of salamander model rendered with orange texture map. The results demonstrate the image-driven simplification (IDS) method's sensitivity to the contents of the texture image.



a. QSlim.
T = 2,048.
time = 0:28.
d_2 = 2.48.

b. Memoryless simplification.
T = 2,047.
time = 1:21.
d_2 = 1.18.

c. Original model.
T = 71,312.

d. IDS with orange texture.
T = 2,047.
time = 17:13.
d_2 = 1.10.

e. IDS with green texture.
T = 2,047.
time = 17:30.
d_2 = 0.90.

Figure 7.12: Bottom view of salamander model rendered with green texture map.

determined by two factors: the method for computing per-vertex texture coordinates, and the order in which edges are collapsed. I have found that my memoryless approach to computing texture coordinates is a good alternative for the first of these two components. Even when the edge collapses are ordered without regard to texture quality, as in my geometry-based method, the results are generally better than those produced by QSlim. When texture appearance is additionally accounted for in the edge cost, as in my image-driven algorithm, the increase in visual quality can be quite dramatic.

7.3.5 Sensitivity to Degeneracies

A common problem in geometry-based simplification is how to handle and avoid geometric and topological degeneracies in the mesh. These artifacts may either be present in the input mesh, or may inadvertently be created during simplification. Some example problems include:

- **T-vertices and topological boundaries**. A surface may be geometrically continuous but topologically disjoint, such as at the interface between mesh parts of different resolution, or along material or texture boundaries for which vertices may have to be duplicated (see Figure 7.13a). This may cause cracks to be introduced between the pieces during simplification.

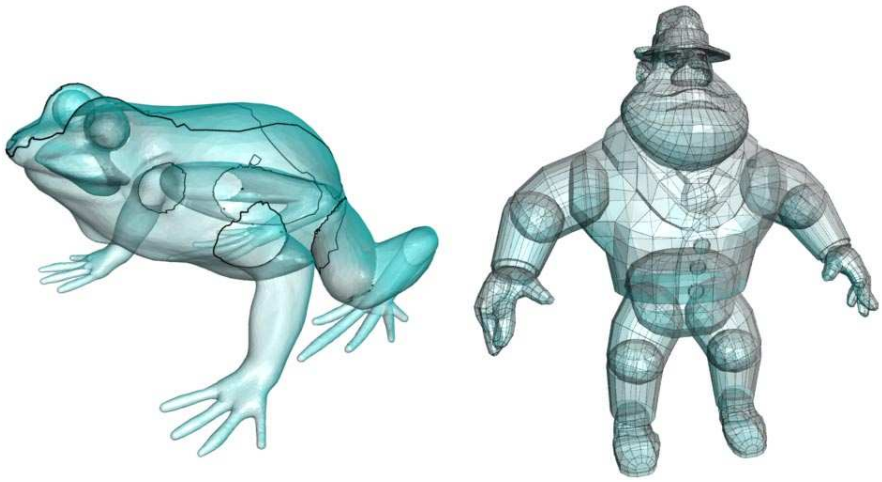
- **Non-manifold surfaces**. Computing vertex normals and determining the surface orientation for culling and shading calculations are both ill-defined operations on a non-manifold mesh.

- **Folded geometry**. The edge collapse operation can introduce folds in the mesh if the substitute vertex is displaced too far laterally. This may flip the orientation of one or more triangles, leading to shading artifacts.

- **Interpenetrations**. Without any constraints on the geometry of the surface, self-intersections may already exist or may be introduced during simplification, potentially causing visual artifacts. Figure 7.13b shows an example of a model that by design has intersecting geometry. The zebra model in Figure 7.15 similarly is made up of separate intersecting body parts.

The simplification process can sometimes avoid introducing artifacts like these. For example, one may prohibit edge collapses that create non-manifold simplices [72], and one can similarly test for folds in the mesh [26, 50, 57, 126] introduced by an edge collapse (but a cascading sequence of edge collapses may also lead to folds). Interpenetrations are particularly difficult to test for, and usually require extensive computation to avoid or even detect. To my knowledge, the only published simplification algorithm that correctly handles self-intersections is the one by Cohen et al. [28]. While some of these artifacts can be eliminated as a preprocessing step (cf. [111]), it can be difficult to detect if degeneracies are introduced during simplification, and if so whether they affect the visual quality of the model. When images are used to drive simplification, however, such artifacts are automatically penalized, and can easily be avoided.

To illustrate how unwanted self-intersections and gaps can be avoided, the frog model from Figure 7.13a was simplified using the image-driven and the memoryless method. This model, shown with texture in Figure 7.14a, is composed of several connected components, with different components for the body, the legs,



a. Texture boundaries. b. Interpenetrating parts.

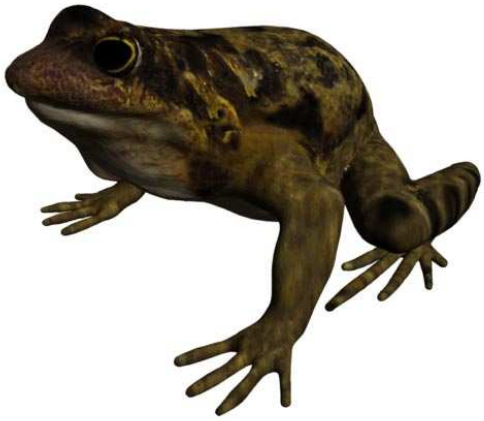
Figure 7.13: Common artifacts in polygon meshes. The boundaries between different texture regions are shown on the frog. Because vertices are duplicated on these boundaries, the different body parts are disjoint. The limbs on the model on the right are also topologically disjoint, but closed and interpenetrating.

and the eyes. Such pieced together models are commonly used in video games and in feature film special effects. Figure 7.14b is a simplified frog model produced by the memoryless method. Two problems are evident: One of the eyeballs is poking through a part of the head and shows up as a white blotch. There is also cracking evident between the legs and the body because these components were never joined in the first place, and edge collapses around the places where they join have caused a mismatch between components. Conventional simplification methods are not often used on such models because few geometric quality measures recognize these problems. Using an image metric, these potential problems are avoided, resulting in the simplified model shown in Figure 7.14c. Note that the image-driven method used the perceptual metric from §6.3, which paid more attention to gaps and other visual artifacts than the mean square metric on this model. In addition, a different edge cost update policy was used, for which the edges $\lceil\lfloor\lceil\bar{v}\rceil\rfloor\rceil$ were updated directly and $\lceil\lfloor\lceil\lfloor\lceil\bar{v}\rceil\rfloor\rceil\rfloor\rceil \setminus \lceil\lfloor\lceil\bar{v}\rceil\rfloor\rceil$ were marked as dirty. These two factors slowed down the simplification speed considerably.

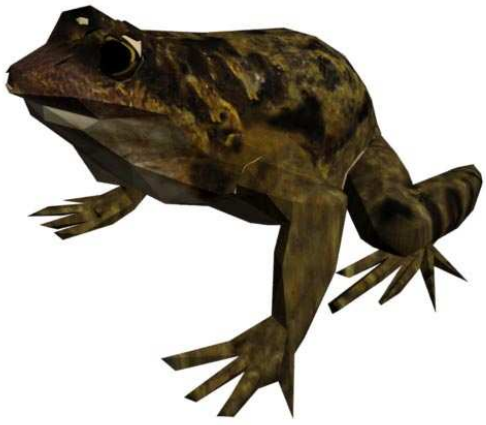
7.3.6 Extreme Simplification

Based on the argument in §7.2.3, image-driven simplification is most valuable when very coarse models are needed. For these models, it often pays off to expend more computational effort, especially in the final stage, as the improvement in quality can sometimes be substantial. We have already seen a few examples where using the image-driven method to produce low-complexity models resulted in higher quality, such as the coarsest versions of the turbine blade model (Figure 7.8). In this section, I will provide a few more examples.

Figure 7.15 shows several levels of detail of a zebra model (see Figure 7.16a for the original model), simplified using each of the three methods. There are a number of apparent problems with the memoryless



a. Original model.
T = 47,028.

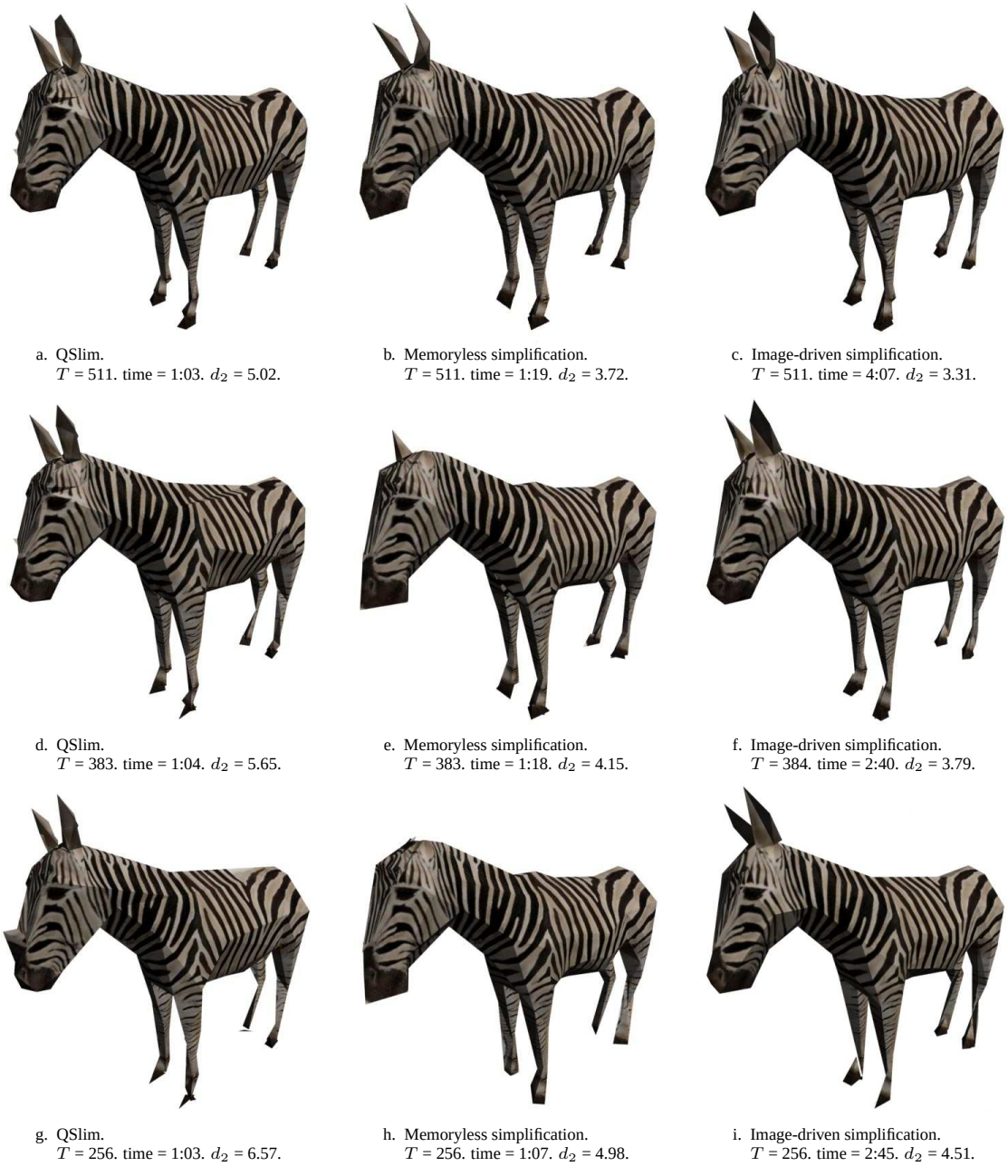


b. Memoryless simplification.
T = 1,023. time = 0:48. d_2 = 3.24.

c. Image-driven simplification.
T = 1,024. time = 3:17:01. d_2 = 2.50.

Figure 7.14: Frog model. Notice the self-intersecting geometry near the eyes and the cracks between the legs and the torso in 7.14b. The perceptual metric from §6.3 was used in the image-driven method.

simplified models (Figures 7.15b, 7.15e, and 7.15h), such as the distorted ears, muzzle, and hooves. In fact, the ears and the hooves are all but eliminated in Figure 7.15h, because they were never joined topologically to the rest of the body in the first place. These problems are less pronounced in the models produced by QSlim (Figures 7.15a, 7.15d, and 7.15g), although other artifacts can be seen, such as severe distortion along the forehead, with several polygons jutting out, and a poorly approximated back with noticeable texture distortion. These artifacts are either absent or much less noticeable in the models simplified by the image-driven method (Figures 7.15c, 7.15f, and 7.15i). Note that this view is particularly favorable for the zebra models produced by QSlim. When seen from other viewpoints, the mismatch in texture becomes more apparent, which is reflected in the numerical image differences included in the captions. Figure 7.16 shows



a. QSlim. T = 511. time = 1:03. d_2 = 5.02.

b. Memoryless simplification. T = 511. time = 1:19. d_2 = 3.72.

c. Image-driven simplification. T = 511. time = 4:07. d_2 = 3.31.

d. QSlim. T = 383. time = 1:04. d_2 = 5.65.

e. Memoryless simplification. T = 383. time = 1:18. d_2 = 4.15.

f. Image-driven simplification. T = 384. time = 2:40. d_2 = 3.79.

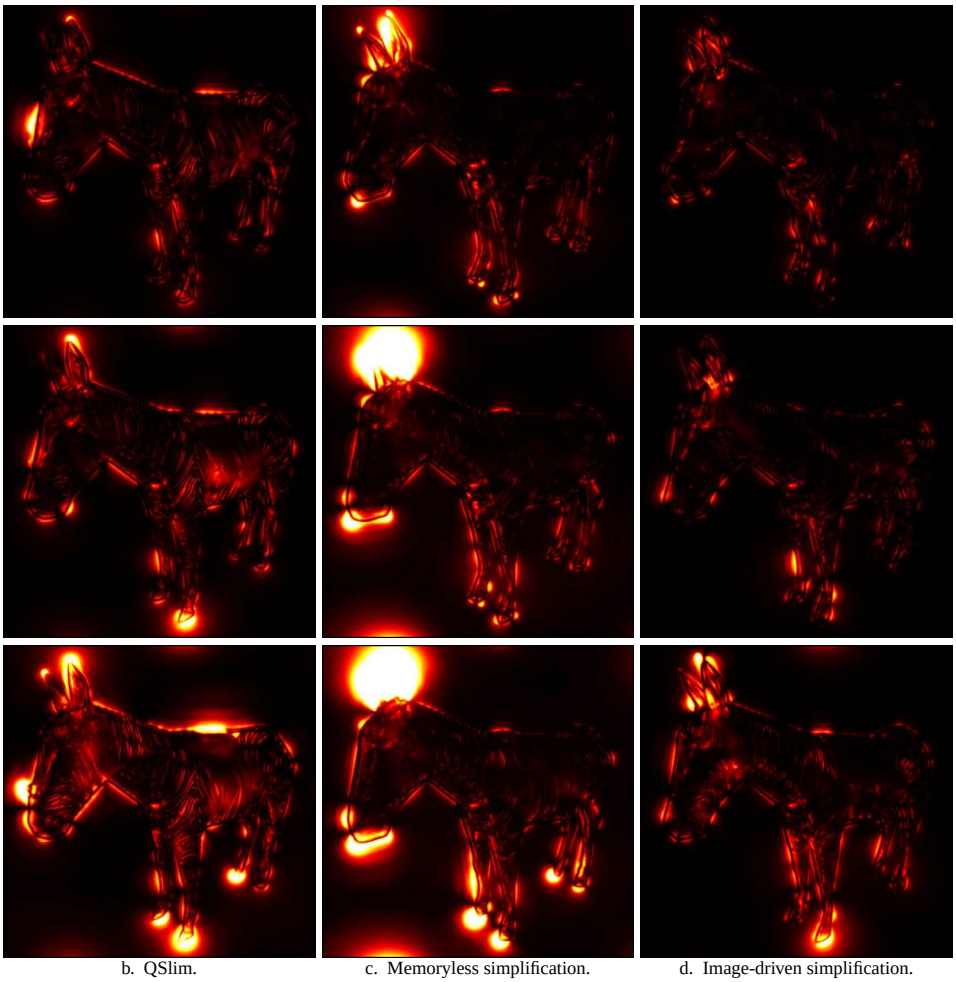
g. QSlim. T = 256. time = 1:03. d_2 = 6.57.

h. Memoryless simplification. T = 256. time = 1:07. d_2 = 4.98.

i. Image-driven simplification. T = 256. time = 2:45. d_2 = 4.51.

