University of Alberta

SEMIAUTOMATIC SIMPLIFICATION

by

Gong Li ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2000

0-612-59831-4

Abstract

In computer graphics, polygons are widely used to represent 3D models or scenes. When complex polygonal models contain more detail than needed, or different levels of detail are required, models need to be simplified. Simplified models have significantly fewer polygons but still preserve similarity with the original. Existing simplification algorithms usually have difficulty in producing good quality simplified models with low polygon counts. In addition, users have limited control over how models are simplified.

In this thesis, we propose that by adding user control to automatic simplifications, the quality of simplified models can be improved. We developed a tool that integrates an existing simplification algorithm and provides users with the ability to interact with the simplification process. The tool produces a multiresolution representation from an input model, and offers a set of functionalities that allow users to modify model geometry, local detail, and the simplification process. By simplifying models semiautomatically, users are able to produce more desirable simplified models with low polygon counts.

To my family

Acknowledgements

I will first thank my supervisor Dr. Benjamin Watson. One and a half years ago, I asked him if I could be his master student, and he kindly agreed. Now upon writing this acknowledgement, I realize more than ever how much I owe him for his guidance, patience and help over this period. Without his help, this thesis would not be possible. I would also like to thank everybody in the graphics lab. They created a warm and friendly environment which made working here an enjoyable experience. I want to thank Paul and Pablo for the daily lunch together and for their help that scattered in the progress of my work. I should also thank Peter who corrected language mistakes in the thesis. I will thank Ehud for his kind suggestions and support that always fill me with courage, and Vivian for the happy chatting together.

I especially thank John who initiated the project in the thesis and provided part of my financial support. I want to express my gratitude to the Computing Science Department for offering me the opportunity to pursue a master degree here, which opened up my mind and triggered my interest in research.

Lastly and most importantly, no word could express my thankfulness to my family. Although thousands of miles away and across the Pacific ocean, they are my constant source of motivation and courage.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In computer graphics, polygonal models are a widely used method to represent surfaces. Compared to other modeling techniques, polygonal models are very simple. The most commonly seen polygonal models are made up of only triangles. However, this simplicity does not reduce its power in representing surfaces. Surfaces of arbitrary type and complexity can be represented by polygons. Another advantage that makes polygonal models so popular is that they are very fast to render in graphics hardware. Simple scan-line rendering enables graphics acceleration chips to render millions of triangles per second. This is especially the case in the personal computer and computer gaming industry.

Because of the importance of polygonal models, numerous techniques have been developed to make them more powerful and efficient. One area that has been a focus of interest in the past decade was the simplification and multiresolution representations of complex models. Various algorithms have been developed and some of them could produce fairly good results. However, we observed that there is an absence of tools that provide user control over these algorithms. We believe that providing such a tool coupled with good existing simplification techniques will further improve model quality. Thus we develop a tool system that allows user intervention in automatic model simplification in a multiresolution context.

1.1 Model Simplification

In many computer graphics applications, very large and complex polygonal models are generated, such as in CAD designs and real models scanned in by cameras. The number of polygons contained in the raw output of these models often go up to tens of thousands or even millions. Such a large number of polygons not only make the model difficult to manipulate, but often contain unneeded detail. The triangles used to represent the fine details often shrink to a single pixel after rendering. Thus simplification algorithms have been developed to simplify the complex models to drastically reduce the number of polygons while preserving visual fidelity to a certain degree. In figure 1.1 (a), the bunny is the original model generated from scanning, which contains around 69000 triangles. In figure 1.1 (b), the model is reduced to 5000 triangles. It can be seen that although more than 90% of the faces are removed, the visual quality of the model is very close to the original.

(a) Original model with 69451 triangles

(b) Simplified model with 4999 triangles

Figure 1.1: Original and simplified bunny model.

Having similar quality with the original model, simplified models are easier to manipulate, require less storage space, and are faster to render.

1.2 Multiresolution Representation

Closely related to model simplification is the multiresolution representation of polygonal models. A simplified model is an approximation of the original model with less detail. If there are many such approximations with different degree of simplification, we have the original model represented at different levels of detail (LOD), or different resolutions. Multiresolution representations provide a trade off between performance and quality. More detail enhances the quality but requires more computing power for display, thus reducing display speed. Based on need, a certain resolution must be chosen that provides the best tradeoff between performance and detail. For example, when a model is viewed from a distance, it will shrink to a small area on the screen. Clearly there is no need to render this model in full detail. A simple and coarse representation of the model should serve. However, if a model is viewed closely, we would want to see more detail.

The straightforward approach to represent multiple resolutions would be to store several copies of the model at different levels of detail. Each copy is used to satisfy some range of detail requirements. When the detail requirement changes, the resolution is changed by rendering a different copy. This simple approach is used quite widely, such as in flight simulation applications. But it is not able to produce a continuous range of multiresolution models.

Recently there have been many advances in multiresolution modeling. Various techniques have been developed to create multiresolution representations of polygonal models [GAR99a]. Compared to the simple method of storing discrete model resolutions, such representations are able to produce continuous levels of detail and are more storage efficient. Many simplification algorithms could be extended to create a multiresolution representation of the original model. In many cases, this is achieved by keeping a record of the whole simplification process. Later the history of the simplification process could be used to create structures that are able to extract levels of detail. Other techniques create a multiresolution representation explicitly such as subdivision methods [PUP97].

1.3 Motivation

Nearly every simplification algorithm that exists today does a fairly good job in simplifying a single complex model down to a certain level of detail, for example at several thousand faces. Although numerical errors of the simplified models are generally different from one simplification algorithm to another, the visual quality perceived by human eyes are almost the same at this level of detail. It is only when the model needs to be simplified further, for example to a few hundred faces, that these algorithms begin to display the differences in quality. In some circumstances, models are required to be simplified to this level. This is especially the case in computer games, where PCs are the dominant platforms. For example, in a 3D first person game, if rendering each character requires displaying several thousand polygons, the display speed might be too low to keep the user's interest. Thus it is desirable to simplify models to a few hundred of polygons, while still maintaining reasonable quality.

However, even with algorithms that produce relatively good quality results at low polygon counts, the simplified models normally leave much to be desired. One reason for this is a common characteristic shared by most existing simplification algorithms: the simplification process is completely automatic and driven by some kind of error measure. When the number of polygons is large, it works well because each simplification step has only a small effect on the visual quality of the model. But when the number of polygons is small, the choice of each simplification step has a much larger impact on the simplified model. At this stage, humans usually do a better job than automatic algorithms.

Hence it is reasonable to enable users to intervene in the simplification process, or to make modifications to the simplified models after it is done. Making modifications to simplified models itself is not difficult. We can apply a simplification algorithm to a complex model and use a 3D modeler to edit the output. However, the limitation is that only a single level of detail can be worked on. If ten levels need to be modified, the work will be tedious. A tool that is more closely associated with simplification and multiresolution models will facilitate this process.

Taking advantage of multiresolution representations, we can modify models in ways that are not possible with a normal 3D modeler. First, the user is able to view different levels of detail and choose one to work on. Second, different resolutions are related in a multiresolution representation. The modifications at one level can be used to affect other levels which may reduce the same kind of edit at different levels. Third, we can apply operations that span different resolutions such as preserving meaningful features. This cannot be achieved in editing levels separately. The end result would be an improved simplified model that fits particular requirements. We also have the option of saving the entire multiresolution representation, so that more levels of detail might be produced later.

1.4 Objective

The major objective of our tool is to provide a way for the user to intervene in the simplification process and improve coarse level models. The tool will integrate a simplification algorithm which provides the user a way to simplify the input model. After the model has been simplified, a multiresolution representation of the model is created. The tool offers a set of functions that enable the user to navigate different resolutions, manipulate the model at different levels of detail, and produce an improved simplified model.

Specifically, our goal is to provide the user with the following functionalities:

- Simplify a polygonal model and produce a multiresolution representation of it.
- The user can navigate through and view different levels of detail. The levels are created in the simplification process and represent a range of quality between the original and fully simplified model.
- Allow the user to make modifications at a chosen level:

  - Edit the geometric shape of the model.
  - Propagate the modifications to other detail levels.
  - Preserve features across levels.
  - Modify the internal structure of the multiresolution representation.

- Handle texture coordinates.
- Output the model.

These functionalities were and are our targets. While many more improvements could be made, we have made substantial progress on each of them.

1.5 Thesis Structure

The rest of the thesis is organized as the following:

- Chapter 2, a survey on the recent research on simplification and multiresolution modeling.
- Chapter 3, presents an overview of the QSlim simplification algorithm, which we integrated into our semiautomatic simplification tool.
- Chapter 4, a detailed description of constructing the multiresolution structure that is used in the system.
- Chapter 5, a detailed description of the set of editing tools.
- Chapter 6, presents the evaluation and results of the tools.
- Chapter 7, is the conclusion and some discussion of future work.

Chapter 2

Background and Related Work

There has been much research on model simplification in the past decade. More recently, attention has been directed at multiresolution modeling. In this chapter, we present some background information, and review some of the most important research advances in these areas.

2.1 Definitions and Terminologies

Before reviewing the research results, let us first formalize some terminology and definitions that will be frequently referred to in later discussions.

Model Representation

There are many ways to represent surface models in computer graphics. Two popular ones are polynomial patches and polygons [ZOR97]. Polynomial patches divide the surface into patches and approximate each patch with polynomial surfaces, such as spline surfaces. At the border of the patches different degrees of continuity can be maintained. This method allows compact representations of smooth surfaces using a small number of control points. Its main applications are approximating smooth surfaces with low curvature. The difficulty with such kind of representations is that it is computationally expensive. When representing fine detail, the number of control points may become very large and difficult to maintain.

Polygons gained wide popularity in representing surfaces in computer graphics because of their simplicity and power to encode detail. Polygons are easy and fast to render for rendering hardware. In particular, triangles are the only type of polygon used in many applications. This is because triangles are the simplest polygons, and any polygonal surface can be converted into triangles.

A polygonal model is a piecewise surface model consisting of a collection of vertices and faces. Let $M(V, F)$ denote the entire polygonal model. V is the collection of all the vertices and F is the collection of all the faces in M [GAR99]. Each face in F is a polygon made up of a number of vertices from V.

There are many ways to store M. One way is to store faces of M by

$$f = ((x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n,))$$

[HOF89], where f is a face and each triple is the geometric coordinates of one vertex in f. Such a structure has lots of redundancies, and updating one vertex requires updating each of its copies in each face corner. A more efficient structure uses pointers to vertices. Vertices are stored in a list with each vertex identified by a unique ID. Each face f contains three vertex IDs, and each pair of the vertices of f defines an edge. The set of model faces defines most of the topology and adjacency information, except for edges and vertices that do not belong to any face. In our context, only faces are visible elements. Thus we will ignore edges and vertices that do not belong to any face.

As polygonal models can be converted to models with only triangle faces, we will only consider models containing exclusively triangles. In the following text, *polygons* will refer to triangles only. The term *model* will refer specifically to triangle models unless otherwise stated.

Manifold and Non-Manifold Surfaces

In topological terms, an *n-manifold M* in E^m, where $m \geq n$ is a subspace that is locally homeomorphic to E^m, i.e., for every point p of M, there exists a neighborhood U of p that is homeomorphic to E^n [HOF89]. E^n is the n-dimensional Euclidean space. Homeomorphism is a bijective (one to one) mapping where every neighborhood is also a neighborhood after such mapping. In short, homeomorphism keeps two spaces topologically equivalent. We are specifically interested in 2-manifolds. A *2-manifold without boundary* is a manifold such that for any point p on the surface, it has a neighborhood that is homeomorphic to a disc. A *2-manifold with boundary* is a manifold that any point p on the surface has a neighborhood that is homeomorphic to a disc or a half disc. Later on, we will abbreviate 2-manifold with the term *manifold*.

Intuitively speaking, a manifold surface is more regular. Each edge of the surface has no more than two adjacent faces. And each vertex has a closed ring of adjacent faces, or a fan of adjacent faces if the vertex is on the boundary.

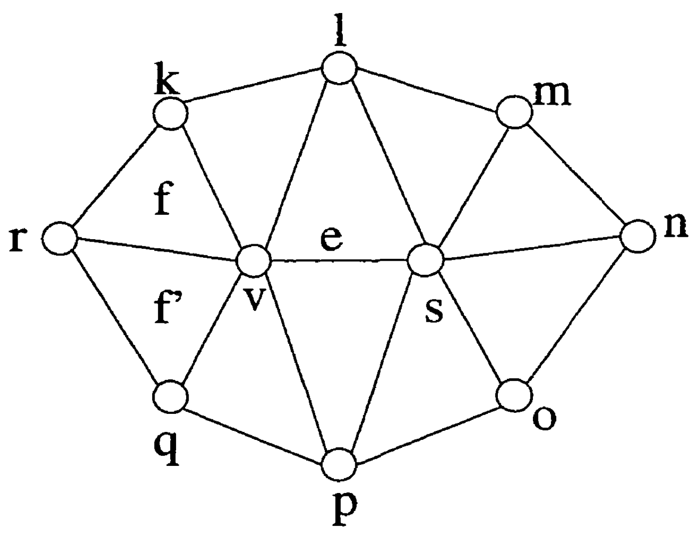


Figure 2.1: Adjacency.

Adjacency

For clarity of later discussions, we will here define the meanings of adjacency between different elements of the model. When two vertices are connected by an edge, we say they are adjacent to each other. If a vertex v is an endpoint of an edge e, we say e is an adjacent edge of v. When a vertex v is a corner vertex of a face f, we say f is an adjacent face of v. All adjacent faces of a vertex form a *face star* of the vertex. Similarly, all adjacent edges of a vertex form an *edge star* of the vertex, and all adjacent vertices of the vertex form a *vertex star*. If an edge e belongs to a face f, we say that they are adjacent to each other. An edge with only one adjacent face is called a *boundary* edge. Two edges are adjacent to each other if two they share one end point. However, if two edges share both end points, they are considered identical. Edge directions are not considered. If two faces share a common edge, they are adjacent to each other. If two faces share more than one common edge (i.e., the two faces share the same vertices) but with different normal directions, they are different. However, if their normals point to the same direction, the two faces are identical. Faces only sharing one vertex are not considered adjacent with each other. For example, in the surface shown in figure 2.1, e is an adjacent edge of v. f and f' are two adjacent faces of v. All faces around v form a face star of v, and vertices k, l, s, p, q, r are the vertex star of v. Edge rv is adjacent to faces f and f', and f is adjacent to f'. All edges residing at the outline of the surface are boundary edges.

Patches and Partitions

A *patch* is a group of connected elements of the model surface. A *face patch* of a model is defined as a collection of connected faces P. By "connected" we mean that if starting from an arbitrary face in P and allowing walking to the adjacent faces within P, all faces of P can be visited. Thus each face patch occupies a contiguous part the model surface. For a number of face patches of a model, if they do not share faces with each other, and their union covers all model faces, the face patches define a *face partition* of the model surface. Similarly, we can define *vertex patches* and *vertex partitions*. The difference with their face counterparts is that they are defined on the vertices. If a set of patches do not share elements with each other but their union does not cover the entire model, they implicitly define a partition of the model surface, because the remaining vertices or faces are implicitly one or more patches.

2.2 Model Simplification

Model simplification deals with the problem of reducing the number of polygons in highly detailed polygonal models and, at the same time, preserving visual or geometric similarity with the original model. Generally speaking, each simplification algorithm contains the following components:

1. The basic simplification operations that actually reduce vertices or faces.
2. The mechanism that preserves the similarity between the original model and the simplified model.
3. The evaluation of the difference (or error) between the simplified and the original models.

The most common difference between simplification algorithms lies in the first component. For example, some algorithms work by deleting vertices, while others may do it by contracting edges. However, the second component is the most crucial part of a simplification algorithm. It is here where the quality of simplification is influenced the most. The third component does not appear in all simplification algorithms. Some algorithms use the error to guide the second component, thus combining the two components. Others may use other kinds of heuristics, and do not explicitly measure error. There are, however, tools that can calculate the error of the simplified models and are independent of the simplification algorithms [CIG98]. Such tools provide a way to evaluate different simplification algorithms in a uniform context. In the following review of simplification algorithms, we will classify the different techniques mainly based on the first component. The classification borrows some ideas from [HEC97] but with some changes. Within each of the major categories, algorithms are further differentiated using the second component.

Algorithm Characteristics

An important property of simplification algorithms is the direction it works in. If the starting point is the original model and the algorithm keeps working toward the simplified model, it is called a *decimation* method. If the algorithm starts from a coarse representation and iteratively refines it, the method is a *refinement* method. The termination condition usually is a threshold which could be the desired number of vertices or faces, or the largest error that is acceptable between the simplified and the original.

Some algorithms work by applying operations that only affect a local area of the model. For example, an algorithm may delete one vertex at a time and patch the hole. Such simplifications use *local operations.* Other algorithms, however, apply simplification on the whole model, and there is no distinct iteration of local simplification operations.

Simplification algorithms also differ in the assumptions they make about the input model. A significant number of algorithms rely on the model surface being manifold. For example, they will assume each edge has only two adjacent faces and use this assumption in the algorithm. Other algorithms accept more general surfaces.

Simplification algorithms also differ in abilities to handle model properties, such as preserving model topology. Some algorithms not only can handle the geometric shape of simplified models, they also can treat texture coordinates, colors or normals. These properties are important for the visual quality of the simplified model.

In the next few subsections, we will review the major simplification techniques.

2.2.1 Vertex Clustering

Vertex clustering techniques work by grouping vertices of a model into cells. For each cell, a representative vertex is computed and is used to replace all other vertices in the cell. These types of algorithms are fast compared to others, but they generally cannot achieve very high quality simplifications.

Rossignac and Borrel

The vertex clustering technique developed by Rossignac and Borrel [ROS93] [ROS97] uniformly divides the model's bounding box into grid cells based on geometric proximity. The size of the cells is decided by how much the model needs to be simplified. Each vertex is assigned a weight value which reflects the perceptual importance of the vertex. The value is estimated by taking the inverse of the maximum angle between all pairs of incident edges on the vertex. For every cell, a representative vertex is calculated by merging all the vertices in the cell. The calculation of the merged vertex could be selected from several approaches: it could be the mean of the vertices in the cell, the weighted sum of the these vertices, or the vertex with the greatest weight. All original vertices are replaced by its cell's representative vertex. Then the model's faces are reconstructed using representative vertices according to the original faces.

This technique is quite general, because it only depends on the vertex set and could work with any type of surface. The original model's topology is not preserved. Thus disjoint parts might be merged and holes on the original model could close.

Low and Tan

The method developed by Low and Tan [LOW97] offers an improvement of Rossignac and Borrel's algorithm. An improved weighting strategy is used, where instead of using uniform grid cells, cells are centered around weighted vertices. The vertices are sorted in non-increasing order according to the weight values assigned to them. The top vertex in the list (the vertex with the highest weighting) is used as the center of a new cell. All the vertices in the cell are replaced by the representative vertex and removed from the vertex list. This process is repeated for the rest of the vertices in the sorted list. To prevent cases where a sliver face is widened by clustering, vertices are checked if they fall into multiple cells. If so, the cell with the closest center is chosen. To further improve the quality of the simplified model, elongated parts that are reduced to edges are rendered as thick edges with dynamic normals.

Brodsky

Both of the previous two algorithms can be viewed as decimation methods, because they start from the original model and end at the simplified model. The R-Simp algorithm [BRO99] is a refinement method using vertex clustering. Initially it treats the whole model as an entire cell and iteratively subdivides it into 2, 4 or 8 sub-cells depending on the nature of model surfaces contained in a cell. Each subdivision is decided by the surface *curvedness*, defined as the sum of all the face normals in the cell. The heuristic is based on the observation that where the curvature is high, the length of the sum of the normals tends to be small, and the cells are divided by planes splitting high curvature faces. Simplification terminates when enough cells have been created, and each cell is replaced by a representative vertex calculated by optimizing the quadric error metric [GAR99].

2.2.2 Face Merging

Face merging algorithms simplify models by merging roughly coplanar faces into sets, and each set of faces is replaced by fewer larger polygons. This kind of algorithms are decimation methods.

Hinker and Hanson

The algorithm developed by Hinker and Hanson [HIN93] has a time cost of $O(n \log n)$, where n is the input size. First, the model faces are grouped into roughly coplanar sets. Before each set is created, a representative normal is selected. A polygon is added to the set if its normal falls within a user-specified error range around the representative normal. After a polygon is added, the representative normal is adjusted to be the average of the polygon normals in the set, in order to avoid missing candidate faces when using fixed representative normals. After coplanar sets are built, a segment list is created. A segment is a list of boundary points that identifies the boundary between the polygon sets. Connected segments are joined together, and new polygons are constructed from them. Finally the new polygons are re-triangulated. This algorithm seems to work best on surfaces with zero curvature in at least one direction, such as cylinders and cones [GAR99a].

Kalvin and Taylor

Kalvin and Taylor developed a face merging method using "superfaces" [KAL96]. A superface is a "nonplanar polygon" defined by the boundary of a face patch. The algorithm works in three phases. In the first phase, superfaces are created. Starting from a random seed face that has not been visited, face sets are grown by adding more faces to it. A set of merging rules decides whether new faces are added. The rules test whether a face is nearly coplanar with faces in the face set and whether adding it causes irregular patches. The second phase straightens the boundaries of the constructed superfaces. Edges between adjacent superfaces are replaced by straight edges which are called "superedges". To avoid over-simplification, the superedges are recursively split until every boundary vertex is within a bounded distance to an adjacent superface. The last phase is the triangulation phase where the superface edges are projected onto the approximation plane. The projected 2D polygons are decomposed into star polygons and are triangulated.

2.2.3 Simplification Envelopes

Varshney et al.

The algorithm developed by Amitabh Varshney et al. [VAR95] [COH96] uses offset surfaces to guarantee an error bound. An offset surface is parallel to the original model surface with a constant distance. At the beginning, two offset surfaces are created inside and outside the original model with a distance ϵ, so that the model surface is contained in a surface envelope. If the envelope intersects with itself, ϵ is reduced to avoid it. After the envelope has been set up, the model is simplified. In [VAR95], a global method is applied by creating a set of candidate triangles. A candidate triangle is defined by three original vertices that are *visible* to each other within the envelope. This ensures that the candidate triangles do not intersect with the envelope. The candidate triangles are then sorted in decreasing order by how many original triangles they cover. Then the model is reconstructed by adding the candidate triangles in order, while deleting original triangles that overlaps with the newly added candidate triangle. Any holes in the process are patched with triangles. In [COH96], a local algorithm is added in complement to the global one, because the global algorithm's complexity grows too fast and is not practical when model becomes big. The local algorithm is a vertex decimation method constrained by the envelope. Compared to the global algorithm, the local one is faster and more robust. This algorithm only accepts models with manifold surfaces.

2.2.4 Re-tiling Surfaces

Turk

Turk adopts a method of re-tiling [TUR92] surfaces suitable for curved surfaces. At first, a user-defined number of points are randomly distributed across the model surface. To make the distribution even, each point is propelled by its neighbors, and the points drift on the model surface until all the points reach a stable state. After this, the points are triangulated and the original vertices are discarded. The triangulation process first creates a *mutual tessellation*, which is an intermediate polygonal surface that triangulates both the original vertices and the new points. The mutual tessellation contains all the original edges.

