Faculty of Mathematics and Natural Sciences

Department of
Mathematics and
Computing Science

Efficient Implementation of KLSDecimate: An Algorithm for Surface Simplification

Chris Döling

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

August, 1999

RuG

Efficient Implementation of KLSDecimate: An Algorithm for Surface Simplification

Chris Döling

Abstract

To represent 3-dimensional objects mostly triangles are used. Nowadays, the representation of these objects consists of a large amount of triangles. However, the available computing power is not always sufficient to render the model in real-time, or even visualize it in reasonable time.
For such cases, algorithms exist which simplify or decimate the model. One of these decimation algorithms, by Klein, Liebich and Straßer, was already implemented. It gives better results than most other algorithms because it uses a better error metric. However, the implementation lacked a lot of speed, due to some inefficient routines and datastructures. They were replaced with more efficient ones. Also the generation of a multiresolution model (MRM) with this algorithm was studied. Suggestions for adapting the implementation to support the MRM are given.

Contents

Chapter 1

Introduction

1.1 Decimation

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. Datasets of models used for medical or geographical applications, for example, contain up to millions of polygons. Due to this increase in the size of datasets, problems arise for their visualization. Often, the available computing power is not sufficient, for instance for interactive (real-time) 3D rendering. The performance of the rendering hardware does not increase as fast as the complexity of the datasets. The bandwith of the current available networks is a limiting factor as well. So to manipulate and visualize datasets of this complexity, clearly some sort of simplification is desired to reduce their size.
Over the years, a variety of algorithms for reducing the size of datasets have been developed. One kind of reducing algorithms is called *decimation* algorithms. A decimation algorithm applies multiple passes over the dataset and progressively removes those vertices that pass a distance or angle criterion. The resulting holes are patched using a local retriangulating process. In this report such a decimation algorithm will be discussed.

1.2 Previous work

The program discussed in this report is based on a decimation algorithm designed by Klein, Liebich and Straßer, as discussed in [KLS96]. This algorithm uses a modified Hausdorff distance between the original and the simplified mesh as an error measure. In 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. The algorithm will be described in detail in chapter 3.
The algorithm as discussed in [KLS96] was already implemented by A. Noord

and R.M. Aten in a program called KLSDecimate, as they have reported in [NA98]. This implementation, however, was rather slow, partly because their goal was to build a working version, rather than a full-featured one. For this same reason the implementation also needed some functional improvements. Some changes with respect to speed and functionality were made by R. van der Zee, as he described in [RvdZ99].
Although a great gain in speed was achieved, the implementation was still rather slow, in comparison with other decimation algorithms, and also in comparison with the decimation times the authors of the algorithm claimed. So, the program was reviewed again, and some improvements were implemented. These improvements will be discussed in the next chapters.

Chapter 2

Background

2.1 Terminology

In order to understand the rest of this report, a number of terms will be explained in this section.

2.1.1 Surfaces

In this report surfaces (2-manifolds) embedded in 3-dimensional Euclidian space are considered, which are approximated by *triangle meshes* or, for short, *meshes*. A triangle mesh is a piecewise linear surface consisting of triangular *faces*, which are pasted together along their edges. Edges of a triangle, or vertices of an edge are called *faces*. Vertices, edges and triangles are referred to as *simplices*: a vertex is a 0-simplex, an edge is a 1-simplex and a triangle is a 2-simplex. x-simplices have $(x-1)$-simplices as their faces. If t is a k-simplex, then k is called its *order*.

2.1.2 Triangle meshes

We can also define a triangle mesh 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.

During the process of mesh simplification, which is the subject of this report, a triangle mesh is transformed into another triangle mesh, 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. In order to obtain a new triangle mesh T', this polygon has to be retriangulated. A *triangulation* of polygon P is a decomposition of P into triangles by a maximal set of non-intersecting diagonals. This set has to be maximal in order to prevent that a polygon vertex is in the interior of a triangle edge. After this retriangulation a new triangle mesh T' is obtained.

2.1.3 Topology

The *topology* of a surface is an important property when decimating a triangle mesh. That is, it is important not to change the topology of the surface. When a given surface S is transformed into another surface S' by an elastic deformation D, that is, an invertible transformation which does not tear the surface apart, then the surfaces S and S' are said to be *topologically equivalent* or *homeomorphic*. The transformation D is then called a *topological mapping* or *homeomorphism* and D is said to be *topology preserving*. Examples of topology preserving and non topology preserving transformations are given in figure 2.1.

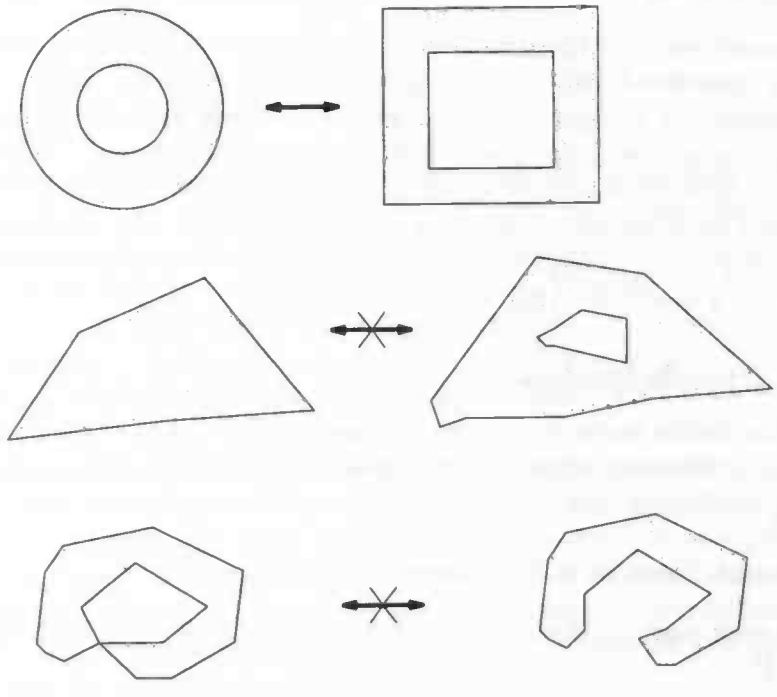


Figure 2.1: *Examples of topological equivalence and inequivalence.*

2.1.4 Classification of vertices

In the implementation of KLSDecimate by Noord and Aten the following a vertex classification is used, consisting of three categories (see also figure 2.2):

1. The *simple vertex* is the most basic type of vertex. A vertex v is classified as a simple vertex when all the triangles in the loop surrounding v have exactly two neighbouring triangles in that loop.

2. A vertex v is called a *border* or *boundary vertex* when it has exactly two triangles, which have only one neighbour in the loop of triangles surrounding v.

3. A vertex is called a *complex vertex* when it can not be classified as a simple vertex or a border vertex. Such a vertex has triangles with more than two neighbours in the loop or are part of more than one border.

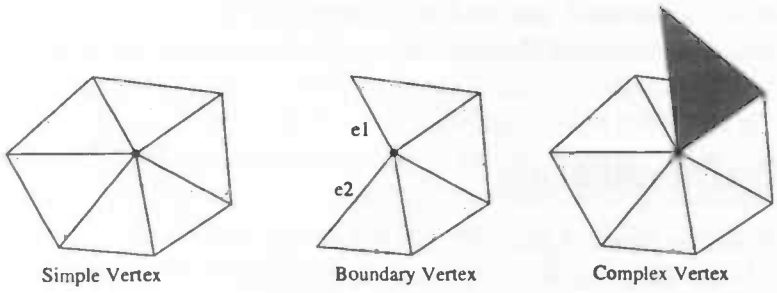


Figure 2.2: *Examples of vertex classification.*

2.2 The Visualization Toolkit

The implementation of the decimation algorithm is based on the the *Visualization Toolkit* (VTK) [VTK98]. The toolkit is written in C++, which is based on objects and classes. VTK features a large range of datastructures to cope with many different types of data. On most of them a huge amount of operations is available. Some of these datastructures or objects of VTK are used within KLSDecimate. The most important ones are:

- `vtkPolyData`: This datastructure is able to store polygons and its vertices. It is highly suited for storing and manipulating triangle meshes.

- `vtkTriangle`: This object represents a triangle. It has some operations defined on it, like for instance computation of the normal of the triangle.

- `vtkIdList`: This dynamic list can be used to store identifiers such as vertex or triangle ID's.
- `vtkPolyDataReader/Writer`: These objects are able to stream polygonal data from and to a file.

Of the above the `vtkPolyData` structure is the most important one used by KLSDecimate. The main reason for using this datastructure is that all functions for the required manipulations on the data are available. Furthermore it stores all relations between the elements (vertices, triangles, etc.) stored in it. These relations are needed by the algorithm. For retrieving these relations the following methods can be used:

- `void GetPointCells(int ptId, vtkIdList& cellIds);`
- `void GetCellPoints(int cellId, vtkIdList& ptIds);`

The first method can be used to get all the cells (triangles in our case) surrounding a certain vertex. This is convenient in some cases, for instance when a vertex is removed, we need to know what triangles surround it. The second method returns all the identifiers of the vertices defining a certain cell.

2.3 Retriangulation

In the current implementation of the algorithm, the triangulation method used is *recursive loop splitting*. The hole that has to be retriangulated is split into two halves along a line. The split line is defined by two non-neighbouring vertices in the loop of vertices of which the hole consists. 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 through the loop. 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*. This is the minimum distance of the loop vertices to the split plane, divided by the length of the splitting line. The possibility with the maximum aspect ratio is selected. A minimum aspect ratio for triangulation to succeed can be set.

Chapter 3

The decimation algorithm

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, as discussed in [KLS96], and also in [NA98], is a decimation algorithm that uses a modified Hausdorff distance between the original mesh and the simplified mesh as an error measure. In 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 lack this global error measure. Many measure the 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, the method by Klein *et al.* ensures a higher geometric accuracy than most other algorithms. In this chapter the algorithm as discussed in [KLS96] and [NA98], is described.