Figure 7.15: Zebra model. These images illustrate how the image-driven method excels at very coarse levels of detail.

a. Original model.
T = 58,503.



b. QSlim. c. Memoryless simplification. d. Image-driven simplification.

Figure 7.16: Difference images for zebra model.

difference images produced by my perceptual metric. From these, it is evident that the memoryless method had great difficulties preserving the geometry, whereas QSlim was less successful in preserving the texture parameterization.

In this chapter, I have shown how image-driven simplification generally produces models of higher visual quality than state-of-the-art geometry-driven methods. These results span a large class of models, including simple "organic" models, topologically complex mechanical parts, models rendered using different shading techniques, and several textured models with greatly varying texture content. These results suggest that using images, even if only during the last few percent of the reduction process, can lead to a substantial improvement in quality, and is particularly useful for producing very coarse models. This is an important result because it allows many models to be simplified using an automated method, for which tedious manual simplification and modeling were previously needed. Image-driven simplification by no means replaces such manual tasks, but is a powerful tool that makes producing high quality levels of detail easier for applications that demand visual similarity. We will see in the next chapter how using mesh optimization can take us yet another step closer to producing the best possible model for any given model complexity.

Chapter 8

IMAGE-DRIVEN MESH OPTIMIZATION

8.1 Introduction

The most common approach to mesh simplification is to perform a sequence of local operations that reduce the complexity of the model. I have already presented two such algorithms in Chapters 4 and 7, and discussed several others in Chapter 3. Nearly all such methods use a greedy approach to selecting coarsening operations—the operation that is performed next is the one that will change the model the least according to some quality measure. Even though the greedy order is not guaranteed to be optimal, it is, by definition, the best we can do without the benefit of foresight. Other than choosing the order of operations, many simplification methods also introduce new vertices in the simplified mesh that must be positioned, preferably in a manner that minimizes the error measure. Because of the locality of the coarsening operations, most simplification methods limit this optimization to a single vertex in each step, and generally avoid making changes to the connectivity other than what is necessary to coarsen the mesh. As a consequence, these greedy methods often result in simplified meshes that can be substantially improved by making further changes to the connectivity, vertex positions, and surface attributes (if present) of the model.

Why is the greedy approach to simplification not optimal? A typical greedy simplification algorithm uses an operation—edge collapse, for instance—to reduce the complexity of the model. Usually a priority queue is used to order the potential edges to collapse according to the estimated change in quality that each edge collapse would make. At each step, the edge with the lowest cost is collapsed, and affected neighboring edges are then re-evaluated and re-inserted into the priority queue. Such algorithms essentially create a path $M^0 \rightarrow M^1 \rightarrow \cdots \rightarrow M^n$ through the space $\mathcal{M}$ of all possible meshes, where each new node in the path is a mesh that has one fewer vertex than the preceding node. Previous decisions in the selection of earlier meshes in this path severely restrict the later meshes that can be reached. Consider the analogous problem in 2D of simplifying a single (possibly many-sided) polygon by performing edge collapses. If the original polygon is a detailed approximation of a circle, then the best (in the mean error sense) five-sided simplification is a regular pentagon. A single edge collapse (the greedy step) cannot produce the best four-sided model, which is a square, without moving several vertices at once. This same problem occurs frequently in 3D simplification. For example, by extruding the circle and pentagon to cylinders, we are faced with a similar problem in 3D, where no combination of two edge collapses yields an optimal model. Not only do such greedy algorithms produce suboptimal geometry, but the mesh connectivity can often be improved as well.

In this chapter, I describe a method for improving the appearance of an already simplified model by making a series of changes to its geometry and connectivity. This optimization method, which is also described in [96], is an extension of the image-driven simplification (IDS) method described in the previous chapter. Similar to IDS, the optimization algorithm uses rendered images of the simplified and original, detailed mesh to help guide the changes. As a consequence, the combined effects on visual appearance due to geometry, surface attributes, shading, visibility, and potential artifacts such as cracks and interpenetrations are automatically taken into account in the optimization process. The mesh optimization algorithm differs from IDS, however, in that the mesh is modified so as to directly minimize the image-based error measure. (Recall that IDS uses the geometry-based heuristic from Chapter 4 to position vertices.) In addition, the optimization method does not perform simplification—the vertex count remains the same throughout the process. The main approach of the optimization algorithm is to make edge swaps and *vertex teleports* to alter the mesh connectivity, and to use the *downhill simplex method* to simultaneously improve vertex positions and surface attributes. These steps will be described in more detail in the following sections.

8.2 Optimization Algorithm

The optimization algorithm described in this section was in part inspired by the work of Hoppe et al. [72], who used a nested optimization procedure based on random descent to minimize the geometric deviation between two meshes. My optimization algorithm is also a logical extension to my previous work on image-driven simplification (Chapter 7). By combining optimization with an image-based measure of visual similarity, this approach has the potential to produce approximating meshes of higher visual quality than previous methods. Before describing the algorithm itself, I will first cover some of the preliminaries. I strongly recommended reading §6.1 and §6.2 before this chapter—I will only briefly review the material covered in those sections here. In a sense, Chapter 7 also sets the stage for my image-driven optimization algorithm and, while not essential, I advise reading it as well. Before going any further, let me first discuss what it is that we are trying to optimize. I will then give a brief overview of the algorithm, followed by a bottom-up detailed description of the different components of the method.

8.2.1 Energy Function

In the context of optimization, I will refer to the image-based quality measure d presented in Chapter 6 as the *energy function* E (cf. [72]). That is, E measures the difference between the rendered images $\widehat{\mathcal{Y}}$ of some ideal model $\widehat{M}$ that we wish to reproduce, and images $\mathcal{Y}$ of the current model M being optimized. Whenever a new mesh M' is produced by making an *optimization move*, i.e. moving some of the vertices in M or changing its connectivity, we must conceptually use the image comparison procedure from §6.1, which requires rendering the entire model from multiple viewpoints, capturing the images, and applying the image metric to each image to measure the visual quality of the mesh. As in the case of image-driven simplification, we can accelerate this process by updating the images incrementally and evaluating the image metric over the affected pixels

only, assuming the difference between consecutive meshes is small. Using these techniques, the overall speed of the optimization algorithm was increased by a factor of six for the bunny model in Figure 8.3a.

For efficiency reasons, instead of computing the actual energy E, the *change* in energy $\Delta E = E(M') - E(M)$ can be used to determine whether a move is beneficial or not, i.e. a move improves the mesh quality if $\Delta E < 0$ since a low-energy state is preferred. The argument for computing the change in image quality—instead of the absolute image quality—was given in §6.2.1, and is based on being able to exploit coherence between meshes to reduce computation time. For the hardware described in §8.3, roughly 100 evaluations of ΔE can be made per second using 20 views and 256×256 images.

8.2.2 Overview

The optimization algorithm begins with two input meshes; the original detailed mesh and a simplified version of this mesh that has the desired number of vertices. It is not critical what method is used to create the simplified mesh, and results from several methods will be shown later. The user picks the number of viewpoints to use (from six to twenty-four is typical), and the algorithm creates this number of rendered images of both the original and the simplified mesh. Then, using a method described in detail later, an edge in the simplified mesh is selected for improvement. The algorithm then attempts a number of changes to the mesh at and around this edge to create a mesh whose rendered images are closer to those of the original mesh. Possible changes to the mesh include moving two or more vertices, swapping an edge, or even teleporting a vertex (moving a vertex between entirely different portions of the mesh). Which of these changes are tried is based on how costly each attempt will be relative to the likely improvement each change will yield. When the method is done considering a particular edge, a new edge is selected and the process repeats.

Mesh optimization can be described as a process of searching the space $\mathcal{M}$ of all possible meshes for the mesh that minimizes some given metric subject to a set of constraints. In my algorithm, the goal of optimization is to produce a model with a few number of triangles that is visually similar to a target model with a larger number of triangles. In contrast to mesh *simplification* algorithms such as the one in Chapter 7 and the method by Hoppe et al. [72], which are also driven by this goal, I will assume that an already simplified mesh is provided, which is used as a starting point in my optimization method, and which is to be improved with respect to some measure of visual quality. Given this coarse input mesh, the optimization is constrained by fixing its number of vertices, although the vertices are allowed to move, and changes may be made to the mesh connectivity.

The space of all meshes $\mathcal{M}$ that we seek to explore can be parameterized in terms of the mesh connectivity, geometry, and surface attributes (e.g. colors, normals, and texture coordinates). Formally, a mesh $M = (K, X, S)$ is a triplet consisting of a *simplicial complex* K that defines the connectivity, a set of vertex positions X that define the geometry, and a set of surface attributes S. As before, I will distinguish between the topological entity $v \in V$ and the corresponding geometric realization $\mathbf{x}^v \in X \subset \mathbb{R}^3$ of a vertex. Each attribute is bound to a vertex v, a triangle t, or a *corner* (v, t) formed by v and one of its incident triangles t. While the geometry and surface attributes considered here are continuous parameters, the mesh connectivity is discrete. To optimize both, I will take an approach similar to that of Hoppe et al. [72] by using a two-level

nested optimization: an inner, continuous optimization in which vertices and surface attributes are modified while fixing the connectivity, and an outer, discrete optimization in which simple, atomic changes to the connectivity are made. The general approach in my method is to select a set of edges in the mesh to improve, as suggested by an *oracle*, interleaved with a sequence of randomly chosen edges. This oracle (described in detail later) identifies edges that may be the cause of large differences between the images of the original and current mesh. For each chosen edge, a sequence of connectivity moves of varying complexity are attempted, and a small set of vertices in the neighborhood of the edge are optimized until the connectivity move results in a decrease in the energy function.

In addition to the use of an oracle to guide the optimization, versus random descent, my optimization method differs from Hoppe et al.'s [72] in several ways. First, whereas their method is used both to simplify and optimize a mesh, my technique assumes that an independent simplification step has already been performed, after which the complexity of the mesh is never changed. Second, my optimization method is not guided by a geometric measure of distances, but rather by image differences. By using an image metric to guide optimization, all of the relevant factors that make up the appearance of a mesh can be captured without explicitly creating an energy term for each one. Therefore, the difficult issue of how to balance such factors as geometric distance, color, and texture against one another is avoided. Third, the method used to optimize vertex positions is entirely different from the conjugate gradient approach used by Hoppe et al. Finally, the selection of which operation to perform on an edge is not random in my algorithm, but is decided based on which operation would result in a non-negligible improvement to the mesh.

Let me at this point stop for a moment and say a few words about my optimization method. As will become evident, there are quite a number of design choices that had to be made in developing this algorithm, including settings for many of the parameters in the algorithm. Some of these components of the algorithm may seem ad hoc, and there may sometimes seem to be little basis for the particular set of choices made here. I confess that this is true to some extent—many of these choices were made by simple trial and error, accepting those changes and additions that worked well for a number of test models. While this may discourage some readers, let me point out that this is more often than not the case in global optimization methods, and particularly in those cases that involve both combinatorial and continuous optimization. Because the search space is so vast, and because there is no clear path to the optimum, making intelligent choices about how to get there as quickly as possible is often very difficult. It is therefore not surprising that many optimization methods rely completely on making a set of random moves and hoping that a better configuration is found (cf. [72]). In my algorithm, I have tried to develop the best heuristics possible for making beneficial changes to the mesh as quickly as possible, while trying to incorporate statistics to improve the odds of doing better than chance. Demonstrably, these strategies have worked in the majority of cases. Nevertheless, I am certain that there is still room for improvement, and that significant performance gains can be attained simply by tuning some of the parameters of the algorithm.

In the remainder of this section, I will first describe the low level details of the continuous and discrete optimization, and then conclude by discussing the strategy for choosing connectivity moves and the set of edges to optimize.

8.2.3 Continuous Optimization of Mesh Geometry

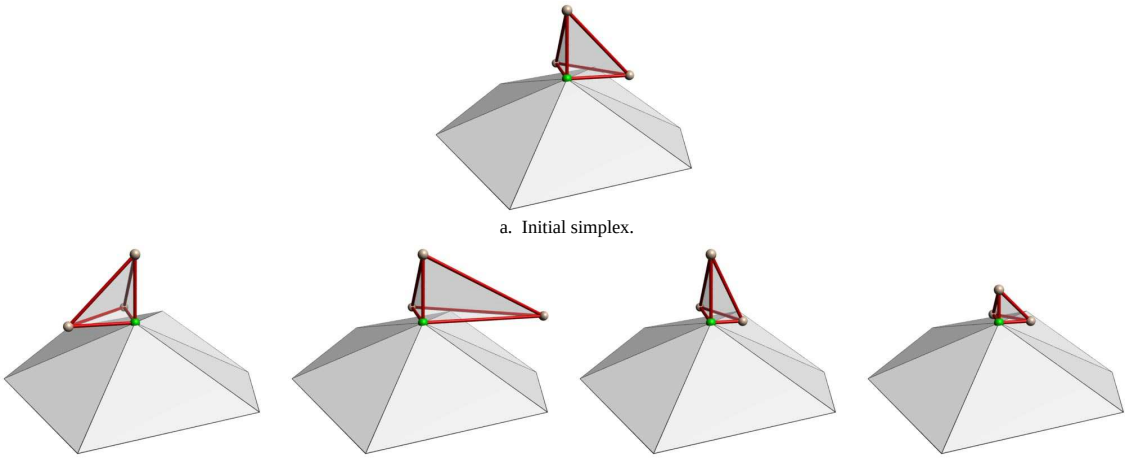
In this section I will explain how to optimize the geometry of a small portion of a mesh. The goal of this optimization is to improve the visual appearance of the mesh by making a series of small adjustments to the vertex positions, such as lengthening a protrusion, smoothing out undesired wrinkles and bumps on the mesh, enhancing creases and other fine details, etc. Specifically, given a mesh with a fixed connectivity and a subset V of its vertices, the vertex positions X^V are moved simultaneously until a local optimum in the visual quality of the mesh is found. We can generalize this procedure to include (continuous) surface attributes simply by concatenating vertex positions and attributes to form a single higher dimensional parameter vector. For simplicity, however, I will restrict this discussion to vertex positions only.

