University of Groningen **Faculty of Mathematics and Natural Sciences**

Department of Mathematics and Computing Science

Geometric Simplification Algorithms for Surfaces

A. Noord
R.M. Aten

Advisors:
dr. J.B.T.M. Roerdink
drs. M.A. Westenberg

June, 1998

RuG

Geometric Simplification Algorithms for Surfaces

A. Noord R.M. Aten

Abstract

To represent 3-dimensional objects in computer graphics, triangles are a popular graphics primitive. To represent a model with sufficient detail usually a large amount of triangles is needed. But sometimes less detail is desired to speed up computation or to reduce memory usage. This calls for an algorithm to simplify the model.
Two such algorithms are studied. The first one, designed by Schroeder, Zarge and Lorensen, was already implemented in the Visualization Toolkit (VTK). The second one, by Klein, Liebich and Straßer, was not. This algorithm promised better results than the first method because it uses a better way to measure the error between the original and simplified model. It was implemented in VTK and the results of both algorithms are compared.

Samenvatting

Om door middel van computer graphics driedimensionale modellen weer te geven, worden vaak driehoeken gebruikt. Om een model met voldoende detail weer te geven is vaak een grote hoeveelheid driehoeken nodig. Maar soms is minder detail gewenst om berekeningen te versnellen of geheugengebruik te beperken. Dit vraagt om een algoritme om het model te vereenvoudigen.
Twee zulke algoritmen zijn bestudeerd. Het eerste, ontworpen door Schroeder, Zarge en Lorensen, was al in de Visualization Toolkit (VTK) geïmplementeerd. Het tweede, van Klein, Liebich en Straßer, was dat nog niet. Dit algoritme beloofde betere resultaten dan de eerste methode omdat het een betere manier gebruikt om de fout te meten tussen het originele en vereenvoudigde model. Dit algoritme is geïmplementeerd in VTK en de resultaten van beide algoritmen worden vergeleken.

Contents

Preface

When the time had come to start thinking about graduating, both of us wanted to do something in the field of visualization, preferably virtual reality (VR). After playing with the VR system for a short time, Jos Roerdink helped us shape our project. Although not directly related with VR, the results of the project can be used with VR applications.
At first, we intended to do more than just studying and implementing one algorithm, but the complexity of the algorithm did not permit us to do more within the given time.
This thesis presents the results of nine months of work on our graduate project.

We would like to thank Jos Roerdink for his guidance and Michel Westenberg for his endless patience.

Arjan Noord and Robert Aten

1 Introduction

In the field of computer graphics and in scientific and engineering applications more and more complex graphical models are used, consisting of thousands or millions of polygons and often including additional information like color or texture. This increase in model size is caused by the application of mesh generation methods to larger datasets (for a precise definition of 'mesh', see section 1.1).
Marching cubes methods, for example, are used in visualization to construct an iso-surface from a 3D field of values. They can easily produce meshes consisting of 500.000 to 2.000.000 triangles. Measuring devices like Magnetic Resonance (MR) scanners, range cameras and satellites generate high resolution datasets. Sampling resolutions are increasing fast, causing ever growing model sizes.

To be able to cope with these growing model sizes, an increase in computing performance is needed. Often the available computing power is not sufficient, for example for interactive (real-time) 3D rendering. Transmission of large 3D meshes over a computer network is slow due to limited bandwidth.

Clearly some sort of simplification is desired to reduce the size of the models. After applying such a reduction routine to a triangle mesh, the new mesh must meet the following requirements:

- It must preserve the original topology of the mesh (see section 1.1).
- It must form a good approximation of the original mesh.
- Optionally, the vertices of the simplified mesh should be a subset of the original vertices. This means that no new vertices can be created.

Over the years a variety of methods for simplifying triangular meshes, modeling height fields and surfaces — originating from a large number of fields — have been explored. Such methods are regular grid methods, hierarchical subdivision methods, feature methods, refinement methods, decimation methods, etc.[10]. In this report we will concentrate on decimation methods.

Decimation methods simplify the original mesh by removing vertices from it and patching the resulting holes by a new triangulation. This is repeated until a certain termination criterion is met.

In the next two chapters two such decimation algorithms, designed by Schroeder, Zarge and Lorensen [2] and by Klein, Liebich and Straßer [6], are discussed. The algorithm designed by Klein *et al.* was implemented and compared to the algorithm by Schroeder *et al.*. This is described in chapter 4. Conclusions are drawn in chapter 5.

1.1 Surfaces, meshes, and topology

In this thesis surfaces[1] (2-manifold) embedded in 3-dimensional Euclidean space are considered, which are approximated by *triangle meshes* ('meshes', for short). In this context, a triangle mesh is a piecewise linear surface consisting of triangular faces which are pasted together along their edges; it is constructed from points (vertices), edges and triangles (interior included). These building blocks are referred to as *simplices*: a vertex is a 0-simplex, an edge a 1-simplex, a triangle a 2-simplex. If t is a k-simplex, the integer k is called its *order*[2]. Edges of a triangle, or vertices of an edge are called *faces*.

Triangle meshes

More precisely, a finite set T of simplices consisting of vertices, edges and triangles, is called a *triangle mesh* if the following conditions hold:

1. for each simplex $t \in T$, all faces of t belong to T;
2. for each pair of simplices $t, t' \in T$, either $t \cap t' = \emptyset$ or $t \cap t'$ is a simplex of T;
3. each simplex t is a face of some simplex t' (possibly coinciding with t) having maximum order among all simplices of T.

The union of all simplices of a triangle mesh T, considered as point sets in $\mathbb{R}^3$, is called the *domain* of T. A triangle mesh T whose domain is a set Ω is called a triangular decomposition or *triangulation* of Ω. Often, the distinction between a triangle mesh T and its domain is not strictly maintained. The intended meaning will be clear from the context.
A surface is called *orientable* when it can be represented by the surface of a sphere with g handles (for example, a torus can be viewed as a sphere with one handle). Models that cannot be represented by such meshes are for example open or closed curves, polyhedral volumes, non-orientable surfaces (e.g. Möbius strips), or non-manifolds (e.g. two cubes joined along an edge).

Since the main subject of this thesis is mesh simplification, triangle meshes will be transformed into other triangle meshes by deleting vertices from the original mesh T. During this process, a face of T may turn from a triangle into a simple (i.e. non-selfintersecting) polygon. To obtain a new triangle mesh T' this polygon has to be retriangulated. A *triangulation* of a polygon P is a decomposition of P into triangles by a maximal set of non-intersecting diagonals (the diagonals have to be maximal to prevent that a polygon vertex is in the interior of a triangle edge). After retriangulation a new triangle mesh T' is obtained.

[1] Surfaces with boundary (for example, a hemisphere) are allowed.
[2] This can be generalized by introducing the notion of *simplicial complexes* [10]

Topology

An important property when approximating a surface by a triangle mesh, or an initial triangle mesh by a simpler triangle mesh, is the *topology* of the two surfaces (original and approximation). This concept will not be described in full mathematical detail, but instead an intuitive explanation of its importance will be presented. One may transform a given surface S into another surface S' by an elastic deformation D, that is, an invertible transformation which does not tear the surface apart. Certain properties of the surface S are preserved under such a deformation. The surfaces S and S' are said to be *topologically equivalent* or *homeomorphic*. The transformation $D : S \to S'$ is called a *topological mapping* or *homeomorphism* and D is said to *preserve the topology* of the surface. Similar considerations apply to deformations of curves. Examples of topological mappings between triples of curves or surfaces are given in figure 1.1.

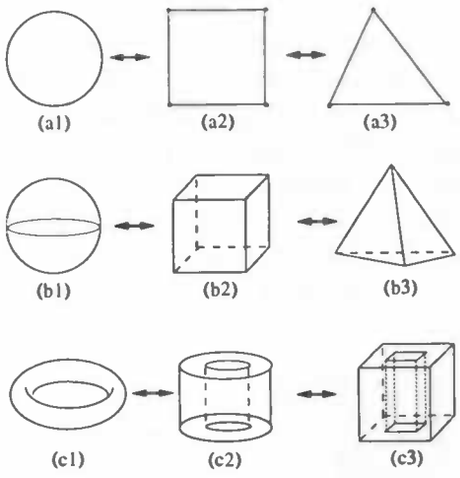


Figure 1.1: *Three rows of topologically equivalent surfaces (equivalence is indicated by a double arrow). Row 1: (a1): boundary of a circle; (a2): boundary of a square; (a3): boundary of a triangle. Row 2: (b1): surface of a sphere; (b2): surface of a cube; (b3): surface of a tetrahedron. Row 3: (c1): surface of a torus; (c2): surface of a cylinder with cylindrical hole; (c3): surface of a cube out of which a smaller cube is deleted.*

2 Decimation of Triangle Meshes

2.1 Introduction

Schroeder, Zarge and Lorensen [2] developed a general vertex decimation algorithm primarily for use in scientific visualization. Their method takes a triangulated surface as an input, typically a manifold with boundaries. In a manifold topology a region around each point is topologically equivalent to a disk (in 2D) or a ball (in 3D). That is, a small disk or ball can be placed on the surface without tearing or overlapping (see also section 1.1). Topology that is not manifold is called non-manifold. Examples of non-manifold topology include triangle meshes where an edge is used by more than two triangles, or triangles connected to each other at their vertices (i.e., not sharing an edge). See figure 2.1.

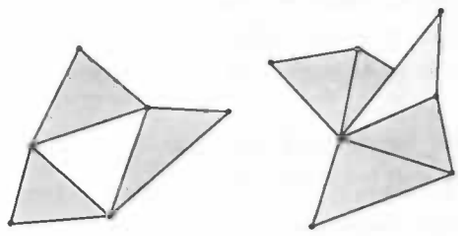


Figure 2.1: *Examples of non-manifold topologies.*

2.2 The algorithm

The decimation algorithm makes multiple passes over the data and on each pass, every vertex is candidate for deletion, except *complex* vertices (see section 2.2.1). If the vertex meets the decimation criteria, the vertex and all associated triangles are deleted. The resulting hole is then patched by a local triangulation. This process is repeated until some termination criterion is met (usually a reduction percentage).

The algorithm consists of three steps:

1. characterize the local vertex geometry and topology
2. evaluate the decimation criteria
3. delete the vertex and triangulate the resulting hole

2.2.1 Characterize the local vertex geometry and topology

In the first step of the algorithm, the local geometry and topology of the vertex are determined. For each vertex v, five different types are distinguished: simple, complex, boundary, interior edge, corner.

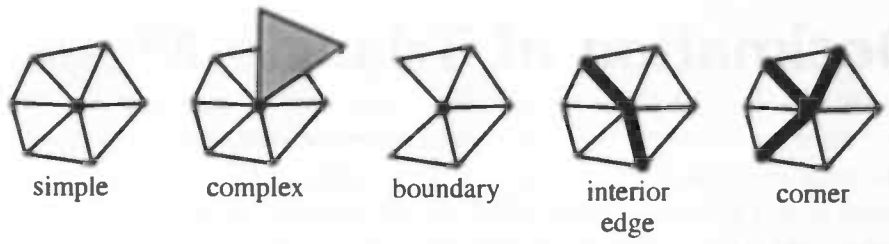


Figure 2.2: *Classification of vertices into five types.*

A *simple* vertex is always surrounded by a complete cycle of triangles (see figure 2.2) and each edge that uses the vertex is used by exactly two triangles. If the edge is used by more than two triangles or if the vertex is used by a triangle not in the cycle of triangles surrounding the vertex, then the vertex is *complex*.

If the angle between two adjacent triangles is greater than a specified *feature angle*, a *feature edge* exists. A vertex is called an *interior edge* vertex if it is used by two feature edges. If one or three or more feature edges use the vertex, it is classified as a *corner* vertex. If the vertex is surrounded by a semi-circle of triangles, it is considered a *boundary* vertex.

A complex vertex can never be deleted, because deleting it would change the topology of the mesh. All other vertices can be removed.

2.2.2 Evaluate the decimation criteria

The previous characterization step produces a list of vertices that surround the vertex v. All vertices in the list must be connected to v by an edge. This list is ordered in such a way that when the subsequent vertices from the list are connected, they form (part of) a circle around v. See figure 2.3.

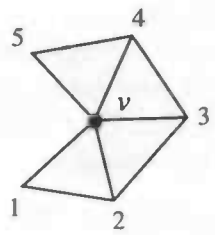


Figure 2.3: *Ordered loop of vertices that surround v.*

In the same way an ordered list of triangles is produced. The evaluation step determines

whether the triangles forming the loop can be deleted and replaced by another triangulation (without the original vertex and associated triangles).
The algorithm uses vertex distance to plane (or edge) as a decimation criterion, but other criteria may be applied as well. The distance is computed as follows. First, an average plane is constructed. Using triangle normals $\vec{n}_i$, centers $\vec{x}_i$ and areas A_i, an average normal $\vec{N}$ is computed, which is then normalized ($\bar{n}$). The center of the average plane is represented by $\bar{x}$.

$$\vec{N} = \frac{\sum_i \vec{n}_i A_i}{\sum_i A_i}, \qquad \bar{n} = \frac{\vec{N}}{|\vec{N}|}, \qquad \bar{x} = \frac{\sum_i \vec{x}_i A_i}{\sum_i A_i}$$

where i runs over all triangles in the loop. The distance of the vertex v with position $\vec{v}$ to the plane is $d = |\bar{n} \cdot (\vec{v} - \bar{x})|$. If this distance is within a specified range v may be deleted, otherwise it is retained.