3.2 The error metric

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

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

where $d(x, y)$ is the Euclidean distance between two points $x, y \in R^n$. We can use this definition to define the distance $d_E(X, Y)$ from a set X to a set Y by

$$d_E(X, Y) = \sup_{x \in X} d(x, Y). \tag{3.2}$$

This distance is called the *one-sided Hausdorff distance* between the set X and the set Y. It is not symmetric, so in general $d_E(X,Y) \neq d_E(Y,X)$.
If, for example, the one-sided hausdorff distance $d_E(T,S)$ from S to T is less than a predefined error tolerance ϵ, then

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

For mesh simplification this condition is sufficient in most cases. However, because the one-sided Hausdorff distance is not symmetric, there can be problems in some cases, especially near borders or interior edges.
Such cases are handled by using the *two-sided Hausdorff distance*, or in short *Hausdorff distance*. It is defined by

$$d_H(X,Y) = \max(d_E(X,Y), d_E(Y,X)). \tag{3.3}$$

This two-sided Hausdorff distance is symmetric and we have

$$d_H(X,Y) = 0 \Longleftrightarrow X = Y.$$

Now, if the Hausdorff distance $d_H(T,S)$ between the original triangle mesh T and the simplified mesh S is less than a predefined error tolerance ϵ, then

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

and because it is symmetric

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

3.3 The algorithm

Like a typical mesh simplification algorithm, the algorithm starts with the original triangle mesh T and successively simplifies it by removing vertices and retriangulating the resulting holes. This process goes on until no more vertices can be removed from the simplified triangle mesh S without exceeding a predefined maximum Hausdorff distance between the original mesh and the simplified one.
The main issue of the algorithm is the computation of the Hausdorff distance. In general, this is quite a complicated task, since the distances between every pair of points from both triangle meshes need to be computed. In our case though, where we have an iterative simplification procedure, this can easily be solved. The idea is to keep track of the actual Hausdorff distance between the original and simplified mesh and of the *correspondence* between these two meshes from step to step. This correspondence ensures that distances are calculated to the correct part of the mesh. By using the correspondence, for every vertex x in the original triangle mesh which has already been removed, the triangle of the simplified mesh that is nearest to

x can be found.
The idea behind the algorithm is to compute and update an error value for every single vertex of the simplified mesh. This value is called the *potential error*, that is the Hausdorff distance that would occur if the vertex was removed. At the beginning of the algorithm, the potential errors of all vertices in the original mesh are calculated and stored in a list or priority queue. This list is sorted in ascending order, according to the potential error of each vertex. In each step the vertex whose removal causes the smallest potential error, that is, the vertex that is at the beginning of the sorted vertex list, is removed. After the vertex is actually removed, this list is updated. When a vertex is complex or if retriangulation of the hole would lead to topological problems, the potential error is set to infinity.
This strategy of sorting automatically preserves sharp edges, since the vertices of which they consist have a relatively large potential error, and are therefore at the bottom of the list.
After building the sorted vertex list, the vertices are iteratively removed from the list and from the triangle mesh. When the potential error of the vertex that is to be removed becomes larger than the predefined Hausdorff distance, the algorithm terminates.
When the vertex v is removed from the triangle mesh its adjacent triangles are removed and the remaining hole is retriangulated. After this retriangulation has been successfully completed, the potential errors of all neighbouring vertices of v have to be recalculated. They are removed from the list and reinserted again according to their new potential error.

3.3.1 Potential error calculation

One of the most important parts of the algorithm is the computation of the potential error of a vertex. The distance d_H between all the points of the two triangle meshes has to be computed. In order to speed up the computation, the one-sided Hausdorff distance

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

is used. When none of the neighbouring vertices of a single vertex 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 $t_i, i = 1..n$, be the triangles that are incident to v. These triangles would be removed if v were to be deleted. Let $s_j, j = 1..m$, be the new triangles produced in retriangulation of the remaining hole, after v was deleted. When the vertex v was a border vertex then $m = n - 1$, and otherwise $m = n - 2$. To calculate $d_E(\{t_i\}_{i=1..n}, \{s_j\}_{j=1..m})$ it is sufficient to compute

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

see figure 3.1.

However, after a few simplification steps 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 becomes a bit more complicated. 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, like already mentioned above. So for each already removed vertex v of the original triangle mesh T, the triangle s of the simplified mesh S that has the smallest distance to v has to be stored. Vice versa, for each triangle $s \in S$ all vertices v of T for which s is the triangle with the smallest distance to v are stored. This information is updated in each iteration step and is sufficient for calculating $d_E(T, S)$. Let $s_i^l, i = 1..n$, 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 mesh T. Let $s_j^{l+1}, j = 1..m$, be the set of new triangles created during the step from triangle mesh S_l to S_{l+1}. Let V be the set of vertices of the original mesh that are already removed. Each $v_k \in V$ must be nearest to one of the removed triangles $s_i^l \in S_l$. For each triangle in the original triangle mesh T incident to one of the vertices v_k the distance to S_{l+1} is calculated. In order to do this, it is sufficient to calculate the distances between triangles of the original triangle mesh and a subset $\widetilde{S} \subset S_{l+1}$. Here, $\widetilde{S}$ contains the newly created triangles of S_{l+1} and the triangles of S_{l+1} sharing at least one vertex with the the newly created ones. This is justified by

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

Because this procedure is a local one, it accelerates the distance calculation considerably. Furthermore, it ensures that the distance is always measured to the correct part of the simplified mesh, so the distance measurement respects the topology.

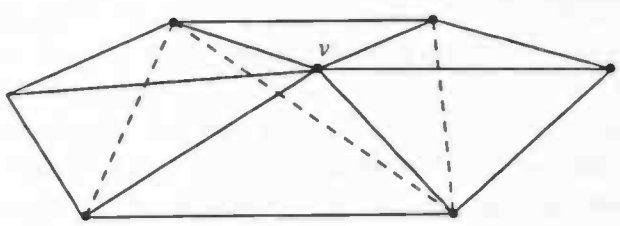


Figure 3.1: *Since none of the neighbouring vertices of v have already been removed, it is clear that the distance from the original to the simplified triangle mesh is the distance from vertex v to the simplified triangle mesh (dashed lines).*

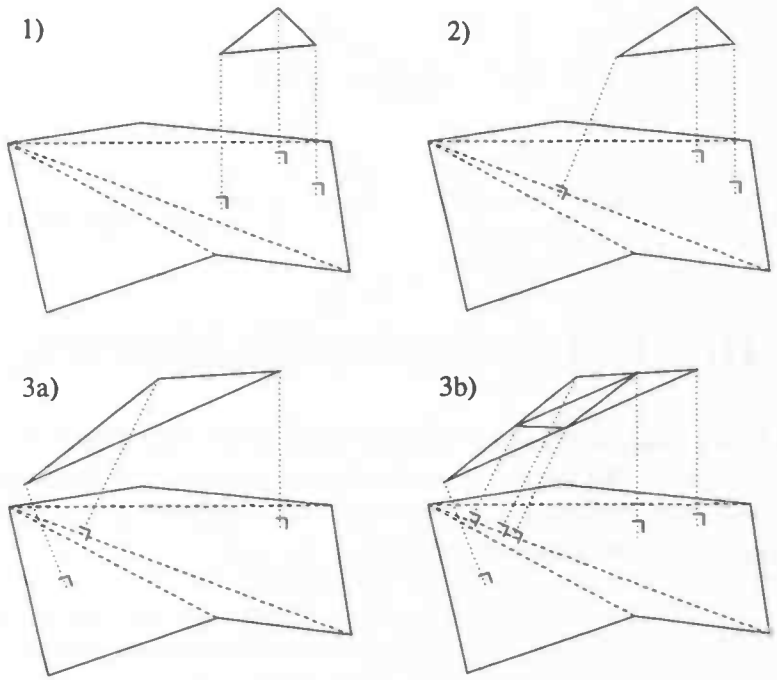


Figure 3.2: *Different subcases distinguished when computing the distance from triangle to mesh.*

3.3.2 Distance calculation

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. If the smallest distances from the vertices of the triangle t exist to different triangles of the simplified mesh S, the distance from t to S may occur between a point on the border or even inside of the triangle t and a point somewhere on the simplified mesh S. To solve this problem, two cases are considered:

1. Triangle t of the original mesh has no vertex in common with the simplified triangle mesh S.
2. Triangle t of the original mesh has one or two vertices in common with the simplified triangle mesh.

In the first case three subcases can be distinguished (see figure 3.2):

1. All three vertices are nearest to the same triangle $s \in S$.
2. The three vertices are nearest to two triangles $s_1, s_2 \in S$ that share an edge.

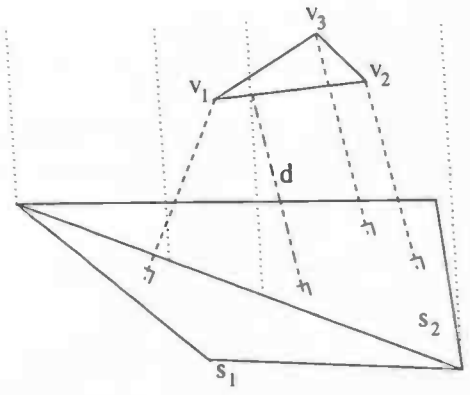


Figure 3.3: *An example of the use of a half-angle plane. The dotted lines lie in the half-angle plane. The distance from the intersection of the triangle t and the half-angle plane to the triangles s_i is larger than the distance from the vertices v_i to the triangles s_i, so the former distance has to be used.*

3. All other cases.

In the first subcase the distance calculation is simple:

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

The second subcase is a bit more complicated. To solve it, 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. This procedure is illustrated in figure 3.3.

In the third subcase, that is all other cases, the triangle t is subdivided until either subcase 1 or subcase 2 applies for the subtriangles (see figure 3.2). When the longest edge of a subtriangle is smaller than a predefined error tolerance ϵ the subdivision also terminates. In that case the maximum distance

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

is used, where v_i^{sub} is the subtriangle 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 subcases of case 1.

Chapter 4

Improved implementation

4.1 Distance calculation

In order to speed up the distance calculation, R. van der Zee suggested in [RvdZ99] to use the optimizations proposed by R. Klein and J. Krämer in [KK97]. These optimizations were considered, but not implemented. Instead another optimization was done. The optimizations proposed by Klein and Krämer, the reasons why they were not implemented and the optimization that was done instead, will be described below.

4.1.1 Accelerating the algorithm according to Klein et al.

According to Klein and Krämer the computation costs of a simplification algorithm can be estimated as follows: If we remove the m-th vertex from a mesh, the potential error of its neighbour vertices has to be updated. If we consider a closed triangle mesh without border, the relation between the number of triangles f and the number of vertices v is given by

$$f = 2v - 4. \tag{4.1}$$

To estimate the cost of this operation for the removal of one vertex we have to retriangulate the remaining hole and to measure the distance between the vertex and the retriangulated area. Also the distances between all vertices corresponding to triangles of the retriangulated area and the new triangles of the retriangulated area have to be calculated. According to Eq. 4.1 these are in average

$$\frac{m-1}{f} = \frac{m-1}{2v-4} \tag{4.2}$$

vertices per triangle. The distance between these vertices and in average 5 triangles in the retriangulated area have to be calculated. In total, in each simplification step we have to calculate in average

$$6 \times 6(\frac{m-1}{2(v-m)-4}+1) \times 5 \tag{4.3}$$

distances between vertices and triangles. Therefore, the total number of distance calculations necessary to remove n vertices is in average

$$\begin{aligned} & 180 \sum_{m=1}^{n} (\frac{m-1}{2(v-m)-4}+1) \\ & = 180(n + \sum_{m=1}^{n} \frac{m-1}{2(v-m)-4}) \end{aligned} \tag{4.4}$$

To reduce this amount of distance computation we exploit the following observations:

- Consider all neighbour vertices of the vertex and let ϵ be the maximum of their potential error. As soon as we find one vertex with a distance greater than ϵ, the distance computation can be stopped, since in this case the neighbouring vertex will be removed first and a further potential error has to be computed for the vertex. If the distance computations are stopped, the potential error is set to infinity.