Then the old vertices are deleted from the mutual tessellation. When an old vertex v is to be removed, the face star around v is found and projected onto a plane, which results in a polygon containing v. Then v is removed and the polygon is triangulated, the result of which is mapped back onto the model. Topology is checked and preserved in the simplified model. The algorithm also detects curvature in the model surface. Where the curvature is high, the density of the points is increased. This will enhance the preservation of detail at high curvature area. The algorithm also maintains mapping of the original vertices and the simplified model, which facilitates interpolation between distinct levels of detail. This algorithm works only with manifold surfaces.

2.2.5 Vertex Decimation

Vertex decimation methods typically delete one vertex at each iteration, and the resulting hole on the surface is patched by new triangulations. These algorithms and later introduced edge and face decimation methods are similar in that they all iteratively apply local decimation operations.

Schroeder et al.

The algorithm developed by Schroeder et al. [SCH92] decimates vertices in multiple passes. It classifies the local geometry and topology around a vertex into five categories called: simple, complex, boundary, interior edge and corner edges. Simple vertices are surrounded by face stars and are manifold. Complex vertices have adjacent surfaces that are not manifold. Vertices lying on boundaries are called boundary vertices. If a vertex is shared by two feature edges, it is an interior edge vertex. Here feature edges are defined as edges with the dihedral angle between two adjacent triangles greater than a specified angle. If three or more feature edges share the same vertex, then it is a corner edge vertex. All kinds of vertices are candidate vertices except complex vertices. The decimation of a vertex is decided by the distance of the vertex to the average plane of surrounding triangles. If the distance is within a specified value, the vertex is removed. If the vertex is on a boundary or feature edge, the distance to the line that connects the two closest neighbors of the vertex is used. After the vertex is removed, the hole is triangulated by a recursive loop splitting procedure.

2.2.6 Edge Decimation

Edge decimation or edge contraction methods simplify models by deleting edges. After an edge is deleted, it is replaced by a target vertex. The surrounding surface of the edge is updated using the new vertex.

Hoppe et al.

Hoppe et al. developed a mesh optimization algorithm [HOP93] which minimizes an energy function. Initially a set of data points X are taken from the input model. These points are used for measuring the error of the simplified mesh, and placing its vertices. The energy function is defined as

$$E = E_{dist} + E_{rep} + E_{spring}$$

The first component measures the error of the simplified model from the original model. It is defined as the sum of squared distances from the data points X to the simplified mesh. The second term is proportional to the number of vertices in the mesh. The last component places a spring with rest length zero on each of the edge.

Minimization of the energy function is achieved in two nested minimization sub-problems. The inner minimization is continuous and optimizes geometry, in which vertex positions are updated to minimize the first and third term of the energy function. The outer minimization is discrete and optimizes topology. Three edge operations, namely *edge collapse*, *edge swap* and *edge split* are tried in turn to attempt to minimize the energy function. With inner and outer minimizations, the model is simplified and adjusted to fit the original model. Topology is preserved during simplification. This algorithm is slow, but is able to produce very good results for various models.

In later work, the mesh optimization method is modified to the *progressive meshes* algorithm [HOP96]. The detail of this algorithm is introduced later. The major improvements of progress meshes are the speed of simplification and the multiresolution representation.

Garland

The QSlim algorithm [GAR99] developed by Michael Garland is basically an edge contraction algorithm. But it is also extended to merge vertex pairs which are disjoint and close enough to each other so as to merge disjoint parts. QSlim offers a very compact representation of error metric, called *quadrics*, for face patches on the original model. Given a vertex position, quadrics can be used to calculate the error of the vertex to the patch of faces associated with the quadric. In addition, a quadric can be used to get the optimized position of a vertex. A quadric is initially derived from the formula which calculates the distance of a vertex to a plane. Each quadric consists of a matrix, a vector and a scalar. An important property of quadrics is that merging two face patches corresponds to the addition of the associated quadrics. With the quadric representation, there is no need to maintain a list of original faces associated with each vertex during simplification. Instead, each vertex has a quadric and the quadric is enough to calculate the error of the vertex and optimize the vertex's position. When an edge is contracted, the target vertex's quadric is the sum of the quadrics of the two end points, and the target vertex's optimized position is calculated from its quadric. The algorithm itself is simple. All candidate edges or vertex pairs are first put into a priority queue, sorted by the error that will be introduce if the edge is contracted. Then the edge with the least error is contracted first. The two end points of the edge is replaced by the target vertex and all affected edges is updated. This process repeats until the specified number of faces is reached or the edge queue is empty.

With some extension, the quadric error metric can be generalized to accommodate not only the position coordinates, but also vertex attributes such as color, normal or texture co-ordinates. The algorithm remains the same. QSlim is the simplification algorithm currently used in our semiautomatic simplification implementation.

2.2.7 Triangle Decimation

Triangle decimation algorithms remove one triangle at a time. With each removal the hole is re-triangulated or a new point is inserted.

Hamann

The method developed by Bernd Hamann [HAM94] is a triangle decimation algorithm. First it weights all the faces depending on the curvature at each vertex of the face and the angles of the face. The curvature weight is used to preserve portions of the surface with high curvature. The angle weights are used to penalize sliver faces. Then the triangle with the minimal weight is chosen for removal. After a triangle T has been removed, a new vertex will be inserted for better triangulation of the hole. The position of the new vertex is obtained by first choosing a point p in the projected local surface of T, then project p in the direction of T's normal onto a quadratic surface that approximates the model surface before removing T. To triangulate the hole, initially triangles are created centering around the new vertex. Then a series of edge swaps are applied for better shaped triangles. After a triangle removal, affected triangles' weights are updated. The simplification process continues until the desired number of vertices is reached.

2.2.8 Summary of Simplification Algorithms

Simplification algorithms are generally evaluated by their speed of simplification and quality of results. Quality is usually evaluated by the error between the original model and the simplified model. A complete and fair comparison of these algorithms should be done in the same environment and using the same input. Evaluation of the error should also be provided by a uniform measurement. However, these are often hard to do. Here we gave a general description of the speed and quality of the algorithms. More detailed comparisons could be found in [BRO99]. Due to lack of data, not all algorithms introduced in previous sections are covered.

Vertex clustering algorithms are generally the fastest among all the simplification algorithms, but usually produce relatively poor quality results. The vertex clustering algorithm in [ROS93] is the fastest simplification algorithm. Other simplification algorithms vary in speed, and do not necessarily relate with their category. QSlim [GAR99] is a fast algorithm with fairly good quality. It is a little slower than vertex clustering algorithms but generally faster than other algorithms. Progressive meshes [HOP96] and simplification envelopes [COH96] come next in speed and with better quality. The mesh optimization [HOP93] method is very slow but produces the best quality results.

2.3 Simplification of Attributes

Besides geometric position, a vertex often has more attributes, such as texture coordinates, vertex normals and colors. The model shape has been the major focus of interest in simplification algorithms. However, these other attributes are also important in the model's appearance. Some simplification algorithms have started taking them into consideration.

Cohen et al.

Cohen et al. [COH98]'s appearance-preserving simplification algorithm decouples the vertex positions from its attributes. Surface attributes are converted to color and normal textures. During simplification, texture deviations are considered in addition to geometry. The algorithm first divides the model surface into patches, and each patch is parameterized using a spring system and mapped to a rectangular region in the texture space. With this parameterization, each triangle in the model is scanned and assigned texture coordinates.

Texture deviation is defined as the distance between two vertices having the same texture coordinates. The texture deviation of a triangle is the maximum of all such distances of its vertices. In practice, the deviation is approximated by axis aligned bounding boxes, and these boxes are enlarged during simplification to ensure that they cover the corresponding original vertices. Texture deviations are integrated with a geometry metric to affect the order of simplification such as edge collapses.

Garland

The quadric error metric in the QSlim algorithm can be generalized and extended to integrate vertex attributes [GAR99]. The vertex attributes are treated in the same way as the position coordinates, thus placing the vertices into higher dimension spaces. All the previous concepts remain the same, except that their calculation is now in higher dimensional spaces, instead of the previous 3-dimensional space. Other than such an extension, the overall algorithm remains the same.

Hoppe

Based on Garland's quadric error metric, Hoppe designed a new quadric error metric that handles surface attributes [HOP99]. Instead of extending the quadric error metric to higher abstract dimensions, another quadric term is introduced specifically for simplifying attributes. The introduced quadric is added to the original geometric quadric resulting in a new quadric. Compared to the extended quadric in [GAR99], the new quadric error metric requires fewer coefficients, and is more precise in simplifying the attributes. To handle multiple values for a single attribute at each vertex (such as different texture coordinates for each face), *wedges* are introduced. Each wedge is assigned a quadric for a single value for each attribute. During simplification, each vertex attribute is optimized from wedge quadrics similar to optimizing geometrical positions.

2.4 Multiresolution Modeling

Model simplification is used to generate simplified versions of the original model with different amounts of detail. These levels of detail could be used to meet different requirements. However, instead of just a collection of simplified models, more complicated structures, called multiresolution models, can be designed to represent a model at various resolutions. Multiresolution models also provide more capability for editing the model, for example, changing a coarse level affects the fine levels in similar ways. The problem of multiresolution modeling could be broken into two sub-topics: the representation or structure of the multiresolution model and the manipulation of the multiresolution model.

2.4.1 Discrete Representation

The most simple and intuitive way of representing multiresolution models is to create a collection of models at different resolutions and store them as separate models. Each of these resolutions is used for a range of detail requirements. When switching between d-ifferent resolutions, the complete model is replaced. The advantage with this approach is its simplicity. But the disadvantage is that memory constraints could limit the number of distinct levels can be kept, especially for large and complex models. The transition between different resolutions is usually discontinuous, which results in a visually distracting popping effect. Some algorithms have been developed to interpolate between different resolutions to make the transition smooth [GAR99a].

2.4.2 Multiresolution Analysis

Lounsbery introduced a method of applying wavelets to analyze multiresolution models [DER93]. It decomposes the model into a base mesh with the same topology as the original model. Then the the base mesh is subdivided repetitively. For each subdivision, a triangle is split into four sub-triangles. The new vertices inserted are treated as detail and detail coefficients are computed. The multiresolution model is the base mesh with the subsequent detail. This method is mathematically elegant. But it requires the original model to have subdivision connectivity. Eck et al. [ECK95] developed a method to convert arbitrary surfaces to surfaces with subdivision connectivity, so that multiresolution analysis could be applied. But this conversion introduces some error into the model. Multiresolution analysis is also unable to resolve creases in the model very well.

2.4.3 Progressive Meshes

Hoppe [HOP96] developed the progressive mesh representation of multiresolution model-s. The mesh optimization method [HOP93] is modified such that only the edge collapse operations are kept. This makes the simplification algorithm work just like a typical edge decimation algorithm, in which candidate edges are sorted in a priority queue and collapsed in order. The energy function is also modified to eliminate the E_{rep} term. Two new terms E_{scalar} and E_{disc} are added which are used for interpolating vertices' scalar attributes and preserving creases respectively. When interpolating scalar attributes, the scalar energy term E_{scalar} is minimized and the optimal attribute value is found. To preserve discontinuity curves, it first checks if the topology of the curve is modified by a set of rules. If the topology of the curve is modified, the edge collapse is disabled or penalized. Otherwise, additional points are sampled on the curve, and the E_{disc} term is minimize in the similar way as E_{dist} in the energy function.

The simplification algorithm produces a series of edge collapses which reduces the original mesh M to a simplified mesh M^0 along with the edge collapses

$$ecol_{n-1} \rightarrow ecol_{n-2} \rightarrow \cdots \rightarrow ecol_0$$

The inverse of each edge collapse is a vertex split. Reversing the edge collapse sequence, the mesh could be represented as a simplified mesh M^0 along with a series of vertex splits

$$vsplit_0 \rightarrow vsplit_1 \rightarrow \cdots \rightarrow vsplit_{n-1}$$

Applying the vertex splits to M^0 will recover the original mesh. This progressive mesh representation can be used to progressively transmit a multiresolution mesh. The representation also allows selective refinement based some conditions, such as the viewing frustum.

2.4.4 Vertex Tree

The vertex tree structure can be generated naturally from some simplification algorithms such as vertex clustering, edge decimation and face decimation methods. A vertex tree is a tree structure consisting of vertices. It has a root node which contains several children. These children have a number of children of their own. Each node can only have one parent. The single node without a parent is called the root of the vertex tree. Any simplification algorithm that works by collapsing a number of vertices and replacing them with a representative vertex could be modified easily to create a vertex tree. Currently, vertex trees find application mostly in realtime view dependent simplification, where the model is dynamically simplified based on the current view point. However, we will use it for a different purpose which will be explained in later chapters.

Luebke and Erikson

In the hierarchical dynamic simplification (HDS) method developed by Luebke and Erikson [LUE97], the vertex tree is used for view-dependent dynamic simplification. The algorithm aims at dynamically simplifying complex CAD models depending on the viewpoint of the user. A vertex clustering algorithm is applied that creates the vertex tree in a preprocessing phase. Each node contains a number of descendent nodes and related triangles. The nodes are classified as *active* and *inactive* nodes. Active nodes are located at the top portion of the tree and form a active tree, while inactive nodes occupy the lower portion of the tree. The active nodes located at the boundary between the active and inactive nodes are called *boundary* nodes. The boundary nodes form a cut through the tree. A list of triangles called the active triangle list is maintained, which contains all currently visible faces. Boundary nodes are expanded and replaced by their children when more detail is needed. A number of new triangles will be added to the active triangle list after an expansion. If less detail is needed, boundary nodes are collapsed and some triangles are deleted from the active triangle list. Whether to collapse or expand nodes or which nodes are chosen depends on the nodes' screen space error. Each node has a bounding sphere containing all the descendents of the node, and the sphere is projected onto the screen. If the screen space occupied by the sphere of a node is below an error bound, the node will be collapsed. Otherwise it will be expanded. Thus the view of the model will dynamically adjust its detail at various parts according to how much detail is needed from the current view point. Besides the screen space error, the model silhouette is also detected and nodes potentially on the silhouette are tested with a tighter screen space error. This improves the shape of the silhouette from the current view point. Several optimizations are applied, which includes exploiting temporal coherence, culling invisible nodes and parallelization.

Xia and Varshney

The multiresolution structure developed by Xia and Varshney [XIA96] is a special vertex tree. The algorithm extends the edge collapses in progressive meshes and creates a *merge tree* which is essentially a binary vertex tree. For each edge collapse, the vertex that is merged into the other vertex is called the child and the other vertex is called the parent. A series of edge collapses are applied and the parent-child relationships are established. To prevent merges that causes face fold-overs, a set of merging dependencies are enforced.

The coarsest level model is used to initialize the vertex display list and the face display list. Then the vertex and face list is updated by adaptive refinement or collapse based on image space error. This process is similar to that of the Luebke's approach. When updating the vertex display list, merging dependencies are taken into account to avoid fold-overs.

Hoppe

Similar to Xia and Varshney's work, Hoppe [HOP97] extended the progressive mesh representation to allow view dependent selective refinement. The structure is a vertex hierarchy with explicit parent-child relationship stored. Two operations, *vsplit* and *ecol* are defined which split an edge or collapse an edge respectively. Some preconditions are defined as to whether a vsplit and ecol is legal. Three refinement criteria are established which are based on the current view frustum, the surface orientation and the screen-space geometric error. If all of the three refinement requirements are met then the vertex is split. Otherwise it is collapsed.

Gueziec et al.

The structure developed by Gueziec et al. [GUE98] does not create the vertex tree explicitly. But it is essentially a vertex tree structure as we have described. The structure is used for interactive navigation of LOD and progressive transmission. The algorithm assumes the input surface is manifold. An edge collapse algorithm is used to simplify the input model. At first, the model is simplified in a series of edge collapses. For each edge contraction, one vertex is kept which is colored red, and the other is merged into the red vertex and is colored blue. The red vertex becomes the representative vertex of the blue vertex. The red-blue coloring is the similar as the parent-child relations in [XIA96]. A vertex representative array is maintained which records the representatives of all the original vertices. Each vertex is also associated with a level value which is incremented depending on the edge neighbors during simplification. After simplification, the order of edge collapses is built into a directed acyclic graph. Navigating through the LODs is achieved by replacing the vertex index array with the original or the representative vertex index. If the original index is used, it is a split. Otherwise it is a collapse. The algorithm also outputs a progressive representation which transmits the levels through batches of triangles and vertices, along with the vertex representative array.

2.4.5 Multiresolution Parameterization

Multiresolution adaptive parameterization of surfaces (MAPS) developed by Lee et al. [LEE98] constructs a smooth parameterization of the original mesh over a base domain. The parameterization allows applications such as remeshing the original mesh into meshes with subdivision connectivity. The algorithm uses a vertex decimation simplification algorithm to reduce the original mesh into a base mesh. The hole produced by removing vertices is flattened using conformal mapping and triangulated by constraint Delauney triangulation. During simplification, a bijection Π from the original mesh M^L to the base mesh M^0 is maintained. For each simplification step M^l to M^{l-1}, barycentric coordinates are computed for each vertex in the original mesh according to M^{l-1}. This process continues until the base mesh M^0 is reached and the original vertices are all mapped to the triangles in M^0. Once the parameterization is finished, it can be used to remesh the original mesh into a subdivision connectivity mesh. This is achieved by subdividing the base mesh and mapping the vertices created from subdivision back to the original mesh. For any vertex p, first find the base triangle that contains p, from which the original triangle that contains p can be found. Then the mapping of p from the base mesh to the original mesh can be calculated by Π^{-1} which is the inverse mapping of Π. To smooth subdivision across base domain boundaries, the base domain is first smoothed using modified Loop subdivision which maps the base mesh to itself. Then the inverse mapping Π^{-1} is applied resulting in smoothed remeshing. When subdividing the base mesh, uniform subdivision could be quite inefficient, which leads to adaptive subdivision. Each triangle to be subdivided is associated with an error value. Since each triangle has a set of original vertices mapped onto it, the error is defined as the maximum of the distances of the vertices to the triangle. When the error is below a threshold, the triangle is not further subdivided.

2.5 Multiresolution Editing

The multiresolution representations presented so far allow displaying, transmission and navigation of the multiple levels of detail. However, sometimes we want to edit the multiresolution models. Editing a multiresolution model is different from traditional modeling in that we edit the model at multiple levels of detail. Multiresolution modeling offers many options that traditional modeling cannot provide. For example, after we have edited at a certain level, we can propagate the change to other levels, providing user control of scale. Or we can take surface patches extracted from different levels and combine them together, so that the resulting model has a desired distribution of detail.

Cignoni et al.

The *Zeta* tool [CIG98a] implemented by Cignoni et al. proposes a data structure and algorithm that enables a user to interactively refine or simplify a user-specified region on models with manifold surfaces. The algorithm introduces a concept called the *lifetime* of a face. Each face is associated with two errors ε_{birth} and ε_{death}. ε_{birth} is the mesh error when the face is first introduced into the mesh and ε_{death} is the mesh error when the face is deleted. The interval $[\varepsilon_{birth}, \varepsilon_{death}]$ is the lifetime of a face. The key data structure is called a *packed facet-edge* denoted as a *pfe*. A pfe is a directed edge attached with all the adjacent faces that ever appeared during simplification. Since edges have directions, each original edge on the model is divided into two opposite edges, and each of the two opposite edges takes responsibility for one side of the manifold surface. Within a pfe, the faces' life intervals are disjoint with each other. A pfe and its faces have pointers to other pfes which allows traversal of all pfes. Pfes are constructed from a vertex decimation algorithm. When a vertex is removed and the hole is retriangulated, new triangles as well as the old ones are all added to their edges' pfes. The entire model is not stored explicitly, but implicitly defined by the pfes. The multiresolution representation consists of a vertex list, a face list and the collection of pfes.

Given an error ε, extracting a mesh corresponding to error ε is achieved by traversing pfes starting from an initial seed pfe. For each visited pfe, one of its faces is selected whose lifetime contains ε. Since faces within a pfe have disjoint lifetimes, the selection is unique. After a mesh level has been constructed, the user can select a region by defining a focal point and a radius. This region is to be selectively refined or simplified. The user provides two error values ε_f and ε_r, which decide the error at the focal point and at the boundary of the region. The user can also specify a distribution function between the two errors. Given these information, the selected region is reconstructed in the similar way as constructing the mesh, only that the error constraints follow the new specifications. The order of traversing pfes are constrained by some conditions so as to avoid inconsistent face expansion.

Zorin et al.

The interactive multiresolution mesh editing algorithm developed by Zorin et al. [ZOR97] uses subdivision to achieve multiresolution representation. The multiresolution model is first constructed from an analysis process followed by a synthesis process. Analysis creates a coarse mesh from a fine mesh. A version of the Taubin [TAU95] smoothing filter is used to enhance the quality of the coarse representation. Synthesis refines the coarse mesh and builds the fine mesh. In the synthesis process, Loop's subdivision scheme and local frames are used for propagation in the fine direction. To prevent the exponential storage space and processing time resulting from uniform subdivision, the algorithm is modified into an adaptive and a local version. Adaptive analysis and synthesis are used to cut off further subdivision where the difference between the fine level and the coarse level is below a threshold. As user edits are typically restricted to a local area of the model, local versions of analysis and synthesis are used to update only the triangles that are affected by the editing. With the analysis and synthesis process, user edits can be propagated both in the coarse direction and in the fine direction. The algorithm also uses adaptive rendering to display the model, where the children of a subdivided triangle are only drawn when considered necessary, and larger triangles are rendered whenever possible and thus increasing rendering speed.

Kobbelt et al.

The multiresolution modeling algorithm developed by Kobbelt et al. [KOB98] is capable of editing arbitrary meshes. The algorithm simplifies the original model using a decimation algorithm, and a hierarchy is built from it. For multiresolution editing, the algorithm uses smoothing and local frames. Smoothing a mesh is achieved by applying a discrete fairing algorithm called the *umbrella algorithm*. Local frames are built based on faces, where each triangle and its three adjacent triangles are approximated by a quadratic surface. Distance vectors to the approximation surface is used as the detail coefficients. Editing is a multi-level smoothing process. It is defined as a V-shaped process $\Phi_i = \Psi P \Phi_{i-1} R \Psi$, applied from right to left. Ψ is the smoothing operator which smoothes the current mesh. R is the decimation operator which decimates the mesh to the next coarse level. Φ_{i-1} is a recursive call to smooth the next coarse level. R is the downhill part of the V-shaped recursion. It continues until the coarsest level is reached. The operator P re-inserts the decimated vertices and the final smoothing operator Ψ eliminates noise. In interactive modeling, however, the multi-level smoothing starts directly from the coarsest level and omits the pre-smoothing phase. The mesh is reconstructed each time after an editing operation is applied. This eliminates dependency on the history of operations. When the user applies a modification to the mesh, first a boundary is designated to define the scale of influence. A handle is defined within the scale which is a strip of triangles. The handle is controlled by the user and all vertices on the handle are transformed in the same way. The rest of the vertices within the defined scale interpolate the fixed scale boundary and the controlled handle, and the fairing algorithm makes sure the interpolation is smooth.