Multidimensional methods for continuous optimization problems fall into one of two categories: methods that make use of derivative information of the objective function in order to make an educated guess about where, or at least in what direction, the minimum lies, and slower methods that rely on function evaluations only. Unlike in [72], where the energy function is a closed form quadratic expression, the energy function E is in our case given by discrete image differences that depend non-trivially (although generally smoothly) on the input parameters (the vertex positions and attributes). Therefore, we must use an optimization procedure that relies only on sampling the energy function itself. I have chosen to use the *downhill simplex method* for this task, because it is easy to implement and generally requires only a small number of function evaluations before converging on a minimum [120]. This method takes as input $n + 1$ vectors that specify the vertices of an n-simplex, evaluates the function at these vertices, and proceeds by making a sequence of moves, such as reflections, contractions, and expansions, which are chosen based on the current function values at the vertices of the simplex (Figure 8.1). The energy function is then evaluated whenever a vertex in the simplex is moved. Near a local minimum, the simplex contracts until the function values at its vertices become sufficiently close to one another. Thus, by tracking the hyper-volume of the simplex, which always expands or contracts by a power of two, we can estimate when a minimum has been found.

To apply the downhill simplex method to the mesh optimization problem, a basis is first constructed for the set of m mesh vertices V, with positions $X^V = \{\mathbf{x}^v\}_{v \in V} = \{\mathbf{x}_j\}_{j=1}^m$, that are to be optimized. Even though these vertices need not be related geometrically or topologically whatsoever, I will assume that they are confined to a small neighborhood in the mesh. For example, if $m = 4$ (a number of vertices frequently optimized at a time in my algorithm), then $n = 3m = 12$ linearly independent 12-dimensional vectors (the xyz-coordinates for the four vertices) need to be produced. The current vertex positions X^V in M collectively make up an n-vector

$$\mathbf{p}_0 = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix}$$

for one of the vertices in the initial simplex. The remaining n vertices $\{\mathbf{p}_i\}_{i=1}^n$ of the simplex can be computed by choosing the unit coordinate axes in $\mathbb{R}^n$ as a basis, and displacing these vertices a small distance δ from



a. Initial simplex.

b. Reflection. c. Expansion. d. Single contraction. e. Multiple contraction.

Figure 8.1: Illustration of operations used in the downhill simplex method. This figure shows the case of $n = 3$ dimensions, for which the simplex is a tetrahedron (here shown in red). A part of the mesh is shown, for which the position of one of its vertices (shown in green) is being optimized. The other three vertices of the tetrahedron are possible candidate positions for the mesh vertex. The four examples in the bottom row illustrate the result of applying the set of possible moves to the simplex in 8.1a. Each time one of these operations is performed, the energy function is probed at the new vertex or set of vertices, and the lowest valued vertex becomes the pivot vertex for subsequent moves.

$\mathbf{p}_0$ along each corresponding basis vector, i.e. $\mathbf{p}_i = \mathbf{p}_0 + \delta\hat{\mathbf{e}}_i, 1 \leq i \leq n$. Each such $\mathbf{p}_i$ naturally constitutes an initial estimate of the location of the optimal X^V, and it is important that these estimates are reasonably close to the expected minimum for fast convergence. Consequently, the magnitude of the displacement δ is based on the local geometry around the vertex set V. In my current implementation, δ is set on a per vertex basis to the average length of the vertex's incident edges. One might suspect that using local coordinate frames derived from the geometry of the mesh (as opposed to using the arbitrary canonical basis in $\mathbb{R}^n$) would produce better and less biased offsets. However, I have not found this to be true in practice.

Once the initial simplex has been formed, the continuous optimization of X^V is performed by making repeated evaluations of the energy function, until the process converges or a predefined limit on the number of evaluations is exceeded. In the current implementation, a limit of $32n$ evaluations is imposed to avoid spending too much effort on one small region of the mesh.

So far I have not discussed how to choose the set of vertices V to optimize, as this decision is tightly linked to the outer, discrete optimization, which will be discussed in the following subsection.

8.2.4 Discrete Optimization of Mesh Connectivity

Most simplified meshes can be improved greatly by optimizing the positions of their vertices alone. After a while, however, a point of diminishing returns will be reached as changes to the connectivity are needed to further improve the mesh. This is generally required for one of two reasons: either the local mesh connectivity

is not appropriate for its given geometry, which can be handled by making one or more *edge swaps*; or the mesh tessellation is too fine or too coarse in relation to the geometric complexity, which is addressed by transferring vertices from one area to another using a *vertex teleport* operation. These two connectivity moves are described in the remainder of this section.

To explore the entire space of all meshes, we need a way of generating all possible mesh connectivities K for a given set of vertices V. While the number of meshes with a fixed number of vertices is finite, the vast majority of these meshes are not useful to us. Rather than generating the complexes from scratch, this type of combinatorial optimization is often done by making incremental changes to a good initial estimate of the optimal connectivity. Most simplification methods produce mesh connectivities that are at least reasonably good starting points. For manifold meshes of fixed topological type, it can be shown that the *edge swap* operator (Figure 2.2) is sufficient to produce any desired (manifold) connectivity. While this operation is useful for making local changes to the connectivity, it is not practical for distributing vertices over the mesh, as a long chain of edge swaps in conjunction with geometry optimization might be required to transfer a single vertex from one area to another. Instead, vertices are transferred using a more global and atomic operation. In essence, two atomic operations are needed to keep the vertex count fixed; one for vertex removal, and one for adding a vertex to the mesh. To remove a single vertex, I use *edge collapse*, while *edge split* is used to introduce a vertex (Figure 2.2). These two operations, when used together, make up the vertex teleport operation. I chose to use edge split instead of *vertex split*—the dual of edge collapse—for two reasons: The edge split results both in a uniquely defined connectivity and a unique position for the new vertex (assuming the edge is split at its midpoint), whereas vertex split requires not only the specification of which edges to "pull apart" but also how to assign coordinates and surface attributes to the new vertex. Second, by using edge split, the discrete optimization can be treated as a sequence of improvements made to the edges of the mesh via a small set of well-defined atomic operations.

Recall that the discrete optimization is wrapped around an inner continuous optimization. Whenever a connectivity move is made, the geometry of the nearby vertices is optimized, and the move is accepted if it leads to a decrease in the energy function. I will discuss how to choose what moves to make on what edges in the following sections, and focus the remainder of this section on providing the final details of how to perform each move.

Since the initial connectivity might be far from optimal, one should avoid expending too much effort optimizing the geometry during the early stages. Therefore, I have chosen to associate multiple levels of geometry optimization with each connectivity move, ranging from simple vertex placement heuristics to optimizing successively larger sets of vertices. The idea is to allow an efficient but less accurate optimization strategy as long as the mesh quality can be improved, and to employ higher degrees of optimization to fine-tune the mesh near an optimum. Since the expected number of function evaluations is roughly linear in the number of vertices to be optimized, small sets of vertices are favored for optimization, which can then be expanded whenever insufficient progress is made. Table 8.1 contains the vertex set optimized for each connectivity move. In addition to the edge swap, split, and collapse operations, a "no-op" move is included that corresponds to optimizing the local geometry without making any changes to the connectivity.

connectivity move	*optimization level*			
	0	1	2	3
no-op		$\lfloor e \rfloor$	$\lfloor \lfloor \lceil e \rceil \rfloor \rfloor$	$\lfloor \lceil \lfloor e \rfloor \rceil \rfloor$
swap		$\lfloor e' \rfloor$	$\lfloor \lfloor \lceil e' \rceil \rfloor \rfloor$	$\lfloor \lceil \lfloor e' \rfloor \rceil \rfloor$
split		v	$\lfloor \lceil v \rceil \rfloor$	$v \cup \lfloor \lceil \lfloor e \rfloor \rceil \rfloor$
collapse	v	v	$v \cup \lfloor \lfloor \lceil e \rceil \rfloor \rfloor \setminus \lfloor e \rfloor$	$\lfloor \lceil v \rceil \rfloor$

Table 8.1: Vertex sets optimized for each connectivity move and optimization level. Examples of e, e', and v are shown in Figure 2.2. For the edge split and collapse operations, v is initially placed at the edge midpoint. Geometry optimization is performed on levels 1–3, but not on level 0, e.g. the position of v is optimized on level 1 in the edge collapse operation, but remains at the edge midpoint on level 0.

While the set of connectivity operations used in my algorithm allows a large space of meshes to be explored, it is not complete, i.e. it is not possible to construct *all* meshes with a fixed number of vertices from a given initial mesh. For example, there is no way to merge two disjoint components or to change the genus of the mesh. Nor is it possible to open a hole in the mesh or to split a boundary loop in two. In order to keep the algorithm simple, the optimization method relies on starting from a good initial model that has the appropriate *topology*, but not necessarily the optimal *connectivity*.

8.2.5 Choosing Connectivity Moves

After developing the necessary tools for locally optimizing the geometry and connectivity of the mesh, I will now turn my attention to the issue of how to intelligently choose what parts of the mesh to optimize and what operations to use in order to most efficiently reduce the mesh energy. In this section, I assume that we are given an edge e to optimize, but are left with the decision as to what connectivity moves to attempt and what vertices around e to optimize. Because we can expect some moves to be more efficient at lowering the energy than others, but also require a healthy mix of moves, we would like to balance the use of different moves so as to reduce the energy as quickly as possible. Below I will present a statistical argument for making the decision when to attempt a move.

As mentioned in the previous section, there are multiple, nested sets of vertices to optimize associated with each connectivity move. For flexibility, I will treat the different optimization levels with each move as independent operations. (There are thus 13 possible moves.) By keeping a history of the performance of each operation, we can choose whether to attempt a move on an edge based on its efficiency, i.e. the expected reduction in energy per time unit. For each connectivity move and optimization level, statistics are maintained for the average change in energy ΔE and its standard deviation σ, the average completion time[1] t, as well as the frequency of utilization f. These averages are all computed using a nonuniform weighting function that exponentially attenuates the statistics over time. Adopting the conventions used in statistics, I will use $x = -\Delta E$ below to denote the random variable associated with the energy reduction, and μ to denote its (weighted) average.

[1]The time is measured in number of energy evaluations instead of seconds to ensure that the optimization is deterministic and reproducible.

Following the goal to make quick downhill moves in the energy function whenever possible, the algorithm generally attempts only the most efficient connectivity move on an edge. However, in order to keep the statistics up to date, less efficient operations must sometimes be attempted. This choice is balanced using statistical probabilities by computing a confidence interval for the expected energy reduction. Let the variables (μ^*, t^*, f^*) refer to the currently most efficient operation O^*, i.e. the one with the largest ratio $\frac{\mu}{t}$. For large enough samples, we can assume that x follows a normal distribution. We can estimate the odds that an operation O is more efficient than the currently best operation O^* as follows. First, find the n that satisfies

$$\frac{\mu + n\sigma}{t} = \frac{\mu^*}{t^*}$$

The probability $P(\frac{x}{t} > \frac{\mu^*}{t^*})$ that O is more efficient than O^* is then $P = 1 - \operatorname{erf}(\frac{n}{\sqrt{2}})$, where erf is the error function

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

This follows from the assumption that the distribution of x is normal. Given this probability P of success, i.e. of outperforming the best known move in efficiency, when should O be attempted? A reasonable choice is to use O in proportion to its probability of success. Thus, when P is small, O is used less often, and vice versa. More precisely, O is attempted if its current relative frequency of utilization is lower than the probability of success, i.e. if $\frac{f}{f+f^*} < P$.

By consolidating the equations above, we can rewrite the condition for attempting O as a single inequality:

$$\mu > \max\left\{ \frac{t}{t^*}\mu^* - \sqrt{2}\operatorname{erf}^{-1}\left(\frac{f^*}{f+f^*}\right)\sigma,\ x_{min} \right\} \tag{8.1}$$

where x_{min} is the total change in energy accumulated from previous operations, which is reset to zero each time an edge is optimized. This term is included to avoid attempting a new, possibly expensive move when significant progress has already been made optimizing the current edge.

By using a probabilistic algorithm to determine if a move should be performed, several moves per edge may be attempted in addition to O^*. I have chosen to use a predetermined order of operations, and optimization levels are ordered from lowest to highest within each move. The simplest move—the no-op—is attempted first. For each of its three optimization levels, Inequality 8.1 is evaluated, and the corresponding operation is attempted if the condition is satisfied. If insufficient progress is made (or if none of these operations is efficient enough to attempt), we may conclude that the geometry is locally optimal for the given connectivity, but we must allow for the possibility that the connectivity is not optimal with respect to the geometry of the target model. Consequently, the next cheaper move is attempted; the edge swap. Note that this move is only defined if e is manifold and does not form a surface attribute boundary (e.g. between two textures). If the edge swap optimization does not significantly reduce the energy either, we need to investigate whether the surface is locally undersampled by attempting a vertex teleport.

The vertex teleport operation begins by splitting e via insertion of a vertex v at its midpoint. In order for the edge split to be accepted, it must lower the energy enough to offset the expected increase in energy

associated with the "cheapest" edge collapse. While the exact value for the lowest collapse energy is not always known ahead of time, it can be estimated using the lowest energy known when the previous edge collapse was completed. This energy is attenuated over time to ensure that one bad estimate does not entirely inhibit future teleport attempts. If the edge split does not meet this energy constraint, the operation is undone and the next optimization level is attempted. Otherwise, we must find an edge to collapse commensurate with the decrease in energy provided by splitting e.