For boundary and interior edge vertices the distance to edge criterion is used instead of the distance to plane criterion. In this case the algorithm determines the distance to the line defined by the two vertices creating the boundary or feature edge. The example in figure 2.4 shows why this might be necessary.

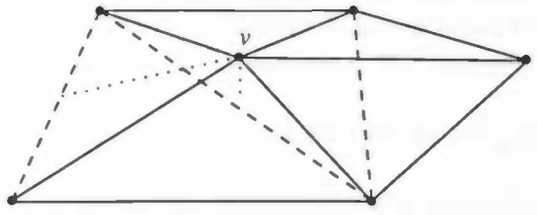


Figure 2.4: *The dashed lines represent a possible triangulation after the deletion of vertex v. The dotted lines represent the vertex distance to edge and vertex distance to plane, respectively. Although the distance to the average plane is relatively small, the error introduced by deletion is fairly large.*

However, sometimes it is not desirable to retain feature edges, for example if they are the result of noise in the original mesh. In this case interior edge vertices can be evaluated using distance to plane. This is called *edge preservation.*

After deletion of the vertex v and the triangles in the loop, the resulting hole must be triangulated. In case of an interior edge, the loop must be split in two halves first, with the split line connecting the vertices forming the feature edge. The two halves are then triangulated separately, so that the feature edge is preserved.

2.2.3 Triangulate the resulting hole

The loop (or two loops in case of a feature edge) created by the previous step must now be triangulated. A 3D triangulation method has to be used since the loops are generally non-planar. If a loop cannot be triangulated, for example when no split plane can be found (see section 2.3.2), the vertex will not be removed. Recursive loop splitting triangulating schemes

have proven to be effective and are quite simple. This algorithm uses such a triangulation method.

After the triangulation is completed successfully, the vertex is *actually* deleted.

2.3 Implementation

The algorithm is available as a filter in the Visualization Toolkit (VTK) [5] and is also available as a separate program called the Decimate System.
In both implementations the initial distance, distance increment and maximum distance can be set to control the decimation criterion (in the VTK implementation this is done by setting the initial error, the error increment and the maximum error). By gradually increasing the distance, first the flat parts of the mesh are simplified and later the more curved parts follow. Both implementations also allow modifying parameters for the feature angle, the target reduction and the triangles' aspect ratio. The `MaximumSubIterations` parameter limits the number of passes that are made over all vertices in the mesh without increasing the distance. The `MaximumIterations` parameter limits the number of distance increments.
Because the Decimate System is commercial software and no license was available, no further study of this implementation was done. In the following sections, the VTK implementation will be discussed and compared to the article [2].
First some important design steps are reviewed.

2.3.1 Data structure

The Visualization Toolkit features a large range of data structures to cope with many different types of data. Structured grid data, unstructured grid data, rectilinear grid data, polygonal data, etc. can all be represented. The decimate algorithm only uses polygonal data, represented by the class `vtkPolyData`. This will be discussed extensively.

The data objects in VTK are called *datasets*. The dataset can be divided in two parts: the organizing structure and the supplemental data attributes.

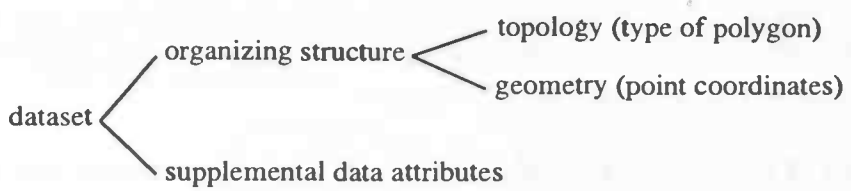


Figure 2.5: *Data subdivision.*

The organizing structure can be split into the topology (the type of polygon) represented by *cells* and the geometry (the point coordinates) represented by *points*. A dataset consists of one or more cells. See figure 2.5. The supplemental data attributes represent additional information, such as temperature values, inertial mass, etc.

The main building block of the data structure is the cell, which is defined by specifying

a type combined with an ordered list of points. This list is called the *connectivity list.* Together, the connectivity list and the type specification define the topology of the dataset. The x-y-z point coordinates define the dataset geometry.

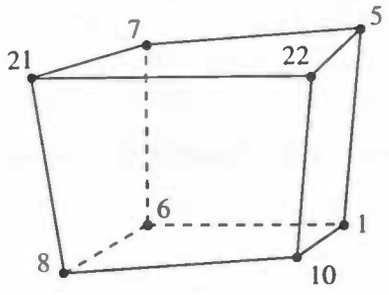


Figure 2.6: *A hexahedron example.*

For example, the hexahedron in figure 2.6 can be represented by

```
TYPE         : hexahedron
CONNECTIVITY : (8,10,1,6,21,22,5,7)
```

The numbers in the connectivity list (and in figure 2.6) are point IDs and thus refer to points in the point list. In the point list the actual coordinates are stored.

Each cell is a subclass of the abstract class `vtkCell` (see figure 2.7).

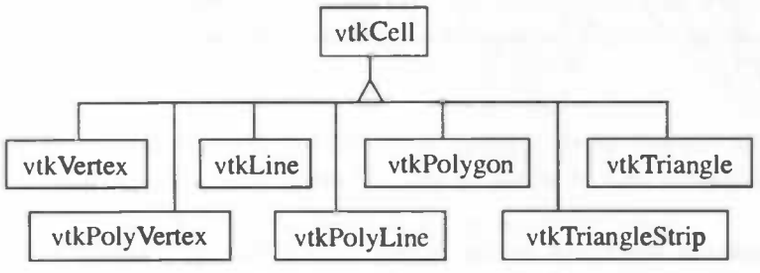


Figure 2.7: `vtkCell` *class and its subclasses.*

Within the `vtkPolyData` structure the following cell types are possible:

- vertex, a single point
- poly-vertex, an ordered list of points
- line, defined by two points, direction from first to second point
- poly-line, one or more connected lines; defined by an ordered list of $n+1$ points, where $(i, i+1)$ defines a line
- triangle, a counter-clockwise ordered list of three points
- triangle strip, one or more triangles; defined by an ordered list of $n+2$ points, where $(i, i+1, i+2)$ defines a triangle (see figure 2.8)
- polygon, an ordered list of three or more points lying in a plane

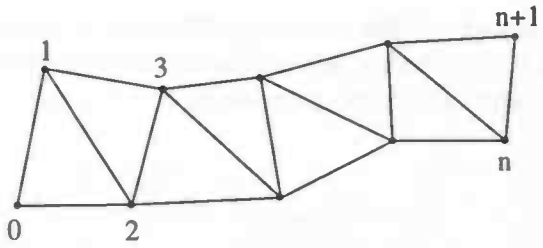


Figure 2.8: *Example of a triangle strip.*

A collection of cells of the same type can be stored in the `vtkCellArray` structure. This is a list of integers. The first number indicates the number of points in the cell connectivity, followed by a number of point IDs. This sequence is repeated for each object.

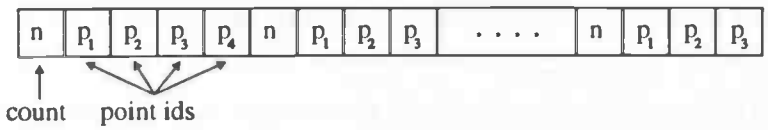


Figure 2.9: *The cell array structure.*

In this structure, type information is not explicitly represented. Instead, `vtkPolyData` maintains four separate lists to *vertices*, *lines*, *polygons* and *triangle strips*. The cell type can be derived from the list the cell belongs to (sometimes the number of points that define the cell is also needed).

A cell list can be implemented to speed up searching the cell array. The cell list is a structure with type information and an offset in the cell array (see figure 2.11).

The implementation described so far is not very efficient. To implement an operation to retrieve vertex or edge neighbors (often done during decimation), the cell list has to be searched, requiring $O(n)$ time. The reason for this inefficiency is that the data representation is downward, as shown in figure 2.10 a).

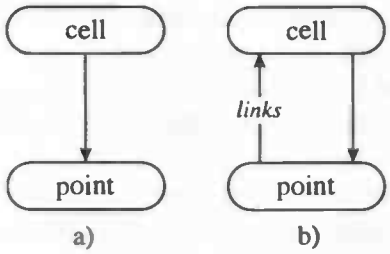


Figure 2.10: *The data representation hierarchy.*

So, in order to improve the efficiency of this representation, extra information has to be added

that allows upward traversal in the hierarchy.

The solution Schroeder *et al.* found was extending the structure with a *link list.* The link list is a list of lists of cells that use each point. It corresponds to the upward links in figure 2.10 b). At position i the link list holds data on the point with ID i. It holds the number of cells that use point i (`ncells`) and the list of cells that use i (`cells`).
Now that the upward link has been implemented, the structure has been transformed into a ring.

The full data structure is shown graphically in figure 2.11.

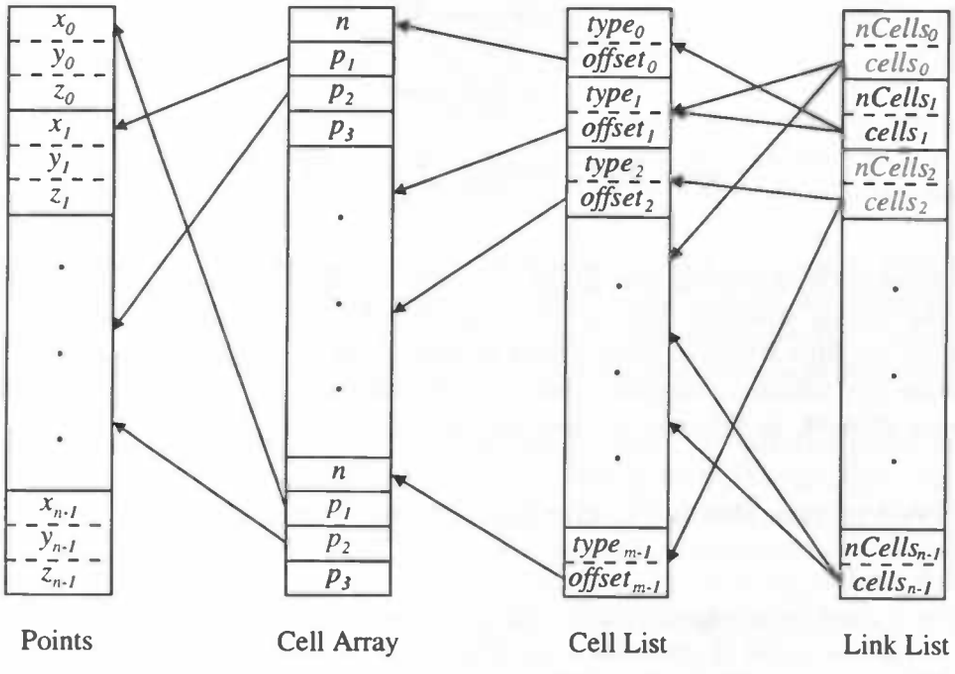


Figure 2.11: *The data representation structure.*

In short, the `vtkPolyData` structure contains four cell arrays for vertices, lines, polygons and triangle strips. It contains a cell list to keep track of cell types and and their place in the cell array, and a link list to make upward traversal possible and ensure very efficient data access.

2.3.2 Triangulation

In the VTK implementation, the triangulation method used is *recursive loop splitting.*
Each loop to be triangulated is split into two halves along a line (the split line). The split line is defined by two non-neighboring vertices in the loop. Each new loop is divided again, until only three vertices remain in each loop, which then form a triangle.
Because the loop is usually non-planar, a split plane is used. This is a plane through the split line and orthogonal to the average plane. By checking that every point in the new loop is on the same side of the split plane, it can be assured that the two new loops do not overlap.
Each loop may be split in many different ways, however. The best possibility is selected,

based on the aspect ratio. The *aspect ratio* is the minimum distance of the loop vertices to the splitting plane, divided by the length of the splitting line. The possibility with the maximum aspect ratio is chosen. A minimum aspect ratio for triangulation to succeed can be set.

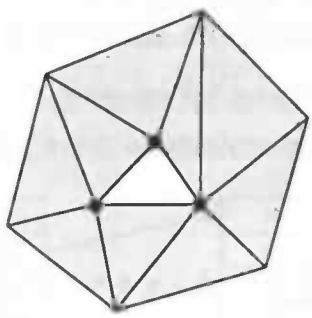


Figure 2.12: *Example of a tunnel in a mesh.*

Special cases such as tunnels (see figure 2.12) are handled accurately: if one of the vertices around the 'tunnel' would be deleted, the tunnel would collapse. A new edge would be created, duplicating an edge which is already part of one of the surrounding triangles. By checking for creation of duplicate triangles or edges, these cases can be identified preventing a vertex from being deleted. In this way the topology of the mesh is preserved.