2.6 What is Missing

In previous sections, we have surveyed research on simplification and multiresolution modeling. Many simplification algorithms have been designed. These algorithms are able to take a complex model as input, and produce a simplified model with much fewer faces and vertices. However, nearly all such simplification algorithms share some common disadvantages. One is that the algorithms offer limited control to the user. Typically, a user can only control the simplification process by supplying a set of parameters. Such control may not be enough in circumstances where complex control is required. Another disadvantage is that at low polygon counts, a human user would probably do a better job than the algorithms. Moreover, when a model is handed to a simplification algorithm, all meaning conveyed in the model will be lost. For example, a bunny model resembles a bunny only to a human, but to a simplification algorithm, it is no more than a collection of vertices and triangles.

The drawbacks with automatic simplification algorithms warrant a tool that offers the user more control and interactivity in the simplification process. Multiresolution editing can be applied in this respect. However, the existing multiresolution modelers do not fit very well because existing multiresolution modelers emphasize editing the original model with the help of coarse representations, instead of improving model simplification. These observations motivated us to develop the semiautomatic simplification tool.

Chapter 3

QSlim Overview

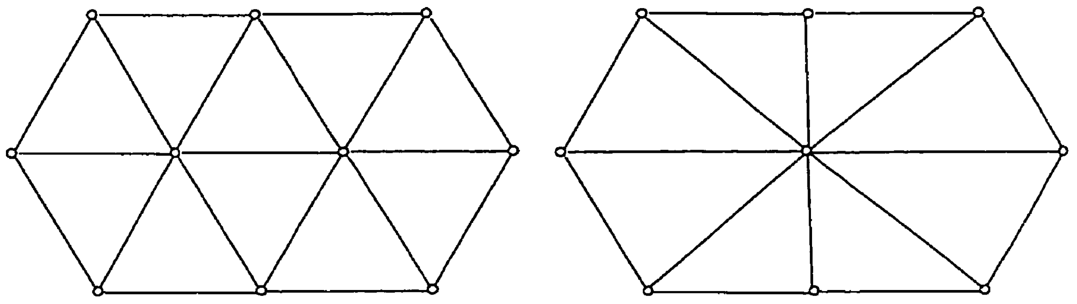
Our work utilizes a simplification algorithm to produce the multiresolution model. Many different simplification algorithms could be chosen for this. Initially we used the R-Simp algorithm which is a vertex clustering algorithm. Currently we are using the QSlim algorithm because of its good simplification quality, reasonable speed, and its use of decimation rather than refinement. In this chapter we will give a review of QSlim [GAR99] in more detail.

3.1 The Basic Algorithm

As first introduced in Chapter 2, QSlim is an edge decimation algorithm. The basic decimation operation is an edge contraction. At each step, the algorithm chooses a candidate edge and contracts it, reducing the vertices as well as the faces.

3.1.1 Edge Contraction

The contracted edge will be replaced by a single vertex v. This vertex is used later on to represent the edge. After an edge is contracted, the local neighborhood of the edge will be affected by the contraction and must be updated. Figure 3.1 shows a normal edge contraction. In the figure, the contraction causes the number of vertices to be reduced by one. The two adjacent faces of the edge degenerate to single edges and are eliminated. The number of faces that degenerate because of each edge contraction depends on the topology of the surface. It might be one if the edge is on the boundary, or more than two if the edge has more than two adjacent faces. The set of faces surrounding the contracted edge is the union of each endpoint's face star. Each face within this set must be updated using the new vertex. The edge queue (see "selecting edges for contraction" below) also needs to be updated. The edges surrounding the contracted edge is the union of each end point's edge



(a) Before edge contraction (b) After edge contraction

Figure 3.1: Contracting an edge.

star. Each edge in this set will use the new vertex v as an end point. Some edges will be merged and become redundant. Such edges must be deleted from the edge queue.

Vertex Placement

There are several ways to get the position of the replacement vertex v. We can merge one end point into the other. This approach has the advantage that there is no need to store the target vertex information separately. The progressive mesh representation [HOP96] using this approach can achieve similar size of representation as the original model.

However, when the best approximation is desired, optimized vertex placement should be used so that the error introduced by the edge contraction is minimized.

Selecting Edges for Contraction

The algorithm iteratively selects an edge and contracts it. The choice of the next edge to contract has a significant impact on the quality of the result. In QSlim, edges are sorted by the error their contractions will introduce. Each time we select an edge to contract, we pick the one that will introduce the least amount of error. After an edge contraction, the surrounding edges are affected, and their error must be recalculated, and consequently their position in the queue must be adjusted.

Vertex Pairs

Since edge contractions only merge connected components of a model, separate components will remain disjoint after simplification. This is a desired result in some situations. Sometimes better simplification can be achieved if some disjoint components are merged together, especially when they are close to each other. For example, a grid of disjoint but closely packed rectangles is better joined together and approximated by a larger rectangle, rather than contracting each individual rectangle. QSlim extended edge contraction to vertex pair contraction. Not only edges are candidates for contraction, vertex pairs that are within a distance threshold τ are also contracted. The τ is specified by the user and is usually small. For our work, however, we ignore τ and only use edges as candidates for contraction. However, such a feature could be added in future versions.

3.1.2 Simplification Process

The simplification algorithm works by iteratively selecting an edge, contracting it, and updating the local neighborhood affected by the contraction. Candidate edges are put into a priority queue, sorted by the error of the edges. Each time the top edge (introducing the least error) in the queue is popped off and contracted. The algorithm stops when the queue is empty or some predefined condition is met, such as the number of faces reaching a threshold. If the queue is empty, the whole model will be reduced to a single vertex.

The algorithm could be described in the following steps:

1. Initialize the model for simplification.
2. Select candidate edges for contraction and calculate the error for each edge.
3. Place all the edges in a priority queue sorted by edge error, with minimum error edge at the top.
4. Repeat until queue is empty or termination condition is met:
    (a) Pop an edge from the priority queue.
    (b) Contract the edge, replacing it with a new vertex.
    (c) Update affected neighboring edges and faces.

This algorithm in fact applies to other iterative edge decimation algorithms as well. The fundamental difference between these algorithms is how the edges are sorted, how the error is evaluated, and how target vertices are positioned. QSlim introduces *quadrics* that offer a compact representation of the error measurement and allow optimizing replacement vertex positions.

3.2 The Quadric

The major contribution of QSlim is the introduction of quadric based error metric. A quadric offers a compact representation of surface characteristics. Given a vertex position, it can evaluate the vertex's error relative to the associated surface and can also find out the optimal vertex position by minimizing the error. [1]

3.2.1 The Definition

The derivation of quadrics starts from a vertex's distance to a plane. Given a plane P

$$\mathbf{n}^T\mathbf{v} + d = 0$$

where $\mathbf{n}$ is the plane's normal vector, the squared distance of a vertex v to P is

$$D^2(\mathbf{v}) = (\mathbf{n}^T\mathbf{v} + d)^2$$

This could be further expanded as

$$\begin{aligned} D^2 &= (\mathbf{n}^T\mathbf{v} + d)^2 \\ &= (\mathbf{v}^T\mathbf{n} + d)(\mathbf{n}^T\mathbf{v} + d) \\ &= \mathbf{v}^T\mathbf{n}\mathbf{n}^T\mathbf{v} + 2d\mathbf{n}^T\mathbf{v} + d^2 \\ &= \mathbf{v}^T(\mathbf{n}\mathbf{n}^T)\mathbf{v} + 2(d\mathbf{n}^T)\mathbf{v} + d^2 \end{aligned}$$

Denoting $\mathbf{n} = [x, y, z]^T$, the term $\mathbf{n}\mathbf{n}^T$ is a matrix

$$\mathbf{n}\mathbf{n}^T = \begin{bmatrix} x^2 & xy & xz \\ xy & y^2 & yz \\ xz & yz & z^2 \end{bmatrix}$$

and $d\mathbf{n}^T$ is a vector. Denoting $\mathbf{n}\mathbf{n}^T$ as A, $d\mathbf{n}^T$ as $\mathbf{b}$, and the scalar d^2 as c, a *quadric* Q is defined as

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

Thus the squared distance of v to P is

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

where $Q(\mathbf{v})$ is the evaluation of Q with vertex v.

[1] In later discussions, vectors representing vertices are in boldface only in formulas, not in text.

3.2.2 The Error Metric

Since each face of a model defines a plane, the error of a vertex v to a face f is defined as the squared distance from v to the plane P defined by f, which could be calculated by evaluating the quadric of P. For a set of faces F, each face of F defines a quadric from its plane. The error from v to all the faces in F is defined as the sum of errors from v to each face of F:

$$E_F(\mathbf{v}) = \sum_i Q_i(\mathbf{v}) = \sum_i (\mathbf{n}_i^T \mathbf{v} + d_i)^2$$

where $E_F(\mathbf{v})$ denotes v's error to the face set F. It can be shown that quadrics have the property

$$Q_1(\mathbf{v}) + Q_2(\mathbf{v}) = (Q_1 + Q_2)(\mathbf{v})$$

where $Q_1 + Q_2 = (A_1 + A_2,\ \mathbf{b_1} + \mathbf{b_2},\ c_1 + c_2)$. Thus

$$E_F(\mathbf{v}) = \sum_i Q_i(\mathbf{v}) = (\sum_i Q_i)(\mathbf{v}) \tag{3.1}$$

This indicates that when evaluating the sum E_F, there is no need to calculate the error from v to each face in F. Instead, we calculate the sum of the faces' quadrics and the error could be evaluated from the quadric sum.

Each original model vertex v_i has a number of adjacent faces which is the face star of v_i. When contracting an edge $(v_i,\ v_j)$, both end points are replaced by a new vertex v. Thus v is associated with the union of v_i and v_j's face star. If a later contraction collapses an edge containing v, lets say $(v,\ v_k)$, and replace it with a new vertex v', v' will again inherit faces associated with v and v_k. As simplification goes on, associated face sets merge and become larger. If the whole model is simplified to a single vertex r, the associated face set of r will be all the faces in the model. Therefore, for any vertex v that appears during simplification, it has an associated set of faces, which we denote as $F(v)$. Considering equation 3.1, we can assign each vertex v a quadric

$$Q_v = \sum_{f \in F(v)} Q_f$$

which is the sum of quadrics of each face in $F(v)$.

Initially all the original vertices' quadrics are initialized by adding quadrics from their face stars. When contracting an edge $(v_i,\ v_j) \to v$, we calculate the quadric of v by

$$Q_v = Q_{v_i} + Q_{v_j}$$

By maintaining a quadric for each vertex, we eliminate the need for storing $F(v)$ explicitly. If the error for a particular vertex v is needed, it can be obtained by evaluating $Q_v(v)$.

Normalized Quadric

Suppose we have a triangle f with quadric Q_f. The total quadric of f is Q_f. If we subdivide f into 2 triangles, since the plane defined by f is the same, each sub-triangle also has a quadric Q_f. Now the total quadric of f is $2Q_f$. Clearly the total quadric is dependent on the tessellation. This is a problem when we are initializing the original vertices' quadrics. We would want the quadrics dependent only on the model geometry. Thus it is desirable to *normalize* the face quadrics such that they are independent of tessellation. In QSlim, a face's quadric is normalized by dividing the face into fragments. Each fragment has a quadric weighted by its area and assigned to a face vertex. The vertex adds the quadrics of all fragments assigned to it. In this way, the total quadric of a model would be independent of tessellation.

3.2.3 Calculating Optimal Vertex Position

With the error metric defined, finding the optimal vertex position means minimizing the error value. For a vertex v with quadric Q, its error is

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

Taking partial derivative for Q on each coordinate component of $\mathbf{v}$: (x, y, z), we have

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

Solving $\nabla Q(v) = 0$ yields the optimal vertex position $\mathbf{v} = -A^{-1}\mathbf{b}$. This position has the minimal error for the faces associated with Q. The optimal replacement vertex positions are calculated this way after an edge has been contracted.

3.2.4 The Extended Quadric

Sometimes a vertex has additional properties, such as texture coordinates, colors or normals, other than geometric positions. Additional attributes are treated in the same way as positions. For example, a vertex with RGB color values can be generalized to a vertex in 6-dimensional space. All previous concepts remain the same except that their implementation is now in higher dimensions. The squared distance from a vertex to a plane can be reduced to the same form $D^2 = \mathbf{v}^T A\mathbf{v} + 2\mathbf{b}^T\mathbf{v} + c$, where A is a symmetric matrix, $\mathbf{b}$ is a vector and c is a scalar value. Thus the quadric definitions are mostly the same.

With the quadric extended to allow more vertex attributes, the algorithm can treat additional vertex attributes with little modification. However, there are restrictions to the kind of attributes that can be included. The attribute values should be on a per vertex basis, i.e., each vertex should only have a single set of attributes. For geometrical positions, this is implicitly true because the vertex can only have one position in geometric space. But for other attributes such as texture coordinates and vertex normals, a vertex could have one set for each adjacent face. By "duplicating" vertices at the same geometric position but different texture or normal coordinates, QSlim can handle these problems to some degree. Even this approach has its limitations. For example, if two vertices have texture coordinates from the same texture map, a small difference between the texture coordinates means the vertices are also close to each other on the texture map. However, if the coordinates come from different texture maps, the distance calculated from the two coordinates does not make much sense, since even when the distance is very small, it does not mean the textures are more continuous. In Chapter 2, we have described *wedges* introduced by Hoppe [HOP99]. With such an extension, the problem can be fixed.

3.2.5 Constraints

Feature edges and boundaries may be important to the visual quality of a model. Preservation of boundary or feature edges is achieved by weighting quadrics of vertices on these edges, such that the error and position of the replacement vertex is biased toward the edge to be preserved. For an edge e that is to be preserved, an imaginary face for each adjacent face of e is inserted along e, and imaginary faces are vertical to their corresponding adjacent face (figure 3.2). The quadrics of the imaginary faces are weighted by a large factor and added to the initial quadrics of edge points of e. Thus when optimizing vertex positions, the result will bias toward the vertical faces because these faces have larger weights. In this way, the replacement vertices tend to be closer to the preserved edge and the edge shape is preserved.

3.3 Integrating Quadrics in Simplification

Given the quadric error metric, we can add detail to the simplification algorithm presented at the beginning of this chapter.

In the initialization phase, the quadric for each vertex is calculated by adding the normalized quadrics of its adjacent faces.

When initializing the edge queue, each edge is assigned a quadric by adding the quadrics

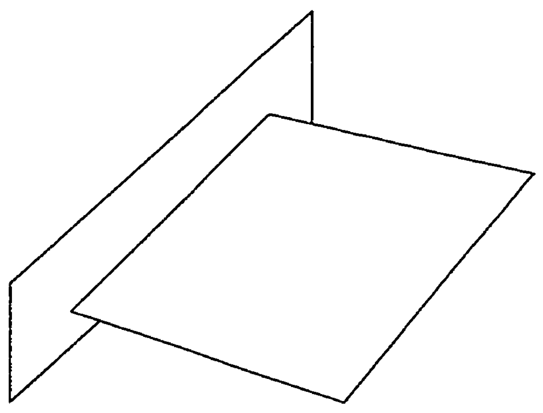


Figure 3.2: The vertical imaginary face is added for edge constraints.

for the vertices adjacent to the edge. The optimal replacement vertex position is calculated, and the error of an edge is evaluated by applying the optimal vertex position to its quadric. Then the edge is inserted to the edge queue sorted by its error.

For each edge contraction, a new vertex is created taking the optimal position calculated previously, and with its quadric being the quadric of the edge. All affected neighboring edges must be updated. The update includes recalculating each affected edge's quadric and error. The edge's position in the edge queue must be adjusted to preserve the order of the queue.

Chapter 4

Multiresolution Representation

In this chapter, we will introduce the multiresolution representation used in our system. This multiresolution structure is a vertex tree as introduced in Chapter 2. It is similar to those used in [XIA96] and [LUE97]. Many simplification algorithms are able to create a vertex tree naturally. Such algorithms typically map several vertices to one vertex during each simplification step. Edge decimation and vertex clustering algorithms are good examples of these. Because of the many-to-one mapping, a tree structure is suitable to record the simplification process.

4.1 Vertex Tree Terminology

Here we define the notations that we use frequently in our later discussions.

A *vertex tree* is a tree structure whose nodes are vertices in a certain space. For our discussion, the space is 3-dimensional geometrical space. Each *node* of the tree has zero or one *parent* node, and may have a number of *child* nodes. The connection between a parent and a child is called a *link*. If a node has no parent node, it is called the *root* of the tree.

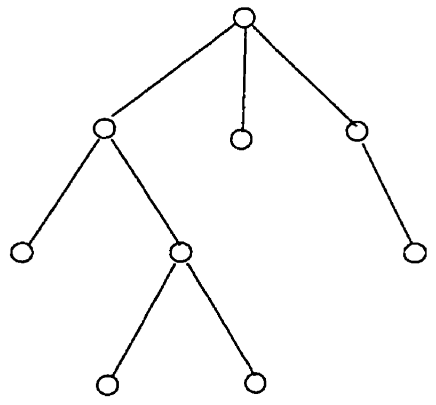
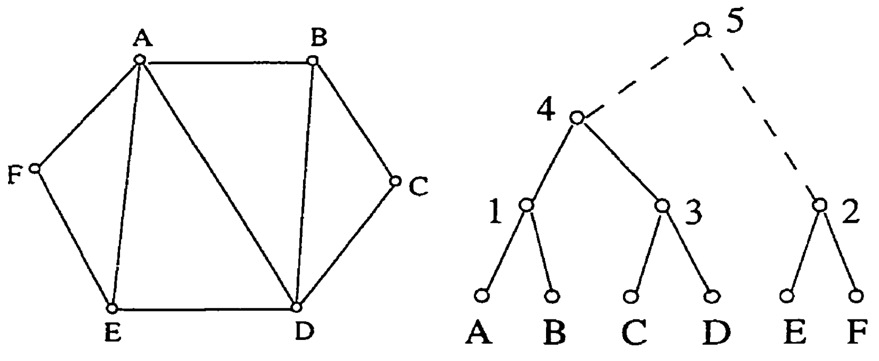


Figure 4.1: An example of a vertex tree.



(a) The faces to be simplified (b) The constructed vertex tree

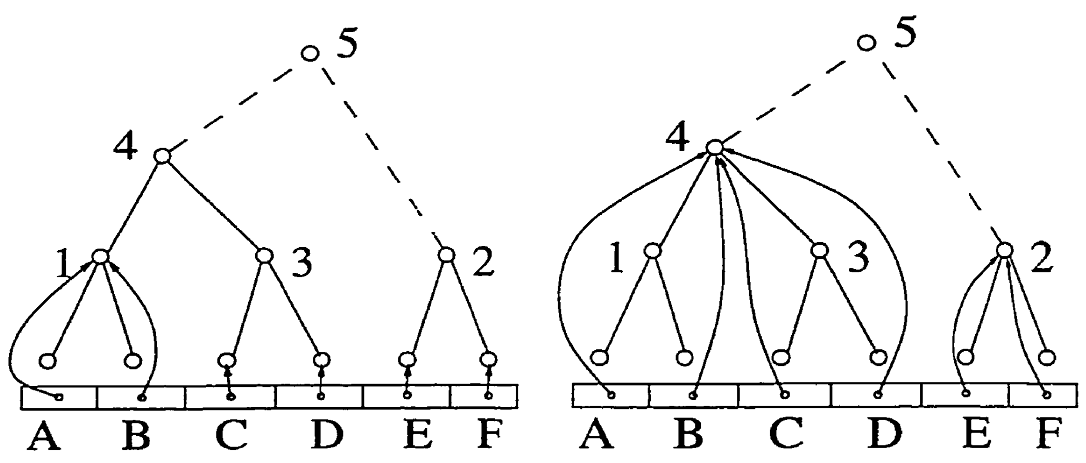
Figure 4.2: Building a vertex tree from a surface with 6 vertices. (a) are the faces to be simplified. (b) is the constructed vertex tree. The number at the internal nodes denote the order they are created. The dotted links indicate that node 5 is the pseudo root added to create a single tree.

A tree has only one root. If a node has no children, it is called a *leaf* node, which is also called an *original vertex*. We call nodes that are not leaf nodes the *internal nodes*, which could also be called branch nodes. Each internal node also defines a *subtree* or a *cluster*, with the internal node being the *subroot* of the subtree. Nodes that share a common parent are *siblings* to each other. Each node has a unique *path* made up of links connecting the node to the root. If a node v has a path p, all nodes on p except v are called *ancestor* nodes to v, and v is a *descendent* node to its ancestors. The number of links on p is called the *level* or the *depth* of v. The root has a depth of 0. A set of vertex trees that are disjoint with each other is a *vertex forest*. Vertex forests could be converted to a single vertex tree by adding a root node taking all the roots of the vertex trees as children. We also place the tree such that the root is at the *top*, and the leaf nodes are at the *bottom*. The direction from the leaf nodes to the root is *up*, and the opposite direction is *down*. Figure 4.1 shows an example of a vertex tree.

If all the nodes of a vertex tree is uniquely ordered and put into a list, we say that the tree is *ordered*, and the list is called the *order list*. If the order list only contains the internal nodes of the tree, we call it *internally ordered*.

4.2 Construction of Vertex Tree

A vertex tree is constructed from the bottom up during the simplification of the input model. In the QSlim algorithm, each iteration contracts an edge $(v_i, v_j) \rightarrow v$. The two



(a) Reconnection after a single edge contraction.

(b) Current proxies after several edge contractions.

Figure 4.3: Reconnect current proxies after edge contractions.

vertices v_i and v_j are replaced by the optimized vertex v. This is a mapping of two vertices to one new vertex. Thus a single edge contraction creates a small tree where v is the parent (also the root) and v_i and v_j are its children. All adjacent edges and faces of v_i and v_j are reconnected to v. In later simplifications, v will also be contracted and become a child of the new node. As the simplification progresses, a vertex tree will be built from the bottom up. If the model is not simplified to a single node but down to a certain number of faces, the result of the simplification will be a vertex forest. For simplicity, the forest is converted to a vertex tree by adding a pseudo root node. Figure 4.2 shows a simplification process that builds a vertex tree from a small surface with 6 vertices.

Current Proxies

In QSlim, after each edge is contracted, the edge's vertices are deleted and the new node is used to represent the edge.[1] However, we would like to preserve the old vertices to build a vertex tree. Thus we must have a way to access the newly created node for later reference to the contracted edge. We use current proxies to achieve this.

For each original vertex of the model, we add a pointer that initially points to it. We call the pointers *current proxies*, and use the term $CP(v)$ to refer to the current proxy of the original vertex v. When we refer to the vertices, instead of using the vertices themselves, we use its current proxies. After an edge $e(v_i, v_j)$ has been contracted to the vertex v,

[1]In practice, one end point is deleted and the other is updated with the new node's information. This is logically equivalent to deleting both end points and creating a new one.

both $CP(v_i)$ and $CP(v_j)$ are now pointing to v. Therefore, any reference to v_i or v_j is now actually referring to v. If v is later contracted again and has a parent v', both $CP(v_i)$ and $CP(v_j)$ should be redirected from v to v'. Figure 4.3 illustrates updating the current proxies after edge contractions.