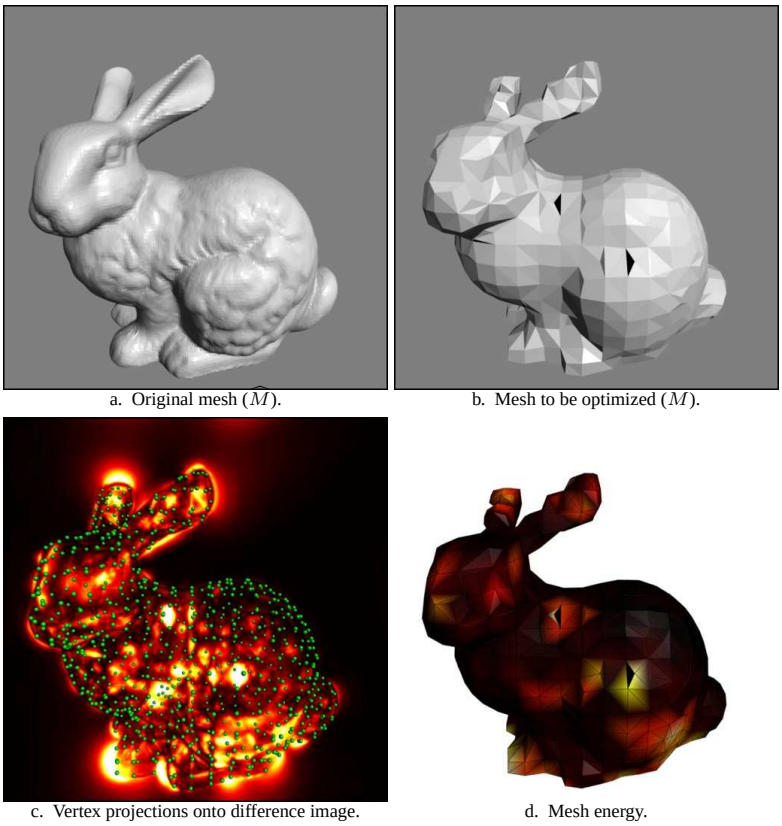
Similar to several simplification algorithms, my optimization method maintains a priority queue of edges, sorted by estimates of the edge collapse energies. As with other operations, an optimization level l is associated with each estimate. Initially, each collapse candidate is set to a default state of zero energy and an optimization level of negative one. After a set of vertices V is optimized, the state of the edges $\lfloor\lceil\lceil V\rceil\rceil\rfloor$ is reset to this default state, likely placing the edges at the front of the queue, which is an indirect request that their energy estimates be updated since they are likely to have changed. When an edge collapse is requested, the lowest energy edge is dequeued. If its estimated energy is lower than the threshold given by the previous edge split, the estimate is verified by collapsing and optimizing the edge at its given optimization level. If, on the other hand, the threshold is exceeded, the optimization level is incremented and a (hopefully) lower collapse energy is obtained. If the edge collapse is still not acceptable, the edge is either reinserted into the queue, if the optimization level is lower than the maximum, or is placed in a temporary list as its energy cannot be lowered, allowing other edge collapses to be considered, and the procedure is repeated.

In each iteration of this search for a valid edge collapse, the edge with the lowest collapse energy and whose optimization level has not reached the maximum is dequeued. Due to this search order, from lowest to highest collapse energy, the likelihood of finding a valid edge collapse decreases rapidly over time, and the search is terminated if the probability of success is lower than 0.3%, i.e. using a 3σ confidence interval. This often preempts a futile search after a few seconds, which might otherwise take a long time to complete.

If an edge is found whose collapse energy is lower than the threshold, the teleport operation completes successfully. Otherwise, the most likely conclusion is that there is no edge collapse compatible with the previous edge split. Instead of undoing the split, however, the method proceeds by collapsing the cheapest edge. While this will result in an energy increase, it is a rare occurrence, but not necessarily a bad one as it allows for occasional uphill moves that may get the optimization out of local minima. Also note that since the edges created in the previous edge split are candidates for collapse, we should always in theory be able to collapse one of these edges to revert back to the mesh as it was before the edge split.

8.2.6 Choosing Edges to Optimize

The outermost loop in the optimization method consists of choosing a set of edges to optimize. Quite naturally, some edges are better candidates than others, yet it is not immediately obvious how to rank them to maximize the reduction in energy. We can, however, order the edges by their *potential* for improvement by making use of difference images. That is, for a given choice of image metric and associated difference images, an *oracle* determines which areas of the mesh are high in energy, and which have a potential for large improvement. I have found this oracle to be useful for detecting artifacts in the mesh that can quickly be



a. Original mesh ($\widehat{M}$). b. Mesh to be optimized (M).

c. Vertex projections onto difference image. d. Mesh energy.

Figure 8.2: The edge selection oracle. Blurred difference images between the two models are computed periodically for all views. The vertices of the coarse mesh (shown in green in 8.2c) are then projected onto each difference image, and energy is collected for these vertices, resulting in the color-coded mesh in 8.2d. The vertex energies are then distributed to the edges of the model, which are ranked by decreasing energy. This allows the oracle to suggest which edges have the highest potential to reduce the total mesh energy.

improved, which is an advantage over methods like [72] that rely solely on random descent.

Periodically, the potential energy for each edge is computed by projecting its vertices onto the screen and summing up the pixel differences from blurred versions of the difference images, similar to [147] (see Figure 8.2). The blurring is performed efficiently by using a separable 5×5 Gaussian convolution kernel and the down and upsampling operations from [17], which results in an effective kernel size of 9×9. The edges are then sorted by their potential energy, and the oracle recommends a small set of the highest energy edges for optimization. The difference images are also used to measure the overall mesh energy, which is useful for monitoring the progress of the optimization. The user can then terminate the optimization when a satisfactory energy level has been reached.

As alluded to above, the oracle does not always produce edges that can be improved greatly, and sometimes outputs roughly the same set of edges twice in a row. For this reason, the set of edges suggested by the oracle is interleaved with a batch of randomly chosen edges. At the beginning of each iteration, in which

a total of 64 edges is optimized, these two sets are balanced based on the amount of progress made in the previous iteration. The resulting optimization procedure is very flexible and adjusts quickly to changes in the mesh that are either beneficial or detrimental.

8.3 Results

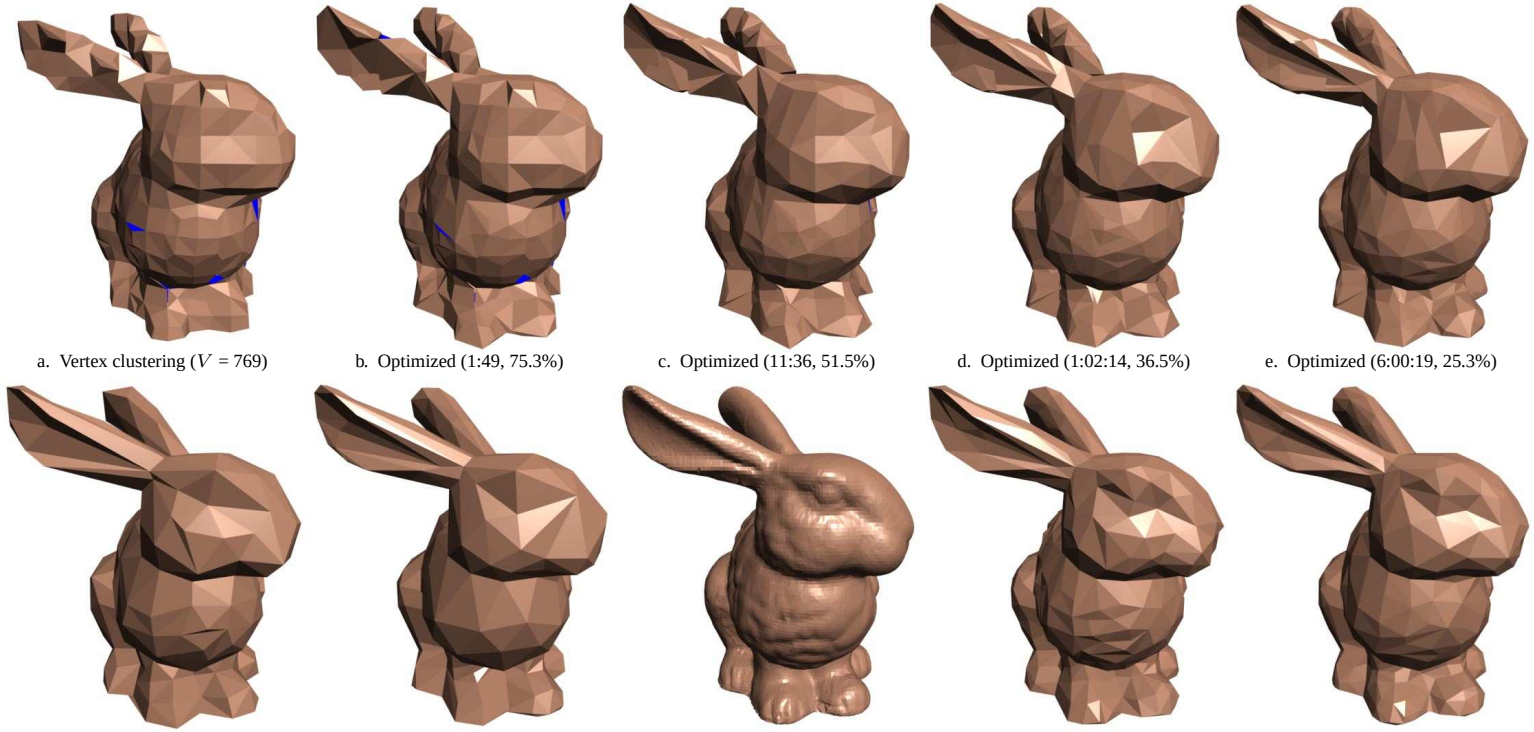
The models discussed in this section were optimized on a one-processor 250 MHz R10000 Silicon Graphics Octane with Maximum Impact graphics and 256 MB of RAM. I include examples of models that were optimized between a few minutes and up to six hours using the mean square metric d_2^2. Other parameters to the optimization algorithm, such as number of views and image dimensions, were the same as for the image-driven simplification algorithm (see §7.3).

The first example of mesh optimization is for a bunny model that has been simplified using a variant of Rossignac and Borrel's vertex clustering method [127], for which all double-sided faces that were introduced during simplification have been removed, thus creating a few holes in the mesh. Figure 8.3a shows the cluster-simplified model, and Figures 8.3b through 8.3e show successively improved models using the image-guided optimization. In addition to smoothing out the rough surface, the optimization is able to close holes in the mesh through properly chosen edge collapses. The percentages in the captions correspond to the mesh energies relative to the input model in Figure 8.3a.

My mesh optimization method is also able to improve models that were simplified using higher quality simplification methods. Figures 8.3f and 8.3i show two models that were produced by memoryless simplification. Figures 8.3g and 8.3j show the results after optimization. Notice that the shapes of the ears are better captured by the optimized meshes, and the dark sliver triangles created by the memoryless method are gone. Figure 8.3h shows the original bunny model for comparison.

Figure 8.4 shows the mesh energy as a function of time for three different versions of the Stanford bunny model, which are shown in Figures 7.1c, 8.3a, and 8.3i. These graphs are proof that the optimization algorithm indeed has the capability of greatly reducing the image-based energy measure. Not only did it yield a 25% improvement over image-driven simplification, which already produces models of very high visual quality, but also reduced the energy for the models simplified using the two geometry-based methods below the energy of the image-driven simplified model. After a rapid initial reduction in energy, the curves eventually level off as the optimum is approached. Ideally, the curves would converge as time approaches infinity. Figure 8.4 shows that, while they all do approach a lowest level of energy, they may settle at slightly different levels. This may be due to the optimization getting stuck in local minima, and that the operations used in the optimization may be too localized to allow breaking out of these minima.

Figure 8.5 shows the (final) mesh energies for several levels of detail of the bunny. These energies were computed using the 24 views described in §6.1. Notice that the optimization method always outperforms the image-driven simplification method using this quality measure, regardless of which model is used as input to the optimization. In fact, I have found that using the memoryless method followed by optimization takes less time than using image-driven simplification to reach the same energy level. However, the best



a. Vertex clustering (V = 769) b. Optimized (1:49, 75.3%) c. Optimized (11:36, 51.5%) d. Optimized (1:02:14, 36.5%) e. Optimized (6:00:19, 25.3%)

f. Memoryless simplification (V = 341) g. Optimized (E = 59.7%) h. Original model (V = 34,834) i. Memoryless simplification (V = 693) j. Optimized (E = 58.8%)

Figure 8.3: Optimized bunnies. The captions indicate the number of vertices (V), the energy in relation to the input model (E), and the time spent optimizing (hours:minutes:seconds).

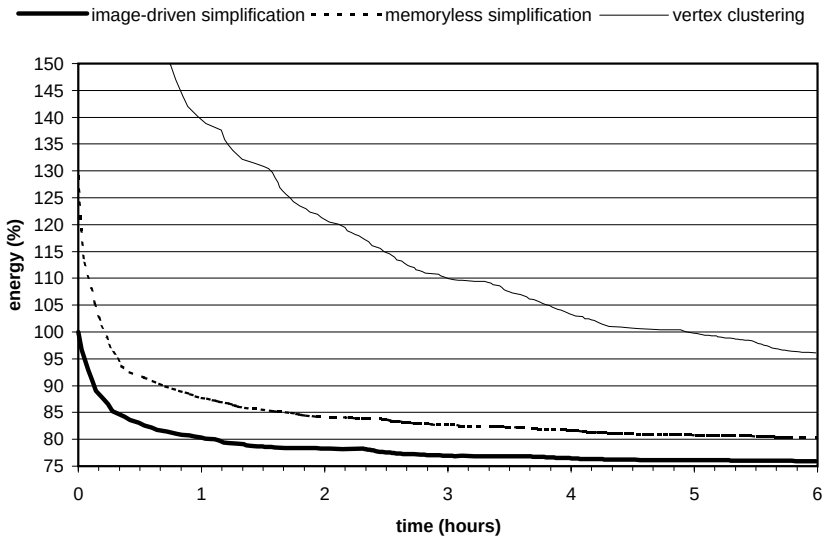


Figure 8.4: Mesh energy as a function of time for various bunny models. Each curve corresponds to a different initial model, produced by the image-driven (693 vertices), memoryless (686 vertices), and vertex clustering [127] (769 vertices) simplification algorithms. The energy, which is based on mean square image differences, is here expressed as a percentage relative to the model produced by the image-driven simplification method, and starts at 380% for vertex clustering.