2.4 Notes on the VTK decimate implementation

Complex vertex
If a vertex is used by an edge which is used by three or more triangles, or is used by a triangle that is not in the circle of triangles, then this vertex is considered complex according to the article [2] (see section 2.2.1). The VTK implementation expands this definition. A vertex is also considered complex if the degree (the number of triangles using the vertex) is higher than a specified number, i.e. 25.

Global error
In the `vtkDecimate.h` file, a global error bound decimation criterion is mentioned. At first, one might tend to believe this is some sort of global error, e.g. a bound on the total error of the approximation. However, this is not the case. Instead, it is the maximum error that can be introduced during each simplification step. Because the new mesh is compared to the previous simplification, nothing can be said about the error of the simplification compared to the original mesh.

3 Mesh Reduction with Error Control

3.1 Introduction

Although several algorithms have been developed to reduce the number of triangles in a triangular mesh, only few of these methods actually measure the difference between the approximation and the original mesh and if so, they are mostly only applicable to special cases, like parameterized surfaces.
The algorithm developed by Klein, Liebich and Straßer [7] is a decimation algorithm that uses a modified Hausdorff distance between the original mesh and the simplified mesh as an error value. This way, the algorithm guarantees a controlled approximation, where the distance between the original and the simplified mesh never exceeds the user-defined maximum Hausdorff distance.
Most methods, like the one by Schroeder, Zarge and Lorensen [2], lack this global error measure. (Schroeder *et al.* use a user-defined maximum error introduced in a *single* simplification step. Because of the possibility of accumulation of errors, this does not give a global bound on the error of the approximation.) By measuring the actual distance between the original triangle mesh and the approximation and not the distance to an average plane, the method by Klein *et al.* ensures a higher geometric accuracy than most algorithms.

3.2 Error metric

The Euclidean distance between a point x and a set $Y \subset \mathbb{R}^n$ is defined by

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

where $d(x, y)$ is the Euclidean distance between two points.
The distance from a set X to a set Y is defined by

$$d_E(X, Y) = \max_{x \in X} d(x, Y).$$

This is called the *one-sided Hausdorff distance* between sets X and Y.[1]

If, for example, the one-sided Hausdorff distance between the simplified mesh S and the original mesh T, denoted by $d_E(T, S)$, is less than a user-defined error tolerance ϵ, then

[1] In the paper by Klein *et al.* *inf* and *sup* are used instead of *min* and *max*. Because the problem deals with finite sets and *inf* and *sup* always belong to the set, *inf* and *min*, and *sup* and *max* can be interchanged.

$$\forall x \in T \text{ there is a } y \in S \text{ with } d(x,y) < \epsilon,$$

where x and y are points on the surface of the mesh.
However, this distance is not symmetrical. In some cases this can lead to problems, especially near borders or interior edges (see section 2.2.1).

To handle these cases the *two-sided Hausdorff distance* is used, defined by

$$d_H(X,Y) = \max(d_E(X,Y), d_E(Y,X)).$$

This distance is symmetric and $d_H(X,Y) = 0 \Longleftrightarrow X = Y$.

Now, if $d_H(T,S) < \epsilon$ then

$$\forall x \in T \text{ there is a } y \in S \text{ with } d(x,y) < \epsilon$$

and

$$\forall y \in S \text{ there is an } x \in T \text{ with } d(x,y) < \epsilon.$$

3.3 The algorithm

Like the algorithm by Schroeder, Zarge and Lorensen, this method starts with an original triangle mesh T and successively simplifies it by removing vertices and re-triangulating the resulting holes. It terminates when no more vertices can be removed from the simplified triangle mesh S without exceeding the predefined maximum Hausdorff distance.

The central issue of this method is the computation of the Hausdorff distance. To compute it, the distances between every pair of points from both triangle meshes need to be computed. This obviously requires a lot of operations. The computation can be simplified by keeping track of the Hausdorff distance between the original and the simplified mesh and of the correspondence between the two meshes from step to step. This correspondence ensures that distances are calculated to the correct part of the mesh. See figure 3.1
By using the correspondence, for every vertex x in the original triangle mesh which has already been removed, the triangles of the simplified triangle mesh $y \in S$ which are nearest to x can be found. That is, $d(x,y) \leq d(x,z), \forall z \in U \subset S$ and vice versa. Keep in mind that U only contains those parts of S that are topologically corresponding. The dashed line between a and b would be the corresponding part U in the example in figure 3.1.

Furthermore, for every single vertex of the simplified mesh a potential error is computed (and updated). It represents the Hausdorff distance that would occur if the vertex were to be removed.
For all vertices the potential error is computed and the vertices are then stored in a list, sorted by error in ascending order. If the vertex at the top of the sorted vertex list is removed from the triangle mesh, the list is updated.
This strategy of sorting automatically preserves sharp edges, because (interior) edge vertices have a relatively large potential error and are therefore at the bottom of the list.
When a vertex is complex (see section 2.2.1) or if re-triangulation of the hole would lead to topological problems, the vertex is not deleted.

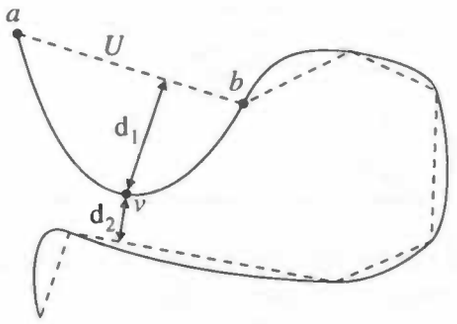


Figure 3.1: *The continuous line represents the original curve and the dashed line represents the simplified curve. Consider the distances d_1 and d_2 from a vertex v to parts of the simplified curve. Although the distance d_2 is less than distance d_1, distance d_1 is the distance needed because it is measured to the topologically corresponding part of the curve.*

First the list of vertices with according potential error is built, sorted by error. Next, vertices are iteratively removed from the list and from the triangle mesh (smallest error first) if their potential error is smaller than the predefined maximum Hausdorff distance ϵ. When this is no longer possible, the algorithm terminates.
If the vertex v is removed from the triangle mesh, its adjacent triangles are removed. Then the remaining hole is re-triangulated by projecting the remaining vertices into a plane and then using a constrained Delauney triangulation (CDT). The CDT triangulates an area confined by a closed polygon, using only the polygons vertices.
After the re-triangulation has been successfully completed, the potential errors of all neighboring vertices have to be updated. They are removed from the list and re-inserted according to their new error.

3.3.1 Calculation of the potential error

The crucial part of this algorithm is the computation of the potential error. The distance d_H between all the points on the two triangle meshes has to be computed. To speed up the computation, the one-sided Hausdorff distance is used.

$$d_E(T, S) = \max_{t \in T} d(t, S)$$

When none of the neighboring vertices have already been removed from the original mesh, the potential error is computed as follows.
Let v be the vertex of which the potential error has to be computed. Let $t_i, i = 1, \ldots, n$, be the triangles incident to v. These triangles would be removed if v was deleted. Let $s_j, j = 1, \ldots, m$, be the new triangles produced in the re-triangulation. When the vertex v is a border vertex then $m = n-1$, otherwise $m = n-2$. To compute $d_E(\{t_i\}_{i=1,\ldots,n}, \{s_j\}_{j=1,\ldots,m})$ it is sufficient to compute

$$\max_{j=1,\ldots,m} d(v, \{s_j\}),$$

see figure 3.2.

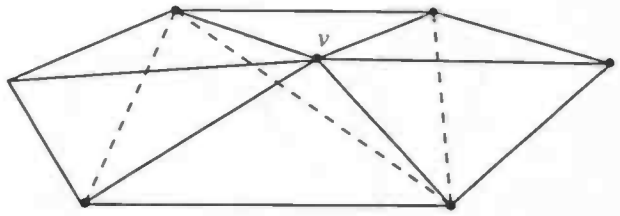


Figure 3.2: *Since none of the neighboring vertices of v have already been removed, it is clear that the distance from the original to the simplified triangle mesh is the distance from vertex v to the simplified triangle mesh (dashed lines).*

After a few simplification steps, however, there are triangles $t_k, k \in K$ in the original mesh with vertices that do no longer belong to the simplified mesh, so the problem gets more complex. For some of the triangles in the original mesh it may not be clear to which triangles of the simplified mesh distances have to be computed. To solve this problem some sort of correspondence has to be stored.
For each already removed vertex v of the original triangle mesh T, the triangle $s \in S$ that has the smallest distance to v is stored. Also, for each triangle $s \in S$ all vertices of T for which s is the vertex with the smallest distance are stored. This information is updated during each step.

Let $\{s_i^l, i = 1, \ldots, n\}$ be the set of removed triangles from S_l, where S_l is the triangle mesh which is created after deleting l vertices from the original triangle mesh T. Let $\{s_j^{l+1}, j = 1, \ldots, m\}$ be the set of new triangles produced during the step from triangle mesh S_l to S_{l+1}. Let $\{v_k \in V\}$ be the set of vertices of the original mesh that already have been removed. Each v_k must be nearest to one of the removed triangles $s_i^l \in S_l$.
For every triangle in the original triangle mesh that is connected to one of the vertices v_k the distance to S_{l+1} is calculated. To do this only the distances between triangles of the original triangle mesh and a subset $\tilde{S} \subset S_{l+1}$ need to be calculated. $\tilde{S}$ contains the newly created triangles of S_{l+1}, $\{s_j^{l+1}\}$, and the triangles that share at least one vertex with the newly created ones. This is possible, because

$$d_E(t, S_{l+1}) \leq d_E(t, \tilde{S}).$$

Because this operation is local, it speeds up calculation considerably and it also insures that the distance is measured to the correct part of the mesh.

3.3.2 Distance from triangle to triangle mesh

To compute the distance from a triangle t to the simplified triangle mesh S a straightforward method would be using the maximum of the distances from all three vertices of the triangle t to the simplified mesh. However, this method will not produce the correct distance in some cases (see figure 3.3).

To solve this, two cases are considered.

1. Triangle t of the original mesh has no vertex in common with the simplified triangle mesh S.

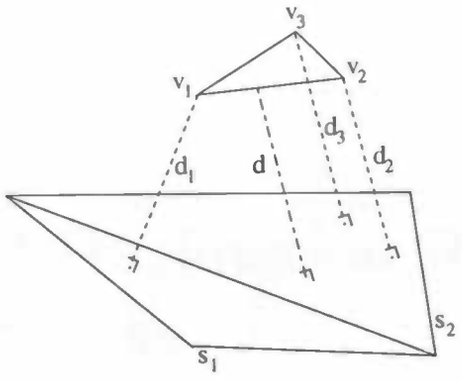


Figure 3.3: *In this example, the distance would be computed in an incorrect way. The computed distance would be the maximum of d_1, d_2 and d_3, but the actual distance d is larger.*

2. Triangle t of the original mesh has one or two vertices in common with the simplified triangle mesh.

In the first case three sub-cases can be distinguished (see figure 3.4):

1. All three vertices are nearest to the same triangle $s \in S$ (see figure 3.4-1).
2. The three vertices are nearest to two triangles $s_1, s_2 \in S$ that share an edge (see figure 3.4-2).
3. All other cases (see figure 3.4-3a).

In the first sub-case, calculation of the distance is simple:

$$d_E(t, S) = \max(d(v_1, S), d(v_2, S), d(v_3, S))$$

The second sub-case is more complicated. To solve this, a half-angle plane is created between the two triangles s_1 and s_2 sharing a common edge. This half-angle plane is intersected with those edges of triangle t having endpoints that belong to different triangles. Taking the maximum of distances of the vertices of t and the distances of the intersection points to triangles s_1 and s_2 gives the error. See figure 3.5.

In the third sub-case (all other cases) the triangle t is subdivided until either case 1 or case 2 applies (see figure 3.4-3b). When the longest edge of a sub-triangle is smaller than a predefined error tolerance ϵ the subdivision also terminates and

$$d(t, S) = \max(d(v_1^{sub}, S), d(v_2^{sub}, S), d(v_3^{sub}, S)),$$

where v_i^{sub} is the sub-triangle that contains vertex i.
In case 2 an upper bound of the maximum distance is also computed using the half-angle plane. Using subdivision, the problem is reduced to either case 1 or case 2.