Redirecting current proxies to a particular node is initially implemented as a recursive call, see algorithm 1.

Algorithm 1 Redirecting current proxies to r.

```
reconnect( Node r, Node d )
{
    if( d is leaf )
    {
        CP(d) = r;
        return;
    }
    foreach c of d's child
        reconnect( r, c );
}
```

At any time in the simplification, current proxies will point to the vertices that the model has been simplified to. Using current proxies makes updating vertex references easy. When a vertex's current proxy is changed to a new vertex, all edges and faces that refer to that vertex will be automatically updated.

Each current proxy is assigned a unique ID, and current proxies are referred by their IDs. In implementation, the current proxies are stored in an array and the index is used as the ID. Thus a face will contain the current proxy IDs for its vertices. Whenever a face vertex is changed, the face shape also changes. If an edge of a face is contracted, two of the face's vertices will have current proxies pointing to the same node, and the face is degenerate (see figure 4.4).

Order List

As introduced in Chapter 3, QSlim iteratively picks an edge from the top of the edge queue and contracts it. As each edge contraction produces a new vertex tree node, the sequence of edge contractions generates a list of nodes v_0, v_1, $\ldots$, v_n. All nodes contained in this list are created during simplification, i.e., they are all internal nodes. These nodes connect the internal nodes of the vertex tree into a particular order. Thus the vertex tree is internally ordered after simplification. Figure 4.2(b) has already shown the ordering among

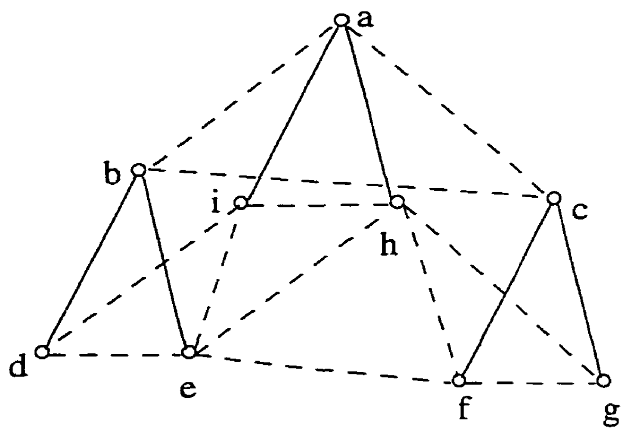
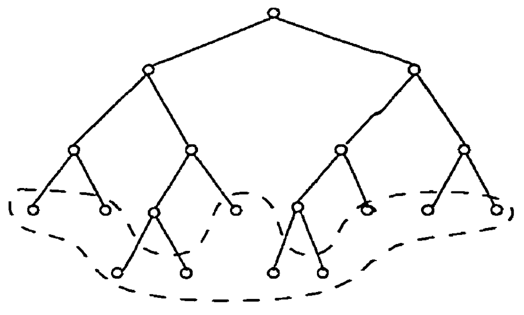


Figure 4.4: The face updates after three edge contractions. The solid lines represents the tree links. Dotted lines are the faces. After updating current proxies, face vertices will be automatically updated. The face (h, e, f) is now updated to face (a, b, c). The other faces are degenerate.

the internal nodes, which is reflected by the numbers. The list that connects all the internal nodes is the *order list*. If we start from the first node and follow the order list, we are able to traverse all the internal nodes of the tree. The list reflects the order that QSlim contracts edges. Since all candidate edges are sorted by the error their contraction introduces, the order list is sorted in the same way. Therefore each node of the order list inherits the edge's error as the sorting key. The direction of the order list is from coarse to fine, i.e., the nodes that are generated later during simplification are closer to the front of the list. Thus the first node in the list will be the last node generated from the last edge contraction. For the order list, we also maintain a pointer that points to a particular node in the list. The pointer is called the *current position* in the order list.

The order list is different from the edge queue in that the edge queue is created by QSlim, and only exists during simplification. Once the simplification is over, the edge queue is destroyed. The edge queue is only an auxiliary structure assisting simplification for selecting candidate edges. The order list, however, is a property of the vertex tree, and exists as long as the vertex tree. It defines a particular order that the internal nodes are connected. The order list can be directly created from the edge queue by recording each edge contraction selected by the edge queue. But it does not depend on the edge queue. We can always sort the internal nodes of the vertex tree according to some other criteria and produce a new order list.


(a) A cut made up of leaf nodes

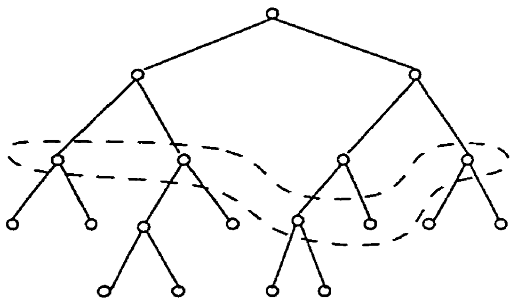

(b) A cut made up of internal and leaf nodes

Figure 4.5: Examples of tree cuts.

Tree Balance

Since an edge contraction is a local operation, and QSlim selects the next candidate edge only based on the error, the constructed vertex tree is usually not balanced. The parts that get simplified more will have deeper subtrees while less simplified parts may have shallow subtrees. The extreme situation is that one node gets contracted and its parent immediately gets contracted and so on. This will create the most unbalanced tree. However, such cases are unlikely to appear because edges are sorted by their error. Where simplification is too severe in one local area, the error of later edge contractions in that area will be greater and thus their contraction will be delayed in the edge queue. In the next iteration the algorithm will pick an edge from elsewhere. Thus the simplification algorithm itself will try to balance the tree to a certain degree. The relative balance of the vertex tree is shown in Chapter 6.

4.3 Cut of the Vertex Tree

A *cut* of a tree is a node set C that satisfies:

1. for any original vertex v_i, $CP(v_i) \in C$;

2. for any two nodes $n_l \in C$ and $n_k \in C$, n_l is not an ancestor of n_k.

3. for any node $n \in C$, there is at least an original vertex v such that $CP(v) = n$.

Intuitively speaking, a cut of the vertex tree is the vertex set of a simplified version of the original model. Figure 4.5 shows examples of tree cuts. As we will see in the next section, a cut is a complete partitioning of the model vertices.

For any face $f(c_0, c_1, c_2)$, where c_i are nodes pointed to by the face's current proxies, if all three nodes belong to the cut of the vertex tree and none of the two nodes are identical, f is called a *current face*. For any cut of the vertex tree, all current faces form the *current level* of the vertex tree.

The cut and current level define the visible nodes and faces when the user navigates the vertex tree, which we will introduce later. Corresponding to a cut is a unique position in the order list.

4.4 Partitioning

It is not difficult to see that each internal node defines a vertex patch of the model, because each internal node is the root of a subtree which contains a set of original vertices. Particularly, the root defines a vertex patch that contains the whole model's vertices. At one level below the root, each child of the root defines a vertex patch, and thus all the children of the root define a vertex partition of the model. We could also go further down to other levels and the subsequent levels will have their own partitions of the tree. At lower levels, the patch size becomes smaller and smaller. Figure 6.13 shows a partition of the bunny model at one level down from the root. Any given cut defines a partition of the model vertices.

Given a partition, a vertex belongs to a unique vertex patch. However, a face has three vertices and each vertex may belong to different partitions. This is why some faces are not colored in figure 6.13 as their vertices belong to different patches. We can assign such faces specifically to one of the patches, for example, we can assign the face to the patch that contains the face's vertex with index zero. In this way, a vertex patch defines a face patch, and the vertex partition also defines a face partition.

4.5 Navigation of the Resolutions

After the vertex tree is constructed along with an order list, we are able to navigate the different resolutions of the model. Suppose we are now looking at the finest resolution, the current position is the last node in the order list. We can decrease the resolution by moving the current position backward in the order list. When the current position moves backward for one node, we have to perform one *collapse*. A collapse is the replay of an edge contraction, which is an operation of reconnecting a node's children's current proxies to that node. The algorithm of collapsing is shown in algorithm 2.

When a node gets collapsed, several nodes on the current resolution become invisible

Algorithm 2 Collapsing a node

```
collapse( Node n )
{
    reconnect(n);
    mark degenerate faces as invisible;
}
```

and are replaced by their parent. The neighboring faces need to be checked for degeneracy. If a face has two corner vertices with their current proxies pointing to the same node, it is degenerate and will need to be marked as invisible.

Similar to collapsing but in the opposite direction, we define the operation of *splitting*. This is a step forward in the order list. The operation is to split a visible node and replace it with its children. The algorithm of splitting is shown as in algorithm 3.

Algorithm 3 Splitting a node

```
split( Node n )
{
    foreach c in n's children
        reconnect(c);
    check for faces that become visible;
}
```

As with collapsing, the affected faces must be checked since some of the invisible faces become visible again and should be marked as such.

By collapsing and splitting, we are able to go back and forth in the order list. As we traverse the order list, faces disappear and reappear depending on the change of their visibility. The order list is actually a record of the simplification process, and our traversal is replaying the simplification process back and forth. At any stage of the traversal, we are looking at the current level of the vertex tree, and the set of visible nodes is a cut.

4.6 Optimizations For the Tree Structure

4.6.1 Model Topology

We need to be able to extract the topology information from any current level of the tree, such as finding the adjacent faces of a node. Topology information can be stored explicitly in each node. This will require a large amount of memory, and with a lot of redundancy. On the other hand, since all nodes are linked in the vertex tree, and the original model has all the topology information, all topology information could be retrieved from the original model.

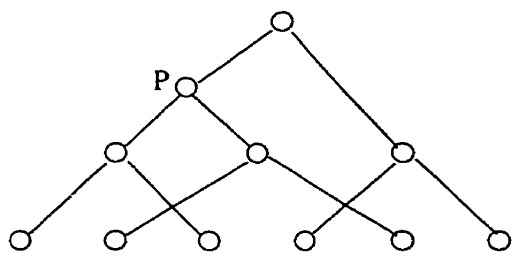
There are many ways to express topology, we can store adjacent edges, vertices or faces in each vertex structure. Storing adjacent faces will encompass both adjacent edges and vertices, since we do not consider isolated vertices and edges. Therefore we keep a list of adjacent faces in each of the original vertex structures. No topology information is stored at internal nodes. When we want to get topology information from internal nodes, we have to find the corresponding original vertices for this information. The basic operation of getting topology information is getting the neighbor faces of a node. This is a recursive call shown in algorithm 4. The leaf nodes that are descendents of this node are found and the adjacent faces are added to the returning list. The third parameter `vis` determines if we want to have the current faces (visible faces) only, or all the adjacent faces regardless of visibility.

Algorithm 4 Getting neighbor faces of a node

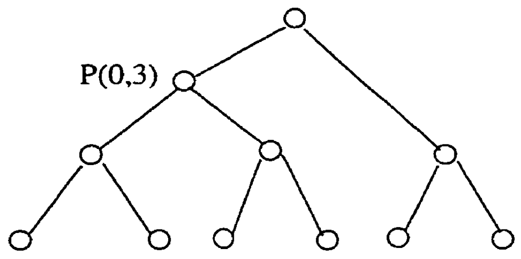
```
nodeAdjacentFaces( Node n, Faces flist, Bool vis )
{
    if( n is leaf node )
    {
        foreach f in n's neighboring faces
        {
            if( (f not in flist) and (!vis or f.visible) )
                flist.add(f);
        }
        return;
    }
    foreach c in n's children
        nodeAdjacentFaces( c, f, vis );
}
```

With algorithm 4, we are able to implement the operation of updating visible faces in the splitting and collapsing operations. In collapsing, we collect the adjacent current faces, and check each one's visibility after the collapse. For splitting, we have to get all adjacent faces including visible and invisible ones, so that some revived invisible faces can be marked as visible again.

In [LUE97], each node maintains a list of original triangles that are *contained* in the node classified as two groups. One group has faces with only one corner contained in the node. The other group with two or more corners contained in the node, but not in its children. The first group is used to update corners of faces that are still visible after collapsing or



(a) Before rearrangement.



(b) After rearrangement

Figure 4.6: Rearrange vertices.

splitting the node. The second group is used to identify which faces should be deleted (invisible) or added (visible) after the node is collapsed or split. In our approach, these functions are accomplished by the current proxies and algorithm 4. There is no need to update the face corners in the first group because this is done automatically by updating the current proxies. The second group of faces are eliminated because we retrieve the face list on the fly and check for visibility. Without having to maintain the list of faces at each node, we save some amount of memory required for each node. If the original model is big, the memory savings could be significant. However, this approach will likely to be slower.

4.6.2 Rearranging Leaf Nodes

Although clear as definitions, recursive calls are expensive and slow. As the user traverses the order list back and forth, the `nodeAdjacentFaces` function is called frequently, and the recursive calls in it could slow down the speed of response. The `reconnect` function in algorithm 1 is also a recursive call. Looking at these functions reveals that what the recursive call does is to have access to the leaf nodes that are descendent to an internal node. If we can get these leaf nodes in another way, we can achieve some optimization.

All the model's original vertices and their corresponding current proxies are stored in an array. An original vertex and its current proxy is accessed by its index. Initially the array is created in the order of reading in the vertices. However, if we rearrange the order of these nodes according to the tree, we can associate the order of the vertices with the tree structure. The idea is to do a depth first traversal and rearrange the vertices in the order that they are visited. See figure 4.6, in which the original tree (a) is rearranged as (b).

For each node v, we add two indices called the *span*(v) that is an index range (a, b). The original vertices within the index range are all descendents of v. For example, in figure 4.6, node P has a span of (0, 3) and the leaf nodes within that range are all descendents of

P.

After the model is simplified we build the span for each node. Leaf nodes will have the span of a single index. With the span information, we can now eliminate the recursive structure of `reconnect` and `nodeAdjacentFaces`, and update the current proxies or get the adjacent face list directly (see algorithm 5 and algorithm 6).

Algorithm 5 Revised reconnect function

```
reconnect( Node n )
{
    for( i=span(n).a; i<=span(n).b; ++i)
    {
        // redirect the i'th current proxy to n
        current_pointer(i) = n;
    }
}
```

Algorithm 6 Revised adjacent face function

```
nodeAdjacentFaces( Node n, Faces flist, Bool vis )
{
    for( i=span(n).a; i<=span(n).b; ++i)
    {
        foreach f in vertex(i)'s neighboring faces
        {
            if( (f not in flist) and (!vis or f.visible) )
                flist.add(f);
        }
    }
}
```

4.6.3 Generalized Tree

By default, QSlim builds a binary vertex tree, because its primary simplification is an edge contraction. However, this might be too fine-grained and requires a large amount of memory for big models. Quad-tree or oct-tree may be required for hierarchies which consume less memory. To generalize this idea, the upper bound of the number of children a node could have becomes a parameter *child_bound* that the user can specify. When contracting an edge (v_i, v_j), if an end point is an internal node, the replacement node v can choose to *adopt* its children if after adoption, the number of children of v is within *child_bound*. Figure 4.7 shows this process.

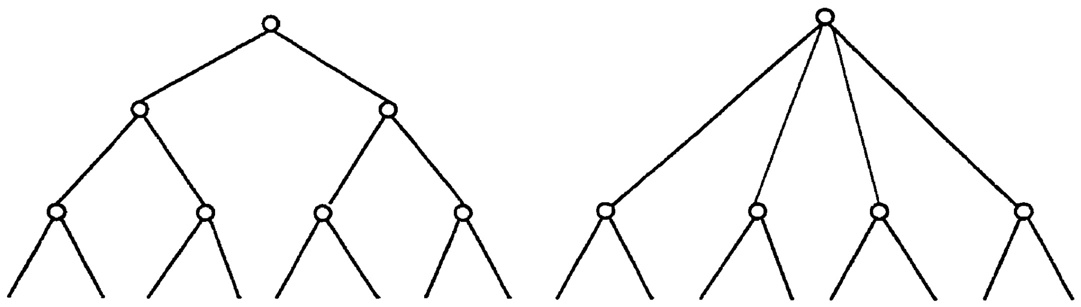

(a) Before adoption. (b) After adoption.

Figure 4.7: The replacement node adopts the edge end point's children.

It is important that such adoption preserves some relationship that exists between previous parent and children. For QSlim, this relationship is that the parent's quadric equals the sum of children's quadrics:

$$Q_{parent} = \sum_i Q_{child(i)}$$

It is not difficult to see that adoption preserves this relationship.

Chapter 5

Semiautomatic Simplification

Numerous simplification algorithms have been introduced in the past decade. Most of them are completely automatic. Some of them offer limited user control, often in the form of input parameters such as error bound, the number of vertices or faces desired, or some criteria for detecting features that need special care during simplification. Once the simplification process has started, the user can only depend on the algorithm to produce a good final result. The limitations of completely automatic simplifications are:

- The suitable parameters are hard to obtain. It is true that with the right set of input parameters, the simplification algorithm would produce some very good results. However, finding such a set of parameters is not easy. Each algorithm has its own context and parameter domain which depends not only on the algorithm but also on the input model.

- The simplification process is not interactive. As indicated above, finding the correct parameters requires many trials which is tedious. When dealing with visual objects, the user would usually want to manipulate the objects and get feedback interactively.

- Not enough control for the user. Setting a few parameters in some circumstances is not enough for complex control over the simplification. It is better to have a way that the user can control the simplification in a flexible and intuitive manner.

- Perceptual quality may not be the consistent with numerical or geometric quality. All simplification algorithms rely on certain numerical measurement of the model quality. However, the perceptual and numerical quality of the simplification may not always be in agreement. One simplification with large numerical errors may appear more similar to the original model than another simplification with smaller errors.

- Automatic algorithms do not *understand* the model. Each model has some semantic value that a human could easily recognize. However, automatic algorithms treat the input model only as a collection of triangles and vertices. With user control, the simplification is more consistent with the meaning of the model.
- Automatic algorithms cannot adapt to model use. For example, a model is usually associated with a skeleton in animation applications. It is better to simplify the model taking the skeleton into consideration.

Automatic simplification has the advantage of dealing with large models, where it is impossible for human to simplify manually. However, when it comes down to a certain level of coarseness, humans usually do a better job than the automatic algorithms. It will be better if advantages of both approaches be combined so that better simplification can be achieved. Our work is an effort toward achieving this goal. Our goal is to build a tool that offers automatic simplification with fairly good results, at the same time providing the user with tools to modify the simplification process and improve the simplification result.

Our tools are different from traditional modelers. There are many excellent 3D modelers that offer rich collections of tools and features for the user to build or edit 3D models. Our work is not a competing modeler in this sense. We are focused on simplification and try to produce better simplification results. One question that could be asked is: why not use a simplification tool combined with a traditional modeler? The user could simplify the model to a certain level of detail, and use the modeler to modify it and continue simplification. The disadvantage with this process is that it is not interactive. Alternating between simplification and modeling could be cumbersome. Our tool will offer a unified environment for simplification and modeling. Another important disadvantage of traditional modelers is that they only operate on a single level of detail. Our tool will exploit the multiresolution structure, where the user modifications are not limited to a single level of detail. Multiresolution models provide another dimension in which to operate: the different detail levels. Once the user improvements are finished, it is possible to extract a number of levels at different resolutions that are also improved. This is an important benefit where multiple levels of detail are expected. Finally, interacting with the simplification process affords a high level of control not available with modelers and manual manipulation.

As introduced in Chapter 2, there is already research being done on multiresolution editing. Our work differs from those. Multiresolution editing is focused on changing the model, in particular the original model. Typically the original model is changed by manipulating the coarse levels, giving control of the scale of the edit. Although our work offers editing tools, we are focused on *improving* the fidelity of *coarse* level representations and we would prefer to preserve the original model instead of changing it. Specific functions unique to this approach include the ability to control distribution of detail on the model surface and the partitioning of the surface.

In the following two chapters, we will give a detailed description about the structure we use and the functionalities that are provided based on the structure.

5.1 Overview

Our tool is an integration of simplification algorithms and a modeler that the user can use to intervene the automatic simplification process. Starting from a highly detailed model, the user applies a series of edits using the provided functionalities and produces an improved coarse level representation.

The typical procedure of using the tool is

1. The system reads in an input model and simplifies it. This produces a hierarchical structure that encodes multiple resolutions of the model.
2. The user navigates the hierarchy and selects a level of detail that he/she wants to make changes. Then using the functionalities provided in this tool, the user manually improves the model at this level.
3. The change made by the user can be propagated to other levels of detail. Or the user could restart the simplification taking the modified level as input. This will recreate all the coarse levels with the fine levels unchanged.
4. Step 2 and 3 is repeated until the model is improved as the user wants it.
5. The whole hierarchy or a single level of detail is output.

In Chapter 4, we introduced the vertex tree structure that is used for multiresolution representation in our tool. Briefly speaking, the vertex tree remembers old vertices instead of deleting them as pure simplification does. Meanwhile, we record the decimation process in an order list, and traversing the list enables us to navigate the simplification process which results in different levels of detail. As our goal is to improve the quality of simplification, we should provide the ability for users to intervene in the simplification process and modify the model, so that the model has a better appearance than that produced from automatic simplification directly. We offer a set of tools that are useful for the user to achieve this goal.

There are many possible ways to apply changes to a model. The fundamental method is changing the geometric positions of model elements, which is provided by traditional modelers. What is unique in our tool is the multiresolution representation. This enables us to modify the model in a multiresolution context. We can exploit the structural power of the representation so that better simplified models can be created.

5.2 Geometric Position Editing

The basic operation that a user can apply to a model is to change the geometric positions of elements. By examining the model, a user locates the vertices that are poorly placed by simplification algorithms and moves them to a better place. The faces that contain this vertex will be updated with the change.

The geometric editing tool in our modeler is quite simple. The user selects a vertex and moves the mouse, the vertex will follow the mouse movement and once the mouse is released, the vertex is put to a new position. Compared to the rich set of geometric editing tools offered by traditional modelers, this is rather simple and incomplete. Providing a complete set of such features would be beyond the scope of this thesis as our work is more focused on exploiting the multiresolution representation.

One thing that needs consideration is how to define the movements of vertices. When the user moves the mouse, the vertex follows them in the eye space. Currently, the user can only move a vertex parallel to the screen, which means parallel to the XY plane in eye space. The model can be rotated to achieve translation in the third axis.

5.3 Propagation

Simply moving a single vertex is too limited. Usually we want the change to propagate to some other area. There are three identifiable directions for propagation, which are:

- Propagation to the parent. The change of a node affects the position of its parent, which in turn affects all the ancestors.
- Propagation to the children. The change of a node affects its children and further descendents.

- Propagation to the neighbors. The change of a node affects the neighboring nodes on a cut that is being viewed.

5.3.1 Propagation to the Parent

As the goal is to improve simplification results, the user might desire that changes in the currently viewed cut influence the coarser levels. This implies that when there is a change in the child nodes, the parent should in some way reflect this change. This requires that the parent's position be dependent on the child's position. However, parent positions do not depend on their children after the vertex tree has been initially constructed. As shown in Chapter 3, the quadrics are initialized from the original faces and accumulated during simplification. Therefore, when using these quadric values to evaluate the optimal node positions, the target position only depends on the original faces. Unless the related original surface is changed, a node's optimized position will not be affected.