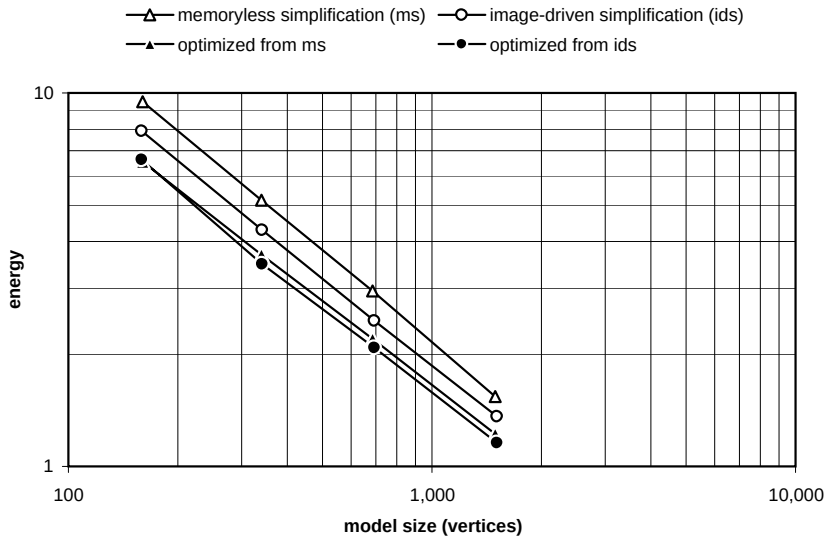
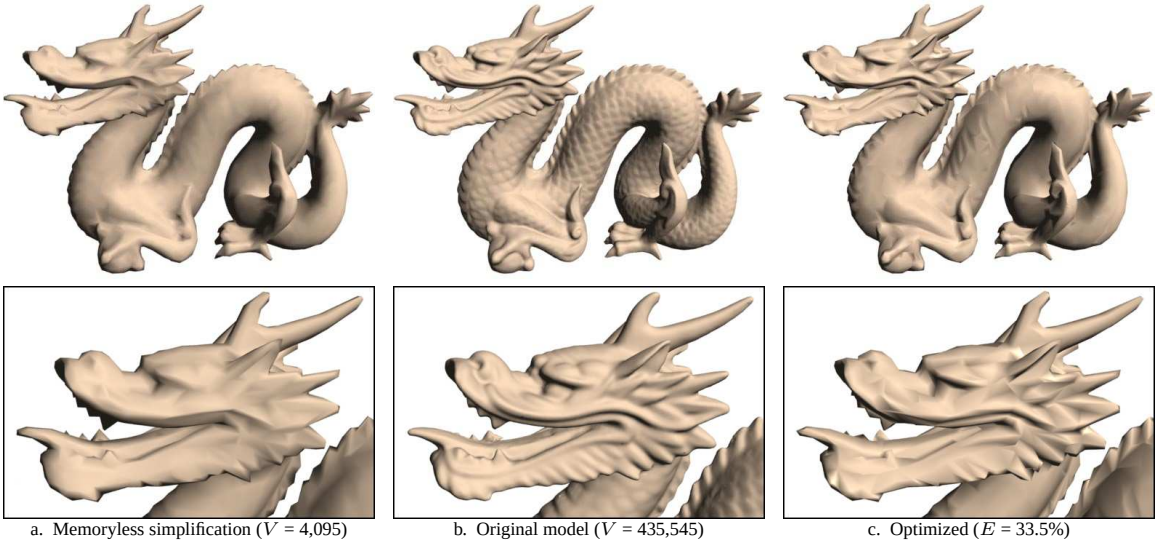


Figure 8.5: Mesh energy for bunny models at different levels of detail. The lower two curves correspond to the final, optimized models, whereas the upper curves correspond to the models before optimization.



a. Memoryless simplification (V = 4,095) b. Original model (V = 435,545) c. Optimized (E = 33.5%)

Figure 8.6: Gouraud shaded dragon model. Both geometry and vertex normals were optimized.

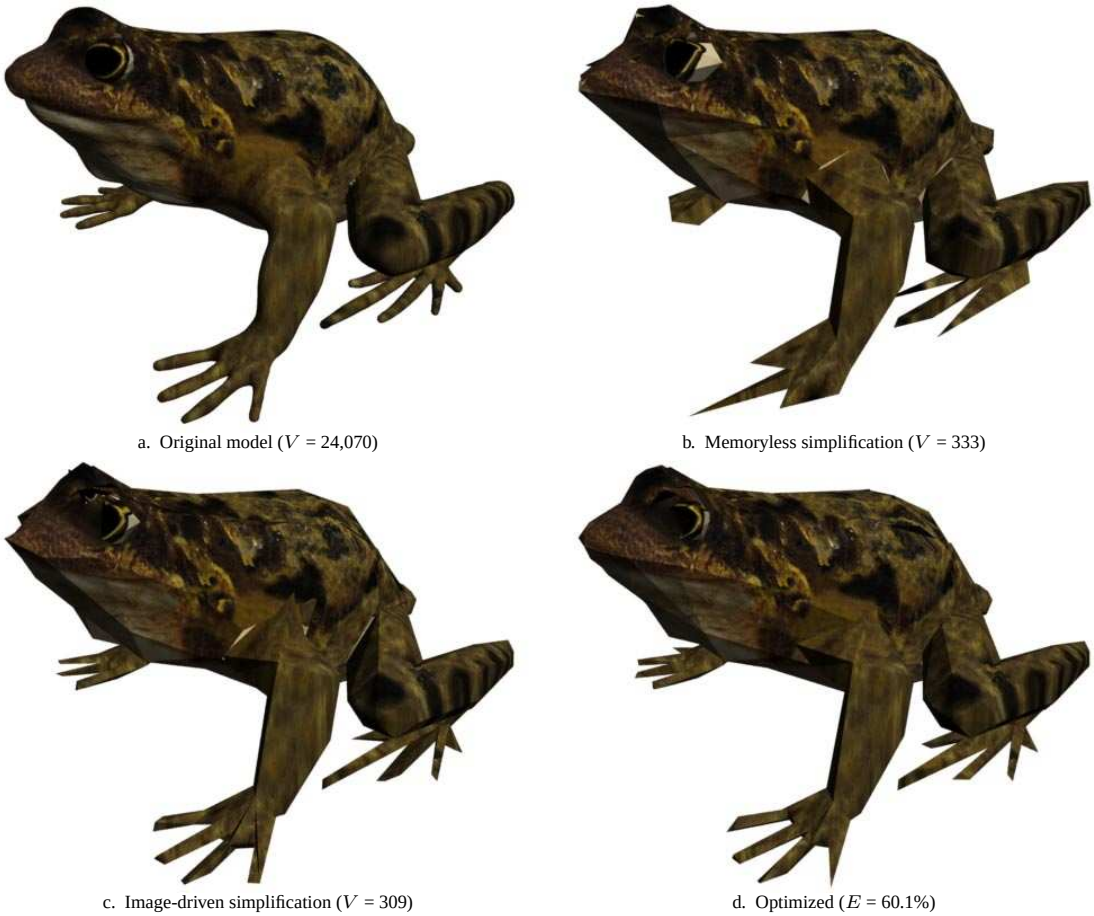
a. Memoryless simplification (V = 580) b. Original model (V = 33,173) c. Optimized (E = 51.0%)

Figure 8.7: Textured torus model.

meshes are obtained when the two image-driven methods are used together. Note that my image-driven optimization method has the potential to degrade the geometric accuracy of a simplified model. If both visual and geometric quality are desired, then image-driven simplification alone should be used. For these reasons, the image-driven techniques are both valuable, whether used together or not.

To demonstrate how the quality of a Gouraud shaded mesh can be improved by making changes to its normals, I used the dragon model in Figure 8.6b. A memoryless simplified version (8.6a) was improved by optimizing both geometry and vertex normals (8.6c). By not constraining the normals to unit length, the algorithm was sometimes able to artificially darken or brighten regions without changing the surface normal direction. As a result, details near the head, legs, and chest have been recovered in the optimized model.

Figure 8.7 illustrates how the texture of a model can be improved. This torus-like object was created by



a. Original model (V = 24,070)

b. Memoryless simplification (V = 333)

c. Image-driven simplification (V = 309)

d. Optimized (E = 60.1%)

Figure 8.8: Optimized frog model. Notice the improvement in both geometry and texture after optimization.

spinning and sweeping an ellipse around a circle, and has a highly non-linear texture parameterization. The texture image consists of several black numbers on a white background, with a number of colored squares and lines. Distortion to such simple geometric shapes should easily be recognized. Shown in this figure is a memoryless simplification, the original, and an optimized version of the torus for which both geometry and texture coordinates were modified. Notice that the large black zeros are better placed after optimization, and that the long curved lines are better matched.

The final example is a textured frog model, shown in Figure 8.8a. This is the same model as the one shown in Figure 7.14a, and is composed of several topologically disconnected body parts. Figure 8.8b shows the frog model simplified by the memoryless algorithm, and several problems are evident. The front right foot has been entirely eliminated, and the remaining feet are also poorly preserved. One striking artifact is the distorted eye shape, caused by severe texture stretch and interpenetrating geometry. There is also cracking evident between the legs and the body. Image-driven simplification produces a considerably better looking model, shown in Figure 8.8c. However, several artifacts remain, mainly because this method does not

optimize the vertex positions with respect to the image metric. The image-driven optimization method, on the other hand, not only recognizes all these problems, but successfully "repairs" the damaged parts. Figure 8.8d shows the result of optimizing the vertex and texture coordinates of the model from Figure 8.8c. Optimization has fixed the cracks and improved both geometry and texture, resulting in a model of substantially higher visual quality.

The results above show how my image-driven optimization method is able to improve the visual quality of a variety of models, including models with surface properties such as normals and texture. Because the method uses images to directly measure visual similarity, it accounts for and repairs visually distracting artifacts, such as cracks in the mesh, which are common in drastically simplified models produced by geometry-based methods. Indeed, my optimization method is very well suited for improving such coarse models, and provides an automatic approach for this task that otherwise may require tedious editing by hand. Of particular value is the fact that my optimization technique is not constrained by what simplification method was used to produce the input model. As shown above, my optimization method is able to improve the visual quality of the input model even when supplied with very low quality simplifications. Consequently, efficient simplification schemes such as my out-of-core and memoryless methods can be used to rapidly simplify very complex models to a few hundred triangles, after which optimization is used to fine-tune the mesh, resulting in a higher quality approximation of the same complexity.

Chapter 9

CONCLUSIONS

9.1 Summary

I have presented three new methods for automatic simplification of polygonal models, and a global optimization method for improving the appearance of a simplified mesh. These methods are targeted at models of varying complexity, from meshes with billions of faces, down to models with as few as a hundred triangles. Because computational speed must be traded for simplification quality, my methods have been designed to cover different ranges of the model complexity spectrum, allowing very fast reduction of extremely complex models, while resorting to slower but higher visual quality optimization of less densely sampled surfaces. Collectively, the methods complement each other well, and should cover the different needs of most users of simplification.

Based in part on the quadric error metric introduced in [50], I proposed two fast and memory efficient simplification algorithms. One of these, which I call "memoryless simplification," uses edge collapse to simplify a model, but differs from most previous edge collapse methods by measuring errors in terms of the amount of geometric *change* made by an edge collapse relative to the partially simplified model. This obviates the need for maintaining memory-costly information about the original, dense model. I showed that this method has a very high quality versus time complexity ratio, and results in models with lower mean geometric error than several other well-known simplification methods. My other quadric-based simplification method uses vertex clustering to decimate a model of arbitrary complexity in a single pass. This output sensitive out-of-core method for simplification is the first of its kind, and compares favorably in quality against previous in-core clustering schemes.

An important theme in this thesis is the notion of *visual similarity*. Two of the methods described here—one for simplification and one for optimization—make use of rendered images of a partially simplified model taken from a number of different viewpoints. The difference between these images and those of a target model are measured using an *image metric*. This is a significant departure from mainstream work on simplification, which so far has focused on developing metrics for accurately measuring the geometric deviation between the two surfaces. Using image metrics to drive simplification and optimization, I have shown that models of substantially higher visual quality can be obtained. In particular, the image metric allows the complex interactions between geometry, surface properties, rendering styles, and visibility to be directly accounted for using a single, unified error measure. Whereas previous simplification methods attempt to preserve the

texture coordinates of a model, my methods can additionally account for the texture image used, allowing more aggressive simplification in regions of nearly uniform color. Yet another benefit of using an image metric is that geometric degeneracies, such as cracks and interpenetrations in the mesh, can be detected and avoided. For maximum speed, both of these image-driven methods make use of commonly available hardware graphics rendering to update the images.

For edge collapse based simplification, I presented a new method for preserving texture coordinates and other surface attributes. This technique is independent of the method used for placing the new vertex after an edge collapse, and has proven useful both in my geometry- and image-driven simplification algorithms. Even without an image metric to guide the simplification, this algorithm for computing surface attributes compares favorably against other methods.

Finally, I proposed a new image metric that is based on visual perception, and that accounts for phenomena such as visual masking and luminance-dependent contrast sensitivity. This metric is significantly more computationally efficient than several other perceptual metrics, and can therefore be put to use directly in simplification. Using my perceptual metric, it is possible to exploit the limitations of human vision by allowing more aggressive simplification to be performed in regions that are less perceptually salient. I showed that this metric, when used with my image-driven simplification method, can lead to higher quality simplified models than those obtained with a metric based on mean square error.

9.2 Future Work

There are several potential avenues for future research relating to each of the methods discussed in this thesis. Some of these research ideas involve logical extensions of my algorithms, whereas others are related yet distinctly different approaches to the problems addressed in this thesis. In addition, I will also discuss several potential applications of my algorithms that were not mentioned in previous chapters.

9.2.1 Quadric-Based Simplification

Both of the simplification methods in Chapters 4 and 5 make use of quadric error metrics, therefore some of my proposed directions for future work within this area pertain to both of these methods. In §9.2.2 below, I will discuss issues that are relevant to out-of-core simplification only.

Guaranteed Error Tolerance As I have shown in previous chapters, the quadric error metric produces models of very low *mean* geometric errors. However, no guarantees can be made on the resulting *maximum* (Hausdorff) errors. In fact, as pointed out in §5.2.2, the best possible substitute vertex may be arbitrarily far from the original surface, such as in the case where two nearly parallel parts of the surface meet.

For applications that demand an error tolerance, but would also benefit from a low mean error, it may be possible to extend the quadric-based algorithm to allow only those simplification operations that respect the error tolerance. For instance, a hybrid of Simplification Envelopes [28] and the memoryless edge collapse method may yield an algorithm having both good mean and maximum error characteristics.

Accurate Distance Measurements Using the example just given of unbounded errors, one might ask why such errors occur in the first place? It would seem that such large maximum errors would also lead to large mean errors, yet the quadric error metric does not penalize such deviations. The reason for this is that the quadric error measures orthogonal but not tangential errors by assuming that the distance to each face in the input mesh can be approximated by measuring the distance to the (infinite) plane in which the triangle lies. Ideally, the quadric error metric would account for both orthogonal and tangential error terms, and measure the squared smallest distance to a triangle. The closest point on a triangle may be one of its vertices, a point on one of its edges, or a point in the interior of the triangle. It may be possible to use quadrics to measure all of these distances, and then use the minimum distance. By extension, this strategy can be used to measure the minimum distance to a collection of triangles. It would be difficult if not impossible to develop closed form expressions for minimizing such a piecewise error measure, but there exist continuous approximations to the minimum distance, e.g. $\min\{x, y\} = \lim_{n\to\infty}(x^{-n} + y^{-n})^{-1/n}$ for $x, y > 0$. It may be possible to generalize quadrics to such error measures for special cases of finite n.