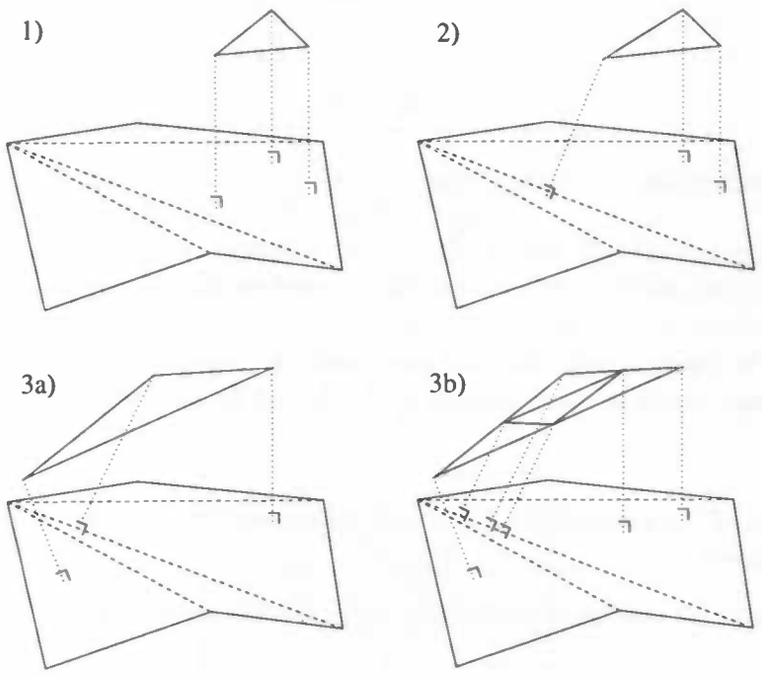


Figure 3.4: *Different sub-cases distinguished when computing the distance from triangle to mesh.*

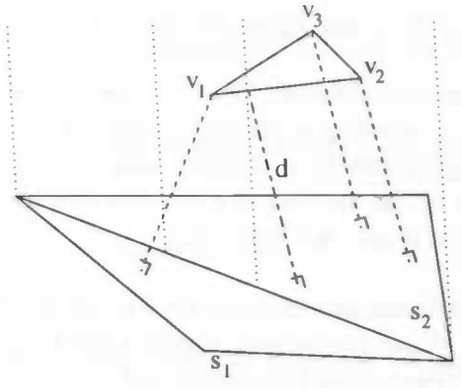


Figure 3.5: *An example of the use of a half-angle plane. The dotted lines lie in the half-angle plane. The distance from the intersection of the triangle t and the half-angle plane to the triangles s_i is larger than the distance from the vertices v_i to the triangles s_i, so the former distance has to be used.*

4 Additional Decimation Routine for VTK: KLSDecimate

4.1 Introduction

Based on the article by Klein *et al.* [6] an additional decimation routine for VTK was constructed. Algorithm details are described in the previous chapter. This chapter discusses some implementation details, like data structures. It shows some results of the routine and compares the results of `KLSDecimate` to the results of `vtkDecimate`. Finally some suggestions for improvements are given.

4.2 Data structure

To represent the three-dimensional object to be decimated a `vtkPolyData` structure is used. The reason this structure was chosen is that it was already implemented in VTK and all the functions for the required manipulations on the dataset were available. For details on this structure, see section 2.3.1. The sorted vertex list and the correspondence list require a separate structure. The following sections will discuss their implementation.

4.2.1 Sorted vertex list

The sorted vertex list is a doubly linked list of elements containing a vertex ID (integer) and a potential error value (float). A separate class – `SortedVertexList` – was implemented for this structure.

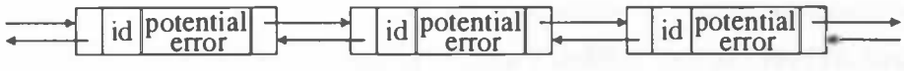


Figure 4.1: *Structure of the sorted vertex list.*

Some functions are implemented to modify or access the list. These are:

`Insert(id, pot_error)` inserts an element with ID *id* and potential error *pot_error* into the sorted vertex list.

`Delete(id)` deletes the element with ID *id* from the sorted vertex list.

`GetPotentialError(id)` returns the potential error for the element with ID *id*.

`GetFirstPotentialError(id)` returns the potential error for the first element of the sorted vertex list. Also returns the ID of this element in *id*.

`Print()` prints the sorted vertex list on the screen.

4.2.2 Correspondence list

The correspondences (see section 3.3.1) are stored in a structure which consists of a vertex list and triangle list. Both lists are doubly linked lists.
Each element of the vertex list contains, besides the vertex ID, a pointer to an element of the triangle list. Each element of the triangle list contains, apart from the IDs of the vertices that define the triangle, a list of IDs of vertices. These IDs refer to vertices in the vertex correspondence list.

The structure will look as indicated in figure 4.2.

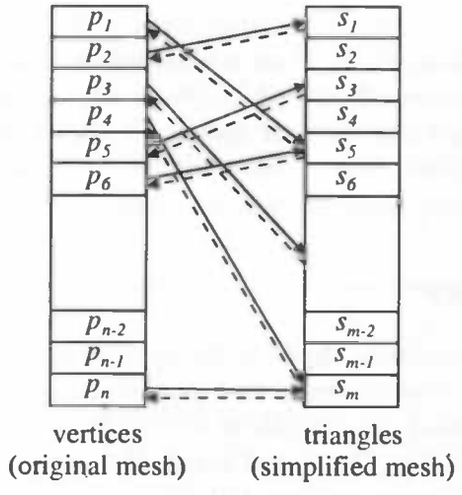


Figure 4.2: *Representation of the correspondence between the original and simplified mesh.*

Vertices are stored according to ID, lowest first; triangles are sorted according to the lowest ID of their vertices.

Available functions on the correspondence list are:

`GetVertexCorr(vertex_id)` returns a pointer to the vertex element with ID *vertex_id*.

`GetTriangleCorr(tid1, tid2, tid3)` returns a pointer to the triangle element with IDs (*tid1*, *tid2*, *tid3*).

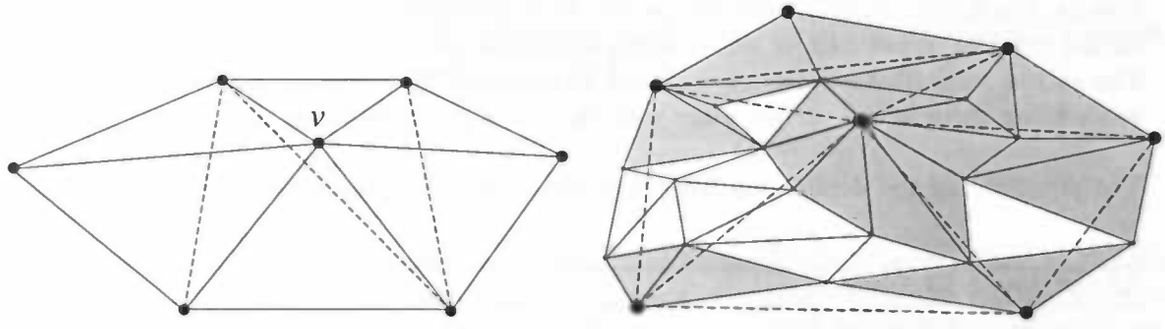
`InsertVertexTriangleCorr(vid, tid1, tid2, tid3)` inserts a new correspondence between the vertex with ID *vid* and the triangle (*tid1*, *tid2*, *tid3*).

`DeleteTriangleCorr(tid1, tid2, tid3)` deletes the element of the triangle list with IDs (*tid1*, *tid2*, *tid3*). Returns the list of vertex IDs it contained.

`Print()` prints the correspondence list.

4.3 Algorithm structure

The structure of the algorithm is typical for a mesh reduction algorithm. First of all a copy of the original mesh is made. Vertices and triangles will be deleted from this copy. During the decimation process the original mesh is referenced for correspondence purposes. Therefore it has to be retained.
After copying the mesh the sorted vertex list is initialized. For every vertex in the mesh the potential error is calculated. A new entry is created in the sorted vertex list with the vertex' ID and potential error. The new entry is inserted according to the potential error, lowest first. For this initialization step a simplified version of the potential error calculation routine is used. This is possible because in the initialization step there is no need to check correspondences of deleted vertices or distances from surrounding triangles to the new triangle mesh. The error induced when this vertex would be deleted is the distance from the vertex to the set of newly created triangles. See figure 4.3(a).



(a) When none of the neighboring vertices have been removed – like in the initial situation – it is clear which triangles of the old and new triangle mesh have to be considered to calculate the potential error.

(b) For the white triangles of the original triangle mesh (solid lines) it is not clear to which triangles of the simplified triangulation (dashed lines) distances have to be computed, because they do not share a vertex with any triangle of the simplified triangle mesh.

Figure 4.3: *In the initial situation, calculating the potential error is simple because no correspondences need to be considered (figure a). Later, correspondences need to be included in the calculation (figure b).*

After initialization is completed, the reduction loop will be entered. The loop will terminate when the first vertex of the sorted vertex list has a potential error which is greater than the predefined maximum Hausdorff distance, or when the target reduction rate is achieved (the percentage of triangles that have to be deleted can be specified). The user can determine which of these two criteria will be used.

When entering the reduction loop the first ID is retrieved from the sorted vertex list. This vertex is now candidate for deletion. A list of vertices and a list of triangles that surround this vertex are created. The list of surrounding vertices represents the hole which would be created if the vertex were to be deleted. This hole is then triangulated using a recursive loop splitting algorithm. This triangulation routine returns a list of new triangles.

After that, the vertex is actually deleted from the mesh and the mesh is updated to represent the new situation. This means resizing the triangle list of each vertex in case the number of surrounding triangles increases. To be able to compute the change in size of the triangle lists, the number of original and new references are stored. Next, for every vertex in the loop the references to the deleted triangles are removed. Because the number of new triangles is always smaller than the number of original triangles some triangles will have to be deleted. The rest can be replaced by the new triangles.

If removal was successful, i.e. re-triangulation of the hole was possible, so the vertex could safely be deleted, the correspondence list has to be updated. The entry in the correspondence list of the vertex that was deleted from the mesh and all the entries of the vertices in the vertex correspondence list that pointed to one of the deleted triangles have to be updated. This means that for all those vertices a new closest triangle has to be calculated. This triangle is one of the newly created triangles. The deleted triangles will also be removed from the triangle correspondence list.
Finally, the entry of the sorted vertex list with the current vertex ID is removed and for all the vertices in the loop of surrounding vertices a new potential error will be calculated. The sorted vertex list entries are updated accordingly, by removing them from the list and re-inserting them in the correct place with their new potential error. See section 3.3.

The structure of the decimation routine is presented using pseudo code. See algorithm 1.

Algorithm 1 KLSDecimate

```
Initialize sorted vertex list
while OK to continue do
    // Continue until termination criterion is met
    Retrieve first vertex from sorted vertex list
    Create loop L of surrounding vertices
    Triangulate loop L
    Remove vertex and its triangles from mesh
    Insert new triangles into mesh
    Update correspondence list
    Update sorted vertex list
end while
```

4.4 Potential error

When calculating the potential error, the distance between the current mesh and the topologically corresponding part of the original mesh has to be calculated. See section 3.3.1.
To be able to calculate the potential error of vertex v, several things are needed:

`corrOrigVerts`: a list of vertices in the original mesh which have already been deleted and refer to one of the surrounding triangles of vertex v as being the closest one. This list is built using the correspondence list. The vertex v is also included in the list.

`newTris`: a list of triangles which would be created if vertex v were to be deleted and the triangles that have at least one vertex in common with one of these newly created triangles.

`origTris`: a list of triangles that surround the vertices in the `corrOrigVerts` list.

The distances between every pair of triangles in `origTris` and `newTris` have to be calculated. The potential error of vertex v is the maximum of these distances.

4.5 Constraints

Some constraints apply to the input the routine is capable of processing. The routine was designed to decimate surfaces. Arbitrary 3D triangular meshes (non-manifold) can not be decimated by the `KLSDecimate` routine.
However, the program can be easily adapted to handle these inputs by expanding the vertex classification with the complex vertex type (see section 2.2.1).

4.6 Results

The Visualization Toolkit comes with two terrain models: `honolulu.vtk` and `fran.g`[1]. These two files were used to test the `KLSDecimate` routine and compare it to the method created by Schroeder *et al.* Since the efficiency of `KLSDecimate` routine still needs some work (see section 4.7.1), the main focus of the comparison will be on the quality of the reduced mesh.

The original `honolulu.vtk` is a 16 Megabyte file, so for test purposes the file was first decimated to 1 Megabyte using `vtkDecimate`. This new file was used as an input and consisted of 13171 vertices and 26276 triangles. It was reduced to 20, 10 and 5 percent of the original size and results are shown in figures 4.5 through 4.7. All figures show the same models, but represented in different ways, using flat shading, Gouraud shading and wireframe representation, respectively.

The figures show that `KLSDecimate` is capable of creating a good approximation of the original mesh. The large features are present in all approximations. The 5 percent model shows a clear loss of detail, so it is up to the user to specify a reduction rate that meets the desired amount of detail.

The other dataset, `fran.g` is a triangle mesh consisting of 26460 vertices and 52260 triangles. This mesh was also decimated to 20, 10 and 5 percent of the original number of triangles. See figures 4.8 through 4.10.

These figures also show that `KLSDecimate` created an accurate approximation of the original.

[1] Files with the `.g` extension can be read with the `vtkBYUReader` instead of the regular `vtkPolyReader`.