A way to work around this is to use faces in the current level to recalculate the quadrics so that the parent's position is affected by the child's change. Suppose we are moving a node v, and the parent node of v is v_p. Let us assume that v is an internal node. Initially, the quadric of v is derived by summing its children's quadrics. Now we recalculate v's quadric not from its children, but from its current faces. The calculation is exactly the same process that the original vertices' quadrics are initialized. For each current face f_i that are adjacent to v, denote its quadric as Q_{f_i}. Thus v's quadric is

$$Q_v = \sum_i p_i Q_{f_i}$$

where i iterates all the adjacent faces of v. p_i are weighting factors associated with each face quadric depending on the weighting strategy (see section 3.2.2). After v's quadric is recalculated, v_p's quadric is updated by summing its children's quadrics again. For v_p's children that have not moved, the old quadrics are used. Thus v_p's quadric is sum of a mixture of original face quadrics and current face quadrics.

5.3.2 Propagation to the Children

When an internal node is moved, it might be desirable to move its descendents as well. There are two ways that we might like the change to be propagated toward the fine level nodes. One is to propagate the change without any modification. This will change all the fine level nodes including the original vertices. On the other hand, sometimes we might want to preserve the original model. This is true when we are especially concerned with improving only simplified models. In addition, levels that are closer to the original should be affected less than those farther from the original. The total effect would be the descendents' changes become weaker and weaker as they approach the original vertices, and when the original vertices are reached, the changes disappear. We denote this effect as *attenuation.*

Global Interpolation

One way to achieve the propagation is by globally interpolate the change. Suppose we move the node v and the difference vector between the old and the new position is $\vec{d}$. If we add $\vec{d}$ to all the children of v, this will give all the descendent nodes of v the same movement. If we want to attenuate the child node changes, we can interpolate $\vec{d}$ based on an *attenuation factor* at each node.

Choosing an Attenuation Factor

We want the changes to get smaller and smaller as nodes get closer and closer to the original model. Adding an attenuation factor and denoting it as $t\,(0 \leq t \leq 1)$, the interpolated position v' is $v' = v + t \cdot \vec{d}$. Thus t should be 1 at the node that is being manipulated, and 0 at the leaf nodes. One way of getting t is to relate it to the node depth. The closer a node is to the leaf nodes, the smaller it is. At leaf nodes $t = 0$, and at the node being manipulated $t = 1$. Nodes between v and leaf nodes have values between (0, 1), and should decrease as nodes' depth increases. A difficulty is that the vertex tree is usually not balanced. For an internal node, the number of links on the path to each leaf node within v's span is different. Thus we cannot use the link numbers directly because for each node there should be only one t value associated with it.

One approach we have tried to overcome this problem is to assign a length value to each link, and use the length instead of the number of links. We stretch the links to make all leaf nodes have the same path length, which we normalize to one. For a node v, its attenuation factor is calculated by

$$v.t = v.parent.t + (1 - v.parent.t)/(max\_leaf\_distance(v) + 1)$$

where *max_leaf_distance(v)* calculates the number of links from v to the deepest leaf spanned by v. This function evenly divides the path lengths where possible. Otherwise, it will stretch a short path to match a longer path. Figure 5.1 shows an example.

Now that we have a consistent t value for each node, we can interpolate by it. However, a problem with this approach is that the calculation of t is purely based on the structure and does not relate to the model geometry. Thus discontinuities easily appear after interpolation.

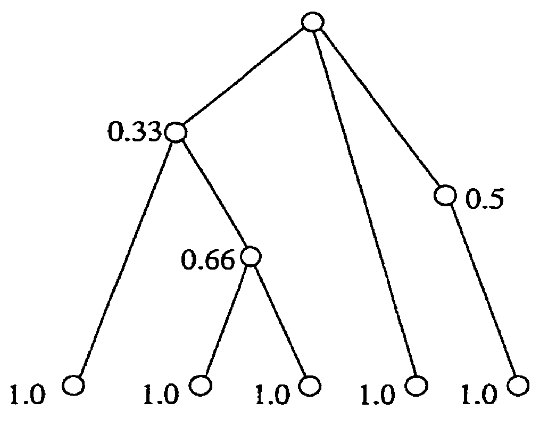


Figure 5.1: Stretching links to get unified t value.

Another approach we tried is to use the error evaluated by the quadric at each node to obtain the attenuation factor. Suppose the vertex v being manipulated has error ε. For each descendent of v with error ε', a pre-attenuation factor is calculated by $t' = \frac{\varepsilon'}{\varepsilon}$. Since the finer level nodes have smaller error than coarser level nodes, and leaf nodes (original vertices) have zero error, t' possesses the property required by an attenuation factor. However, the curve of t' is usually very steep, because the error at the manipulated vertex is often much larger than its descendents. To smooth the curve, we define the attenuation factor to be $t = \sqrt{t'}$. This produces better results than the previous approach.

Since the purpose of user edits is to improve the model quality, modifications are usually small. Attenuation is suitable for these tasks. However, if the edit introduces drastic modifications to the model surface, there might be discontinuities in some levels. This usually happens when an intermediate level is made up of nodes with very different changes due to different attenuation factor.

Propagation by Local Frames

Another approach of propagating to the children is through local coordinate spaces or local frames. The global interpolation approach has the disadvantage that all movements are parallel. Sometimes it is desirable to see the changes include some orientation change. For example, when we grab the nose tip and move it, the orientation of the nose also changes. We would like the fine details of the nose to also rotate to their new orientation instead of moving in parallel with their parent. This is achieved by creating local frames and detail vectors.

In [KOB98], local frames are based on faces. To avoid artifacts, the face and its adjacent three faces are approximated by a quadratic surface, and detail coefficients are distance

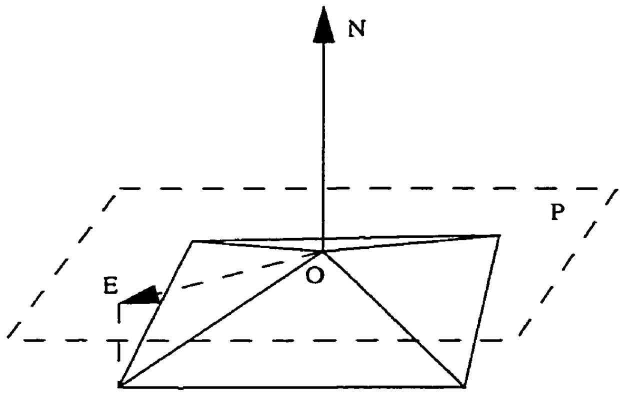


Figure 5.2: Setting up a local frame. ON is the average of O's neighboring face normals. P is ON's orthogonal plane. OE is the projection of an edge onto P. The third axis will be a cross product of ON and OE.

vectors to the surface. As our structure is a vertex tree, it is natural to use vertex based local frames. In order to let local frames' orientation change with the surface, one axis X is set up by averaging the neighboring face normals. Another axis Y is obtained by projecting one adjacent edge onto the orthogonal plane of X. The third axis Z is just the cross product of the first two (see figure 5.2). Thus when a node is being moved, its orientation will be adjusted according to the local surface.

Detail vectors are obtained by transforming the children's position in world space to the parent's local space. Given a local frame L and a vector v in world space, we denote $v' = L(v)$ to be the conversion from world space to L. Conversely, we denote the inverse operation as $v = L^{-1}(v')$ which converts a local vector to world space. When the parent's position is changed, the children's positions are updated by adding the detail vectors to the parent's local frame.

The local frames are implemented in a nested manner. Suppose we have a path containing nodes: $n_0, n_1, n_2, \dots n_k$, where n_0 the node with the smallest depth on the path, and is in the world space. n_1 is defined in the space of n_0, including its position and local frame. n_2, in turn, is in the local frame of n_1. Thus n_2 is two levels down the nesting. This goes on to the last node in the chain. The basic operation in dealing with local frames is converting a position between the world space and a nested frame of a certain level, which is shown in algorithms 7 and 8.

Note that in nested local frames, the operation L^{-1} transforms a vector to the space that is one level higher than the frame L. When the user manipulates a node, its local frame's origin is changed, and the orientation is also changed by averaging the neighboring

Algorithm 7 Convert a vector from local frame to world space

```
globalize( Node n, Vector v )
{
    $v$ = $L^{-1}(v)$;
    if( n.frame is nested in higher level frame )
    {
        globalize( n.parent, v );
    }
}
```

Algorithm 8 Convert a vector from world space to local frame

```
localize( Node n, Vector v )
{
    if( n.frame is nested in higher level frame )
    {
        localize( n.parent, v );
    }
    $v$ = $L(v)$;
}
```

face normals. All its descendents' positions are recalculated by converting their position from the nested local frame space to the world space. The local frame structure, however, remains the same except for the node that is being manipulated.

The nested frame structure is created when the user selects a node, and destroyed when the node is released. Ideally it would be nice to have a static nested local frame structure because setting up the nested frames costs extra computation. But the difficulty with static local frames is that the adjacency relationship between the nodes as well as between a node and its faces is not static in the vertex tree, depending instead on the current cut and the order list, which can be interactively modified (see section 5.4). A node could exist in multiple cuts of the vertex tree, each creating a different set of adjacent faces and a correspondingly different local frame. For example, in figure 5.3, the node v could have different adjacent faces and nodes in different cuts. So we cannot set up its local frame until the cut it currently resides is determined.

To achieve the attenuation effect, we could use the attenuation factor to interpolate the old position and the new position of each descendent node.

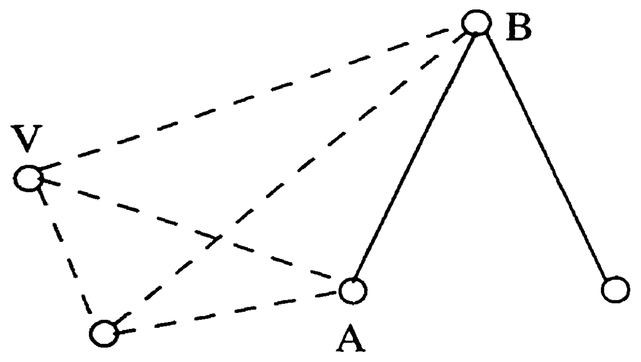


Figure 5.3: Neighborhood is not static but depends on the cut. The dotted lines indicate the face edges. The solid lines are the links in the vertex tree. After A is collapsed into B, V's neighbor node is changed to B. The adjacent face is also changed. This requires setting up the local frame again.

5.3.3 Propagation to Neighbors

It is often too tedious to manipulate a single vertex at a time. Often when a user moves a particular node, it is expected the neighboring vertices would be affected in some way. In [KOB98], the neighbor interpolation is achieved by discrete fairing, which is used to smooth out the surface between control points. Our approach interpolates the change directly. The user can specify the range of neighboring nodes to be affected. The range is a topological circle defined in the number of edges around the selected node. When the user moves the selected node and the vector between the old and new position is $\vec{d}$, the neighboring nodes within the range will interpolate $\vec{d}$ by multiplying an interpolation factor t, where t is a real value between the interval [0, 1]. The t factor here is similar to the attenuation factor in child propagations. To provide the user with more control on the shape of neighbor interpolation, we use a Bezier curve segment to produce the interpolation factor $t = B(\#edges)$. Given the topological distance from a neighboring node to the selected node, the curve will generate a t value, which is guaranteed to be 1 at the selected node and 0 at the specified range border. The internal distribution of t is controlled by two control points of the Bezier curve. Figure 6.7 shows the curve segments, and corresponding propagations. For any affected neighbor nodes, if propagations to the parent or child are required, they will be applied at these nodes.

5.4 Order List Manipulation

The order list associated with the vertex tree initially records the order in which QSlim contracts the edges. This order produces the best series of approximations with respect to the quadric metric. However, it may not be necessary to preserve this order list. Allowing changes of the order list will provide the user with ability to edit across different resolutions.

The order list is built by memorizing each edge contraction. At each edge contraction, the replacement node is added to the list. At the end of the simplification, the order list

$$v_k \rightleftharpoons v_{k-1} \rightleftharpoons \cdots \rightleftharpoons v_0$$

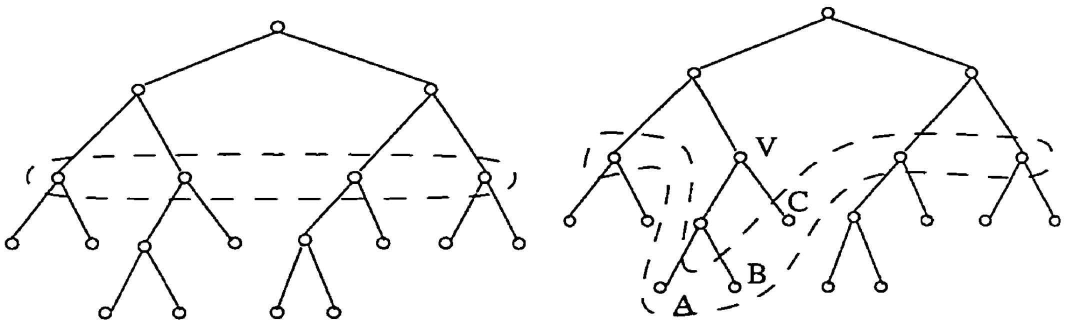
is a list of internal nodes. The subscript k indicates the kth edge contraction, and the node v_i is the replacement vertex produced by that contraction. The reason the list is reversed from its index order is because the model is in the simplified state after simplification. Nodes are added to the list by pushing it onto the front. Two directional arrows indicate that list nodes may be visited backwards and forwards.

As introduced in Chapter 4, following the list will traverse the entire vertex tree. At any point in the traversal, the current position corresponds to a current level or cut in the tree. If we walk through the order list from the start to the end, the list will define a way to selectively refine the model. If we change the permutation of the order list, we can create other ways to selectively refine the simplified model. This will create novel simplified levels that are not produced in the original simplification process. In the following we describe two applications of manipulating the order list, which are local traversal and feature preservations.

5.4.1 Local Refinement and Simplification

Given a crude simplified model, some parts that are simplified may be acceptable as the final output. For example, the faces that are more planar could be replaced by larger triangles. However, there are parts where more detail is desired, like facial expressions or regions that bear important characteristics of the particular model. The order list produced by a simplification algorithm like QSlim always compares error across the entire model, and cannot refine important regions without refining satisfactory regions as well.

User controlled local refinement is natural with the vertex tree structure. Users simply select a vertex in the cut which is an internal node. As each internal node is the root of a subtree, we can just expand the node. For simplicity, when expanding the subtree, all children of a node are used to replace it. If the user wants to further expand a particular node in the expanded subtree, that node could be selected and the same procedure is applied. In figure 5.4, the node V is locally refined, and the cut after refinement contains finer level nodes A, B and C.



(a) The coarse level cut. (b) The cut after order manipulation.

Figure 5.4: Order manipulation changes the cut of the tree.

Local simplification is the reverse of local refinement. But the difference is that while local refinement is a completely *local* operation that only affects the subtree, local simplification affects the neighboring nodes. When a node v is to be simplified, its parent node must be collapsed, which will always simplify v's siblings. Thus local simplification is really not that "local". If the parent of the manipulated node is the root of the vertex tree, the whole model could be simplified.

After local refinement and simplification, the order list must be updated to reflect the changes in the current cut. For nodes that become visible due to local traversal, we adjust their positions in the order list so that the current position in the list corresponds to the modified cut. The update involves a series of node movements in the order list similar to those introduced in the following discussions of feature preservations.

5.4.2 Preserving Features

Some parts of the model surface carry more meaning than other parts, such as the outline of eyes on a human face model. We call such parts *features*. Typically, features are in the form of edges, but it can also be vertices or faces. Automatic simplification algorithms usually do a poor job in preserving features, as it is hard for the algorithms to detect precisely which area has more semantic value than the rest. However, with the aid of a human operator, features can be specified and preserved.

The order list could be used to preserve features. Basically, we achieve this by delaying the collapse of certain vertices and forcing them to appear in coarser levels. Suppose we have an order list

$$v_i,\ v_{i-1}, \ldots,\ v_{j+1}\, v_j, \ldots, v_{k+1},\ v_k,\ v_{k-1}, \ldots,\ v_0 \quad (0 < k < j < i)$$

and v_k is specified as a node in a set of feature nodes. To clarify our discussions, v_i is called the *front* of the list and v_0 the *end* of the list. Suppose we start navigation from v_0, which corresponds to the finest level (or cut), and navigate toward v_i corresponding to the coarsest level. Such a navigation is a series of simplifications. We would like the feature nodes to be retained to a coarser level, say at the position of v_j. Normally, by the time our navigation (i.e., the current position in the order list) reaches v_j, v_k has already been collapsed. Now we can manually move v_k in front of v_j so that when our navigation reaches v_j, v_k is still visible. Thus the order list now becomes

$$v_i,\ v_{i-1}, \dots,\ v_{j+1}\ v_k,\ v_j, \dots, v_{k+1},\ v_{k-1}, \dots,\ v_0$$

Now when we follow the order list and get to v_j, v_k will always be visible. However, there are some dependencies among the nodes. Ancestor nodes cannot be collapsed later than descendents because this will also collapse the descendents. In other words, if v_m is an ancestor to v_n, it must be that $m > n$. When we move nodes around in the order list, such relationship must be maintained otherwise the list will not produce a monotonic series of approximations. Therefore, when v_k is being moved towards the front of the order list, we must check if it passes any of its ancestors. If so, the ancestor must also be moved to keep ahead of v_k. The adjusted ancestor is positioned immediately in front of v_k. The ancestor could be placed at other positions front of v_k, but that does not have an obvious advantage. Moreover, v_k and its adjusted ancestors should have a smaller error than any of the nodes between $[v_i,\ v_{j+1}]$, so that if we sort the nodes between $[v_i,\ v_k]$ after moving v_k, v_k and its ancestors will always be adjacent.

We can specify a group of nodes such as a feature edges or a feature region and do the same thing for each node within the feature. The features would be preserved in the coarse levels. Looking at figure 5.4 in another way, nodes A, B and C could be a number of feature nodes that a user wants to preserve. Before preserving the features, the cut in (a) corresponds to the current position in the order list. After preserving the feature nodes, the current position corresponds to the cut in (b).

In implementation, the user examines the model and identifies features on the model surface. The features are selected either as a line of edges or a patch of faces. Then the user navigates to another level (typically a coarser level) and preserves the features to that level.

5.5 Modifying the Vertex Tree Structure

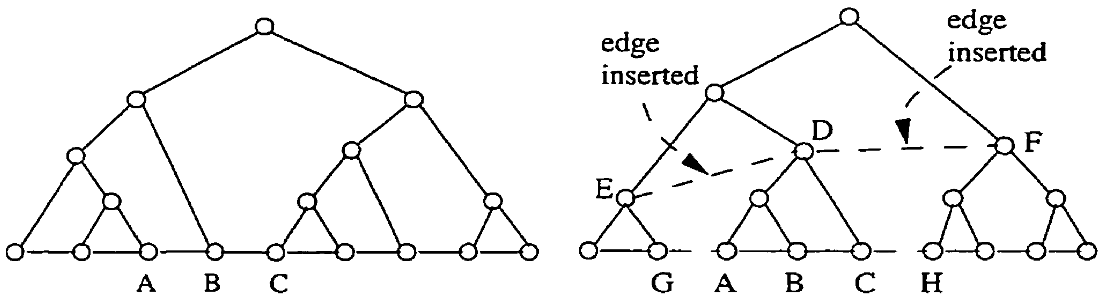
Another possibility of applying modifications is to modify the vertex tree structure itself. Once the vertex tree is built, we rely on it to navigate multiresolution models and make changes in the multiresolution context. However, the vertex tree itself might have room for improvement. As introduced in Chapter 2, the vertex tree divides the model surface and vertices into different patches. A subtree will group a number of faces and vertices into it. If this job is done by a human user, the natural way is to divide the model into several logical parts, such as the head, the body or the tail, as for the bunny model. However, this is hard to be done by simplification algorithms. Thus the vertex tree might be changed in a way that is perceptually, semantically, or utility based, rather than error based.

The modification of the vertex tree structure is denoted as *reclustering*. Our objective is to let the user provide guidance on how the vertex tree should be structured. Since the structure is created during simplification, it could be modified by applying simplification again, but based on some user input as constraints. Viewed in another perspective, this is a way to freeze the simplification process, apply some changes and continue with the simplification.

5.5.1 Reclustering

One useful functionality lets the user specify a patch on the model surface at a certain level, and after simplification, the patch will become a subtree. If we want a specific subtree to be formed from a patch, the nodes inside the patch are allowed to be joined with each other, but no nodes in the patch are allowed to be joined with nodes from outside before the subtree is already constructed.

In QSlim, nodes are joined together by contracting edges. Thus we force the construction of a subtree by not allowing edge contractions between nodes in the patch and the outside nodes. Since QSlim inserts all candidate edges into the edge queue before simplification, we can delay the insertion of these edges. Thus during simplification, no edge contraction would occur between the nodes in the patch and nodes outside the patch, and the patch will be simplified until no more edge contractions are available within the patch, which is now a single node. After the patch has been simplified to a node, we will have constructed a subtree based on the patch. At this time, we would like the simplified patch to participate in the rest of the simplification process otherwise the patch will remain as an isolated node. We now re-insert all the edges that we have delayed at the start of simplification into the



(a) Vertex tree without reclustering (b) Vertex tree after reclustering

Figure 5.5: Reclustering. (a) is the vertex tree before clustering. In (b), the vertices A, B and C are defined as a patch. G and H are border vertices. The edges AG and CH are deleted at the beginning. After the subtree is constructed, the deleted edges are re-inserted.

edge queue (of course, many of the re-inserted edges are now redundant and should be removed), and later simplifications will incorporate the subtree into the resimplified tree structure.

Care should be taken at the border of patches. If we allow the border nodes to be simplified by both the patch and its surroundings, they will compete to simplify the same nodes, corrupting the order list. Therefore, we choose to treat the border nodes as outside of patches, and edges connecting the border with inside nodes are inserted into the edge queue only after the patch is simplified. Figure 5.5 shows an example of reclustering. The vertices A, B, and C are defined as a patch from which a subtree is built.

Preserving Feature Edges by Reclustering

As a variant of reclustering, we treat a set of user defined feature edges separately using the reclustering techniques. Basically, we make feature edges a patch. The internal nodes are the nodes on the edge. Similarly, we do not insert edges connecting the edge with outside nodes at the beginning of simplification. In addition, the user designates a number of nodes on the feature edge which are defined as *critical points*. Critical points are nodes that are important in keeping the shape of the feature edge. When contracting an edge, if one end point is a critical point and the other is not, the edge is merged into the critical point, which means the replacement node copies the critical point. If both end points are critical nodes, the edge is not contracted. In this way, the critical points are completely preserved. Different from reclustering, the deleted edges are not re-inserted later.