- For the vertex itself, instead of computing its distance to all new triangles of the retriangulated hole, the distance of the border vertices of the remaining hole to an average plane passing through the vertex itself is measured. If all vertices are on the same side of the average plane, this distance is always smaller than the minimum distance of the vertex to the triangles. Therefore, if it exceeds the ϵ defined in the previous item, the distance computations can be stopped as well.

Although Klein and Krämer make it all seem very logical at first sight, it is not the case when looked at closely. First, they do not say anything about the consequences their proposals might have. For instance, they don't say anything regarding the accuracy of the approximation after applying the optimizations. Secondly, and most important, the description of the proposed optimizations is not really understandable. For instance, in the first proposal all neighbour vertices of *the vertex* are considered, but it is not clear which vertex is meant by *the vertex*. Also in the first proposal, there is a sentence which says that the distance computation can be stopped as soon as one vertex with a *distance* greater than ϵ is found. In this sentence, it is not clear which part of the whole distance computation has to be stopped, neither is it clear exactly which *distance* is meant. In general, a lot of terms like *the vertex* and *the distance* are used in a context where it is not directly clear what they mean exactly.
For the reasons mentioned above, the proposed optimizations were not implemented.

4.1.2 Keeping an administration of calculated distances

While examining the program, it appeared that the distance calculations needed for updating the potential error were not very efficient. It appeared that many distances were calculated two, or even more times.
When the potential error is updated, a large amount of distances are computed. For each removed vertex which refers to one of the removed triangles as being nearest, the distances of its surrounding triangles to a subset S of the simplified mesh have to be computed. This subset S contains the newly created triangles and the triangles of the simplified mesh sharing at least one point with the newly created ones. This procedure is described in detail in section 3.3. When the distance between a triangle and a mesh is calculated, the distances between the corner vertices of the triangle and the mesh are computed. As the description of the potential error calculation above shows, many triangles for which the distance to S is calculated, are neighbours. Thus, these triangles have many vertices in common. Since the distance of each triangle to S has to be determined seperately, a large amount of redundant distance computations were done in the original implementation.
To solve this problem, three global datastructures were created:

- `vtkIdList` **calculatedVerts**: the vertex ID's of vertices for which the distance is already calculated.
- `vtkIdList` **calculatedTris**: the triangle ID's of the triangles to which the vertices in `calculatedVerts` refer as being closest.
- `vtkFloatArray` **calculatedDists**: the already calculated distances.

These datastructures are reset each time the calculation of a potential error starts. They are linked through their indices, that is: the vertex with the ID stored at index i in `calculatedVerts` refers to the triangle with the ID stored at index i in `calculatedTris` as being nearest, and the distance between these two is the value stored at index i in `calculatedDists`.
Using the `vtkIdList` and `vtkFloatArray` is quite convenient, since they are available within VTK and are easy to use. Both the `vtkIdList` and the `vtkFloatArray` does range checking when inserting a value, and allocates more memory if necessary, so the datastructures never grow out of bounds. Every time a minimum distance is found the datastructures mentioned above are updated. When computing the distance from a triangle to the mesh, the datastructures are checked whether the distance of one or more of the corner vertices is already calculated. When this is the case, the stored value is used, instead of recalculating it. Clearly this optimization saves a large number of calls to `Distance2BetweenPointAndTriangle`, which according to [RvdZ99] is the slowest routine in the program.

4.2 Correspondence list

As described in chapter 3, the correspondences between already removed vertices in the original mesh and triangles in the simpified mesh have to be stored. For that reason the `CorrespondenceList` datastructure was implemented. However, since the original implementation was intended to be a working version rather than a full-featured one, this datastructure is not very fast, nor efficient. Therefore this datastructure was replaced. The original `CorrespondenceList` and the new implementation are discussed below.

4.2.1 Original correspondence list

In the original implementation the correspondences are stored in a structure which consists of a vertex list and a 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 ID's of the vertices that define the triangle, a list of ID's of vertices. These ID's refer to vertices in the vertex correspondence list. These vertices refer to the concerning triangle as being nearest. Vertices are stored according to ID, lowest first; triangles are sorted according to the lowest ID of their vertices.
The structure looks as indicated in figure 4.1.

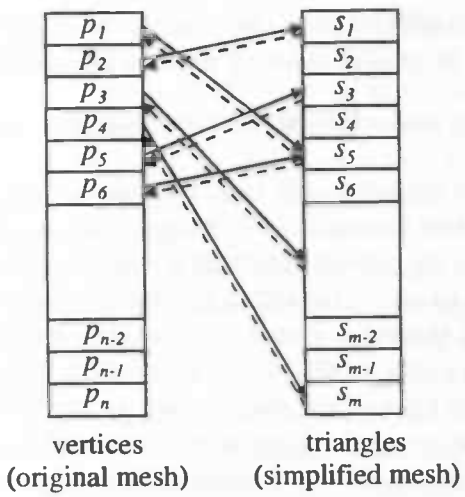


Figure 4.1: *The original correspondence list.*

The datastructure described above is clearly not very efficient. Both lists, the vertex list and the triangle list, are doubly linked lists. This implicates that linear searching has to be done to retrieve information from the structure. As a consequence, almost all operations (inserting, deleting and

retrieving a correspondence) take $O(n)$ time in the worst case. For small datasets this is not a very big problem, but since our goal is to decimate large datasets, this is not a satisfactory solution.
Another problem is the inefficient data storage. A triangle is stored in the list, by storing the ID's of the vertices that define the triangle. Before storing, these ID's first have to be sorted according to their lowest value. However, each triangle has its own ID, and its defining vertices can be retrieved through the `vtkPolyData` datastructure, so it is better to store that ID. In that way, storage space is saved, and also the sorting does not need to take place.

4.2.2 Optimized implementation

While replacing the old correspondence list, it appeared that the vertex list that is used in that implementation, is not needed. It is only referred to once within the correspondence list, and then only to update that same vertex list. Outside the correspondence list it is not used at all. So, the only thing that is needed is the triangle list.
The optimized `CorrespondenceList` makes use of a simple array for the triangle list. This array has as its size the number of triangles in the original mesh. This way it is always large enough and all operations stay within boundaries. The index of an array record is the ID of the triangle the record belongs to. This can be done because the triangle ID's are reused when a vertex is being removed, so the value of an ID never exceeds the number of triangles. A record of the array contains a pointer to a list of vertex ID's. The vertices belonging to the ID's in this list refer to the concerning triangle as being nearest. The new correspondence list is shown in figure 4.2.

The most important operations on the correspondence list are:

- `GetTriangleCorr(tid)` - returns a pointer to the list of corresponding vertices of the triangle with ID *tid*.
- `InsertVertexTriangleCorr(vid, tid)` - inserts a new correspondence between the vertex with ID *vid* and the triangle with ID *tid*.
- `DeleteTriangleCorr(tid)` - clears the element of the array belonging to the triangle with ID *tid* and returns the list of vertex ID's it contained.

These operations now only need $O(1)$ time, since accessing an array at a certain index only needs $O(1)$ time. So, this replacement provides a major speedup of the program. The results will be discussed in chapter 5.

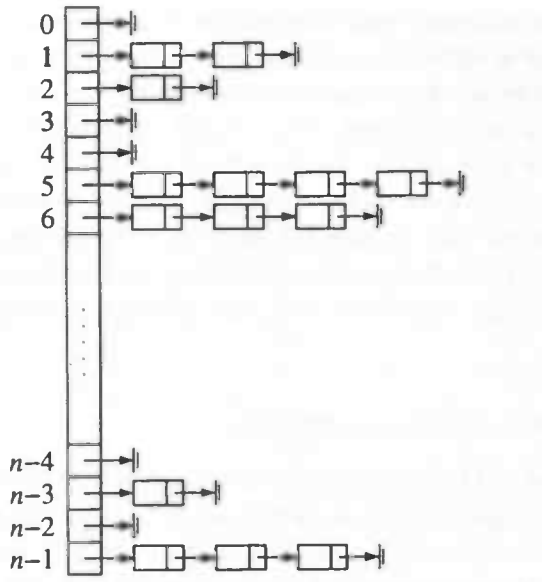


Figure 4.2: *The new correspondence list.*

4.3 Sorted vertex list

In order to store the potential errors of the vertices, we need a certain datastructure. Obviously this structure has to be a fast one, since it can be quite large and there are many operations performed on it when a vertex is removed. The vertex with the smallest potential error has to be determined and removed from the structure, the potential errors of the surrounding vertices have to be recalculated and thus modified within the structure, and of course when decimation is started, the potential errors of all vertices have to be inserted in some logical way.
In the original implementation as discussed in [NA98] a doubly linked list was used, in which the elements were sorted according to their potential error. Thus, in this list the element with the smallest potential error is in front. However, when a vertex has to be retrieved from this list according to its ID, it is not an ideal situation. In the worst case the whole list has to be traversed, since you can only perform linear searching on a linked list. So this action is of order $O(n)$, where n is the length of the list. Also, when the potential errors are recalculated, they have to be moved in the list in order to maintain the sorting order. Especially this operation is slow, because the element has to be deleted and reinserted, these last two actions both being of order $O(n)$.
This doubly linked list was already replaced with a more efficient datastructure, as described in [RvdZ99]. This datastructure is briefly described in section 4.3.1. For reasons that will be discussed below, this datastructure

was again replaced with a new one, which will be described in section 4.3.2.

4.3.1 A combined heap and hash table

The datastructure designed by R. van der Zee uses a combination of a heap with a hash table. A heap is a datastructure designed to have the smallest (or largest) element available in $O(1)$ time. It is a binary tree where the elements are ordered according to a certain criterion. When we consider a heap with the smallest element at the top, the criterion that has to be maintained is that the key-value of every element of the tree is smaller than the key-value of its children. An example of such a heap is shown in figure 4.3.