4.6.1 Comparison with vtkDecimate

The VTK toolkit comes with the decimation routine designed by Schroeder *et al.*, called vtkDecimate. Both the vtkDecimate and the KLSDecimate routine were given the same datasets as input and output results were compared.
KLSDecimate, when running in '*target reduction*' mode, does not require the user to set any parameters, apart from the desired reduction rate.[2]. In this case, the routine uses default values for parameters that were not set. The first comparison was done by just setting the target reduction rate. It uses the fohe.g dataset as an input. This dataset comes with the VTK 2.0 distribution.
Figure 4.11 shows that KLSDecimate creates a better approximation, especially in the bended parts of the triangle mesh.

The second comparison uses the fran.g dataset. Again, vtkDecimate was run with default parameters.
Figure 4.12 shows that vtkDecimate produces a result with some peculiar spikes on the face. Even though the resulting dataset produced by KLSDecimate has less triangles, the result is much better.

By fine-tuning some parameters, like InitialFeatureAngle, AspectRatio, InitialError and ErrorIncrement, vtkDecimate will produce a much better result, but it is not clear how a user can determine the optimal values for the parameters apart from trial and error. Parameters for this particular dataset were found in a tcl-script that came with the VTK distribution. Results are shown in figure 4.13.

Figure 4.13 shows a much better approximation than figure 4.12. Still, the approximation produced by KLSDecimate appears less rugged than the one produced by vtkDecimate, especially around the nose and lips.

Based on these two datasets, it can be said that KLSDecimate promises better results than vtkDecimate, at least when it is run with default parameters. Results can be improved, but that requires fine-tuning some parameters.

4.7 Improvements

4.7.1 Efficiency

KLSDecimate is currently too slow for practical use. It needs just over three hours to create the 95 percent decimated face in figure 4.13. Surely, that is too long, when vtkDecimate needs only several tens of seconds for the same job.

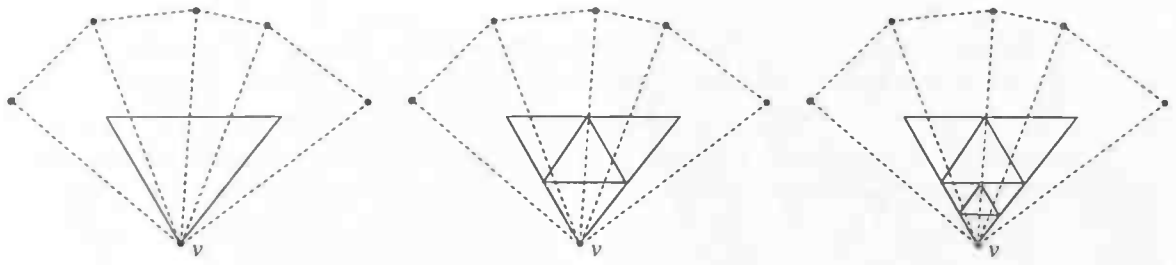
When looking at the structure of the algorithm, it may become clear why the KLS algorithm is slower than the vtkDecimate routine.
First of all, the sorted vertex list has to be initialized. This means that for every vertex in the triangle mesh a potential error has to be calculated.

[2]Currently the user still needs to set a value for the maximum error, but this does not effect the quality of the image. See section 4.7.

During the simplification process an entry is created in the correspondence list for every vertex that is deleted, and for every neighboring vertex the entry in the sorted vertex list has to be updated. This means re-calculating potential errors. When re-calculating these potential errors, the correspondences have to be considered, causing more triangles to be used in the calculation.
The correspondence list will grow in size when more and more vertices are deleted from the triangle mesh. This means that re-calculating the potential error will take a growing amount of time.

Using the sorted vertex list and correspondence list requires a lot of lookup action, which also slows the routine down. Currently, these lists are implemented as doubly linked lists. Search times may be considerably reduced if these were implemented as search trees.
Clearly, maintaining these lists for large datasets requires a large amount of memory. The number of entries in the vertex correspondence list is at most the number of vertices in the original mesh. There is a similar relationship for the triangle correspondence list. A triangle mesh with n vertices has at most $n-2$ triangles, resulting in a triangle correspondence list with at most $n-2$ entries. All entries in the sorted vertex list have the same size. Entries in the triangle correspondence list can vary in size, because a vertex list is included, but the total number of elements of all entries will never exceed n. In short, the extra memory needed to maintain the lists is of $O(n)$, where n is the number of vertices in the triangle mesh.

It appears that the efficiency of the algorithm is also greatly influenced by the amount of triangle splitting in case 3 of the potential error calculation. The following example will show why.



(a) This is neither case 1 nor case 2, so the triangle is subdivided

(b) The bottom triangle causes exactly the same problem as presented in figure (a).

(c) Subdivision will not terminate.

Figure 4.4: *Example of subdivision of triangles. The dotted lines represent triangles from the new triangulation. The continuous lines represent (sub) triangles in the original triangulation.*

As described in section 3.3.2, when calculating the distance from a vertex to the triangulation, three cases can be distinguished. When the situation is classified as case 3, the triangle in the original mesh is subdivided. Sometimes this situation can lead to problems: as shown in figure 4.4(b) the triangle is split, but the bottom triangle represents the same problem as in the initial situation. No matter how often the bottom triangle is subdivided, the new bottom

triangle will never correspond to two neighboring (or even one) triangle. The EDGE_EPS variable controls how long this splitting will continue by limiting the length of the edges of the triangles. By choosing a relatively large value for EDGE_EPS the quality of the approximation will decrease, but the efficiency will improve. Klein *et al.* suggest

$$\texttt{EDGE\_EPS} = (1 + \frac{1}{\sqrt{3}})^{-1}\epsilon$$

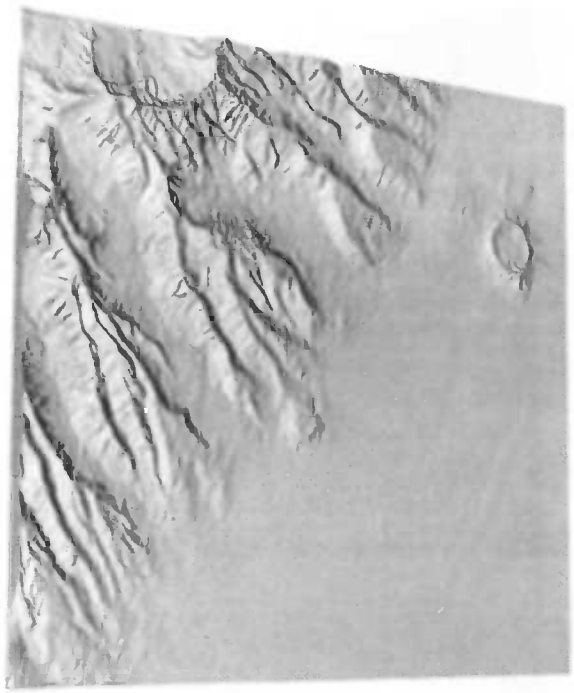
to ensure error ϵ. Here, ϵ is the maximum potential error allowed.

Efficiency might also be improved by storing the subdivided triangles until they are no longer needed because the adjacency of the triangles in the simplified mesh has changed.

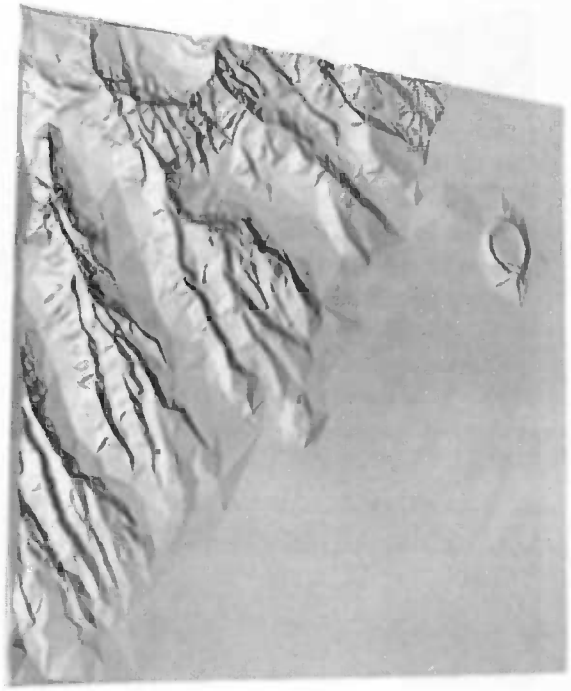
4.7.2 Other improvements

Some other points for improvement are:

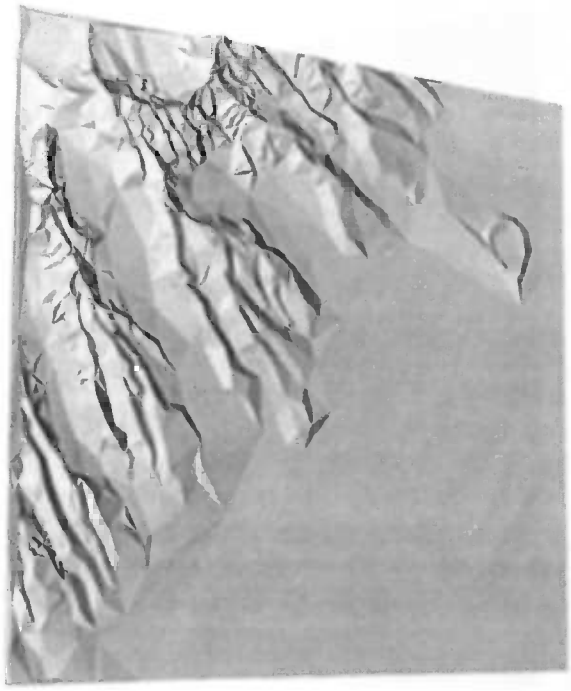
- The routine has to be adapted to work with the new version of VTK, version 2.0. It was designed for use with version 1.3.
- The algorithm has to be adapted to be able to cope with complex vertices, which will make decimation of arbitrary 3D objects possible.
- Currently the routine is implemented as a function, but it needs to be converted into a class.
- Instead of the recursive loop splitting, Klein *et al.* suggest a Delauney triangulation method.
- When using the '*target reduction*' mode the user currently needs to set the potential error, which is used to calculate EDGE_EPS. This should be done automatically, by taking some average value for the potential error or by taking the value of an entry in the sorted vertex list. For example, a target reduction rate of 80 percent might use the entry at 80 percent of the sorted vertex list.



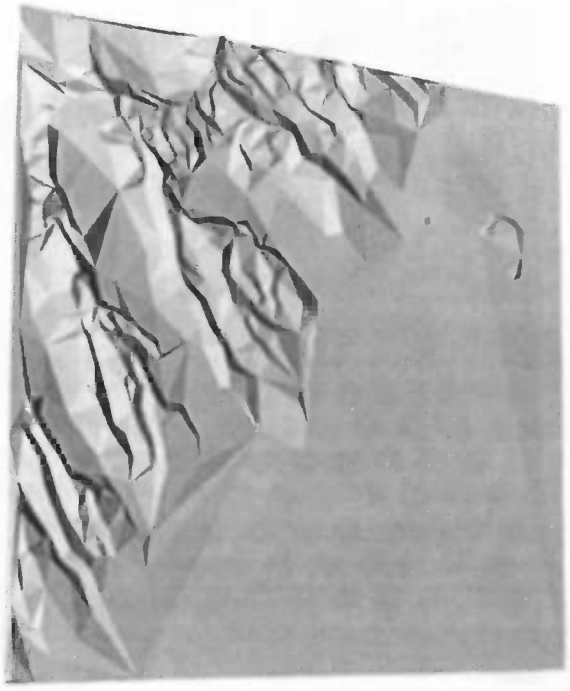
(a) Original dataset with 26276 triangles.



(b) Dataset reduced to 20 percent using 5256 triangles.

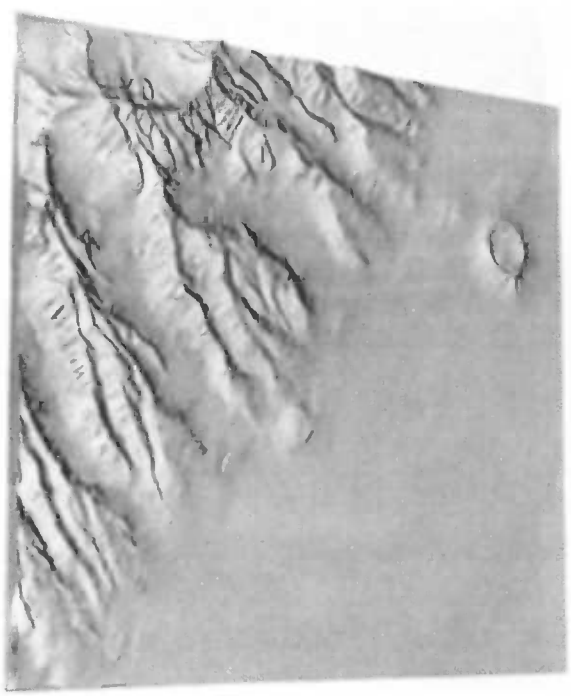


(c) Dataset reduced to 10 percent using 2628 triangles.