In section 5.4.2, we have introduced preservation of features by manipulating the order list. In that approach, features could be made to appear in lower levels by changing the order in which the nodes are refined and collapsed. When a fine level node is forced to be collapsed later, it will appear in lower levels. However, order list manipulation is severely limited by the partial order (or the parent-child relationship) enforced by the vertex tree structure. Reclustering eliminates the partial order. The advantage of reclustering is that reclustering allows one to manually define the structure of vertex trees. Within a user defined patch, simplification continues, but patch boundaries are not crossed. This allows the user to define semantic constraints on model simplification. Combined with order manipulation, reclustering enables users to effectively traverse meaningful regions, and preserve utilization or functionally defined regions.

Chapter 6

Results and Evaluation

The traditional method to evaluate a simplification algorithm is to test its speed and quality. Given the same model and similar environments, the time that an algorithm uses to simplify it to a certain degree, and the quality of the result are the two very fundamental measurements of a simplification algorithm. However, because of the semiautomatic nature of our tool, the traditional testing and evaluation method of simplification algorithms does not fit here. User manipulations emphasize visual effect rather than quantitive precision. It should not be surprising that a user-improved model has larger error than automatically generated ones. However, from the stand point of a user, the modified model will be more similar to the original model because the modifications directly correspond to user requirements. For this reason, strict comparisons of numerical error between the results of our tool and those of automatic simplification algorithms do not provide much useful information.

Interactive multiresolution modeling is the basic nature of our system. The ideal way to test such a system would be a user study. Due to time and resource constraints, such a study is not included in the thesis. Thus the major testing for the work is to demonstrate the effects that we are striving to achieve. This will be in the form of a set of images showing these effects. Additionally, the space and time requirements of the tool should also be examined. Thus we will evaluate the system in these three aspects.

6.1 The Vertex Tree

The size of the tree depends on the size of the model and the structure of the tree. Since the number of model faces remains constant once the model has been read in, the size of the vertex tree depends mainly on the number of nodes in it. If we generate a binary tree, the total number of nodes (including the original nodes) will be roughly twice the original nodes. When the tree is not binary, the total number of nodes generated will depend on the

Child bound	2	4
Total # of nodes	69666	51968
Memory	≈18MB	≈13.5MB
Maximum depth	20	14

Table 6.1: Vertex tree of the bunny model. The original model has 34834 vertices and 69451 faces. The model is simplified to 2 faces and 3 vertices.

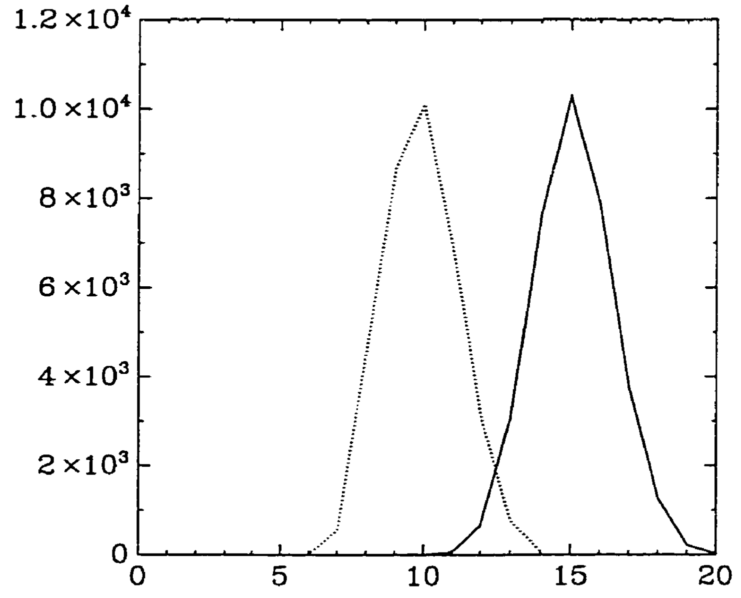


Figure 6.1: The distribution of leaf depths. The X axis is the range of depths. The Y axis is the number of leaf nodes with a depth. The left curve is the distribution of leaf depths with child bound set to 4. The right curve is the distribution with child bound set to 2.

upper bound of the number of children at each node as well as the simplification process. Balance is another property of the vertex tree. More balance generally indicates better structured tree (in a balanced tree, all leaf nodes are at the same depth). The vertex tree is normally not perfectly balanced after construction, but will be balanced to a certain degree because of the nature of simplification algorithms. Table 6.1 shows the space and balance of the vertex trees generated from the bunny model. The *child bound* parameter specifies the upper bound of the number of children each node could have. By default this value is 2, as QSlim is an edge contraction algorithm. When adjusted to larger values, a node could have no more than that number of children. We see that the default binary tree has almost twice as many vertices as the original, with the maximum depth of 20 links, the memory consumed in real implementation is around 18 megabytes. When this parameter is adjusted to 4, the number of nodes and the maximum depth have decreased significantly, and the memory consumed dropped to around 13.5 megabytes.

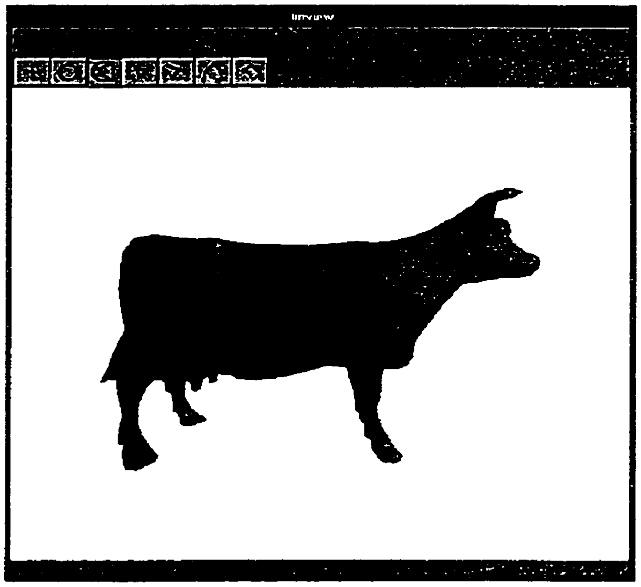


Figure 6.2: The interface of the tool.

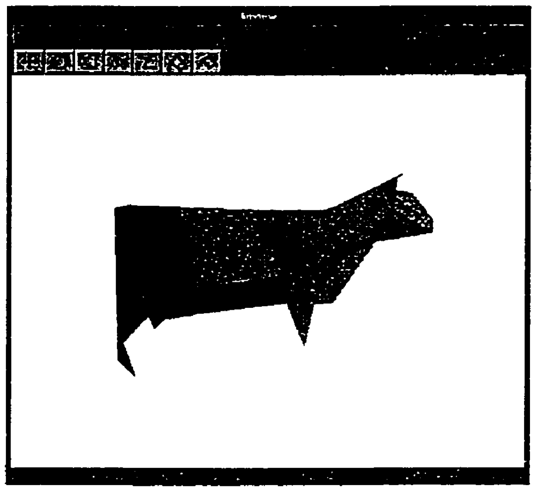
Figure 6.1 shows the distribution of the depths of leaf nodes. The left curve corresponds to the child bound parameter set to 4, and the right curve is the default binary tree. With the majority of leaf nodes centered at a specific depth, it is obvious that the simplification algorithm tries to produce a vertex tree with certain degree of balance.

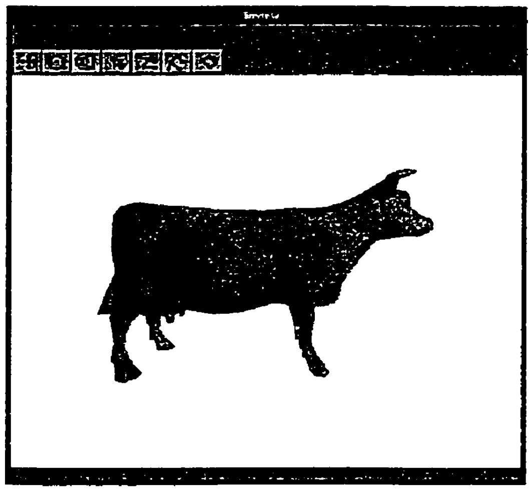
6.2 Interactivity

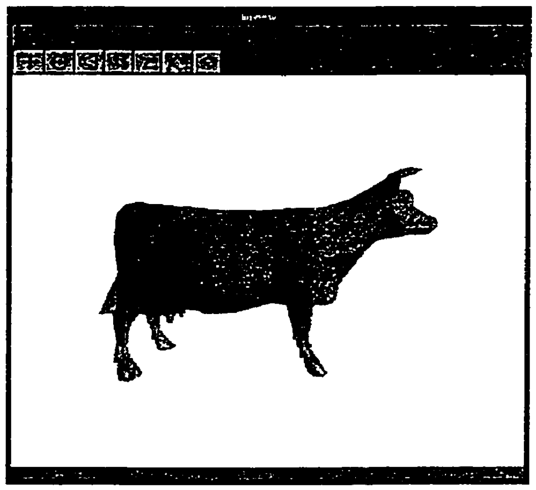
The main interface of the system is shown in Figure 6.2. The viewing area is also where the user manipulates the model. Users can select vertices, define edges or patches. The tool buttons control the interaction modes.

6.2.1 Navigation of Multiresolutions

Navigating different resolutions is achieved by traversing the order list. After the vertex tree is built, the model is at its coarsest level. By dragging a slide bar, the user can change the current order list position, which changes the resolution of the current model view. This is shown in figure 6.3. Because traversing the order list is sequential, the required time is dependent on the degree of change in resolution (or the magnitude of change in order list position). In Chapter 4, we have described reordering the vertices so that each node could be associated with a span of current pointers. Changing the resolution requires a linear traversal of the span of each involved node. The larger the span is, the more time a traversal takes. We have tested the time used in changing resolutions for the horse model,


(a) Simplified cow model with 100 faces


(b) Simplified cow model with 1500 faces


(c) Simplified cow model with 3000 faces

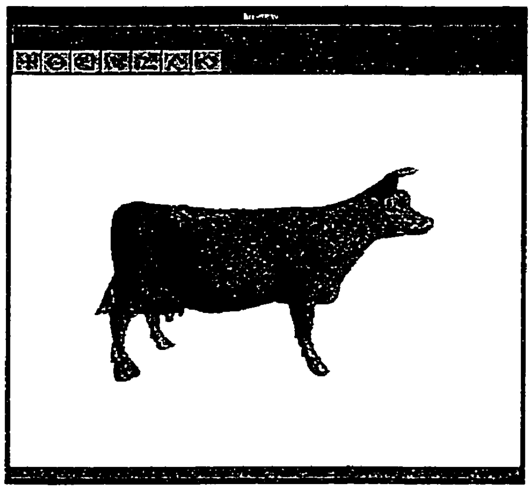

(d) Original cow model with 10862 faces

Figure 6.3: Navigating resolutions

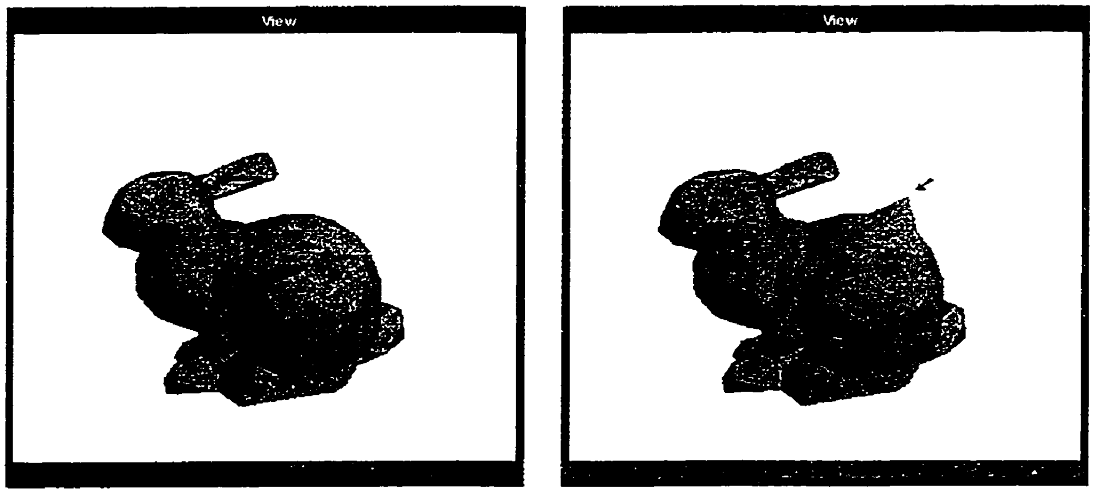
Traversal range	Vertex #	Face #	Time (sec)
0	52	100	-
500	552	1100	6.21
1000	1052	2100	1.84
1500	1552	3100	1.16
2000	2052	4100	0.93
2500	2552	5100	0.71
3000	3052	6100	0.67
3500	3552	7100	0.56
4000	4052	8100	0.56
0 $\rightarrow$ 48433	48485	96966	33.44
48433 $\rightarrow$ 0	48485	96966	28.01

Table 6.2: Navigation time for the horse model.

shown in table 6.2. The vertex tree created is a binary tree. The first row is the simplified model. The last two rows are traversals of the whole order list in both directions. The rows in the middle are traversal times with 500 node increments. The speed test is run on an SGI Octane with two 175MHz R10000 MIPS Processors[1] and 128 Mbytes main memory. It can be seen that it takes more time to traverse the first 500 nodes than the following, and for later traversals the times do not differ very much. This could be explained from the structure of the vertex tree. For a nearly balanced tree, the closer a node is to the root, the larger is its span size, and the size grows geometrically. Since the nodes at the beginning of the order list are closer to the root, traversing these nodes takes significantly more time. From the last two rows we see that collapsing the model is faster than refining the model. This can be explained by the fact that (a) refining requires visiting the span of each of the children of a parent node, while collapsing only requires a single loop through the span of the parent node; (b) refining requires checking a larger face set for visibility, because invisible faces must be checked to see if any of them become visible again, while collapsing only requires checking currently visible faces. The normals of the affected visible faces need to be updated too.

The time used for changing resolutions is dependent on the model. Since we do not store the adjacency information explicitly as in [LUE97], it requires more time for checking for face visibility and adjusting face normals. In practice, users generally do not need to work on the whole vertex tree generated from simplifying the original model, because for a fine detailed model such as the bunny model, a major portion of the intermediate levels

[1]However, only one processor is used.



(a) Original (b) Edited

Figure 6.4: Editing a single vertex. The level is simplified from the bunny model used in table 6.1. It contains 314 vertices and 607 faces.

will be quite similar to the original model and do not need manual improvements. Users typically only need to generate and work on the top portion of the whole vertex tree, which has fairly small spans and interactive response.

6.3 Improving Simplification Quality

In Chapter 5, we described three approaches to improve the simplified model. These include geometric editing with propagation, manipulating the order list, and reclustering. We will next demonstrate the effects of these functionalities. Please note that the following edits shown as images are not real examples of improving model quality. The edits are deliberately exaggerated for clearer demonstration of the effect of each operation. In real applications, edits are generally much subtler, as the purpose of the edits is to improve model quality. At the end of the chapter, we will present a more realistic example.

6.3.1 Geometric Editing

The basic editing tool is picking a vertex and changing its position. The user clicks the mouse on the model and selects a vertex. By dragging the mouse the vertex will be moved to a new position. Figure 6.4 shows an editing of a single vertex.[2]

[2]In the following images involving vertex editing, an arrow points to the vertex being manipulated.

6.3.2 Propagation to Parents

To make editing at the current level affect the coarser levels, we propagate the change to the parent. This is achieved by recalculating the ancestor quadrics and get the optimized position from the recalculated quadrics. Figure 6.5 shows propagating to the parent after a vertex has been edited. To highlight the effect of propagation to the parent, we chose two coarse levels with different degree of detail. One vertex on the bunny's head is dragged outward on the finer level, and the relatively coarser level changes similarly.

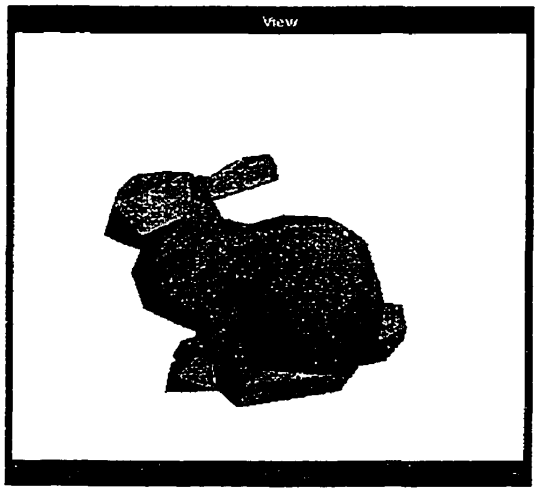
6.3.3 Propagation to Children

Here we demonstrate the effect of propagating an edit to descendent nodes. When the user selects a vertex at a certain level, a nested coordinate space is set up in the subtree below the selected vertex. Changing the selected node will move the descendent nodes along with it. Taking the two levels from figure 6.5, we reverse the operation in figure 6.6, where the edit is now on the coarser level and the change is propagated to the finer level. We see that two vertices in the finer level protruded from their original positions, due to editing on a single vertex in the coarser level. This indicates that the two vertices in the finer level are children of the vertex that was edited in the coarser level.

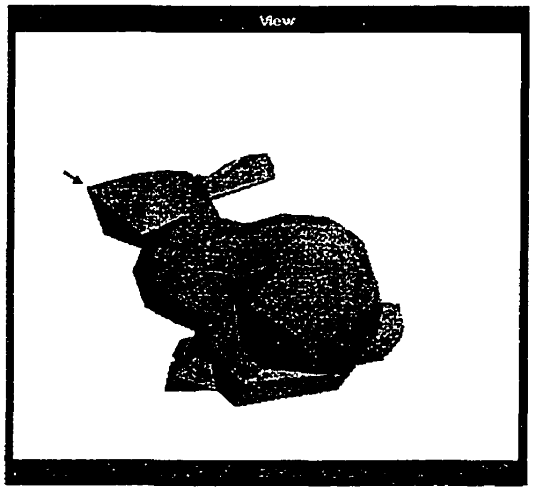
In Chapter 5, we have discussed retaining the original model and attenuating the editing effect. An example of the desired effect is given in figure 6.9. The current attenuation scheme works well for mild edits, which should be the major type of edits in practice. However, if in some occasions drastic change is needed, there might be discontinuities in some intermediate levels.

6.3.4 Propagation to Neighbors

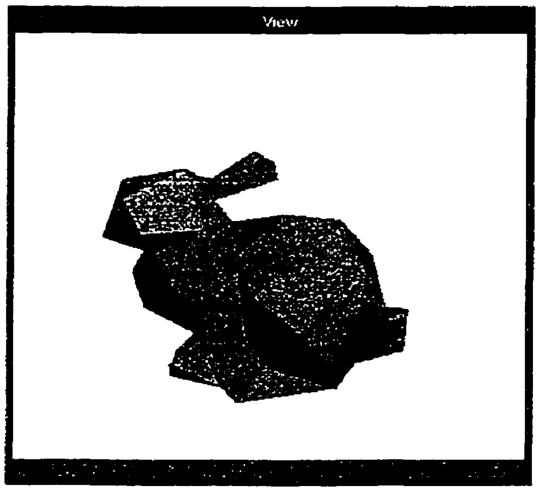
From the previous sections, we have seen propagation to the parent and children for editing a single vertex. Simply editing a single vertex is clearly too limited for larger models. To enlarge the effect of editing a single vertex, we add neighbor propagation to allow the changes of a single vertex to affect its neighbors. The user specifies a radius of a topological ring centered at the selected vertex. A segment of Bezier curve defines the shape of the propagations. Figure 6.7 shows the curve and the associated neighbor propagations. The curve in (c) has a sharper shape, which results in a sharper surface in (d).



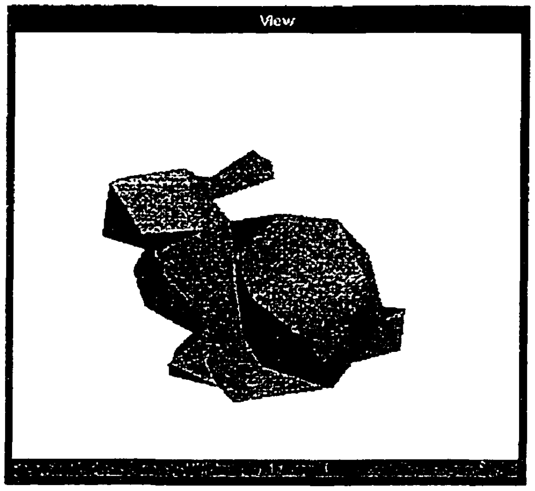
(a) The simplified bunny with 167 vertices and 319 faces.



(b) A vertex on the bunny's head is dragged outward.

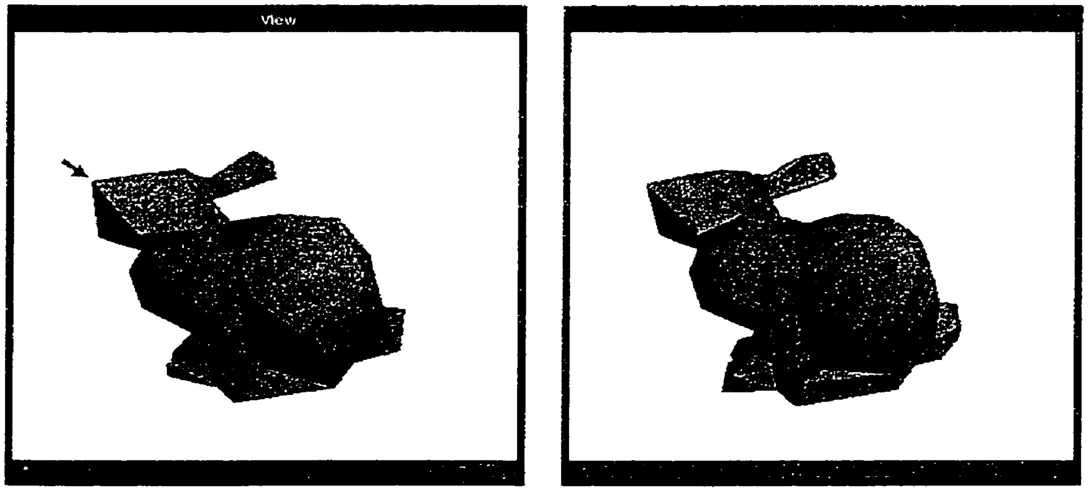


(c) A coarser level with 80 vertices and 150 faces.



(d) Change is propagated from the finer level edit.

Figure 6.5: Propagation to the parent after a single vertex editing.



(a) Bunny model with 80 vertices and 150 faces. A vertex on the bunny head is dragged out.

(b) Bunny model 167 vertices and 319 faces. The change is propagated from the coarser level edit.

Figure 6.6: Propagation to the children after a single vertex edit.