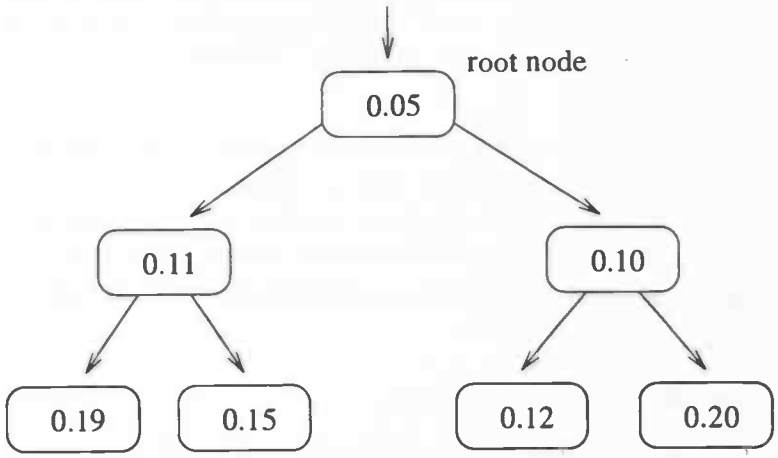


Figure 4.3: *An example of a heap.*

Because we need to search in the datastructure, and searching in a heap takes $O(n)$ time, the combination of a heap with a hash table was chosen. The time it takes to find an element with a certain ID in a hash table is of order $O(m)$, where m is the average depth the of node from the hash table. A comparison of time complexities of the separate hash table and heap, and the combination of both is given in table 4.1.
In the implementation the heap and the hash table are linked through

	Hash table	Heap	Heap & Hash table
smallest-element	$O(n)$	$O(1)$	$O(1)$
search	$O(m)$	$O(n)$	$O(m)$
delete	$O(m)$	$O(n)$	$O(m)$
insert	$O(1)$	$O(\log(n))$	$O(\log(n))$

Table 4.1: *The complexity of the datastructures, where m is the average depth of a node from the hash table.*

pointer structures, to find the corresponding elements. When an operation is performed on the datastructure, it is accessed through the structure which is best suited for that operation. An example of a combined heap and hash table is shown in figure 4.4.

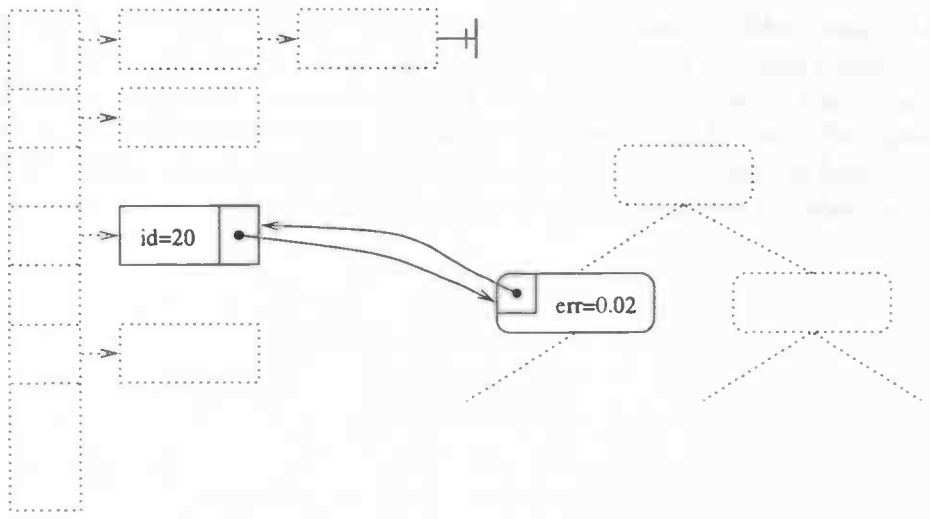


Figure 4.4: *An example of a combined heap and hash table.*

4.3.2 The new priority queue

Although the datastructure described above is quite efficient, it was nevertheless replaced with a new one. The implementation of the combined heap and hash table is quite complex, and it seemed that a more simple implementation was possible.
In [CLR90] another implementation of a heap is described. This implementation makes use of an array for the heap, instead of a binary tree with links between the parent and child nodes, as was used in the implementation made by R. van der Zee. So, this heap datastructure is an array object that can be viewed as a complete binary tree, as shown in figure 4.5. This implementation of a heap should be background knowledge for every computing scientist. Nevertheless it will be discussed here, as an introduction to the new implementation of the priority queue.

Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: $length(A)$, which is the number of elements in the array, and $heapsize(A)$, the number of elements in the heap stored within array A. This means that no element past $A[heapsize(A) - 1]$, where $heapsize(A) \leq length(A)$, is an element of the heap, even though $A[0 \ldots length(A) - 1]$ may contain valid numbers.

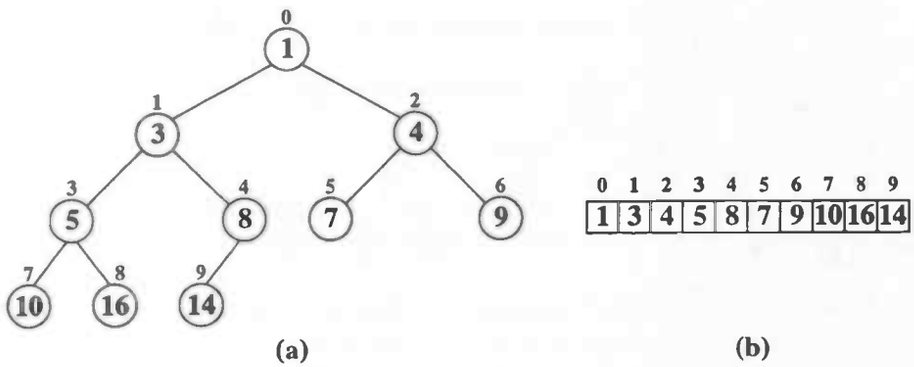


Figure 4.5: *A heap viewed as* (a) *a binary tree and* (b) *an array. The number within the circle at each node in the tree is the value stored at that node. The number next to a node is the corresponding index in the array.*

The root of the tree is $A[0]$, that is in a C/C++-like array. For languages in which arrays start at index 1, the above and following definitions are slightly different, but they will not be discussed here. Given the index i of a node, the indices of its parent `Parent(`i`)`, left child `Left(`i`)`, and right child `Right(`i`)` can be computed simply:

```
Parent(i)
    return ⌈i/2⌉ − 1

Left(i)
    return 2i + 1

Right(i)
    return 2i + 2
```

As already mentioned in section 4.3.1 heaps also have to satisfy the *heap property*: for every node i other than the root,

$$A[\texttt{Parent}(i)] \leq A[i], \tag{4.5}$$

that is, the value of a node is at least the value of its parent. Thus, the smallest element (or the largest, depending on the criterion) in a heap is stored at the root, and the subtrees rooted at a node contain larger values than does the node itself. For maintaining the heap property, the subroutine `Heapify` is available. Its input is an index i into the array. When `Heapify` is called it is assumed that the binary trees rooted at `Left(`i`)` and `Right(`i`)` are heaps, but that $A[i]$ may be larger than its children, thus violating the heap property (4.5). The function of `Heapify` is to let the value at $A[i]$ 'float down' in the heap so that the subtree rooted at index i becomes a heap.

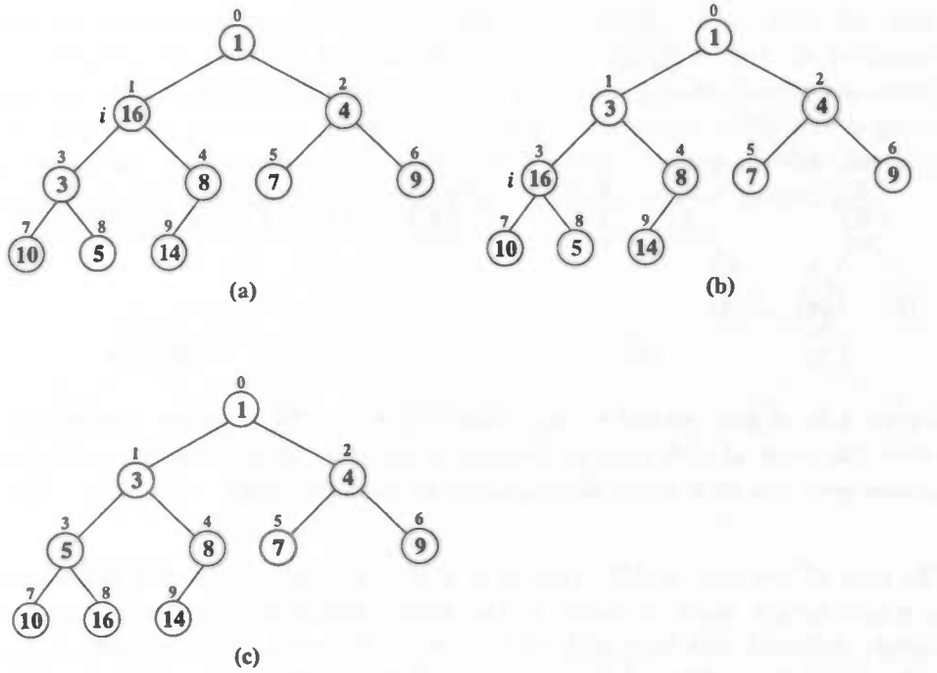


Figure 4.6: *The action of* `Heapify` *on node* $i = 1$*, where heapsize* $= 10$. **(a)** *The initial configuration of the heap, with the value at node* $i = 1$ *violating the heap property because it is larger than both children.* **(b)** *By exchanging the nodes 1 and 3 the heap property is restored for node 1. Now the heap property is violated at node 3.* **(c)** `Heapify` *is called recursively on node 3, and now the heap property is restored. No further recursive calls of* `Heapify` *are needed.*

Figure 4.6 illustrates the action of `Heapify`. At each step, the smallest of the elements $A[i]$, $A[\texttt{Left}(i)]$ and $A[\texttt{Right}(i)]$ is determined, and its index is stored in *smallest*. If $A[i]$ is smallest, then the subtree rooted at node i is a heap and the procedure terminates. Otherwise, one of the two children has the smallest element, and $A[i]$ is swapped with $A[smallest]$, which makes sure that node i and its children satisfy the heap property. The node *smallest*, however, now has the original value $A[i]$, and thus the subtree rooted at *smallest* may violate the heap property. To solve this, `Heapify` must be called recursively on that subtree. The running time of `Heapify` is $O(\log n)$ for an n-element heap.