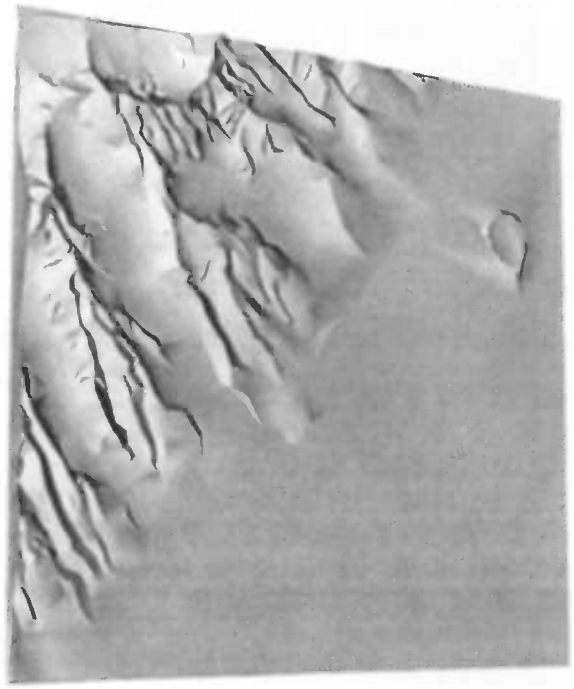
(d) Dataset reduced to 5 percent using 1314 triangles.

Figure 4.5: *Results of the decimation of the* **honolulu** *dataset with various reduction rates, represented using flat shading.*



(a) Original dataset with 26276 triangles.

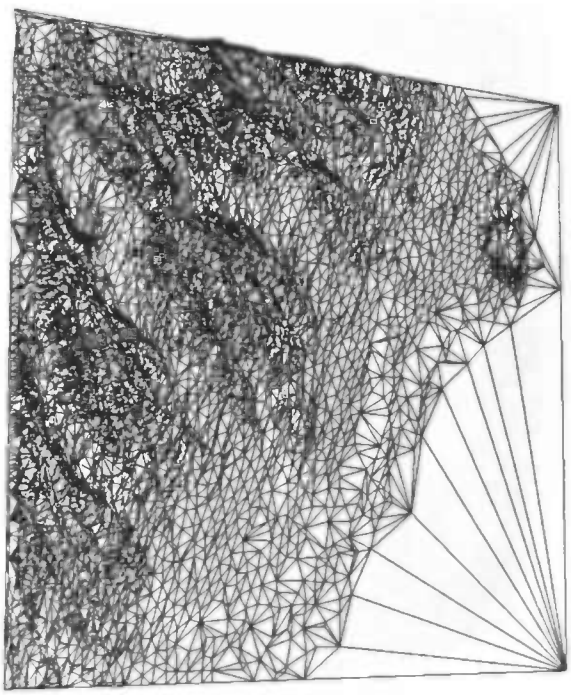
(b) Dataset reduced to 20 percent using 5256 triangles.



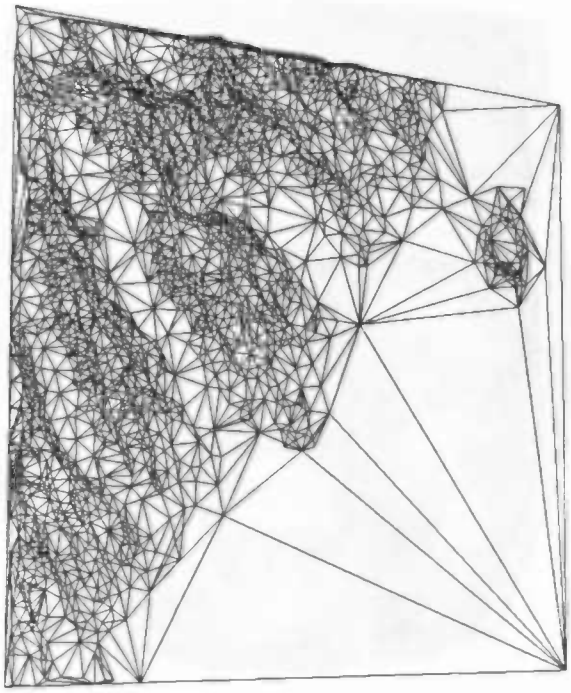
(c) Dataset reduced to 10 percent using 2628 triangles.

(d) Dataset reduced to 5 percent using 1314 triangles.

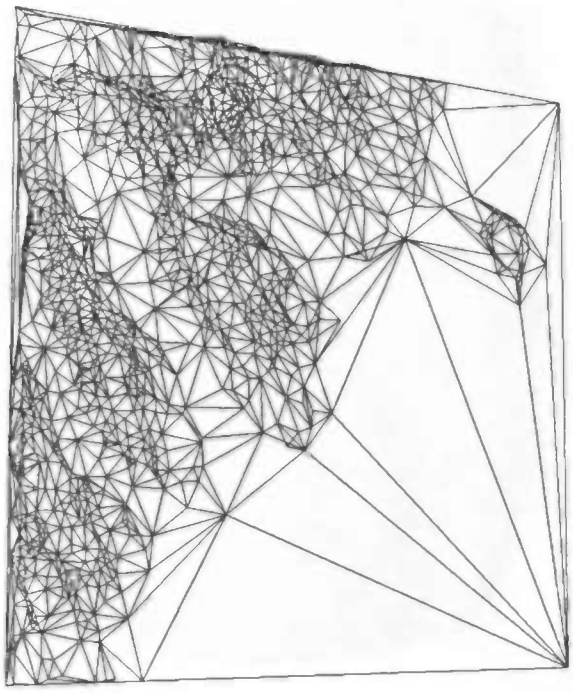
Figure 4.6: *Results of the decimation of the* **honolulu** *dataset with various reduction rates, represented using Gouraud shading.*



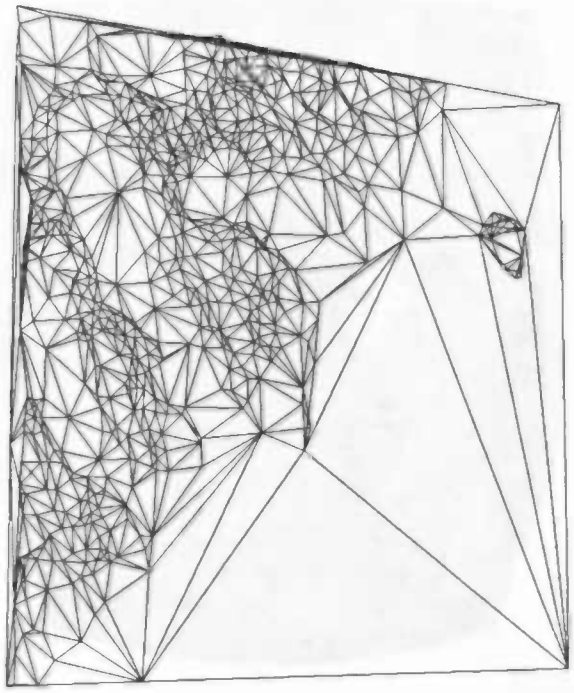
(a) Original dataset with 26276 triangles.



(b) Dataset reduced to 20 percent using 5256 triangles.

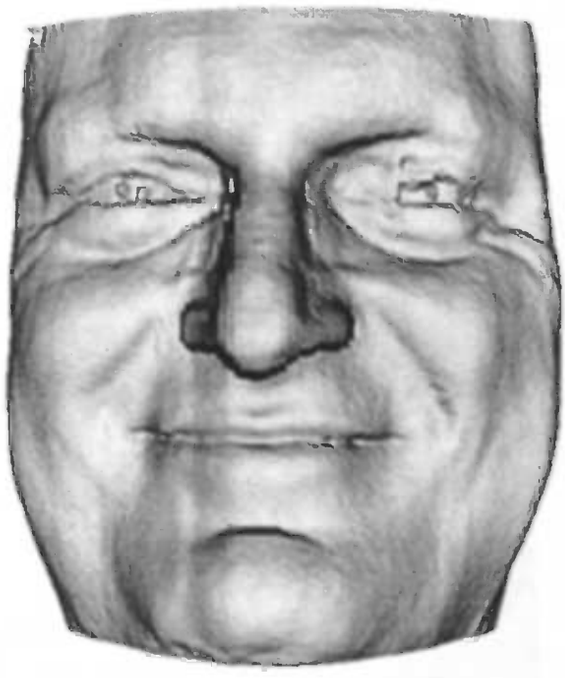


(c) Dataset reduced to 10 percent using 2628 triangles.

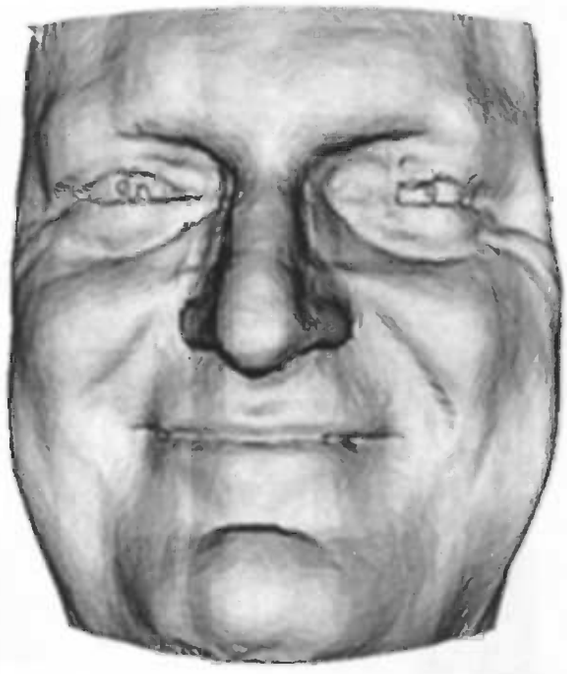


(d) Dataset reduced to 5 percent using 1314 triangles.

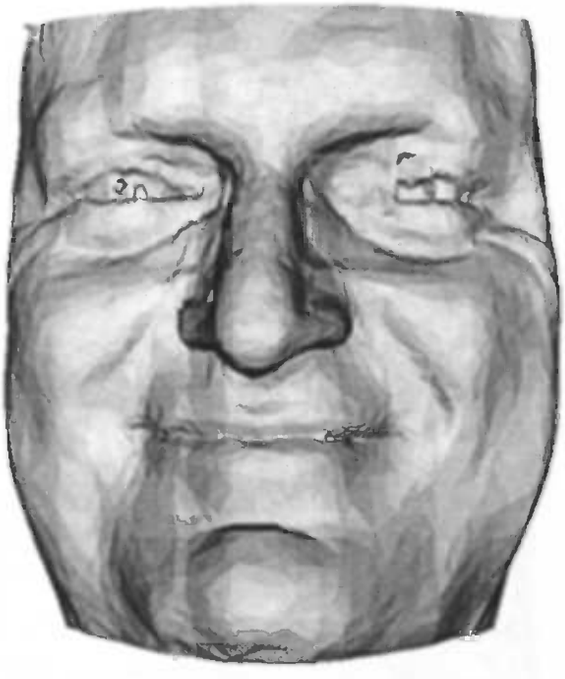
Figure 4.7: *Results of the decimation of the* **honolulu** *dataset with various reduction rates, represented using wireframe.*



(a) Original dataset with 52260 triangles.



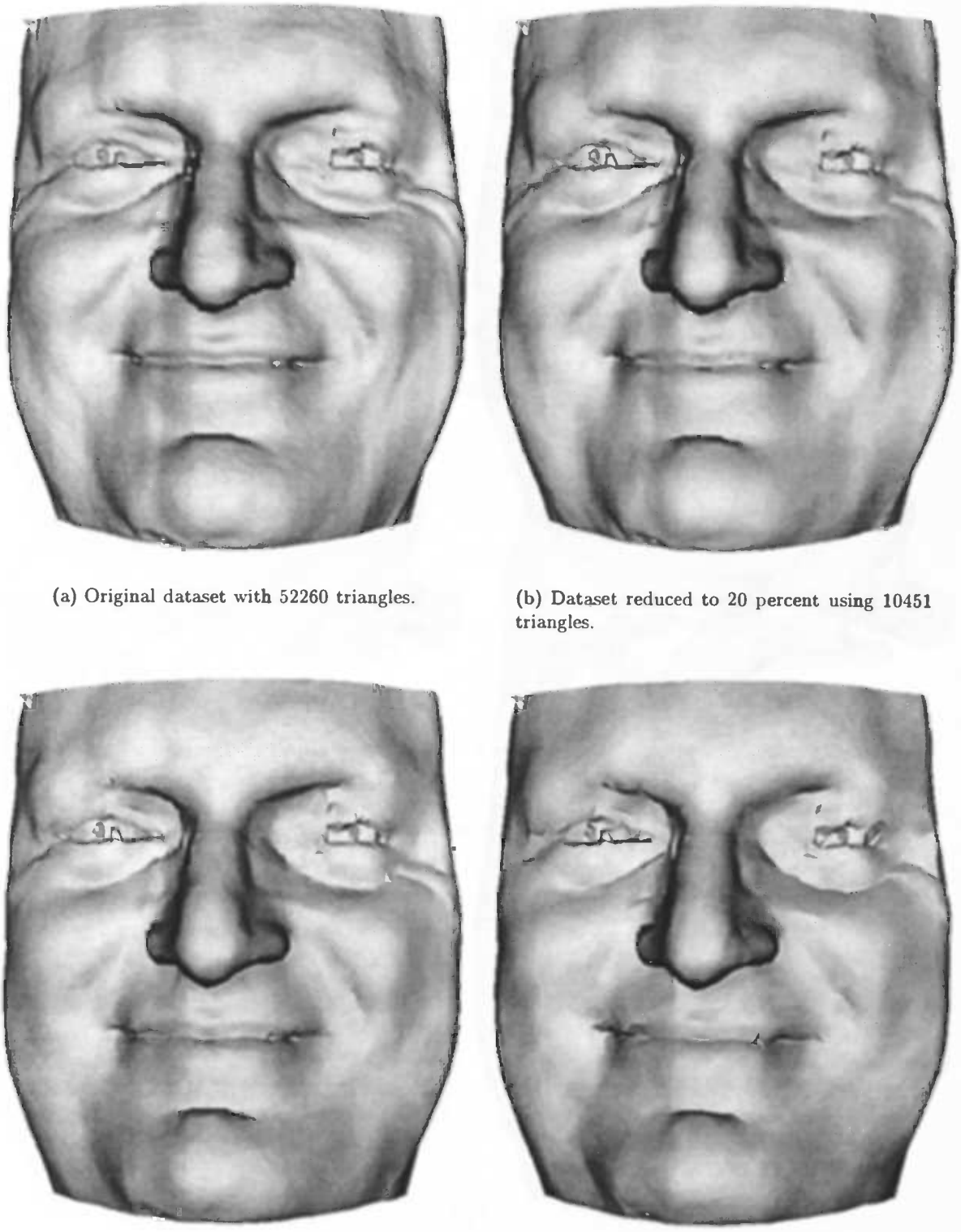
(b) Dataset reduced to 20 percent using 10451 triangles.