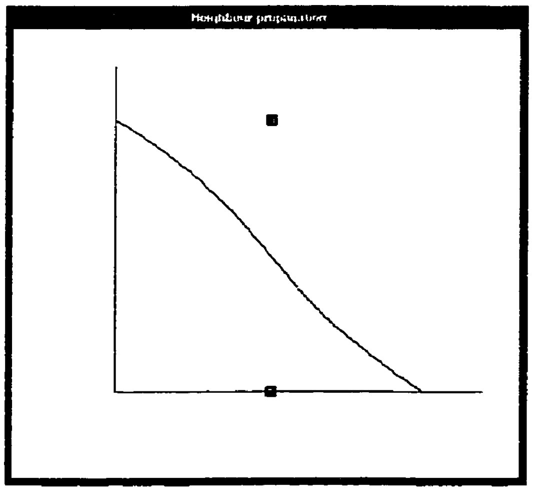
6.3.5 Combined Propagation

With neighbor propagation, we have seen that the change affects a region of the model surface. If we combine the neighbor propagation with parent and child propagations, we can affect corresponding regions in both coarse and fine resolutions. Figure 6.8 demonstrates the result with all three types of propagations turned on. For each row, the left image is the model before editing or propagations. The right image is after the editing or propagation has been applied. The first row is a level which the user manipulates. It resides between a finer level and a coarser level. After the user edit this level, the finer and coarser levels are both affected in the similar fashion.

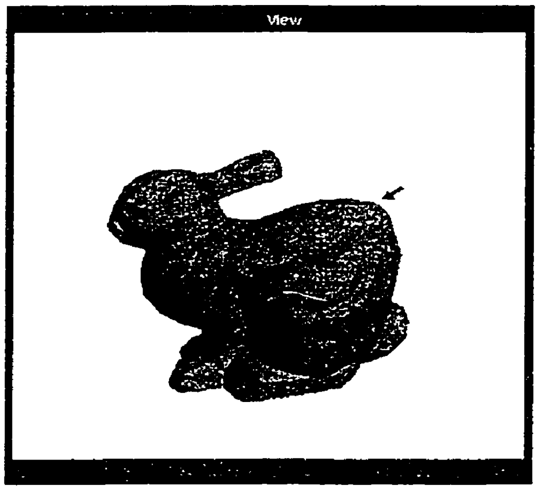
Here we also demonstrate the attenuation effect in propagation to the finer levels in figure 6.9. The vertex tree is generated from a simplified bunny model with 10000 faces. The first image is a coarse level that the user operates. Using neighbor propagation, the bunny's head is raised. The change is propagated across the finer levels with weaker and weaker effects, which can be seen from the rest of the images. When the original model is reached, the change has disappeared.

6.3.6 Order Manipulation

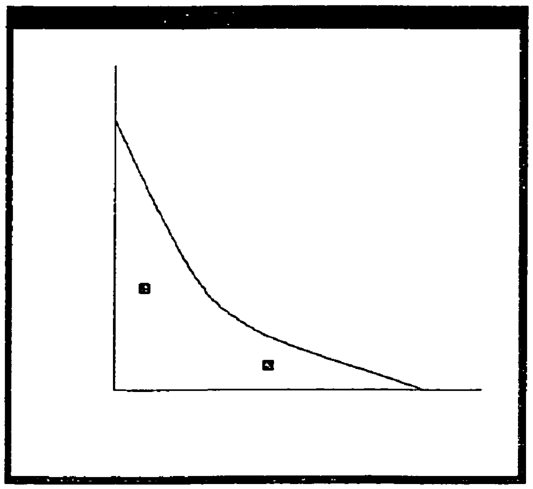
In this section we demonstrate the effect of preserving features by manipulating the order list. A series of vertices are first defined to be features. Then the user selects a lower level in



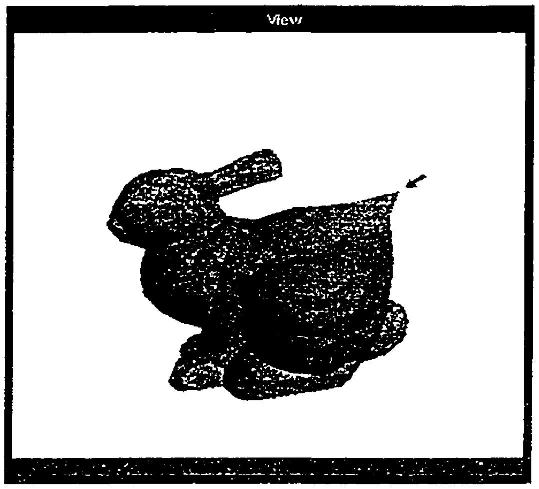
(a) The segment of Bezier curve with default shape.



(b) Edited model with propagated neighbors according to curve in (a).

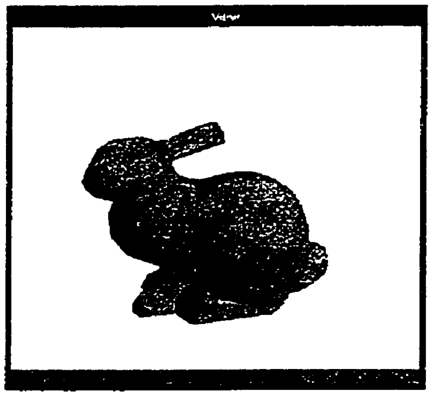


(c) The Bezier curve with an edited shape.

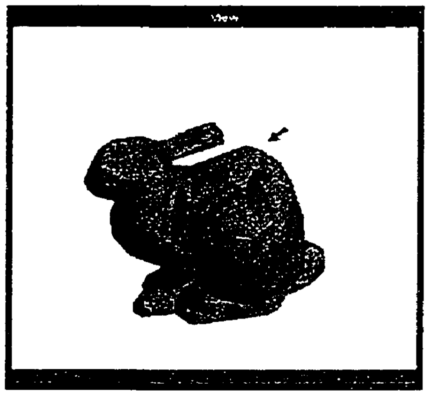


(d) Edited model with propagated neighbors according to curve in (c).

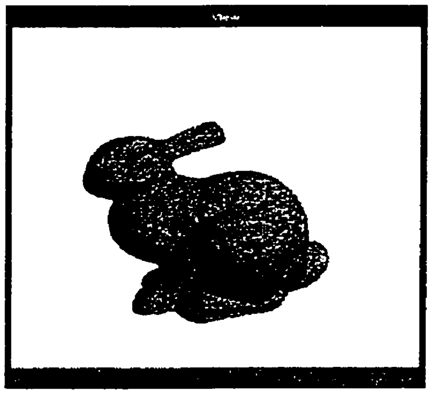
Figure 6.7: Propagation to the neighbors. The model is simplified with 791 vertices and 1557 faces. The radius of neighborhood is 5.



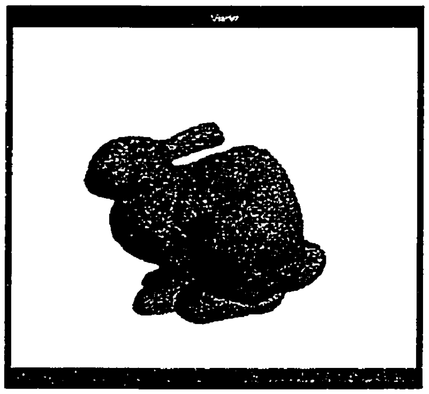
(a) Before editing



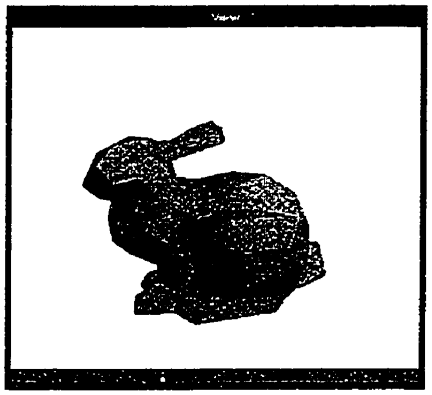
(b) After editing



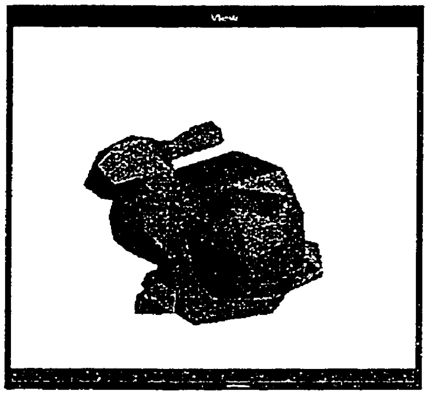
(c) Before propagation



(d) After propagation

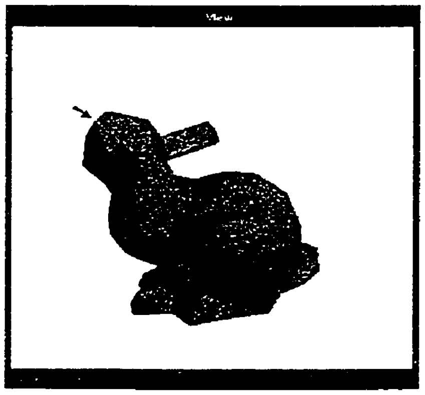


(e) Before propagation

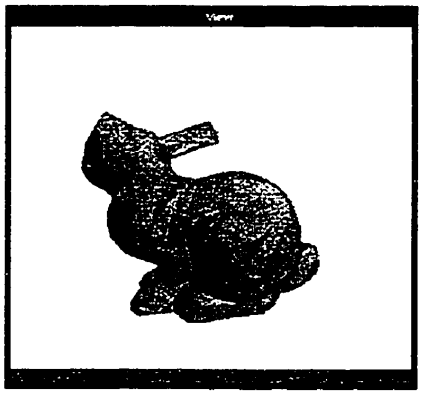


(f) After propagation

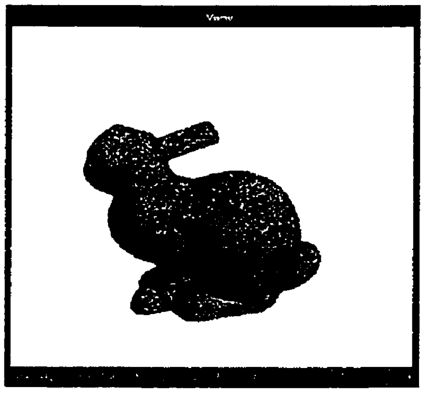
Figure 6.8: Combined propagations. (a)(b) are the level the user works on with 411 vertices and 800 faces. The neighbor radius is 5. (c)(d) has 5048 vertices and 10000 faces. (e)(f) has 157 vertices and 299 faces.



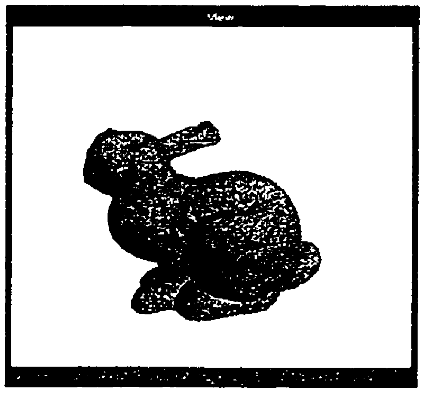
(a) Edited level with 259 vertices and 499 faces



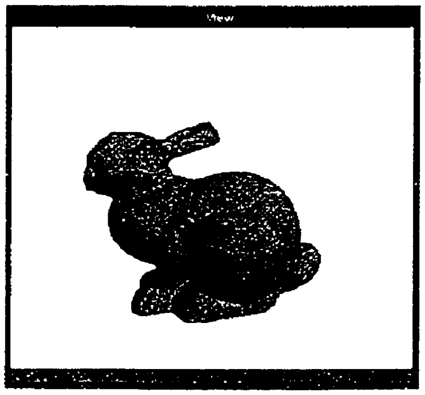
(b) Propagated level with 512 vertices and 999 faces



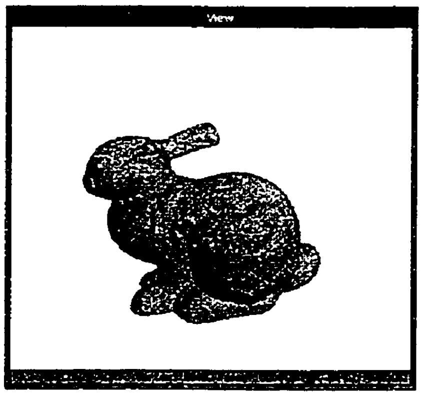
(c) Propagated level with 1017 vertices and 2000 faces



(d) Propagated level with 1521 vertices and 3000 faces



(e) Propagated level with 2026 vertices and 4000 faces



(f) The original model is left unchanged

Figure 6.9: Propagation with attenuation.

which the features are desired to appear. In figure 6.10(a), a line of feature edges are defined on the fine level. In (b), a coarse level is selected around which the feature is desired to appear. In (c), after the feature has been preserved, the current level shifted toward the fine direction by 81 nodes due to node dependencies. Compared to the total of 4993 levels, this shift is not very big. The image in (c) clearly shows the fine curve that has been preserved, while the other parts still remain crude.

6.3.7 Local Traversal

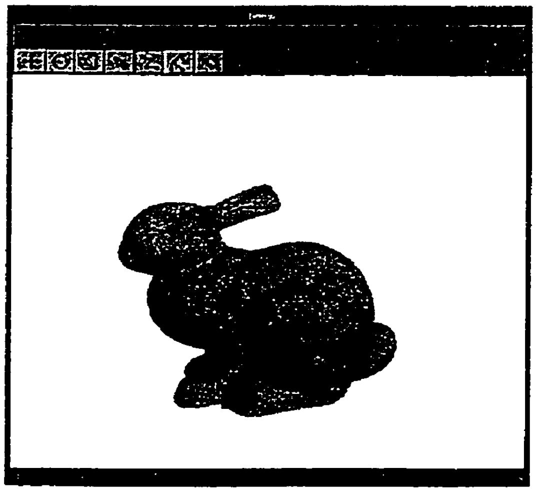
In this section we demonstrate the result of local refinement. This is one of the basic functions also achieved by the *Zeta* tool [CIG98a]. Local simplification can only be done for one collapse at a time for the moment. We demonstrate local refinement on the simplified bunny shown in figure 6.11. We iteratively refine the bunny's head on the coarser level. This added detail at the mouth and eye region. The ear is also refined a little. The refined model now has more vertices and faces. By carefully selecting important regions to refine, we can limit the number of added faces while improving the model quality.

Local simplification is shown in figure 6.12. A region is selected on the bunny's leg. Two collapses are made in the region, which fill the patch with simplified triangles.

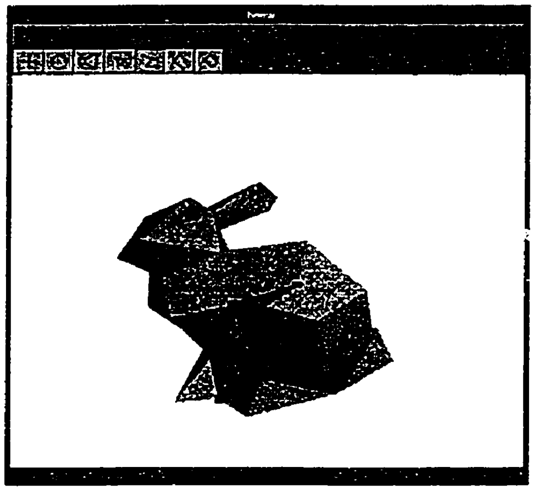
6.3.8 Reclustering

In Chapter 4, we have mentioned that the vertex tree partitions the model surface into patches. Each cut of the vertex tree corresponds to a partition of the model surface. Figure 6.13 shows the partition of the model surface defined by the direct children of the root. Since the model is simplified to 3 vertices, there are three patches in the partition, each of which have a different color in the figure. The band between the patches are faces that have vertices in different patches, which is the reason they are not colored as they do not belong to any single patch.

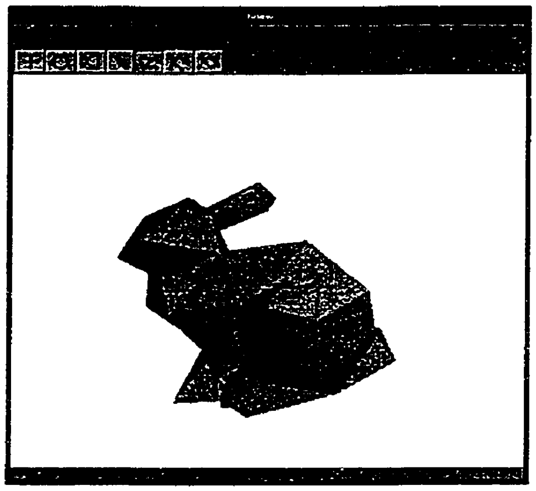
From figure 6.13 we see that the partition generated by the algorithm has no correlation to the semantic structure of the model. A human user would normally divide the model surface based on semantic meaning, making the bunny's head a patch. This indicates that there should be a subtree containing the bunny's head. We use reclustering to achieve this. The user defines a patch that should form a separate subtree. After simplification, a subtree will be created as desired. The location of the subtree depends on the size of the patch and the simplification algorithm. Patches could be defined before simplification begins, or on a intermediate level after simplification. In the latter case, the upper portion of the vertex



(a) Feature edges are defined on the fine level (5048 vertices and 10000 faces).

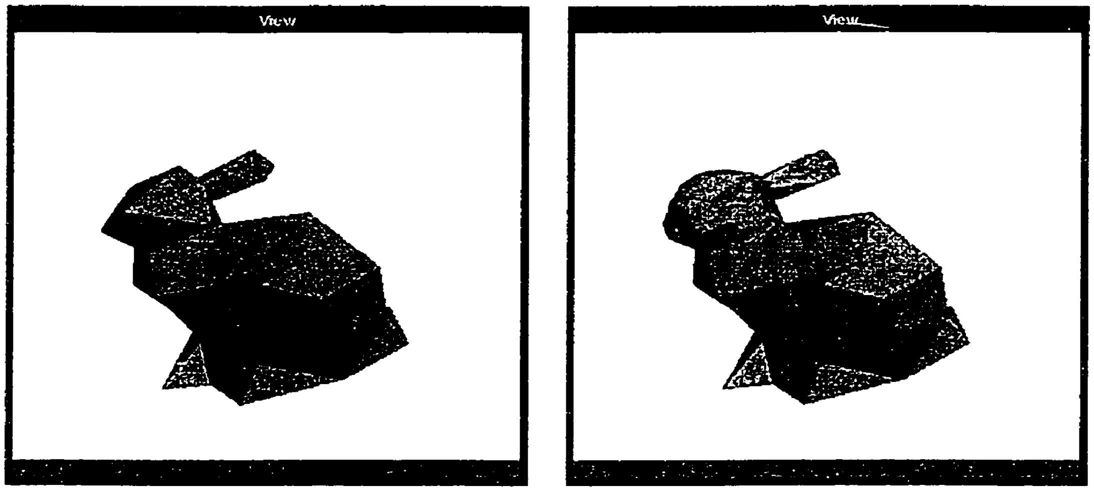


(b) A coarse level (55 vertices and 100 faces) with simplified feature.



(c) The feature is preserved to the lower level, with a shift of 81 nodes.

Figure 6.10: Preserving an edge by changing order position.



(a) The coarse level with 55 vertices and 100 faces

(b) The refined level with 98 vertices and 330 faces

Figure 6.11: Local refinement.

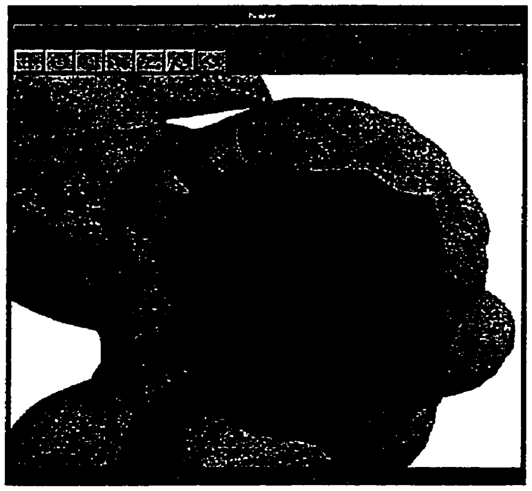
tree will be destroyed and the model is resimplified from the intermediate level. In either case, the defined patch will be guaranteed to form a subtree after simplification. Figure 6.14 shows the reclustered version of the bunny in figure 6.13, where the bunny's head is defined as a patch and reclustered as a subtree. The partition of the rest of the surface is generated by the simplification algorithm.

With proper partitioning, the model is easier to manipulate. For example, when propagating to the children, it is easier to predict the propagation when the patches are defined by the user. Another example would be that local traversal will be well confined within the local patch.

6.3.9 Preserving Feature Edges

Here we demonstrate preservation of feature edges by reclustering. The effect is similar with that of manipulating the order list. But the difference is that here the structure of the vertex tree is actually modified. Following the example in figure 6.10, we define similar feature edges and preserve them in the simplified model, which is shown in figure 6.15. The large points rendered in (a) are treated as critical points and are preserved during simplification. Other vertices are simplified within the feature edges only, so that the edge shape is better maintained.