Such a heap can be used as an efficient priority queue, which is quite convenient for our purposes. That is, we have to store the potential errors, and extract the vertex with the smallest error. This can be done efficiently in a heap. There are some routines available for the use of a heap as a priority queue:

- `Minimum`: returns the minimum value in the heap.
- `Insert`: inserts a node into the heap;
- `Delete`: deletes a node from the heap;

The routine `Minimum` is an easy one: it simply returns the value stored at $A[0]$, which is the minimum value in heap A. So the running time of `Minimum` is $O(1)$.
The `Insert` routine inserts a node into heap A. To do so, it first expands the heap by adding a new leaf to the tree, that is, by increasing *heapsize* with 1. Then it traverses a path from this leaf towards the root to find a proper place for the new element. It does this by iteratively sliding down the parent node until the right place is found, starting at the newly created node. An example of an `Insert` operation is shown in figure 4.7. The running time of `Insert` on an n-element heap is $O(\log n)$, since the path traced from the new leaf to the root has length $O(\log n)$.

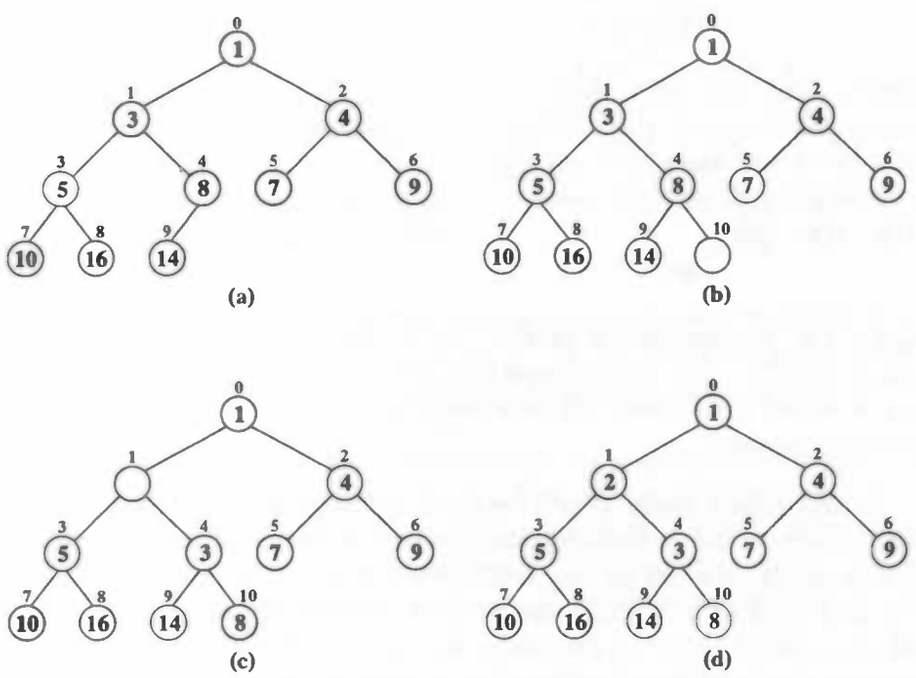


Figure 4.7: *The operation of* `Insert`. **(a)** *The heap before a node with key 2 is inserted.* **(b)** *A new leaf is added to the tree.* **(c)** *Values on the path from the new leaf to the root are copied down until a place for the key 2 is found.* **(d)** *The key 2 is inserted.*

The `Delete` routine is quite straightforward as well. It has as its argument the index i in the array A, where the node that has to be deleted is stored. The deletion is done by simply overwriting the node at index i with

the last node in the heap. Then *heapsize* is decreased with 1. Now the *heap property* may be violated, because the value now stored at $A[i]$ can be larger than the values stored at its left and right subtrees. So now `Heapify` is called at index i of A, to restore the heap property. An example of `Delete` is given in figure 4.8. The running time of `Delete` is $O(\log n)$, since it performs only a constant amount of work on top of the $O(\log n)$ time for `Heapify`.

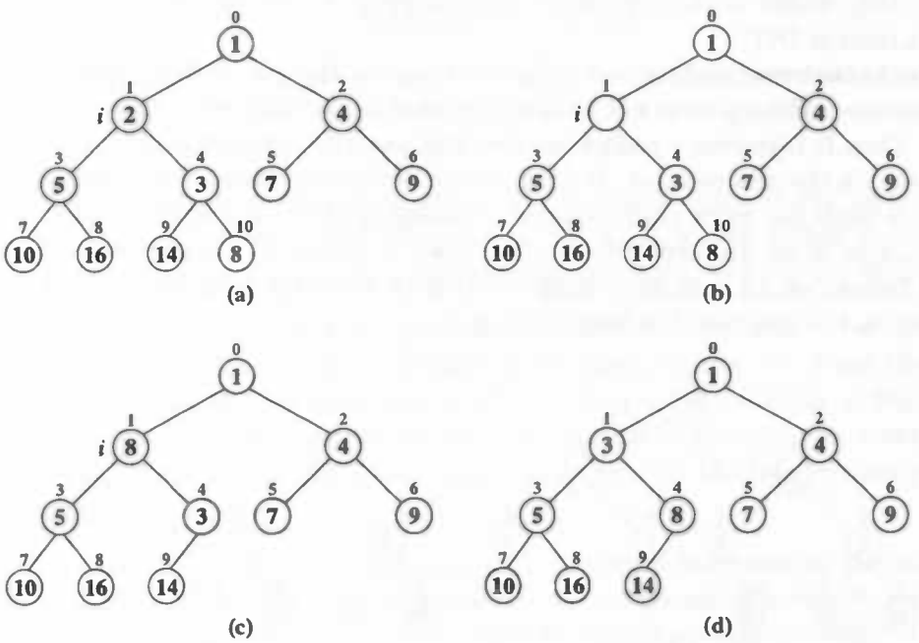


Figure 4.8: *The operation of* `Delete`. **(a)** *The heap before the node i is deleted from it.* **(b)** *The value at i is cleared from the heap, and* **(c)** *the last node in the heap is moved in its place.* **(d)** *Now* `Heapify` *is called on node i and the heap property is restored.*

As said before, every time a vertex is removed from the mesh, the potential errors of the surrounding vertices have to be updated. Finding these vertices in a heap is not an easy task, since a heap has no searching abilities. It is only good for extracting the smallest (or largest) value. This problem was also noticed by R. van der Zee, as he discussed in [RvdZ99]. He solved it by using a hash table to find the place of elements in the heap, as described above in 4.3.1. In the new implementation a simpler solution is used. The place of the vertex with its potential error in the heap is simply determined by using an array V. The size of the array V is the total number of vertices in the original mesh. The index i into V corresponds to the vertex ID. The integer value stored at $V[i]$, where i is the ID of a vertex, is the index number of the element in the heap belonging to that vertex. So, when we consider a heap A, the potential error value of a vertex with ID id is stored at $A[V[id]]$.

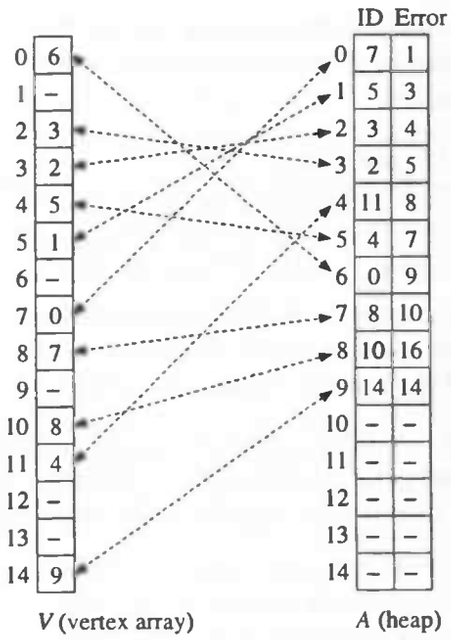


Figure 4.9: *An example of the new sorted vertex list. When we consider a vertex with ID id, the index into the heap A is stored at V[id]. So the potential error value of this vertex is stored at A[V[id]]. The virtual correspondences between the arrays V and A are indicated by the dashed arrows. The heap A is equivalent to the heap shown in figure 4.5.*

When a vertex is removed from the mesh it is also removed from the heap. Additionally, the index value in V is set to -1, to indicate that the vertex is not in the heap anymore. An example of the new implementation of the sorted vertex list is given in figure 4.9.

The operations on this new datastructure are slightly different from the operations on a heap described above. The difference is just that when there is a modification in the heap, the vertex array V has to be modified too, to make the correspondence right again. The `Heapify` routine was also changed to take V into account. So the most important operations on the new sorted vertex list are:

- `Insert(vid, pot_err)` - inserts a node with ID *vid* and potential error *pot_err* into the sorted vertex list. This is done by applying the above described heap insert routine, and additionally updating the index array V.
- `Delete(vid)` - deletes the node with vertex ID *vid* from the sorted vertex list. First the place in the heap is determined with the help of the index array V. Next the node is deleted from the heap. This is

done according to the heap deletion routine described above, meanwhile updating the correspondences stored in V. The correspondence in $V[vid]$ is set to -1.

- `DeleteFirstID()` - deletes the node with the smallest potential potential error from the sorted vertex list. Goes similar to `Delete(vid)`, with the difference that the place in the heap does not have to be determined first, since it is the root of the heap.
- `GetFirstPotentialError(&id)` - returns the smallest potential error in the heap. This value is stored at the root of the heap. Also returns the vertex ID of the corresponding vertex in *id*.
- `GetPotentialError(vid)` - returns the potential error of the vertex with ID *vid*. The place of the node in the heap is first looked up in the index array V, and after that the value is retrieved from the heap.
- `SetPotentialError(vid, pot_err)` - sets the potential error of the vertex with ID *vid*, who is already in the heap, to *pot_err*. First the index in the heap is looked up in V, and then the error value at that index in the heap is set to *pot_err*. After this it is moved up or down in the heap, to find the correct place for it, so that the heap property is satisfied again.

The interface to the new implementation is almost the same as the interface to the old one. As a result, replacing the old datastructure with the new structure within the implementation of KLSDecimate was not difficult. The time complexity of the new implementation is slightly better than the combination of a heap and a hash table. This is because of the use of the index array V. That is, the time to find the place of a vertex with its potential error in the heap is now $O(1)$, since the vertex ID can be used as an index into V. With the hash table, this used to be $O(m)$, with m the average depth of a node from the hash table. The table below shows a comparison of time complexities between the combined heap and hash table and the new implementation of the sorted vertex list.

	Heap & Hash table	New implementation
smallest-element	$O(1)$	$O(1)$
search	$O(m)$	$O(1)$
delete	$O(m)$	$O(\log n)$
insert	$O(\log(n))$	$O(\log n)$

Table 4.2: *The time complexity of the datastructures, where m is the average depth of a node from the hash table.*

We can see that for searching the time complexity is better in the new implementation, but that for deletion it is worse. So, in average the time

complexity of the sorted vertex list stays quite the same. Furthermore, the memory usage of the new datastructure is slightly better. The new implementation stores only two integers and a floating point number per vertex, and the old one stores the same floating point number, one integer (the vertex ID) and many pointers (parent, left and right node, pointers between the heap and hash table, and pointers within the hash table) per vertex. Also this implementation was quite invalved, and the code was really hard to read. The new code though, is very readable. Furthermore, it is a quite compact module, with only a few lines of code in comparison with the old implementation. In addition, the datastructure now has a guaranteed time complexity, which was not the case with the combined heap and hash table. Because a hash table was used, only an average time complexity could be given, since the hash function is not guaranteed to be chosen correctly.
As a result of the modifications described above, a really large improvement in speed was achieved. The results of these modifications will be discussed in detail in the next chapter.