Symmetric Quadric Error Metrics Existing simplification methods based on quadric error measure the distance from a few selected points—the mesh vertices—on the simplified surface to the surface of the original model. I will here argue that this approach to measuring error is fundamentally flawed in several ways. First, the more we simplify the model, the fewer distance samples remain, which in a sense is directly the opposite of what is desired. For very modest levels of simplification, the surfaces are all but guaranteed to be close since many vertices in the simplified mesh have not moved. The coarser the simplified model gets, however, the larger the deviation becomes, yet successively fewer samples are used to measure this increasing error. Second, as discussed in §3.3.1.1, the quadric errors are not measured *symmetrically*, i.e. the distance from the original to the simplified model is not accounted for, which may be large at high simplification ratios, even though the opposite distance may be small. (Consider, for example, what happens when a connected component in the mesh is simplified away entirely.) Finally, for most dense meshes, such as range scans and isosurfaces, the original model does not consist of a set of triangles, but rather of a set of *point samples*. The mesh connectivity is often chosen more or less arbitrarily by applying a triangulation algorithm to the point samples, yet these manufactured triangles are the basis for the quadric error evaluation. A more meaningful error measure would be to compute the distance between the dense set of vertices from the original model to the faces of the simplified mesh.[1] Using the quadric error framework, such an approach essentially treats simplification as a least squares plane fitting of the faces in the simplified model to the input points. Note that, contrary to the original model, there is nothing artificial about the triangles in the simplified model and their connectivity, as these large faces are what makes up the simplified surface. (To see what I mean by this, consider changing the connectivity by swapping an edge in a 100-triangle approximation of the Stanford bunny; does the surface geometry change?)

[1] Indeed, this is the approach taken in [72]. However, their method relies entirely on this uni-directional error measure, and does not measure symmetric errors. The method proposed here also differs from [72] in how vertices in one mesh are mapped to triangles in the other mesh, as well as in how the quadratic error functional is evaluated. I propose using the vertex hierarchy from edge collapses to establish the mapping, while using quadric matrices to efficiently track and evaluate the errors.

I believe that by defining the quadric error as a symmetric error measure, more accurate simplifications can be obtained. This error measure would treat the vertices in the original mesh as fixed, and measure symmetric errors based on the vertices and triangles in the simplified mesh. It is possible to write the error entirely in terms of the vertex positions of the simplified mesh (assuming a fixed connectivity). In conventional quadric-based simplification, the vertices in the simplified model are guaranteed to be optimal with respect to the error measure, because the quadric error is measured as though these vertices are independent samples. In contrast, the optimum of the symmetric quadric error can be found only by performing a global optimization. This is because moving one vertex in the simplified mesh affects the geometry of the incident triangles against which the error is computed, which consequently affects the optimal positions of the vertex's neighbors, and so on. One potential approach to this problem would be to optimize vertices individually during simplification, fixing all other vertices in the mesh, and then in a post-processing step perform a global or iterative local optimization of all vertices. Fortunately, all of these steps are relatively straightforward, and would not be significantly more time consuming than the quadric-based technique in Chapter 4.

Improved Connectivity The connectivity of the mesh resulting from my memoryless simplification method is determined entirely by what set of edges are collapsed. This decision is however made based solely on geometric criteria. It may be that swapping an edge, for instance, would improve the quality of the surface. However, the memoryless metric measures errors only in terms of *changes* made to the model. As a consequence, using this error measure has the implication that any connectivity change to the model necessarily degrades the mesh quality.

One potential approach to integrating edge swaps with memoryless simplification is to couple this operation with the edge collapse and measure their combined effect on the (incremental) error. That is, each collapse of a manifold edge is additionally considered in a dual state, for which the edge is first swapped before it is collapsed. This may be beneficial for long edges that are shortened significantly by being swapped, such as a long and expensive to collapse edge along a "cut" in the mesh that would be sealed by a pre-swap and collapse operation.

Note that the set of pre-swapped edges can be thought of as a collection of "virtual edges" (§3.1.1.5), making the pre-swap/collapse two-step operation a specific instance of vertex pair contraction. Using this topological criterion for selecting virtual edges may prove to be more computationally efficient and easier to implement than, for example, computing Delaunay edges [116], while making it easier to guarantee that the surface remains manifold if so desired.

For memorizing quadrics, such as the ones used in my out-of-core method, that allow absolute errors to be measured, it may be possible to use the quadric information at the vertices of the simplified model to determine whether swapping an edge would be beneficial. There is likely much room for improvement in the connectivity of the models simplified by OoCS, since it is currently determined entirely, and somewhat arbitrarily, by how the fine triangles in the original mesh straddle the grid cell boundaries. Because the memorizing quadrics encode information about the original surface, one could measure the (squared) distance from points other than the vertices on the simplified surface to the original mesh, and use these aggregate distances

as a criterion for accepting edge swaps. This technique would be particularly applicable to simplification based on the symmetric quadrics discussed above, for which exact point locations on the original surface are known.

Quadric-Based Surface Representations Subdivision and NURBS are examples of smooth surface representations. Techniques for designing low complexity control meshes for these primitives to approximate dense polygon meshes or point sets are often cumbersome and computationally expensive [62, 70]. Sometimes one simply wishes to apply subdivision to a simplified mesh to make it appear smooth. This approach cannot recover fine details in the original model, however, as this information is not readily available. An interesting approach to encoding this detail would be to retain the error quadrics in the simplified mesh. As pointed out in [49], these quadrics succinctly incorporate surface position, normal, and curvature information, which is computed and propagated during simplification. From these differential properties, it may be possible to reconstruct parts of the more detailed model as a piecewise smooth surface, by interpolating the quadric information between vertices to produce intermediate points on the surface.

9.2.2 Out-of-Core Simplification.

Because my out-of-core simplification method is restricted to make use of minimal information, it is challenging to come up with ways in which to better make use of this limited information in order to improve the quality of the algorithm. I have already discussed possible ways of improving the quadric error metric, using for example symmetric evaluation, as well as a method for optimizing the mesh connectivity. Below, I will discuss a few additional techniques for improving the quality of the simplified models.

There are also important issues other than quality, two of which I will mention here. In its current form, OoCS does not handle surface attributes. Several large data sets [1, 9, 90] come with color and texture, which need to be preserved. I am confident that an extension of the memoryless technique for surface property preservation from §4.7.1 can be used for this purpose. There are also many applications that would benefit from the output sensitivity and the one-pass capability of OoCS, including isosurface extraction. Instead of producing a dense isosurface that immediately requires simplification, it would be possible to retrofit the popular *marching cubes* algorithm [99] to work directly with OoCS. This integration is made particularly straightforward by the fact that marching cubes operates on a regular grid and visits one grid layer at a time. Combining the two algorithms, OoCS would accept triangles from a few layers at a time, and simplify the model on a downsampled version of the grid used in the isosurface extraction stage. Using this approach eliminates the need to output an overly complex intermediate model.

Surface Boundaries Because OoCS does not make use of connectivity information—a sacrifice that opened up the possibility of performing fast out-of-core simplification in the first place—it has no way of detecting whether an edge is a boundary edge or not. Consequently, surface boundaries are not well preserved by my method. I envision that a variation on the technique used by Garland and Heckbert [50], which makes use of planes parallel to the boundary edges and orthogonal to their incident triangles, can be used

to geometrically encode such boundaries. Consider the case of two adjacent triangles incident on an edge e. By fitting two planes to e that are perpendicular to its incident triangles, we can measure oriented tangential distances to e's triangles along the plane normals. If the two triangles have opposite orientation and lie in the same plane, these two oriented distances cancel. If the triangles are not coplanar, then the oriented distance lies in the bisecting plane of the two triangles and is orthogonal to e. Thus, the sharper an edge is, the larger this distance becomes. When used as part of an error measure, this would tend to preserve sharp edges, which is often desired.

What does this have to do with surface boundaries, one might ask. When e is a boundary edge, there is no adjacent triangle to cancel the tangential oriented distance associated with e's single triangle. Thus, by adding these implicit plane equations, and then squaring the resulting implicit function value, we have an error quadric that penalizes positions that deviate from the surface boundary, as well as positions that are far from sharp edges.[2] Based on this argument, non-manifold edges would also tend to be preserved, which is likely desirable since they typically form sharp creases in the mesh. Note that this scheme makes no use of connectivity information, yet implicitly accounts for feature edges in the mesh.

Enforcing Maximum Errors As discussed in the previous section and in §5.2.2, the optimal cluster representative (as given by the quadric error) may sometimes lie outside the cluster cell itself. Using such outliers would violate the error tolerance imposed by the cluster grid. I suggested a few methods in §5.2.2 for handling this case, such as clamping the vertex coordinates or moving the vertex radially towards the cluster center. A somewhat better solution would be to perform a constrained minimization of the quadric error on the surface of the grid cell. Because grid cells are rectilinear, such positional constraints are linear. I have already described a solution to the linearly constrained quadratic optimization problem in Chapter 4, and I believe a similar technique could be used to more accurately position vertices that fall outside their corresponding grid cells.

Adaptive Simplification My out-of-core method has no knowledge about the shape of the input model before it is read, and cannot therefore perform an adaptive sampling of its geometry. Ideally, the simplified mesh should adapt to the geometry of the original surface, allowing large triangles to be used in flat regions, and requiring a finer tessellation in regions of high curvature. Instead, OoCS produces a nearly uniformly sampled mesh. A potential solution to this problem would be to simplify the model hierarchically, by initially using a fine cluster grid, and then recursively merging grid cells whose quadrics agree on the local surface characterization. This could be determined simply by adding the quadric matrices of adjacent clusters and measuring the resulting minimum quadric error. By successively increasing the quadric error threshold, a progressively coarser model is obtained, whose triangulation adapts well to the geometry of the original.

[2] This quadric error metric bears some similarities to the one used for boundary preservation in §4.4.6, but measures only tangential, and not arbitrary orthogonal, distances to the edge.

9.2.3 Image-Driven Simplification

There are few obvious extensions to the image-driven simplification algorithm, and probably less room for improvement than in my other algorithms. I have however identified a few areas that require further investigation.

Vertex Placement Heuristics A study of different vertex placement strategies, similar to the one performed in §4.8, would be useful for identifying other, perhaps faster, geometry-based heuristics for placing vertices and computing surface attributes. Using an image metric, a rating of the visual quality of different vertex placement methods on a number of test models would be useful for choosing a method that has also has desirable geometric characteristics.

Automatic Parameter Selection One aspect particular to my image-driven algorithm is the option of choosing a set of rendering parameters, such as number of views, viewpoint placement, image dimensions, background image, lighting, etc. While the user has considerable freedom to specify these parameters in a manner suitable for the application, finding a good set of parameters may be a tedious process. Based on my experience, the default values for many of these parameters generally work well, although it may be worthwhile to explore different parameter settings to emphasize certain features of the model, such as its silhouettes, or a smaller part of the object such as its eyes.

Algorithmically, the most difficult issue related to parameter selection is the choice of viewpoints for objects with complex geometry and topology, such as thin, sparse structures, e.g. a tree, and objects with partially or totally occluded geometry, e.g. an automobile. This is a problem currently being investigated in a number of fields within computer graphics and related disciplines, with range scanning being a prime example. For objects with high depth complexity, it may be desirable to take an approach entirely different from optimizing the viewpoint distribution. In §7.3.2, I suggested a set of possible alternatives for making occluded parts visible, including the use of front-face culling, translucency, cutting planes, and model segmentation. It is not immediately clear whether pursuing these somewhat exotic alternatives would pay off in terms of elevated visual quality.

Image Metrics My work on image-driven simplification and suitable image metrics for comparing 3D models is still in the experimental stage. As such, I have only had the opportunity to try a few different image metrics. There are likely many other metrics available, including perceptually-based ones, that would make good candidates for comparing polygonal models. So far, I have only used metrics that are based on achromatic images. A logical next step would be to design metrics that also take color into account.

Another intriguing challenge would be to design a (perceptual) metric that operates on more uniformly sampled 4D light fields. While my camera sphere parameterization of the light field is dense in direction (2^{16} directions or more are sampled from each viewpoint), it is sparse in position (two dozen viewpoints, at most). Using some of the light field parameterizations employed in the image-based rendering community

can potentially allow more information about the model to be captured, while simultaneously providing more redundancy in the coverage of the surface, ensuring that parts of the surface are seen from many viewpoints.

9.2.4 Image-Driven Mesh Optimization

I mentioned in Chapter 8 that there is likely much room for improvement in my mesh optimization method, particularly at the higher levels of the optimization algorithm. I will touch on this below, and also discuss potential alternative approaches to optimization. In addition, some of the research problems mentioned for image-driven simplification, including better image metrics and parameter choices, are also relevant here.

Algorithmic Improvements The low-level stages of my optimization algorithm are based on common optimization techniques, such as the downhill simplex method and a set of efficient connectivity operations, which have proven to perform well in this and other applications. The high-level procedure that governs the choice of what operations should be applied where is not as firmly rooted in optimization theory, however, largely because these decisions are by nature very application specific. In fact, I arrived at the procedure described in §8.2 starting from random descent and using my intuition and trial and error to eliminate some of the major bottlenecks. I envision that entirely different strategies, perhaps with more theoretical support, might lead to a more efficient algorithm. As far as my current algorithm is concerned, I believe that many of its parameters can be tuned more carefully to yield large speedups. The oracle used for selecting edges can also be improved. Currently, image differences are distributed to vertices in the mesh whether the vertices are visible or not from the given viewpoint. By taking occlusion into account, a more accurate assessment of which parts of the mesh are high in energy can be made. The statistical model for choosing connectivity moves is also subject to improvement for more reliable predictions.

The set of connectivity moves considered allows a large category of connectivities to be reached from any given mesh. This set of operations is not complete, however, in that it does not generate all possible meshes, nor is it necessarily the best possible set of operations for optimization. In particular, the edge split operation may provide fewer degrees of freedom than are needed to successfully teleport vertices. Note that the new vertex introduced by an edge split always has a valence that is twice the number of triangles incident upon the edge. If the target model is a sphere, for example, then it is unlikely that a valence four vertex resulting from splitting a manifold edge would be optimal, since such vertices are usually sharp and pointy instead of smooth. Even though edge swaps near the split edge are able to change the valence, these operations will not be invoked unless the split by itself is first accepted. Instead, I believe that a more general operation, such as *vertex split* (the inverse of edge collapse), has the potential to be more effective, while simultaneously allowing a larger set of topology changes to be made.

Alternative Strategies There are approaches to optimization other than making repeated local changes. Indeed, there are situations for which more global operations are needed to escape from a local minimum, requiring simultaneous geometry and connectivity changes to a larger region within the mesh. A particularly confounding problem with local connectivity operations is that a sequence of several moves may be needed

to traverse a hill in the energy function, which quickly leads to an explosion in combinatorial complexity. An alternative way of doing optimization is based on the principle of annealing, for which a heating phase is followed by cooling. By performing alternating phases of refinement (e.g. smooth subdivision) and simplification, one almost always has a chance of getting out of local minima. The idea is to refine the mesh everywhere using, for example, Loop subdivision, and then reduce the mesh complexity back to its original level via image-driven simplification. These two phases are followed by a fine-tuning of the geometry by employing the continuous optimization procedure from §8.2.3. Using this approach, vertices are quickly distributed over the mesh to where they are needed. The simplification phase then culls away the less important vertices and settles on a hopefully more optimal connectivity.

Applications Mesh optimization is not limited to simplified meshes. There are in fact several applications that would benefit from an optimization guided by visual similarity. One such example is remeshing for subdivision and lossy geometry compression. Using the image-driven optimization framework, the vertices and connectivity of a coarse control mesh can be optimized, which is then subdivided and possibly enhanced using small corrective detail vectors (cf. [11, 61, 87]). Such remeshing methods are usually driven by geometric deviation. Using a perceptually-based image metric, it may be possible to ensure that the loss in visual quality due to compression is imperceptible.