(c) Dataset reduced to 10 percent using 5225 triangles.

(d) Dataset reduced to 5 percent using 2612 triangles.

Figure 4.8: *Results of the decimation of the* **fran** *dataset with various reduction rates, represented using flat shading.*



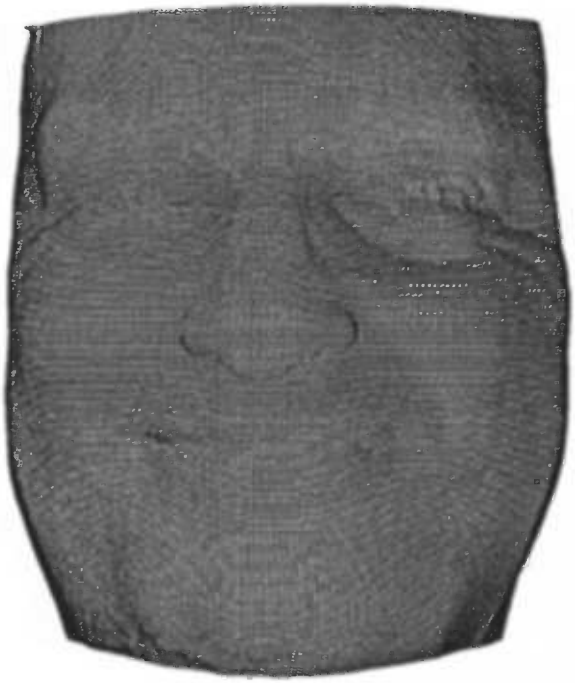
(a) Original dataset with 52260 triangles.

(b) Dataset reduced to 20 percent using 10451 triangles.

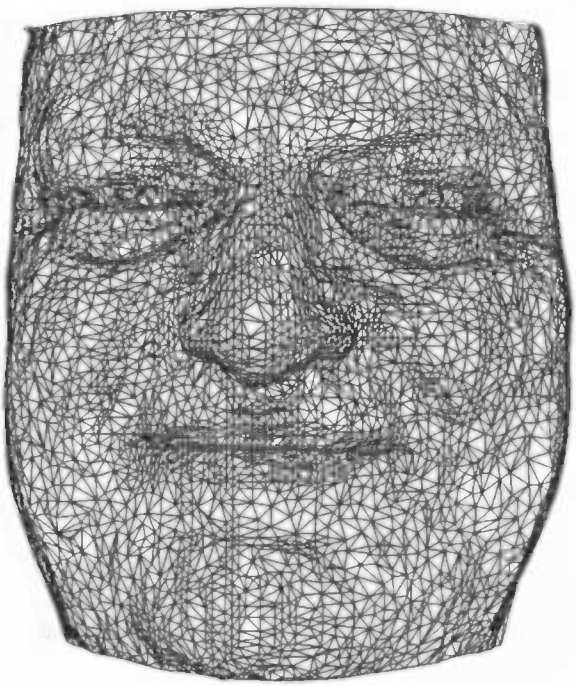
(c) Dataset reduced to 10 percent using 5225 triangles.

(d) Dataset reduced to 5 percent using 2612 triangles.

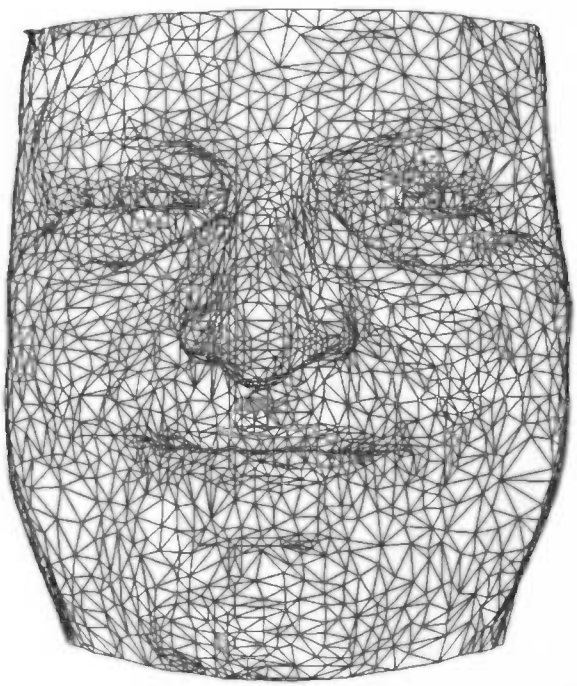
Figure 4.9: *Results of the decimation of the* `fran` *dataset with various reduction rates, represented using Gouraud shading.*



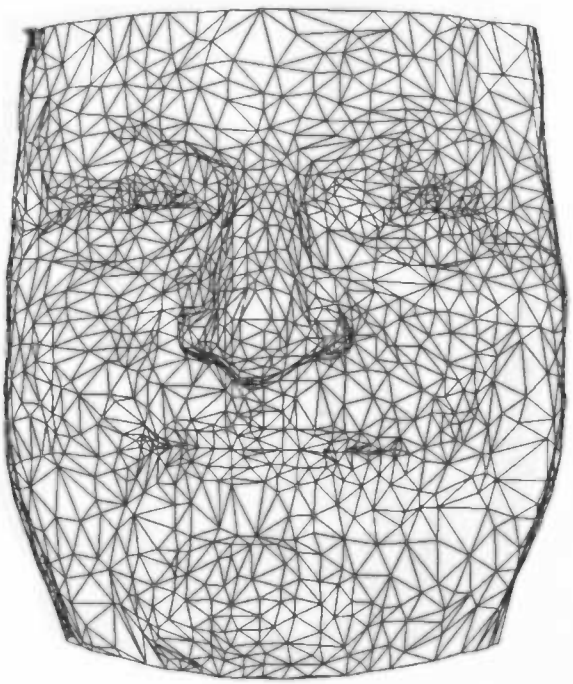
(a) Original dataset with 52260 triangles.



(b) Dataset reduced to 20 percent using 10451 triangles.

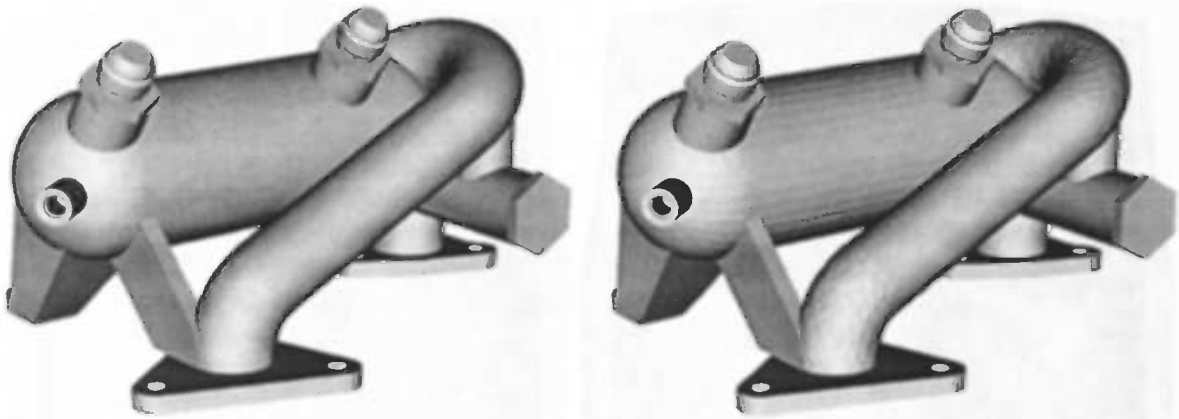


(c) Dataset reduced to 10 percent using 5225 triangles.

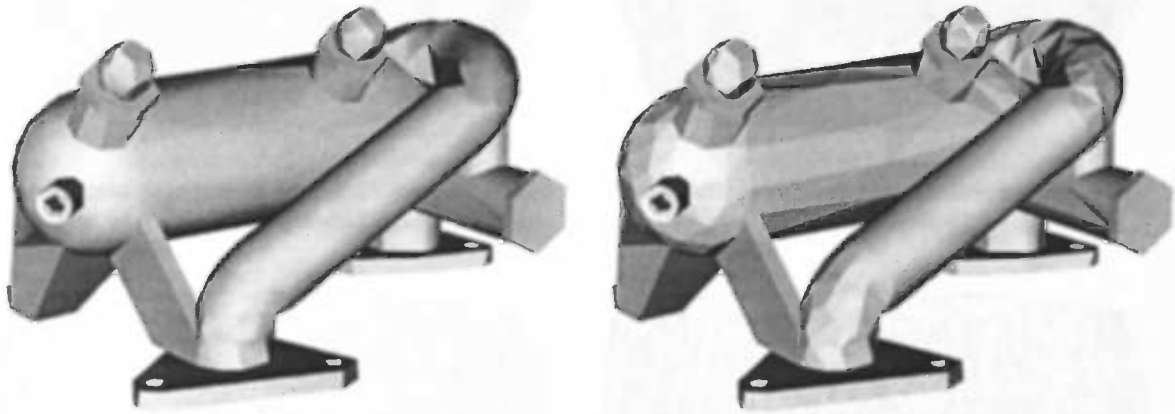


(d) Dataset reduced to 5 percent using 2612 triangles.

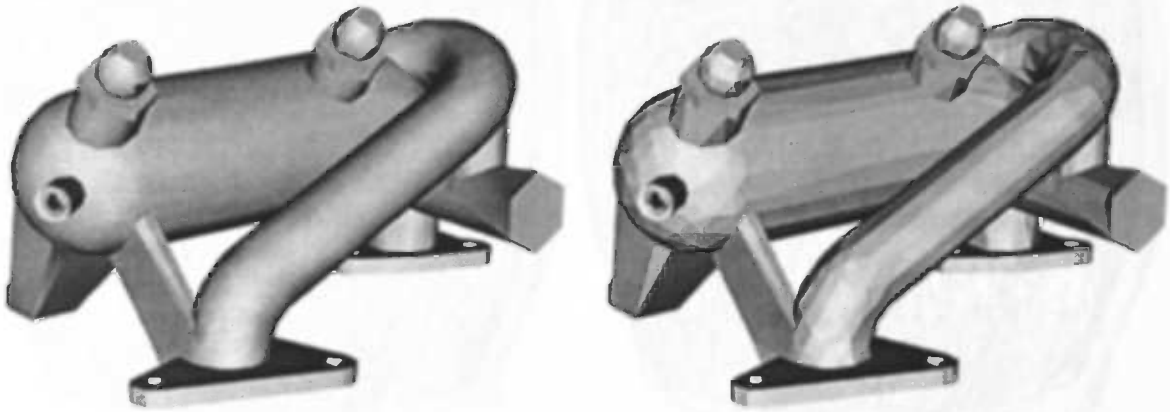
Figure 4.10: *Results of the decimation of the* `fran` *dataset with various reduction rates, represented using wireframe.*



(a) Original dataset consisting of 8038 triangles.