Chapter 5

Results of the new implementation

After the improvements described in chapter 4 were made, a great increase in speed was achieved. In this chapter this gain in performance will be discussed, and also a comparison between the old implementation and the new one, with respect to complexity and absolute timing will be made.

5.1 Complexity

In [SL96] by M. Soucy and D. Laurendeau the theoretical complexity of a decimation algorithm based on vertex removal and retriangulation, is calculated. The algorithm by Klein, Liebich and Straßer has the same underlying main loop and should therefore have the same complexity. By comparing the complexity of KLSDecimate with the complexity as calculated by Soucy and Laurendeau, it can be determined if there are still any routines or datastructures which are not efficiently implemented.

5.1.1 Complexity calculation

Let t_{dpt} be the time needed to compute the distance between a vertex and a triangle. Assuming that the number of triangles needed for retriangulation after the removal of vertex v is N_r, the time needed to recompute the error of each removed vertex is equal to t_{dpt} times N_r. When N is the number of vertices in the original mesh and n is the number of removed vertices, then the number of removed vertices N_{rv} in the area of retriangulation can be estimated as follows:

$$N_{rv} = \frac{N}{N-n}. \tag{5.1}$$

These vertices are used in the computation of the retriangulation error. So, the computation time of a retriangulation error for a given number of

removed vertices n can be estimated by

$$t_{re} = \frac{N_r t_{dpt} N}{N - n}. \tag{5.2}$$

If all other computation times are neglected, and we thus focus on the error calculation, the time needed to remove a vertex for a given value of n is

$$t_n = \frac{N_t N_r t_{dpt} N}{N - n} \tag{5.3}$$

where N_t is the estimated number of vertices for which the potential error needs to be recalculated.
Now it is possible to find a function $t(n)$ which models the running time of the algorithm needed to remove n vertices from the mesh:

$$t_{[\mathrm{SL96}]}(n) = \int_0^n t_k dk = N_t N_r t_{dpt} N \log(\frac{N}{N - n}). \tag{5.4}$$

5.1.2 A comparison of complexities

In order to see whether the complexity of the modified implementation is better than the complexity of the original implementation, some measurements were done on both implementations. For this experiment, the 'fran' dataset was used. This dataset, which is provided with VTK, consists of 26460 vertices and 52260 triangles. The running time in milliseconds was recorded after the removal of every 100 vertices.

Original implementation

Vertex	Time(ms)	Vertex	Time(ms)
1000	36014	13000	1167887
2000	84100	14000	1324585
3000	139932	15000	1494353
4000	203938	16000	1707534
5000	282931	17000	1922340
6000	362466	18000	2164952
7000	450373	19000	2435184
8000	548683	20000	2708823
9000	657369	21000	3007926
10000	770501	22000	3346538
11000	893951	23000	3715636
12000	1026167	23800	4020997

Table 5.1: *Experimental results of the original implementation.*

In table 5.1 a subset of the results (for every 1000 removed vertices) of the original implementation is shown. The numbers in the first column are the number of vertices after whose removal the time in the second column was recorded. A graphical representation of the results together with the theoretical curve is shown in figure 5.1.

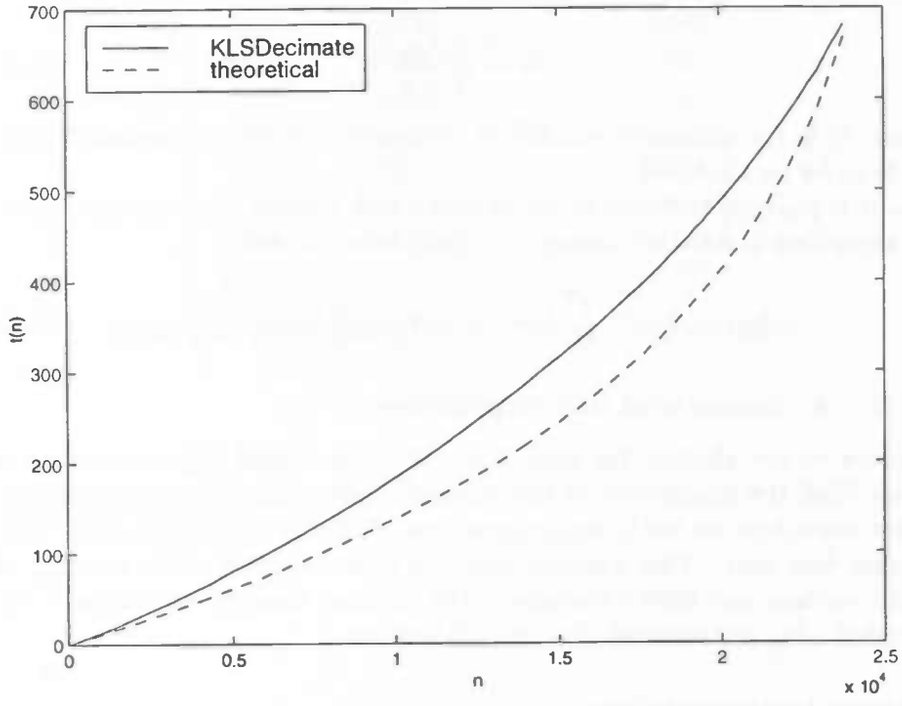


Figure 5.1: *The timing results of the original implementation, using a dataset of 26460 vertices.*

For drawing the theoretical curve $N = 26460$ was used. Furthermore it was normalized against the actual timing. It is the shape of the two curves that matters when a comparison between complexities is done, so the normalization is allowed. We can see that the two curves do not have the same shape. We can also look at the factor between the experimental and theoretical timing values:

$$\frac{t(n)}{t_{[\mathrm{SL96}]}(n)}. \tag{5.5}$$

There is no difference between the complexity of the program and the theoretical complexity when the plot of this formula gives a straight horizontal line. This plot is shown in figure 5.2. Clearly the complexities are not the same.

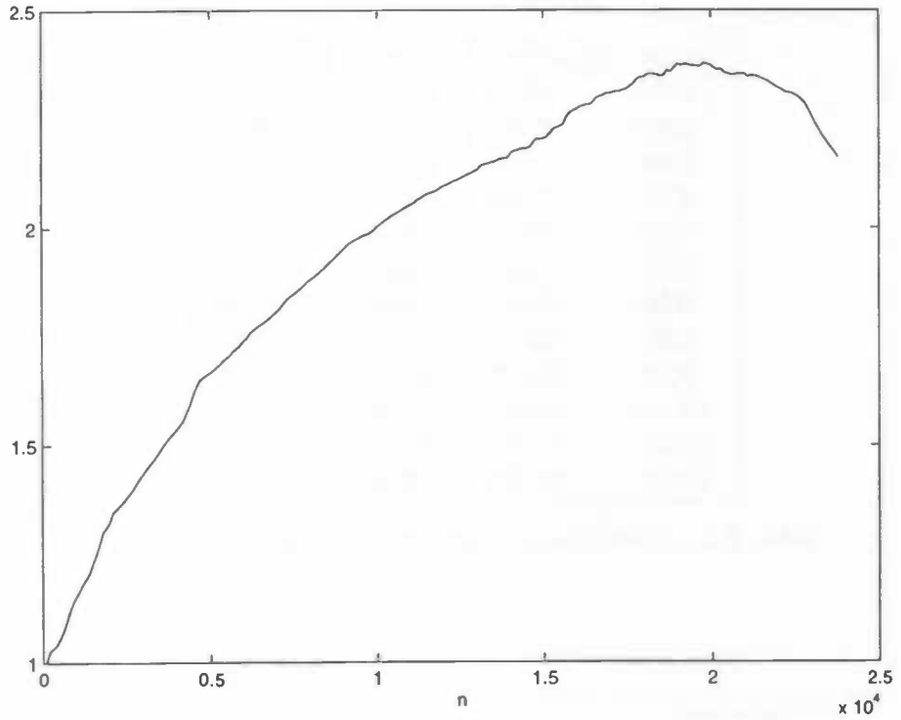


Figure 5.2: *The timing results of* KLSDecimate *divided by the expected timing.*

The presence of the 'bump' in the plot is probably caused by the inefficient correspondence list. This correspondence list grows throughout the running of the program and it is used more often when the number of removed vertices increases. Since this correspondence list is based on linked lists, this might be the bottleneck.

Improved implementation

For the new implementation the same measurements were done. The timing results are shown in table 5.2 and figure 5.3.

Also the factor between the actual and experimental timings according to equation (5.5) is computed again. The results of this computation are shown in figure 5.4.

It seems that this graph is far from a straight horizontal line. However, when we look at the vertical axis, we can see that the graph is in a very small range, close to 1. This means that, though it does not seem to be at first sight, the complexity of the new implementation is quite close to the expected complexity.