Another challenge would be to apply image-driven optimization in the context of image-based rendering. For hybrid geometry- and image-based rendering, the goal is to substitute detailed geometry with images using texture mapped, coarse geometry (see the discussion in §1.1). By allowing both the geometry, parameterization, and texture image to be optimized, this procedure can be entirely automated. A similar approach could be used for related problems, such as the design of bump, displacement, and normal maps. The use of such graphics primitives is motivated by the goal of enhancing the realism and visual quality of polygonal models. Automatic construction of such texture maps and their parameterizations is consequently a problem for which image-driven optimization would appear to be particularly well suited.

Bibliography

[1] Michael J. Ackerman. The visible human project. *Proceedings of the IEEE*, 86(3):504–511, March 1998.

[2] Daniel Aliaga, Jonathan Cohen, Andrew Wilson, Hansong Zhang, Carl Erikson, Kenneth Hoff, T. Hudson, Wolfgang Stürzlinger, Eric Baker, Rui Bastos, Mary Whitton, Frederick Brooks, and Dinesh Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In Jessica Hodgins and James D. Foley, editors, *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206. ACM SIGGRAPH, Atlanta, Georgia, April 1999. ISBN 1-58113-082-1.

[3] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. A new voronoi-based surface reconstruction algorithm. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 415–422. ACM Press, Orlando, Florida, July 1998. ISBN 0-89791-999-8.

[4] Esther M. Arkin, L. Paul Chew, David P. Huttenlocher, Klara Kedem, and Joseph S. B. Mitchell. An efficiently computable metric for comparing polygonal shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(3):209–216, March 1991.

[5] Peter R. Atherton. A method of interactive visualization of CAD surface models on a color video display. In *Computer Graphics (Proceedings of SIGGRAPH 81)*, volume 15, pages 279–287. Dallas, Texas, August 1981.

[6] Chandrajit L. Bajaj and Daniel R. Schikore. Error-bounded reduction of triangle meshes with multivariate data. In *Proceedings of Visual Data Exploration and Analysis III*, volume 2656, pages 34–45. SPIE, San Jose, California, January 1996.

[7] Peter G. J. Barten. Evaluation of subjective image quality with the square-root integral method. *Journal of the Optical Society of America A*, 7(10):2024–2031, October 1990.

[8] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[9] Fausto Bernardini, Joshua Mittleman, and Holly Rushmeier. Case study: Scanning Michelangelo's Florentine Pietà. SIGGRAPH 99 Course #8, August 1999.

[10] Fausto Bernardini, Joshua Mittleman, Holly Rushmeier, Cláudio Silva, and Gabriel Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, October - December 1999. ISSN 1077-2626.

[11] Martin Bertram, Mark A. Duchaineau, Bernd Hamann, and Kenneth I. Joy. Bicubic subdivision-surface wavelets for large-scale isosurface representation and visualization. In Thomas Ertl, Bernd Hamann, and Amitabh Varshney, editors, *IEEE Visualization 2000*, pages 389–396. IEEE, Salt Lake City, Utah, October 2000. ISBN 0-7803-6478-3.

[12] Paul J. Besl. Active, optical range imaging sensors. *Machine Vision and Applications*, 1(2):127–152, 1988.

[13] Volker Blanz and Thomas Vetter. A morphable model for the synthesis of 3d faces. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 187–194. Addison Wesley, Los Angeles, California, August 1999. ISBN 0-201-48560-5.

[14] Mark R. Bolin and Gary W. Meyer. A perceptually based adaptive sampling algorithm. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 299–310. ACM Press, Orlando, Florida, July 1998. ISBN 0-89791-999-8.

[15] Jean-Yves Bouguet. *Visual methods for three-dimensional modeling*. Ph.D. thesis, California Institute of Technology, May 1999.

[16] Dmitry Brodsky and Benjamin Watson. Model simplification through refinement. In I. Scott MacKenzie and James Stewart, editors, *Graphics Interface 2000*, pages 221–228. Morgan Kaufmann Publishers, Montreal, Canada, May 2000.

[17] Peter J. Burt and Edward H. Adelson. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31(4):532–540, April 1983.

[18] Emilio Camahort, Apostolos Lerios, and Donald Fussell. Uniformly sampled light fields. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pages 117–130. Springer-Verlag, Vienna, Austria, June 1998. ISBN 3-211-83213-0.

[19] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges–a scalable representation for triangle meshes. *Journal of Graphics Tools*, 3(4):1–12, 1998. ISSN 1086-7651.

[20] Edwin Catmull and James Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350—355, September 1978.

[21] Andrew Certain, Jovan Popovic, Tony DeRose, Tom Duchamp, David Salesin, and Werner Stuetzle. Interactive multiresolution surface viewing. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 91–98. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[22] Yi-Jen Chiang, Cláudio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 167–174. IEEE, Research Triangle Park, North Carolina, October 1998. ISBN 0-8186-9176-X.

[23] Andrea Ciampalini, Paolo Cignoni, Claudio Montani, and Roberto Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, 13(5):228–246, 1997. ISSN 0178-2789.

[24] Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998. ISSN 1067-7055.

[25] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[26] Jonathan Cohen, Dinesh Manocha, and Marc Olano. Simplifying polygonal models using successive mappings. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 395–402. IEEE, Phoenix, Arizona, November 1997. ISBN 0-8186-8262-0.

[27] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 115–122. ACM Press, Orlando, Florida, July 1998. ISBN 0-89791-999-8.

[28] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Jr. Frederick P. Brooks, and William Wright. Simplification envelopes. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 119–128. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[29] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. ISBN 0-262-03141-8.

[30] Michael A. Cosman and Robert A. Schumacker. System strategies to optimize CIG image content. In *Proceedings Image II Conference*. Scottsdale, Arizona, June 1981.

[31] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 303–312. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[32] Scott Daly. The visible difference predictor: An algorithm for the assessment of image fidelity. In Andrew B. Watson, editor, *Digital Images and Human Vision*, chapter 14, pages 179–206. MIT Press, 1993. ISBN 0-262-23171-9.

[33] Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney. Walkthroughs of complex environments using image-based simplification. *Computers & Graphics*, 22(1):55–69, February 1998. ISSN 0097-8493.

[34] Douglass Davis, William Ribarsky, T.Y. Jiang, Nickolas Faust, and Sean Ho. Real-time visualization of scalably large collections of heterogeneous objects. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 437–440. IEEE, San Francisco, California, October 1999. ISBN 0-7803-5897-X.

[35] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 11–20. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[36] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 81–88. IEEE, Phoenix, Arizona, November 1997. ISBN 0-8186-8262-0.

[37] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In Robert Cook, editor, *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 173–182. Addison Wesley, Los Angeles, California, August 1995. ISBN 0-201-84776-0.

[38] Jihad El-Sana and Amitabh Varshney. Controlled simplification of genus for polygonal models. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 403–412. IEEE, Phoenix, Arizona, November 1997. ISBN 0-8186-8262-0.

[39] Jihad El-Sana and Amitabh Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):83–94, September 1999. ISSN 1067-7055.

[40] Jihad A. El-Sana, Elvir Azanli, and Amitabh Varshney. Skip strips: Maintaining triangle strips for view-dependent rendering. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 131–138. IEEE, San Francisco, California, October 1999. ISBN 0-7803-5897-X.

[41] Carl Erikson and Dinesh Manocha. GAPS: General and automatic polygonal simplification. In Jessica Hodgins and James D. Foley, editors, *1999 ACM Symposium on Interactive 3D Graphics*, pages 79–88. ACM SIGGRAPH, Atlanta, Georgia, April 1999. ISBN 1-58113-082-1.

[42] Carl M. Erikson. *Hierarchical Levels of Detail to Accelerate the Rendering of Large Static and Dynamic Polygonal Environments*. Ph.D. thesis, University of North Carolina at Chapel Hill, 2000.

[43] John S. Falby, Michael J. Zyda, David R. Pratt, and Randy L. Mackey. NPSNET: Hierarchical data structures for real-time three-dimensional visual simulation. *Computers & Graphics*, 17(1):65–70, 1993. ISSN 0097-8493.

[44] James A. Ferwerda, Sumanta N. Pattanaik, Peter Shirley, and Donald P. Greenberg. A model of visual masking for computer graphics. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 143–152. ACM Press, Los Angeles, California, August 1997. ISBN 0-89791-896-7.

[45] Robert J. Fowler and James J. Little. Automatic extraction of irregular network digital terrain models. In *Computer Graphics (Proceedings of SIGGRAPH 79)*, volume 13, pages 199–207. Chicago, Illinois, August 1979.

[46] William T. Freeman and Edward H. Adelson. The design and use of steerable filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(9):891–906, September 1991.

[47] Thomas A. Funkhouser. A visibility algorithm for hybrid geometry- and image-based modeling and rendering. *Computers & Graphics*, 23(5):719–728, October 1999. ISSN 0097-8493.

[48] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In James T. Kajiya, editor, *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 247–254. Addison Wesley, Anaheim, California, August 1993. ISBN 0-201-58889-7.

[49] Michael Garland. *Quadric-Based Polygonal Surface Simplification*. Ph.D. thesis, Carnegie Mellon University, May 1999.

[50] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 209–216. ACM Press, Los Angeles, California, August 1997. ISBN 0-89791-896-7.

[51] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 263–270. IEEE, Research Triangle Park, North Carolina, October 1998. ISBN 0-8186-9176-X.

[52] Tran S. Gieng, Bernd Hamann, Kenneth I. Joy, Gregory L. Schlussmann, and Isaac J. Trotts. Smooth hierarchical surface triangulations. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 379–386. IEEE, Phoenix, Arizona, November 1997. ISBN 0-8186-8262-0.

[53] E. Bruce Goldstein. *Sensation & Perception*. Brooks/Cole Publishing Company, fourth edition, 1996. ISBN 0-534-26622-3.

[54] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996. ISBN 0-8018-5413-X.

[55] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 43–54. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[56] Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, Leonard McMillan, Benedict J. Brown, and Abraham D. Stone. Silhouette mapping. Technical Report TR–1–99, Harvard University, March 1999.

[57] Andre Guéziec. Surface simplification with variable tolerance. In *Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery*, pages 132–139. Baltimore, Maryland, November 1995.

[58] André P. Gueziec, Gabriel Taubin, Francis Lazarus, and William Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 383–390. IEEE, Research Triangle Park, North Carolina, October 1998. ISBN 0-8186-9176-X.

[59] Stefan Gumhold and Tobias Hüttner. Multiresolution rendering with displacement mapping. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 55–66. ACM Press, Los Angeles, California, August 1999. ISBN 1-58113-170-4.

[60] Igor Guskov, Wim Sweldens, and Peter Schröder. Multiresolution signal processing for meshes. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 325–334. Addison Wesley, Los Angeles, California, August 1999. ISBN 0-201-48560-5.

[61] Igor Guskov, Kiril Vidimce, Wim Sweldens, and Peter Schröder. Normal meshes. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 95–102. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, New Orleans, Louisiana, July 2000. ISBN 1-58113-208-5.

[62] Mark Halstead, Michael Kass, and Tony DeRose. Efficient, fair interpolation using Catmull-Clark surfaces. In James T. Kajiya, editor, *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 35–44. Addison Wesley, Anaheim, California, August 1993. ISBN 0-201-58889-7.

[63] Bernd Hamann. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*, 11(2):197–214, 1994. ISSN 0167-8396.

[64] Taosong He, Lichan Hong, Amitabh Varshney, and Sidney W. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2), June 1996. ISSN 1077-2626.

[65] Paul Hinker and Charles Hansen. Geometric optimization. In Gregory M. Nielson and Dan Bergeron, editors, *IEEE Visualization '93*, pages 189–195. IEEE Computer Society Press, San Jose, California, October 1993. ISBN 0-8186-3940-7.

[66] Hugues Hoppe. Progressive meshes. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 99–108. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[67] Hugues Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 189–198. ACM Press, Los Angeles, California, August 1997. ISBN 0-89791-896-7.

[68] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 35–42. IEEE, Research Triangle Park, North Carolina, October 1998. ISBN 0-8186-9176-X.

[69] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 59–66. IEEE, San Francisco, California, October 1999. ISBN 0-7803-5897-X.

[70] Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. In Andrew Glassner, editor, *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 295–302. ACM Press, Orlando, Florida, July 1994. ISBN 0-89791-667-0.

[71] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In Edwin E. Catmull, editor, *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 71–78. Chicago, Illinois, July 1992. ISBN 0-201-51585-7.

[72] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In James T. Kajiya, editor, *Proceedings of SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 19–26. Addison Wesley, Anaheim, California, August 1993. ISBN 0-201-58889-7.

[73] Hugues Hoppe and Steve Marschner. Efficient minimization of new quadric metric for simplifying meshes with apperance attributes. Technical Report MSR–TR–2000–64, Microsoft Research, June 2000.

[74] Charles E. Jacobs, Adam Finkelstein, and David H. Salesin. Fast multiresolution image querying. In Robert Cook, editor, *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 277–286. Addison Wesley, Los Angeles, California, August 1995. ISBN 0-201-84776-0.

[75] Alan D. Kalvin and Russell H. Taylor. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Computer Graphics & Applications*, 16(3):64–77, May 1996. ISSN 0272-1716.

[76] Craig S. Kaplan and David H. Salesin. Escherization. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 499–510. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, New Orleans, Louisiana, July 2000. ISBN 1-58113-208-5.

[77] Renate Kempf and Jed Hartman. *OpenGL on Silicon Graphics Systems*. Silicon Graphics, Inc., 1998. SGI Document Number 007-2392-002.

[78] Andrei Khodakovsky, Peter Schröder, and Wim Sweldens. Progressive geometry compression. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 271–278. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, New Orleans, Louisiana, July 2000. ISBN 1-58113-208-5.

[79] Mark J. Kilgard. A practical and robust bump-mapping technique for today's GPUs. Game Developers Conference 2000, tutorial on Advanced OpenGL Game Development, March 2000.

[80] Davis King and Jarek Rossignac. Optimal bit allocation in compressed 3d models. *Computational Geometry: Theory and Applications*, 14(1–3):91–118, November 1999.

[81] Reinhard Klein, Gunther Liebich, and Wolfgang Straßer. Mesh reduction with error control. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 311–318. IEEE, San Francisco, California, October 1996. ISBN 0-89791-864-9.

[82] Leif Kobbelt, Swen Campagna, and Hans-Peter Seidel. A general framework for mesh decimation. In Kellogg Booth and Alain Fournier, editors, *Graphics Interface '98*, pages 43–50. Morgan Kaufmann Publishers, Vancouver, Canada, June 1998. ISBN 1-55860-550-9.

[83] Leif Kobbelt, Swen Campagna, Jens Vorsatz, and Hans-Peter Seidel. Interactive multi-resolution modeling on arbitrary meshes. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 105–114. ACM Press, Orlando, Florida, July 1998. ISBN 0-89791-999-8.