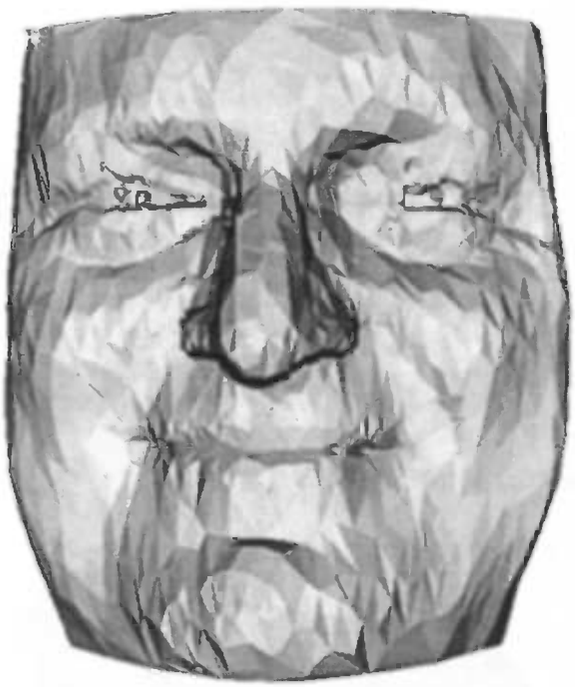


(b) Dataset decimated from 8038 to 2032 triangles using VTK's decimate.



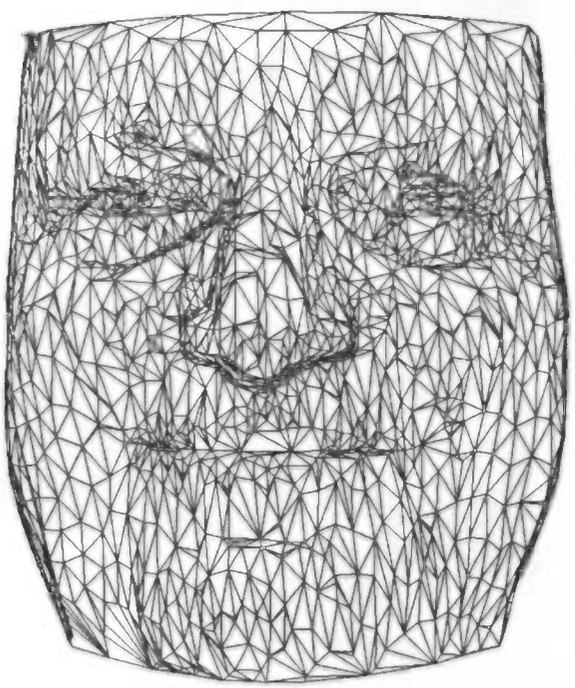
(c) Dataset decimated from 8038 to 2010 triangles using `KLSDecimate`.

Figure 4.11: *Comparison between* `vtkDecimate` *and* `KLSDecimate`, *using the* `fohe.g` *dataset as an input. The figures in the left column are represented using Gouraud shading, the figures in the right column using flat shading.*

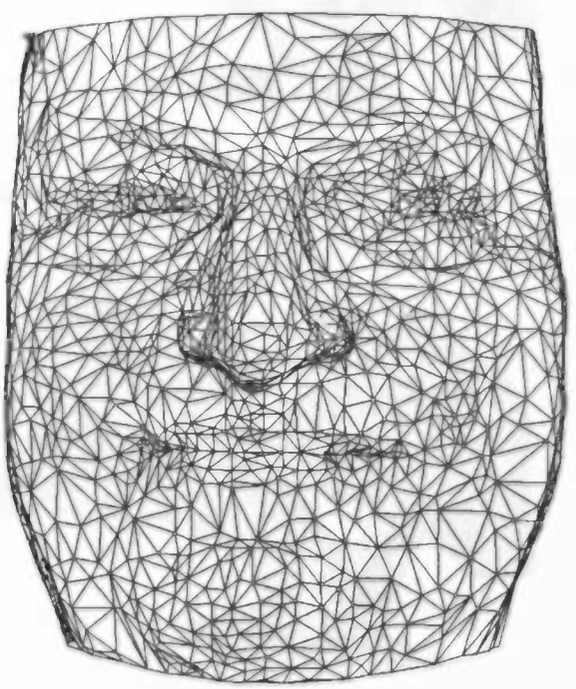


(a) Dataset decimated from 52260 to 3294 triangles using VTK's decimate.

(b) Dataset decimated from 52260 to 2612 triangles using KLSDecimate.

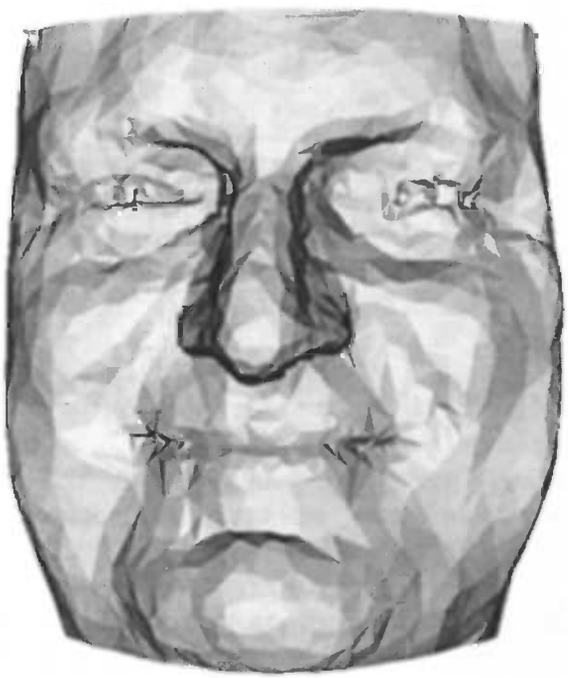


(c) Dataset decimated from 52260 to 3294 triangles using VTK's decimate.

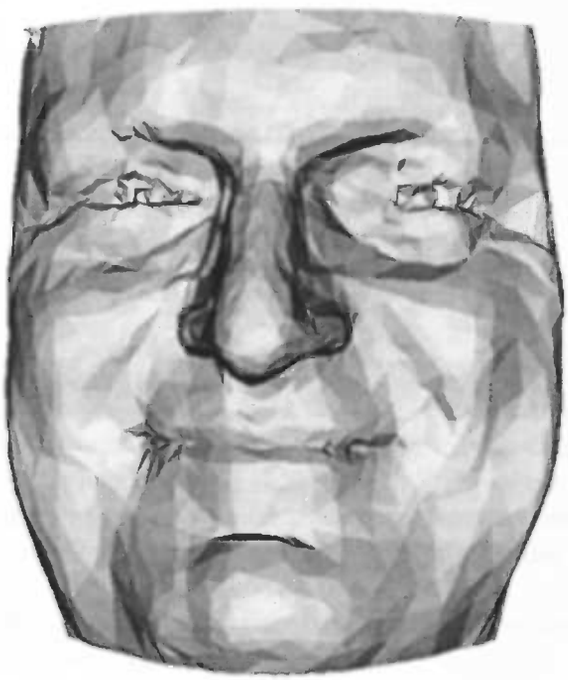


(d) Dataset decimated from 52260 to 2612 triangles using KLSDecimate.

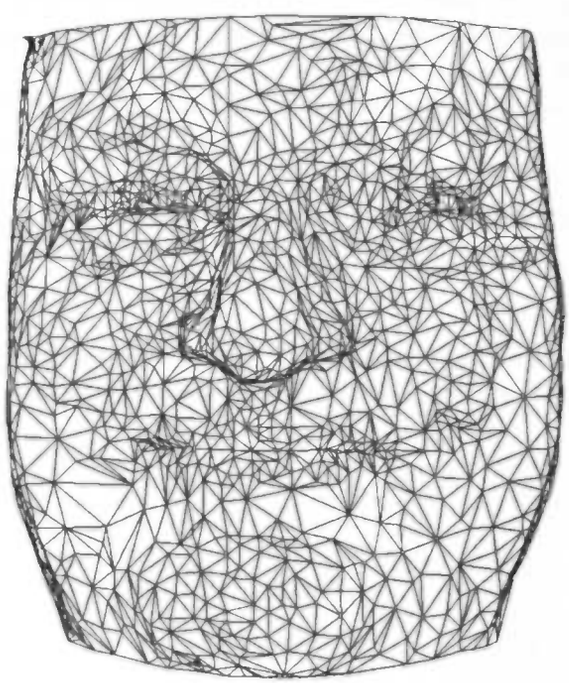
Figure 4.12: *Comparison between* vtkDecimate *and* KLSDecimate. *The figures in the top row are represented using flat shading, the figures in the bottom row using wireframe representation.*



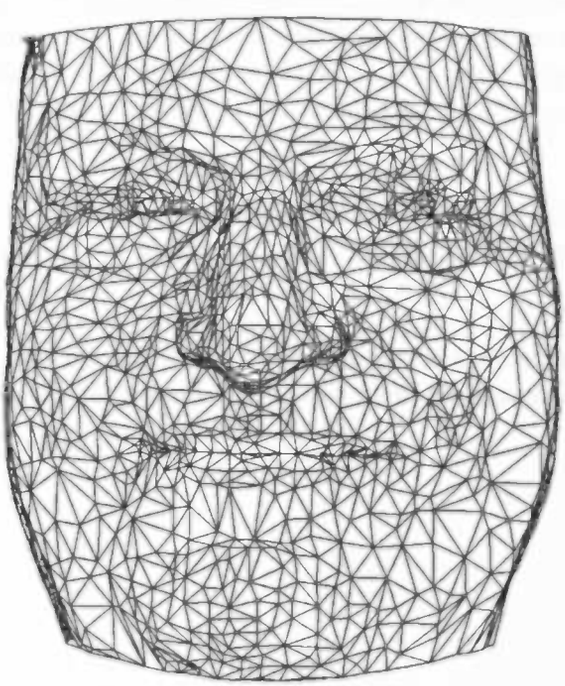
(a) Dataset decimated from 52260 to 2746 triangles using VTK's decimate with optimal parameters.



(b) Dataset decimated from 52260 to 2612 triangles using `KLSDecimate`.



(c) Dataset decimated from 52260 to 2746 triangles using VTK's decimate with optimal parameters.



(d) Dataset decimated from 52260 to 2612 triangles using `KLSDecimate`.

Figure 4.13: *Comparison between* `vtkDecimate` *and* `KLSDecimate` *using optimal parameter settings for* `vtkDecimate`. *The figures in the top row are represented using flat shading, the figures in the bottom row using wireframe representation.*

5 Conclusion

The goal of this project was to implement a decimation routine for VTK and compare the results to the results of alternative methods. At first, the algorithm by Schroeder *et al.* was considered to be implemented, since a global error is mentioned in their paper [2]. However, this global error applied to each simplification step and not to the entire approximation, as explained in section 2.4. Besides, the implementation was already available in the VTK distribution as `vtkDecimate`.
The algorithm by Klein *et al.* promises an actual global error and was therefore chosen to be implemented and the results were compared to the results of `vtkDecimate`.

Although the algorithm implemented is still limited to surfaces, it already shows some good results. Compared to the implementation of `vtkDecimate`, the new routine, `KLSDecimate`, can produce better results with even less triangles. The new routine is easier to use than `vtkDecimate`. `vtkDecimate` will work by simply providing a target reduction rate, but needs some additional parameters to achieve an optimal result. Setting these parameters requires a lot of insight into the data and it will most likely take some trial and error to get everything right. For `KLSDecimate` the user only needs to specify the desired reduction rate to achieve an optimal approximation.

For practical purposes `KLSDecimate` is still too slow. It should be able to decimate a dataset consisting of 50.000 triangles to about 5 percent within a few minutes, but this currently takes about three hours.

When the efficiency of the new `KLSDecimate` routine has been improved and when it is capable of handling arbitrary 3D objects, it will be a very good alternative to `vtkDecimate`.

References

[1] P.S. Heckbert and M. Garland, "Survey of Polygonal Surface Simplification Algorithms", `http://www.cs.cmu.edu/~ph`, May 1997.

[2] W.J. Schroeder, J.A. Zarge and W.E. Lorensen, "Decimation of Triangle Meshes", *Computer Graphics*, Vol. 26, pp. 65-70, July 1992.

[3] J. Rossignac, "Geometric Simplification and Compression", *Siggraph'97*, 1997.

[4] K.J. Renze and J.H. Oliver, "Generalized Unstructured Decimation", *IEEE Computer Graphics and Applications*, Vol.16 number 6, pp. 24-32, November 1996.

[5] W.J. Schroeder, K. Martin and W.E. Lorensen, *The Visualization Toolkit*, Prentice Hall, ISBN 0-13-199837-4, 1996.

[6] R. Klein, G. Liebich, W. Straßer, "Mesh Reduction with Error Control", *IEEE Visualization '96*, pp. 311-318, 1996.

[7] R. Klein, J. Krämer, "Building Multiresolution Models for Fast Interactive Visualization", *Proceedings of the SCCG '97*, 1997.

[8] G. Turk, "Re-Tiling Polygonal Surfaces", *Computer Graphics*, Vol. 26, pp. 55-64, July 1992.

[9] P. Bourke, "Geometry", `http://www.mhri.edu.au/~pdb/geometry`, February 1997.

[10] E. Puppo, R. Scopigno, "Simplification, LOD and multiresolution principles and applications", *Proceedings Eurographics'97, Computer Graphics Forum*, 16(3), 1997.