When we look at the experimental curve in figure 5.3, we can see that it almost fits the theoretical one, especially in the beginning. Later they start

Vertex	Time(ms)	Vertex	Time(ms)
1000	16854	13000	279274
2000	35317	14000	309238
3000	53975	15000	342233
4000	72226	16000	385963
5000	93770	17000	439695
6000	113386	18000	497626
7000	133268	19000	575653
8000	154431	20000	655557
9000	176839	21000	741182
10000	199470	22000	834056
11000	223875	23000	945199
12000	249870	23800	1054704

Table 5.2: *Experimental results of the new implementation.*

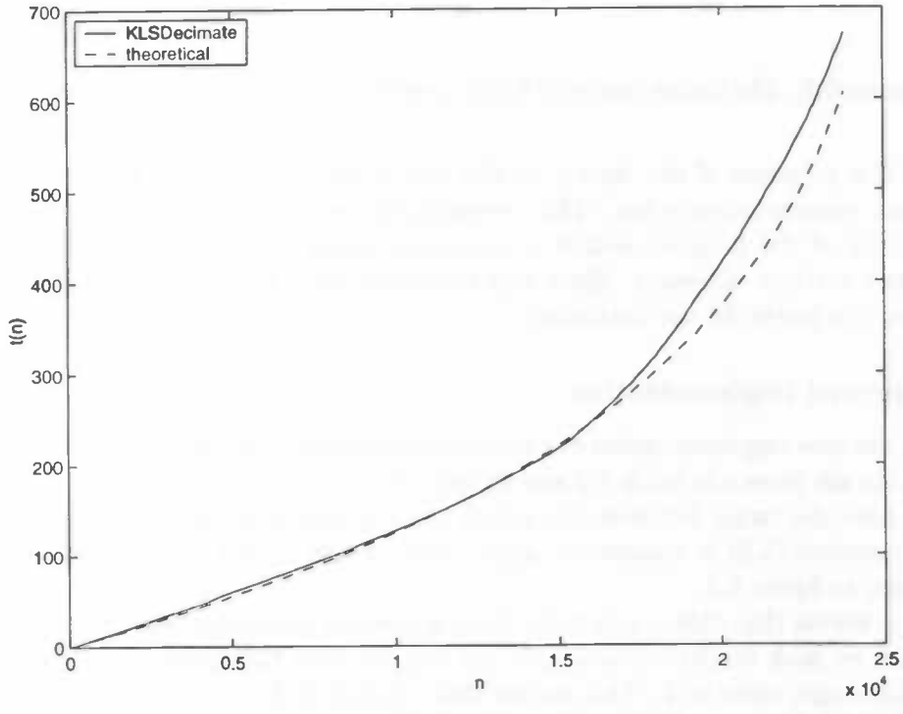


Figure 5.3: *The timing results of the improved implementation, using a dataset of 26460 vertices.*

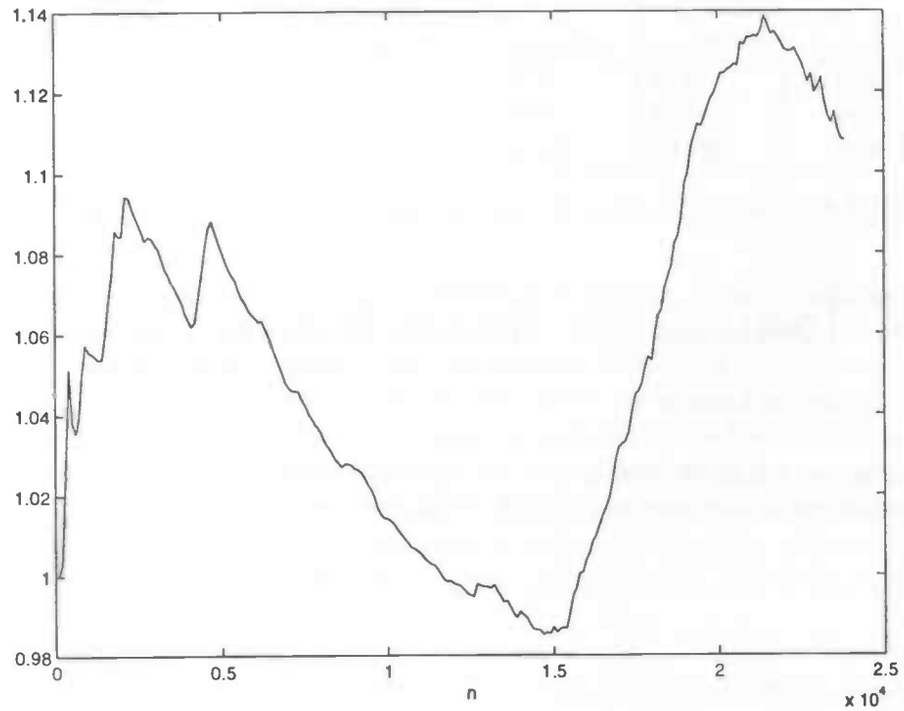


Figure 5.4: *The timing results of the new implementation of* KLSDecimate *divided by the expected timing.*

to run a bit out of pace, as we can also see by the 'bump' in figure 5.4. But this deviation is only a factor 0.14, so this seems acceptable.
It can now be concluded that the complexity has improved much compared to the complexity of the original implementation. Moreover, the deviation from the theoretical complexity is really small. This means that there are no large datastructures in the program which influence the complexity in a negative way. However, there might still be other inefficiencies that influence the complexity now, such as inefficient loops or recursions, or other small problems that can be solved in a more efficient way.

5.2 Measurements

To obtain an indication of how fast the program was before and after the improvements were made, some timings were done for a few different datasets. Table 5.3 shows the running time of the original program and the new version with all the improvements. The program reduced the datasets to 10% of the original number of vertices.

From table 5.3 we can see a really large gain in speed from the original to the improved implementation. The speedup is by a factor of about 2 for

dataset	#vertices	#triangles	original implementation	improved implementation
teapot	1976	3751	2:05.51	0:51.30
fohe	4005	8038	6:27.39	3:19.22
fran	26460	52260	1:07:15.99	17:42.55

Table 5.3: *The running times of both the original and improved implementation.*

the smaller datasets, say less than 10000 vertices, to a factor of 4 for larger datasets. This is quite a good speedup, but the program is still not really fast, considering that these datasets are not so large. We would like to use decimation on datasets for which visualization is really a problem, and that is not the case for the datasets in table 5.3. Only for the 'fran' dataset decimation might be useful, but the size of datasets for which decimation becomes really meaningful is about 50000 vertices or larger.

Chapter 6

A multiresolution approach

The visualization of scenarios with a variety of large models in the context of virtual reality is quite a problem. The visualization cannot be realized in real-time without special techniques to reduce the number of triangles that have to be rendered. For this purpose multiresolution modeling can be used.
Multiresolution modeling means maintaining objects at different levels of detail or approximation, where the level of detail may be different in distinct areas of the object. And with respect to visualization it means to render the model at any given moment with a minimal number of triangles or level of detail. This minimal level of detail should be sufficient to produce an image of reasonable quality. Consequently, when rendering models at a large distance from the camera position, this approach results in a significant decrease in the number of triangles that have to be rendered.
For visualization purposes it is necessary to change between different levels of detail in real time. However, real-time simplification with control of the approximation error is not possible, due to computing power restrictions. To overcome this problem a two stage approach can be used:

1. A view-independent multiresolution model (MRM) is generated from an offline simplification process. This model contains all information needed to reconstruct the object at any given level of detail. The computation of this model is expensive and therefore not appropriate for real-time interaction.

2. View-dependent simplified meshes can be extracted at variable resolution from this MRM, and rendered with the desired image quality in real time.

With regard to the MRM itself, a few desired features can be mentioned. A good multiresolution model should support the following features:

- *Progressive transmission*: When a mesh is transmitted over a communication line, one would like to show a progressively improving ap-

proximation of the model as data is received incrementally.

- *Automatic viewing parameter-dependent approximation*: The level of detail of an object should automatically be adjusted after changes of the viewing parameters. That is, for example, when the camera position changes, or the object moves away from the camera.

- *Error tolerance editing*: Sometimes it is useful to interactively define different error tolerances in different areas of the model. In such a way the user is able to get better approximations in the areas of interest of the model.

R. Klein and J. Krämer developed a multiresolution model, as discussed in [KK97]. This MRM can be generated with the decimation algorithm developed by Klein, Liebich and Straßer [KLS96], as described in chapter 3. In this chapter the MRM will be discussed, and also a few suggestions will be made for adapting KLSDecimate, so that it is able to generate the MRM.

6.1 Structure of the multiresolution model

The multiresolution model described in the following is based on a multitriangulation model first presented by Puppo [Pup96], which was in the context of terrain visualization.
The datastructure of the multiresolution model consists of the following three data sets:

1. List of all vertices,
2. List of all triangles,
3. Set of all fragments.

The set of fragments represents the whole multitriangulation. A fragment (see figure 6.1) can be seen as the representation of a remaining hole in the simplification process (see chapter 3), after a vertex was removed. It consists of two major parts:

- The *floor* of the fragment is a representation of the hole after retriangulation.
- The *ceiling* of the fragment is a representation of the hole before the retriangulation, that is, before the vertex removal has taken place.

Thus, each fragment stores the indices of the triangles contained in its floor and its ceiling. Furthermore, it stores the global one-sided Hausdorff distance between the mesh without the removed vertex and the original mesh. This is necessary for extracting a mesh with a certain approximation error.

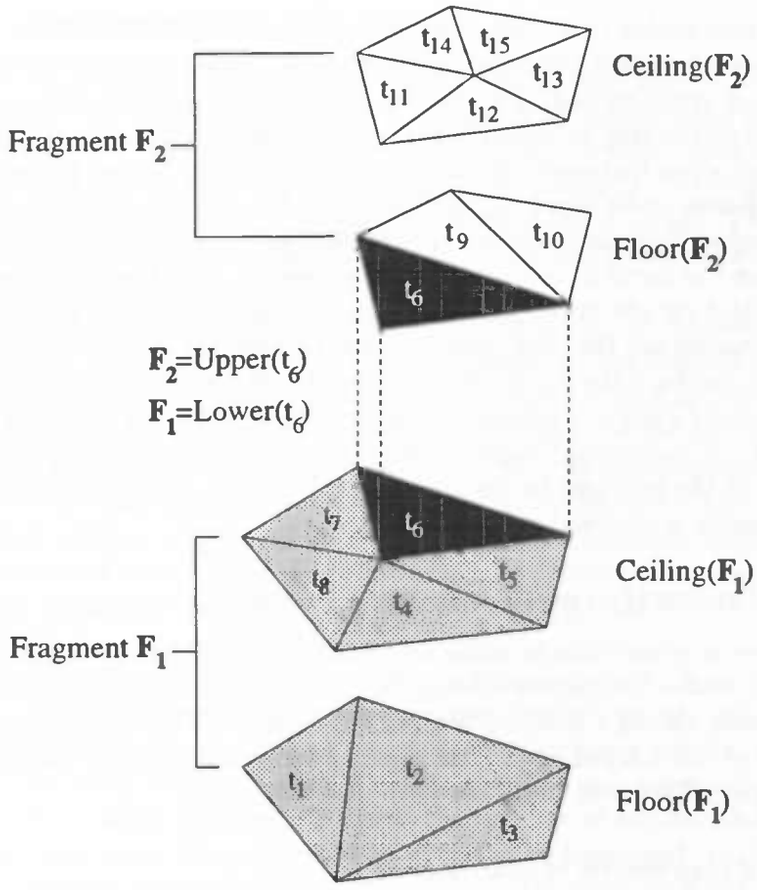


Figure 6.1: *A fragment contains ID's of the triangles in its floor and its ceiling. Each triangle stores two pointers to the fragments containing that triangle.*

For each triangle two pointers to the two fragments containing it have to be stored. These two fragments are called the *upper fragment*, which contains the concerning triangle in its floor, and the *lower fragment*, which contains the triangle in its ceiling (see also figure 6.1).