[84] Leif P. Kobbelt, Jens Vorsatz, Ulf Labsik, and Hans-Peter Seidel. A shrink wrapping approach to remeshing polygonal surfaces. *Computer Graphics Forum*, 18(3):119–130, September 1999. ISSN 1067-7055.

[85] Gregory Ward Larson, Holly Rushmeier, and Christine Piatko. A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):291–306, October - December 1997. ISSN 1077-2626.

[86] Aaron Lee, David Dobkin, Wim Sweldens, and Peter Schröder. Multiresolution mesh morphing. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 343–350. Addison Wesley, Los Angeles, California, August 1999. ISBN 0-201-48560-5.

[87] Aaron Lee, Henry Moreton, and Hugues Hoppe. Displaced subdivision surfaces. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 85–94. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, New Orleans, Louisiana, July 2000. ISBN 1-58113-208-5.

[88] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 95–104. ACM Press, Orlando, Florida, July 1998. ISBN 0-89791-999-8.

[89] Marc Levoy and Pat Hanrahan. Light field rendering. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 31–42. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[90] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The Digital Michelangelo project: 3d scanning of large statues. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 131–144. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, New Orleans, Louisiana, July 2000. ISBN 1-58113-208-5.

[91] Bruno Lévy and Jean-Laurent Mallet. Non-distorted texture mapping for sheared triangulated meshes. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 343–352. ACM Press, Orlando, Florida, July 1998. ISBN 0-89791-999-8.

[92] Peter Lindstrom. Out-of-core simplification of large polygonal models. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 259–262. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, New Orleans, Louisiana, July 2000. ISBN 1-58113-208-5.

[93] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory Turner. Real-time, continuous level of detail rendering of height fields. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 109–118. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[94] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 279–286. IEEE, Research Triangle Park, North Carolina, October 1998. ISBN 0-8186-9176-X.

[95] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April - June 1999. ISSN 1077-2626.

[96] Peter Lindstrom and Greg Turk. Image-driven mesh optimization. Technical Report GIT–GVU–00–16, Georgia Institute of Technology, June 2000.

[97] Peter Lindstrom and Greg Turk. Image-driven simplification. *ACM Transactions on Graphics*, 19(3):204–241, July 2000. ISSN 0730-0301.

[98] Charles Loop. *Smooth Subdivision Surfaces Based on Triangles*. Master's thesis, University of Utah, August 1987.

[99] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In Maureen C. Stone, editor, *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 163–169. Anaheim, California, July 1987.

[100] Pertti Lounesto. *Clifford Algebras and Spinors*. Cambridge University Press, 1997. ISBN 0-521-59916-4.

[101] Michael Lounsbery, Tony D. DeRose, and Joe Warren. Multiresolution analysis for surfaces of arbitrary topological type. *ACM Transactions on Graphics*, 16(1):34–73, January 1997. ISSN 0730-0301.

[102] Kok-Lim Low and Tiow-Seng Tan. Model simplification using vertex-clustering. In Michael Cohen and David Zeltzer, editors, *1997 ACM Symposium on Interactive 3D Graphics*, pages 75–82. ACM SIGGRAPH, Providence, Rhode Island, April 1997. ISBN 0-89791-884-3.

[103] Jeffrey Lubin. A visual discrimination model for imaging system design and evaluation. In Eli Peli, editor, *Vision Models for Target Tracking and Recognition*, Series on information display, pages 245–283. World Scientific, 1995.

[104] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 199–208. ACM Press, Los Angeles, California, August 1997. ISBN 0-89791-896-7.

[105] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In Pat Hanrahan and Jim Winget, editors, *1995 ACM Symposium on Interactive 3D Graphics*, pages 95–102. ACM SIGGRAPH, Monterey, California, April 1995. ISBN 0-89791-736-7.

[106] Lee Markosian, Jonathan M. Cohen, Thomas Crulli, and John F. Hughes. Skin: A constructive approach to modeling free-form shapes. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 393–400. Addison Wesley, Los Angeles, California, August 1999. ISBN 0-201-48560-5.

[107] Joe Marks, Brad Andalman, Paul A. Beardsley, William Freeman, Sarah Gibson, Jessica Hodgins, Thomas Kang, Brian Mirtich, Hanspeter Pfister, Wheeler Ruml, Kathy Ryall, Joshua Seims, and Stuart Shieber. Design galleries: A general approach to setting parameters for computer graphics and animation. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 389–400. ACM Press, Los Angeles, California, August 1997. ISBN 0-89791-896-7.

[108] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In Robert Cook, editor, *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 39–46. Addison Wesley, Los Angeles, California, August 1995. ISBN 0-201-84776-0.

[109] Henry P. Moreton and Carlo H. Séquin. Functional optimization for fair surface design. In Edwin E. Catmull, editor, *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 167–176. Chicago, Illinois, July 1992. ISBN 0-201-51585-7.

[110] T. M. Murali and Thomas A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In Michael Cohen and David Zeltzer, editors, *1997 ACM Symposium on Interactive 3D Graphics*, pages 155–162. ACM SIGGRAPH, Providence, Rhode Island, April 1997. ISBN 0-89791-884-3.

[111] F. S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. Technical Report GIT–GVU–99–37, Georgia Institute of Technology, November 1999. To appear in IEEE Transactions on Visualization and Computer Graphics.

[112] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, second edition, 1998. ISBN 0-521-64010-5.

[113] Renato Pajarola and Jarek Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):79–93, January - March 2000. ISSN 1077-2626.

[114] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 101–108. ACM Press, Los Angeles, California, August 1997. ISBN 0-89791-896-7.

[115] Richard Pito. A solution to the next best view problem for automated surface acquisition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(10):1016–1030, October 1999. ISSN 0162-8828.

[116] Jovan Popovic and Hugues Hoppe. Progressive simplicial complexes. In Turner Whitted, editor, *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 217–224. ACM Press, Los Angeles, California, August 1997. ISBN 0-89791-896-7.

[117] Ian R. Porteous. *Clifford Algebras and the Classical Groups*. Cambridge University Press, 1995.

[118] Charles Poynton. The rehabilitation of gamma. In B. E. Rogowitz and T. N. Pappas, editors, *Proceedings of Human Vision and Electronic Imaging III*, volume 3299, pages 232–249. SPIE, San Jose, California, January 1998.

[119] Emil Praun, Hugues Hoppe, and Adam Finkelstein. Robust mesh watermarking. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 49–56. Addison Wesley, Los Angeles, California, August 1999. ISBN 0-201-48560-5.

[120] 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. ISBN 0-521-43108-5.

[121] Chris Prince. *Progressive Meshes for Large Models of Arbitrary Topology*. Master's thesis, University of Washington, 2000.

[122] Mahesh Ramasubramanian, Sumanta N. Pattanaik, and Donald P. Greenberg. A perceptually based physical error metric for realistic image synthesis. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 73–82. Addison Wesley, Los Angeles, California, August 1999. ISBN 0-201-48560-5.

[123] Martin Reddy. SCROOGE: Perceptually-driven polygon reduction. *Computer Graphics Forum*, 15(4):191–203, 1996. ISSN 0167-7055.

[124] Kevin J. Renze and James H. Oliver. Generalized surface and volume decimation for unstructured tessellated domains. In *Proceedings of IEEE 1996 Virtual Reality Annual International Symposium*, pages 111–121. IEEE Computer Society Press, Santa Clara, California, March 1996. ISBN 0-8186-7295-1.

[125] John Rohlf and James Helman. IRIS Performer: A high performance multiprocessing toolkit for real-time 3d graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM Press, Orlando, Florida, July 1994. ISBN 0-89791-667-0.

[126] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):67–76, August 1996. ISSN 1067-7055.

[127] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In Bianca Falciendo and Toshiyasu L. Kunii, editors, *Modeling in Computer Graphics*, IFIP series on computer graphics, pages 455–465. Springer-Verlag, 1993.

[128] Jarek Rossignac and David Cardoze. Matchmaker: Manifold BReps for non-manifold r-sets. In Willem F. Bronsvoort and David C. Anderson, editors, *Proceedings of the Fifth Symposium on Solid Modeling and Applications (SSMA-99)*, pages 31–41. ACM Press, June 1999. ISBN 1-58113-080-5.

[129] Holly Rushmeier, Greg Ward Larson, C. Piatko, P. Sanders, and B. Rust. Comparing real and synthetic images: Some ideas about metrics. In Patrick Hanrahan and Werner Purgathofer, editors, *Eurographics Rendering Workshop 1995*, pages 82–91. Springer-Verlag, Dublin, Ireland, June 1995. ISBN 3-211-82733-1.

[130] Holly Rushmeier, Charles Patterson, and Aravindan Veerasamy. Geometric simplification for indirect illumination calculations. In Scott MacKenzie and James Stewart, editors, *Graphics Interface '93*, pages 227–236. Morgan Kaufmann Publishers, Toronto, Canada, May 1993. ISBN 1-55860-994-6.

[131] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[132] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 327–334. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, New Orleans, Louisiana, July 2000. ISBN 1-58113-208-5.

[133] Dieter Schmalstieg and Gernot Schaufler. Smooth levels of detail. In *Proceedings of IEEE 1997 Virtual Reality Annual International Symposium*, pages 12–19. IEEE Computer Society Press, Albuquerque, New Mexico, March 1997. ISBN 0-8186-7843-7.

[134] William J. Schroeder. A topology modifying progressive decimation algorithm. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97*, pages 205–212. IEEE, Phoenix, Arizona, November 1997. ISBN 0-8186-8262-0.

[135] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. In Edwin E. Catmull, editor, *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 65–70. Chicago, Illinois, July 1992. ISBN 0-201-51585-7.

[136] Steven M. Seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. *International Journal of Computer Vision*, 35(2):151–173, 1999.

[137] Jonathan Shade, Steven J. Gortler, Li wei He, and Richard Szeliski. Layered depth images. In Michael Cohen, editor, *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 231–242. ACM Press, Orlando, Florida, July 1998. ISBN 0-89791-999-8.

[138] Heung-Yeung Shum and Li-Wei He. Rendering with concentric mosaics. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 299–306. Addison Wesley, Los Angeles, California, August 1999. ISBN 0-201-48560-5.

[139] Cláudio Silva. Personal correspondence, November 1999.

[140] Cláudio Silva and Gabriel Taubin. Curvature-based estimation of surface sampling. Sixth SIAM Conference on Geometric Design, November 1999.

[141] Eero P. Simoncelli, William T. Freeman, Edward H. Adelson, and David J. Heeger. Shiftable multiscale transforms. *IEEE Transactions on Information Theory*, 38(2):587–607, March 1992.

[142] Paul Steed. The art of low-polygon modeling. *Game Developer*, pages 62–69, June 1998.

[143] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann Publishers, San Francisco, CA, 1996. ISBN 1-55860-375-1.

[144] Patrick C. Teo and David J. Heeger. Perceptual image distortion. In *First IEEE International Conference on Image Processing*, volume 2, pages 982–986. November 1994.

[145] Jack Tumblin and Greg Turk. LCIS: A boundary hierarchy for detail-preserving contrast reduction. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 83–90. Addison Wesley, Los Angeles, California, August 1999. ISBN 0-201-48560-5.

[146] Greg Turk. Re-tiling polygonal surfaces. In Edwin E. Catmull, editor, *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 55–64. Chicago, Illinois, July 1992. ISBN 0-201-51585-7.

[147] Greg Turk and David Banks. Image-guided streamline placement. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 453–460. Addison Wesley, New Orleans, Louisiana, August 1996. ISBN 0-201-94800-1.

[148] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In Andrew Glassner, editor, *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 311–318. ACM Press, Orlando, Florida, July 1994. ISBN 0-89791-667-0.

[149] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, October - December 1997. ISSN 1077-2626.

[150] Steve Upstill. *The Renderman Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison Wesley, 1989. ISBN 0-201-50868-0.

[151] Brian A. Wandell. *Foundations of Vision*. Sinauer Associates, Inc., 1995. ISBN 0-87893-853-2.

[152] Benjamin Watson, Alinda Friedman, and Aaron McGaffey. Using naming time to evaluate quality predictors for model simplification. In *Proceedings of the CHI 2000 Conference on Human Factors in Computing Systems*, pages 113–120. Addison Wesley, The Hague, The Netherlands, April 2000. ISBN 0-201-48563-X.

[153] Henrik Weimer and Joe Warren. Subdivision schemes for thin plate splines. *Computer Graphics Forum*, 17(3):303–314, 1998. ISSN 1067-7055.

[154] William Welch and Andrew Witkin. Variational surface modeling. In Edwin E. Catmull, editor, *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 157–166. Chicago, Illinois, July 1992. ISBN 0-201-51585-7.

[155] William Welch and Andrew Witkin. Free-form shape design using triangulated surfaces. In Andrew Glassner, editor, *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 247–256. ACM Press, Orlando, Florida, July 1994. ISBN 0-89791-667-0.

[156] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *Proceedings of New Results and New Trends in Computer Science*, pages 359–370. Springer-Verlag, June 1991. Volume 555 of Lecture Notes in Computer Science.

[157] Andrew Willmott, Paul Heckbert, and Michael Garland. Face cluster radiosity. In Dani Lischinski and Greg Ward Larson, editors, *Eurographics Rendering Workshop 1999*, pages 293–304. Springer-Verlag, Granada, Spain, June 1999. ISBN 3-211-83382-X.

[158] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In Andrew Glassner, editor, *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 269–278. ACM Press, Orlando, Florida, July 1994. ISBN 0-89791-667-0.

[159] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1*. Addison Wesley, second edition, 1997. ISBN 0-201-46138-2.

[160] Julie C. Xia, Jihad El-Sana, and Amitabh Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):171–183, April - June 1997. ISSN 1077-2626.

[161] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 327–334. IEEE, San Francisco, California, October 1996. ISBN 0-89791-864-9.

[162] Johnson K. Yan. Advances in computer-generated imagery for flight simulation. *IEEE Computer Graphics & Applications*, 5(8):37–51, August 1985.

Vita

Peter Lindstrom was born on January 4, 1970, in Stockholm, Sweden, where he grew up. After graduating from high school in 1989, he spent a year in the Swedish Army while working part-time as a freelance writer for a computer magazine, in which he published a series of articles on computer graphics. Peter came to the United States in 1990 to attend Elon College in North Carolina. As a varsity tennis player at Elon, he was awarded athletic and academic *All-America First Team* honors. He graduated Summa Cum Laude from Elon in 1994 with a B.S. degree in Computer Science, Mathematics, and Physics. Soon thereafter, Peter began his graduate studies at Georgia Institute of Technology, where he received an *NCAA Postgraduate Fellowship* and a *Link Foundation Fellowship*. While at Georgia Tech, his research was primarily focused on methods for interactive terrain visualization and polygonal model simplification. During the summer of 1998, he was an intern with the graphics group at Microsoft Research. Under the supervision of Dr. Greg Turk, Peter completed his thesis work and received a Ph.D. in Computer Science in December 2000.