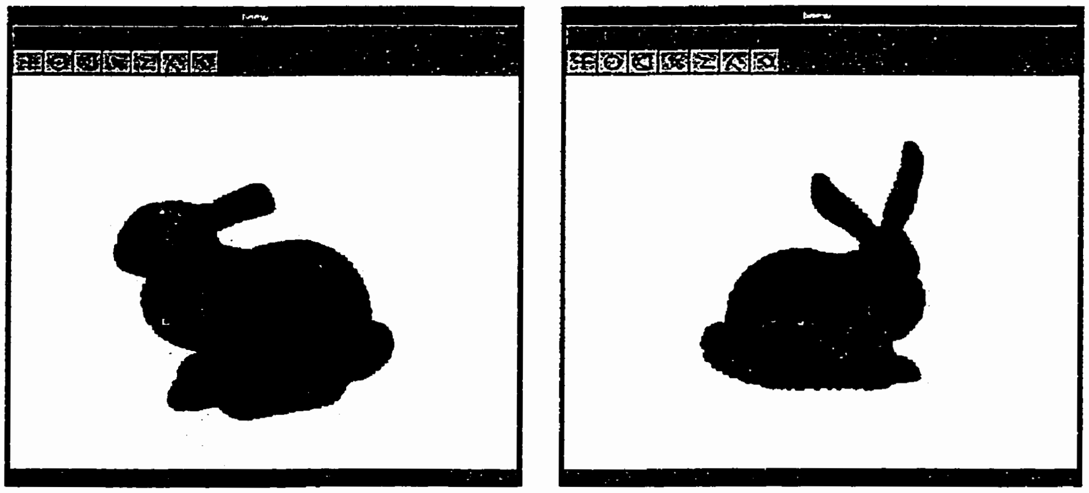
(a) The original leg region on the bunny (5048 vertices, 10000 faces)



(b) A region is selected for collapse

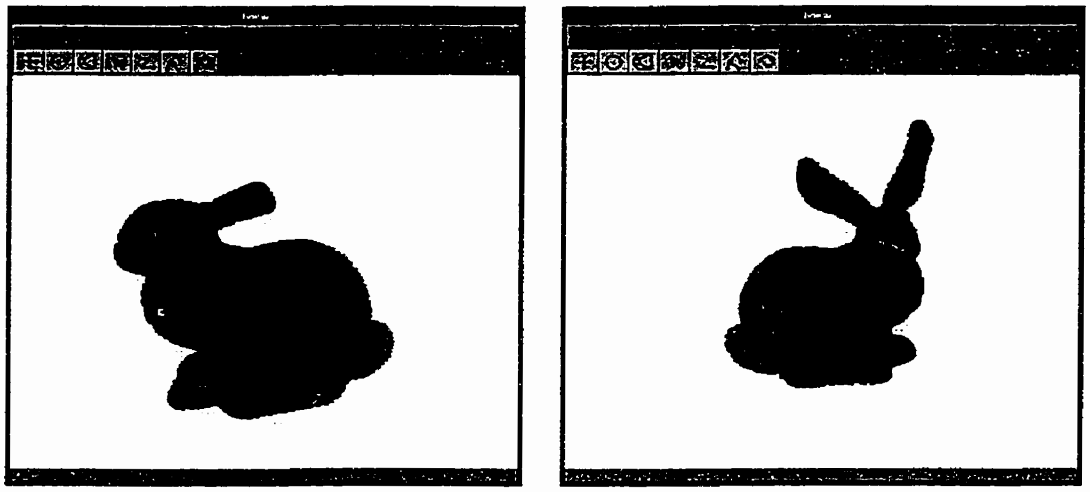
(c) The selected region is collapsed twice

Figure 6.12: Local simplification



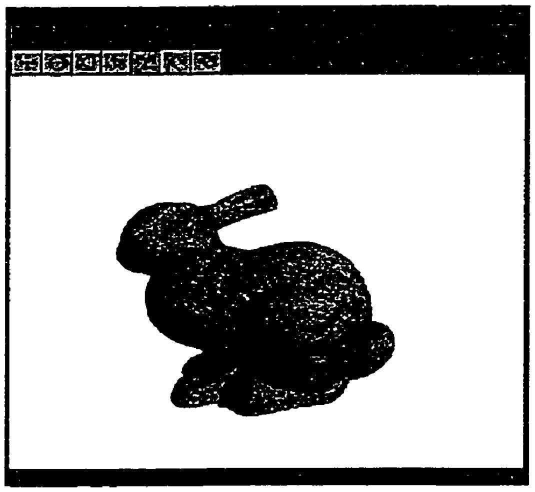
(a) The front of the model (b) The back of the model.

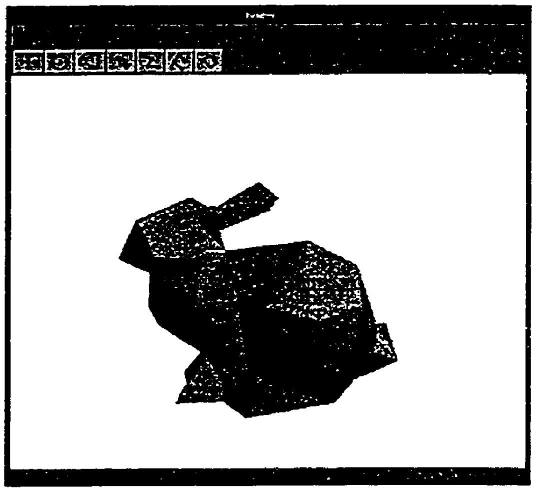
Figure 6.13: A partition of the bunny model (5048 vertices, 10000 faces). The model is simplified to 3 vertices and 2 faces. Subtree roots are children of the root of the vertex tree. The white dot in the patches are used as a seed to find the patches.



(a) The front of the model (b) The back of the model

Figure 6.14: The bunny's head is clustered as a subtree.


(a) Feature edge defined on the fine level (5048 vertices, 10000 faces)


(b) Simplified model (81 vertices, 150 faces) without preserving feature edges

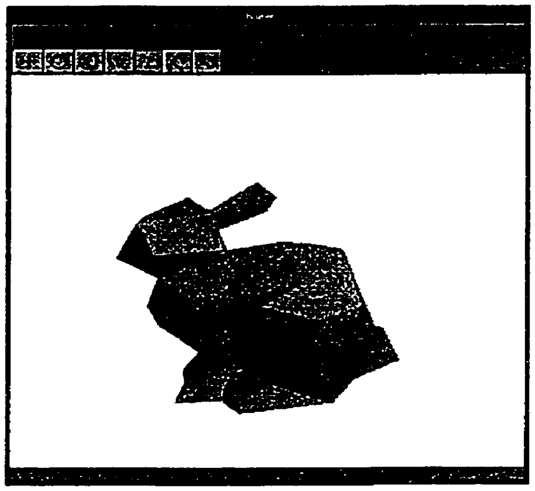

(c) Simplified model (80 vertices, 149 faces) with feature preserved

Figure 6.15: Feature preservation by reclustering

6.3.10 An Example

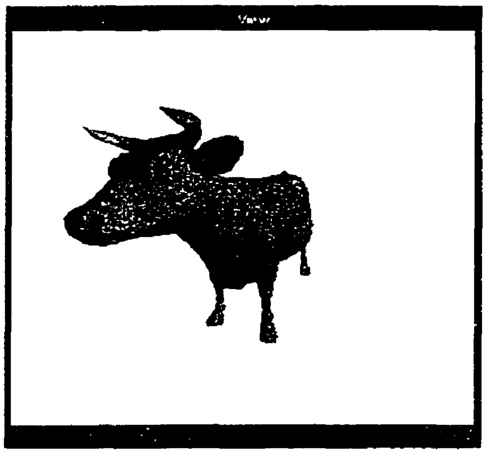
As an example of the application of this tool, we worked on the cow model to try to improve a simplification at low polygon counts. This is not an elaborate example. We mainly used order manipulation with some small usage of other techniques. Here we consider the cow's head to be more important than other parts, so we try to add more detail to the head. To maintain the low polygon counts, the body is represented in fewer number of polygons. Since the body surface is more planar, such reduction does not affect the quality significantly. Figure 6.16 shows the editing effects. Before simplification, we defined a feature edge on the cow's horn with four critical points. Thus the shape of the horn is well preserved after simplification. Then we repetitively applied local refinement and order manipulation to add or preserve important features on the cow's face. The cow's nose, eyes and ears are clearly refined after these set of operations. To reduce the polygon counts, local simplification is applied on the cow's body where the curvature is low. Important features such as the creases, the feet and nipples are especially taken care of by local refinement and order manipulation. Thus the overall quality on the body does not degrade very much. Geometric editing is applied where considered necessary.

6.4 Summary

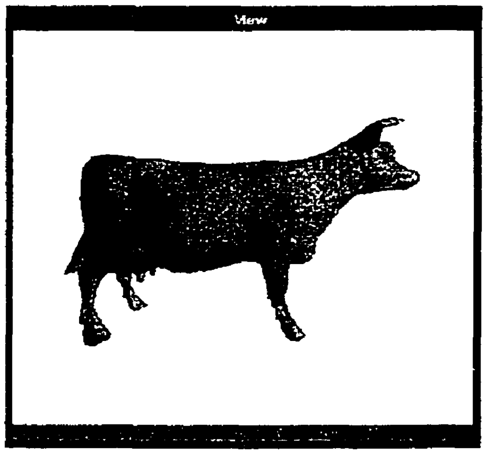
In this chapter, we have demonstrated the various functionalities to modify the multiresolution model. With these, we are hoping to offer users the ability to improve the model by working in a multiresolution context and to control the simplification process. We have been able to achieve encouraging results for the functionalities.

However, there are many places where the tool can be improved.

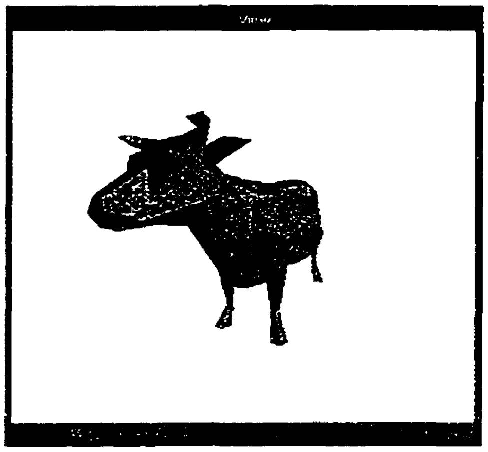
One possible improvement is the interactivity. Currently the tool is not very efficient at handling very large models. As we have seen in table 6.2, the horse is a relatively large model with over 90000 faces, and the navigation time has reached about 30 seconds for a full traversal. Although this could still be called 'interactive' to some degree, it is cumbersome to wait for half a minute to return from the end to the start of the order list. At present, the best way to work with this model is first to simplify the model automatically to a level that looks very similar with the original model, but with significantly fewer faces, and use the tool to improve the partially simplified model. This is the reason why most of the demonstrations in this chapter used a simplified bunny model with 10000 faces. In this way, we work only on the top portion of the full vertex tree and interactivity is increased. The



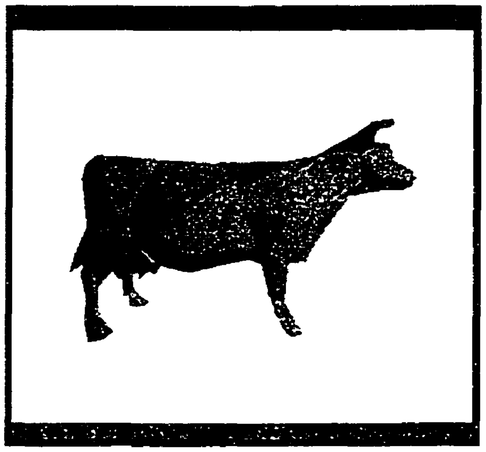
(a) Original cow model.



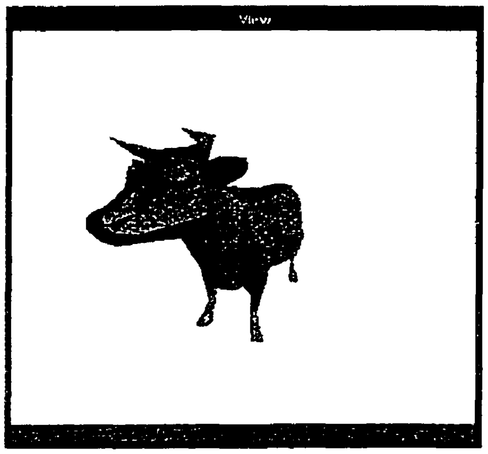
(b) Original cow model



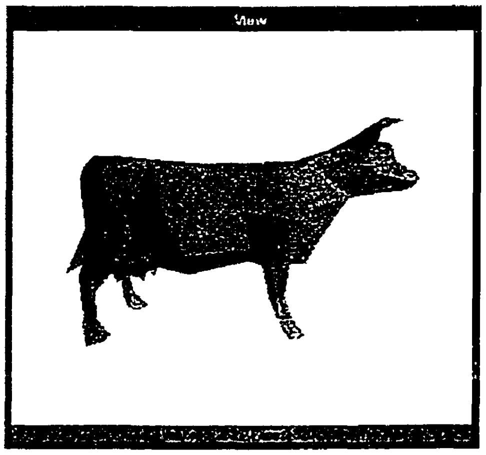
(c) The cow's head before editing



(d) The cow model before editing



(e) The cow's head after editing



(f) The cow model after editing

Figure 6.16: An example. The level being worked on has 262 vertices and 520 faces.

efficiency for our implementation could still be improved, perhaps allowing better interaction with larger models.

Another possible improvement is the attenuation effect for significant editing. Discontinuities are observed when vertex changes are big. The reason for this problem lies in the multiple levels of detail possible and the unbalanced nature of the tree. Since each cut of the vertex tree is a detail level, there are numerous combinations of tree nodes (i.e., the tree cuts) that produce valid level of detail models. The simplification algorithm guarantees that these combinations produce relatively reasonable approximations of the original model. This is the reason that we can manipulate the order list to make nodes appear in different resolutions yet still have a level without much discontinuity. However, with drastic changes on a certain level, this property is hard to preserve.

Other aspects of improvements will be classified as future directions, which we will discuss in the next chapter.

Chapter 7

Conclusion and Future Directions

In this thesis, we have introduced a tool that enables the user to manipulate model simplifications. The tool integrates simplification and modeling into one unified environment. With this tool, the user can make direct modifications to the simplified model, and instead of working on a single level of detail, the user can work in a multiresolution context, which is not offered in traditional modeling tools. It is our hope that the tool provides an environment more suitable for simplification and LOD design.

7.1 Contributions

At the beginning of Chapter 5, we have discussed the limitations of existing simplification algorithms and multiresolution modeling tools. These limitations motivated us in developing the tool presented in this thesis. As a brief overview, existing simplification algorithms are limited in that they lack interactivity, do not offer enough control for users, discard model semantics, and their assessment of simplification quality may not be consistent with perception. Existing multiresolution modeling tools do not meet these needs because they emphasize editing rather than simplification. These limitations demand a semiautomatic simplification tool.

Our work features the following contributions to 3D modeling and simplification. They are also a summary of the major components of the tool system.

- The tool unifies simplification with modeling. Simplification tools and modeling tools have been two distinct components. Coupling these two components into one unified system presents a more suitable tool for creating model simplifications and LODs.
- User edits can be propagated in three directions: the fine, coarse, and neighbor directions.

- By manipulating the order list, detail can be added or deleted as needed, in the context of the vertex tree.

- By reclustering, the vertex tree can be redefined by the user and constructed in a specific way. This will produce a tree structure that has more correlation with the semantics and practical use of the model.

- Patches, feature edges and critical points can be defined by the user and preserved during simplification.

7.2 Future Work

This thesis is our first attempt to present a modeling tool for simplification and LOD. As the work went on, many problems and ideas came up which point to interesting topics that are worth exploring.

7.2.1 A General Framework

While developing the tool, we consciously made it less dependent on the simplification algorithm integrated with the tool. Without dependency on a specific simplification algorithm, the tool can be considered a general framework with the vertex tree as the basic architecture. The simplification algorithm will be more like a 'plugin' for the framework. However, certain assumptions about the simplification algorithm are made.

The most important assumption is that the algorithm should be able to produce a vertex tree. This requires the algorithm to map several vertices to a new representative vertex. This is crucial in setting up the parent-child relationship between the nodes. Vertex clustering, edge decimation and face decimation[1]algorithms are able to generate vertex trees. Algorithms that apply global simplification operations, such as simplification envelopes [COH96], are harder to incorporate. The second dependency is found in reclustering. We force the tree to be constructed in a specific way by deleting certain edges, so that patches can be isolated and evolve into a subtree. This assumes the simplification algorithm to be an edge contraction algorithm. It would be better if a more general solution could be found that does not make such an assumption, which allows more simplification algorithms to be used. Another dependency is the order list. However, many algorithms generate such a list. This dependency could be completely removed by sorting the internal nodes in a post processing phase.

[1]The decimation algorithms should generate a representative vertex for the removed vertices.

7.2.2 Improvement of Propagation

The problem of getting the attenuation effect has been discussed at the end of last chapter. This is a problem that is yet to be solved in our future work. Other than that, the current propagation scheme has a lot of room for improvement. First, the current neighbor propagation solution is rather simple. A more sophisticated propagation scheme such as a fairing method [KOB97] might increase the power of editing. Propagation to the parent is presently achieved by recalculating the quadrics introduced by QSlim. This places another dependency on the simplification algorithm. Propagation to the children has possible room for improvement too. The current nested coordinate space approach is difficult to integrate with the attenuation effect that we have wanted to achieve. A global interpolation approach might be more suitable in this respect.

The fact that an arbitrary cut of the vertex tree produces a reasonable approximation of the original model is an implicit property of the tree generated from simplification. Such a property comes from the fact that all cuts of the vertex tree are approximations of the same surface. However, after user edits and propagation, two different surfaces emerge, in which one is the original model surface, and the other is the edited surfaces. If the two surfaces are similar, there will not be huge discontinuities. However, if the two surfaces are drastically different, some cuts will inevitably contain many discontinuities. For example, we can take some nodes from the original surface and some nodes from the edited surface and make it a cut (following the structural dependency requirements), such a cut obviously has discontinuities. A possible solution is that some more dependency requirements be placed on the possible cuts after a user edit. However, exactly what kind of dependencies should be enforced due to user edits is worth studying more.

After working on one level, the user may navigate to another level and make more modifications. It may be desirable that subsequent propagations preserve the editing on the previous level.

7.2.3 More Editing Tools

The geometric editing tool in our current system is simple, with the only ability to move the position of a vertex. More extensive tool sets should offer functionalities that resemble a traditional modeler. Specifically, it would be nice if the user could add or delete faces and vertices at arbitrary levels. This is not a trivial extension, as the operation will require modification of the vertex tree.

It would also be good to let the user edit attributes other than vertex positions. In the present system, we are able to accept models with a single texture image. This is, however, an asset that comes with QSlim, because QSlim is able to accommodate vertex attributes in generalized quadrics. Textures have an important impact on the visual quality of models. Texture boundaries have special importance similar to geometric creases on the model surface. Using Hoppe's wedges [HOP99] will enable the tool to handle multiple texture coordinates at a single vertex. However, this approach also depends on QSlim. Further work must be done in order to handle general vertex attributes with other algorithms.

7.2.4 Quality Improvement Tools

Another possible set of tools could be integrated to improve the quality of a particular level. Algorithm such as *K-Means* optimization could be used to achieve this. Usually these kinds of tools are slow and expensive. But they are useful when the user especially wants to fine-tune and improve a level, but manually doing so requires too much work.

7.3 Conclusion

Simplification and multiresolution modeling is a new and interesting area of research. This thesis presents some ideas in this area, with the implementation of a prototype. There are still many more topics to explore. Hopefully, the tool can evolve into a useful application as more problems are solved and new techniques are developed.

Bibliography

[BRO99] Dmitry D. Brodsky. R-Simp: model simplification in reverse, a vector quantization approach. Master thesis.

[CIG98] P. Cignoni, C. Rocchini and R. Scopigno. Metro: measuring error on simplified surfaces. Computer Graphics Forum, Blackwell Publishers, vol. 17(2), June 1998, pp 167-174.

[CIG98a] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno. Zeta: a resolution modeling system. GMIP: Graphical Models and Image Processing, vol.60(5), September 1998, pp.305-329.

[COH96] Johnathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, William Wright. Simplification envelopes. Computer Graphics (SIGGRAPH 1996 Proceedings), pages 119-128.

[COH98] Jonathan Cohen, Marc Olano, Dinesh Manocha. Appearance-preserving simplification. Computer Graphics Proceedings, 1998.

[DER93] Tony D. DeRose, Michael Lounsbery, Joe Warren. Multiresolution analysis for surfaces of arbitrary topological type. Technical Report Number 93-10-05.

[ECK95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, Werner Stuetzle. Multiresolution analysis of arbitrary meshes. Technical Report # 95-01-02.

[GAR99] Michael Garland. Quadric-based polygonal surface simplification, PhD dissertation.

[GAR99a] Michael Garland. Multiresolution modeling: survery & future opportunities. Eurographics'99, State of the Art Report (STAR).

[GUE98] Andre Gueziec, Gabriel Taubin, Francis Lazarus and William Horn. Simplicial maps for progressive triansmission of polygonal surfaces. Proceedings of the third symposium on Virtual reality modeling language February 16 - 20, 1998, pages 25-31.

[HAM94] Bernd Hamann. A data reduction scheme for triangulated surfaces. Computer Aided Geometric Design 11, 1994.

[HEC97] Paul S. Heckbert, Michael Garland. Survey of polygonal surface simplification algorithms. SIGGRAPH 97 Course notes.

[HIN93] Paul Hinker, Charles Hansen. Geometric optimization. Visualization'93 Proceedings, pages 189-195.

[HOF89] Christoph M. Hoffmann. Geometric & solid modeling, an introduction.

[HOP93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald and Werner Stuetzle. Mesh optimization. Computer Graphics (SIGGRAPH 1993 Proceedings), pages 19-26.

[HOP96] Hugues Hoppe. Progressive meshes. Computer Graphics (SIGGRAPH 1996 Proceedings), pages 99-108.

[HOP97] Hugues Hoppe. View-dependent refinement of progressive meshes. Computer Graphics (SIGGRAPH 1997 Proceedings), pages 189-198.

[HOP99] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. IEEE Visualization 1999, October 1999, pages 59-66.

[KAL96] Alan D. Kalvin, Russel Taylor. Superfaces: polyhedral mesh simplification with bounded error, lecture slides. IEEE Computer Graphics and Applications, 16(3), May 1996.

[KOB97] Leif Kobbelt. Discrete fairing. Proceedings of the Seventh IMA Conference on the Mathematics of Surfaces, 1997, pp. 101–131.

[KOB98] Leif Kobbelt, Swen Campagna, Jens Vorsatz, Hans-Peter Seidel. Interactive multi-resolution modeling on arbitrary meshes. Computer Graphics (SIGGRAPH 1998 Proceedings), pages 105-114.

[LEE98] Aaron W.F. Lee, Wim Swildens, Peter Schrőder, Lawrence Cowsar, David Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. Computer Graphics Proceedings, 1998.

[LIN99] Peter Lindstrom, Greg Turk. Evaluation of memoryless simplification. IEEE Transactions on Visualization and Computer Graphics, Vol. 5, No. 2, April-June 1999.

[LOW97] Kok-Lim Low, Tiow-Seng Tan. Model simplification using vertex-clustering. The Symposium on Interactive 3D Graphics, SIGGRAPH 1997, pages 75-81.

[LUE97] David Luebke, Carl Erikson. View-dependent simplification of arbitrary polygonal environments, Computer Graphics (SIGGRAPH 1997 Proceedings).

[PUP97] Erico Puppo. Simplification, LOD and multiresolution - principles and applications. Eurographics'97, Tutorial notes.

[ROS93] Jarek Rossignac, Paul Borrel. Multi-resolution 3D approximations for rendering complex scenes. Geometric Modeling in Computer Graphics, June 28-July 2, 1993, pages 455-465.

[ROS97] Jarek Rossignac. Geometric simplification and compression. SIGGRAPH 97 Course notes.

[SCH92] William J. Schroeder, Jonathan A. Zarge and William E. Lorensen. Decimation of triangle meshes. Computer Graphics (SIGGRAPH 1992 Proceedings), pages 65-70.

[TAU95] Gabriel Taubin. A signal processing approach to fair surface design. Computer Graphics (SIGGRAPH 1995 Proceedings), pages 351-358.

[TUR92] Greg Turk. Re-Tiling Polygonal Surfaces. Computer Graphics (SIGGRAPH 1992 Proceedings), pages 55-64.

[VAR95] Amitabh Varshney, Pankaj K. Agarwal et al. Generating levels of detail for large-scale polygonal models.

[XIA96] Julie C. Xia, Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. IEEE Visualization 1996.

[ZOR97] Denis Zorin, Peter Schröder, Wim Sweldens. Interactive multiresolution mesh editing. Computer Graphics (SIGGRAPH 1997 Proceedings), pages 259-268. GMIP: Graphical Models and Image Processing, vol.60(5), September 1998, pp. 305-329.