6.1.1 The extraction algorithm

The algorithm to extract a certain approximation described in [Pup96] always starts from the fragments of the coarsest level of detail. Next these fragments are iteratively replaced by upper fragments in a breadth-first order until all triangles T of the current fragment fulfill a user-specified Boolean condition $c(T)$ or until the finest level of detail is reached.
In practice it is necessary to coarsen the mesh at certain locations and refine it at other locations. An example is the change of the camera position during the visualization process. Klein and Krämer implemented a new al-

gorithm that is able to do this. This algorithm needs a function $c(\Delta)$, which determines whether the approximation error of a triangle stored in its upper fragment is small enough, i.e. whether the triangle approximates a neighbourhood of the original triangle mesh with sufficient accuracy. During the refinement step, fragments are iteratively replaced by upper fragments in a breadth-first order until all triangles T of the current fragment fulfill a user-specified Boolean condition $c(T)$ or the finest level of detail is reached. To coarsen the mesh in certain regions, we need to keep track of those fragments that have the triangles of the current triangle mesh in their ceiling. These triangles are the only ones that can be removed from the mesh. The algorithm checks if the condition $c(\Delta)$ is still true for all triangles contained in the floor of such a fragment. If the condition holds the triangles in the ceiling of the concerning fragment are removed from the current mesh and replaced by the triangles in the floor of the fragment. The queue containing the fragments is updated accordingly.

6.1.2 Interactive error tolerance editing

Sometimes it is desirable to refine or coarsen the mesh *interactively* in some areas. To realize interactive editing the user can mark an area of triangles of the mesh through picking, and assign smaller error tolerances on the triangles of the marked area. The mesh is then automatically refined until all triangles of the area fulfill the new error tolerances.

6.1.3 Progressive transmission

The proposed MRM-model built upon the mesh simplification algorithm also supports progressive transmission. During the simplification process vertices are iteratively removed. This removal causes an increasing approximation error. For progressive transmission, the coarsest mesh is transmitted first. This mesh is the bottom of the MRM. After that, the vertices with their accompanying fragments are transmitted in the reverse order of their removal. Thus, when the bottom of the mesh is transmitted, the receiving side can immediately start to refine the bottom mesh. So, the receiving side gets better and better approximations during the transmission process.

6.2 Adapting KLSDecimate to support the MRM

The adaption of KLSDecimate to generate a MRM during the decimation process is quite straightforward. In the following a few suggestions for the adaption are made.

6.2.1 Datastructures

First of all, three datastructures like the ones mentioned in section 6.1 must be added to the implementation. For the list of all vertices and the list of all triangles the `vtkPolyData` object in which the original mesh is stored, can be used. In the current implementation the data in this object never changes, since a copy of it is made at the start of the program. The simplification process takes place on this copy. The original is only referenced but not changed through the running of the program.
When this datastructure is used it already has a part of the needed data stored in it. It contains all vertices and triangles of the original mesh. In the process of building the MRM no vertices are added to it because no new vertices are created during the simplification process. So the list of vertices is already complete. During the simplification process new triangles are created, so only these triangles have to be added to the datastructure. Keep in mind that when a new triangle is created during the simplification process, its ID is reused within the copy of the original `vtkPolyData` object. In our intended datastructure, a new triangle with a *new* ID has to be *added*, and it should not replace another triangle.
For the set of fragments another datastructure has to be chosen. Within VTK no datastructure is available which resembles the set of fragments described in section 6.1. For the set structure itself an array can be used. The size of this array is the number of fragments that will eventually be created. This number of fragments equals the number of vertices that will be removed. An upper bound for this number can be computed. In this array S the fragments are stored. They are added in the order in which they are created. That is, the first fragment that is created is stored at $S[0]$, the next fragment at $S[1]$, and so on. A fragment can be given an 'ID', which can just be the index into S, where the fragment is stored. Each fragment contains two lists, the *ceiling* and the *floor*. These lists can be linked lists, since most of the times they do not grow beyond a size of about 5. In that case linear searching is not a problem. Furthermore, linked lists are dynamic datastructures, and since we do not know the exact number of triangles that have to be stored in it, a dynamic datastructure is quite convenient. In each element of the lists *ceiling* and *floor* a triangle ID and its error (see section 6.1) are stored. The datastructure for storing the fragments as described here looks as shown in figure 6.2.

Now we only have to store the *upper* and *lower fragment* of each triangle. This can also be stored in an array T. The size of T is equal to the number of triangles in the original mesh plus the number of new triangles created during the simplification process. An upper bound for this number can be computed. In each element of this array the ID's (index into S) of the upper and lower fragment are stored, so each array element contains two ID's. The index i into T is equal to the ID of the concerning triangle in the

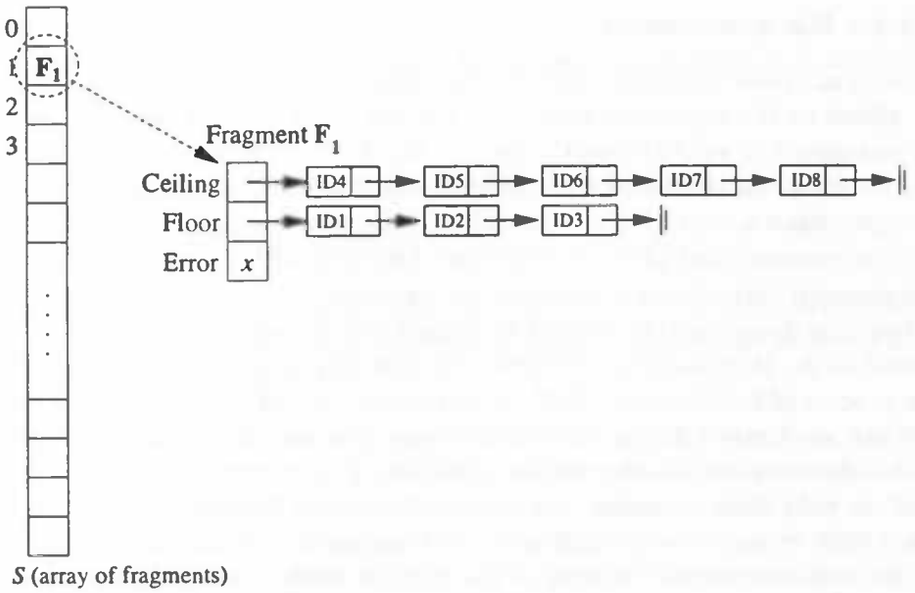


Figure 6.2: *The datastructure for storing the fragments. The elements of the array S on the left are the fragments. One of these array elements (the fragment F_1) is magnified on the right.*

`vtkPolyData` object. The ID can be used as an index because the ID's of the original triangles run from 0 to $n-1$, where n is the total number of triangles in the original mesh, and the newly added triangles are assigned an ID which follows these numbers. That is, the first new triangle is assigned ID n, the next one is assigned ID $n+1$, the next $n+2$, and so on. So the index never exceeds the array bounds of T if the size of T is well chosen.

6.2.2 Algorithms

Klein and Krämer also published the C++-code of the algorithms to extract, refine and coarsen the mesh. Our implementation can be based on this code. Of course it has to be adapted to work with our datastructures, but the basic code stays the same. The C++-code can be downloaded from *http://www.gris.uni-tuebingen.de/people/staff/reinhard/publications.html.*

Chapter 7

Summary and conclusions

7.1 Improvements and conclusions

In [RvdZ99] R. van der Zee suggested some improvements to make KLSDecimate run faster. These improvements were analyzed or implemented (see also chapter 4):

- Optimizing the distance calculation - R. van der Zee suggested the optimizations proposed by Klein et. al., discussed in [KK97]. These were considered, but not implemented. Instead another optimization was done: keeping an administration of already calculated distances.
- Replacing the correspondence list - This used to be a doubly linked list. Now it is implemented using an array, which is simple but fast.

In addition to these improvements, also the sorted vertex list (the priority queue of vertices to be removed) was replaced for simplicity reasons. The old implementation, made by R. van der Zee, was already quite efficient, but really invalved and the code therefore hard to read. Furthermore, the old implementation did not have a guaranteed time complexity as a result of the use of a hash table. The new priority queue is implemented using arrays, which gives readable and short code, and has a guaranteed time complexity. Also a small error was found in the original implementation of the correspondence list, which was eliminated when the correspondence list was replaced. As a consequence, the results shown in appendix A look slightly different from the results shown in [NA98].
In the end, the implementation now has a better time complexity and consequently runs a lot faster (by a factor 2 to 4). Furthermore the code of the datastructures is more readable. Still, the program is not really fast. For instance, a running time of about 17 minutes for the 'fran' dataset of 'only' 26460 vertices is not really fast, although it is a major speedup in comparison with the running time of the previous implementation on the same dataset (more than one hour). However, it should now be fast enough

to generate a multiresolution model (MRM) of a dataset of a similar size as the 'fran' dataset in reasonable time when the implementation is adapted to do this.

7.2 Suggested additions and improvements

There are some features that could be added to the program:

- MRM support - The program should be adapted so that it can generate a MRM. The MRM and suggestions for adaption of the program are discussed in chapter 6.
- Wider file format support - The only kind of data files that can be read at the moment are VTK-files. Furthermore, these files should only contain polygonal data with vertices and triangles. Routines should be added to automatically convert the input when it is not of the described type. For files other than VTK-files there are a lot of filereaders available within VTK to convert these.
- Graphical user interface - At the moment the program does not have much of a user interface. The reduction percentage or maximum error are defined within the code of the program, so the user can not set this value without recompiling the program. Also the names of the input en output files can not be set by the user. A user interface should be added, preferably a graphical one, in which the names of the input and output files, and the reduction percentage or maximum error can be set.

For the first item it should be considered reading [KG98], in which compression of the MRM for efficient data storage is discussed.
Also suggestions for further improvement of the performance can be given:

- Implementation on another machine - The current implementation runs on a SGI Onyx Reality Engine 2. This machine is optimized for visualization purposes and has hardware that accelerates the visualization process. For the decimation process normal (processor) computing power is needed instead of visualization power. So to obtain a better running time, implementing it on another machine should be considered. Adapting the program to run on another platform will probably not be a lot of work.
- Code optimization - The program should be fully reviewed to find the last things that are inefficient, such as inefficient loops or other things that are done in a roundabout way. Especially the routine `Distance2BetweenPointAndTriangle`, the main distance calculation, should be looked at closely. It still is the slowest routine of the whole

program by far, as can be seen from the profiling results shown in appendix B. There might still be code in it that does not run as efficient as it should. Also other routines that are high in the profiling list should be looked at.

- Stop criterion for subcases of the distance calculation - The criterion to stop the distance calculation in the subcases 2 and 3 (see 3.3.2) should be looked at. When many of these subcases occur the running times really are not acceptable anymore (hours and hours). So definitely there is something wrong with that. Probably the criterion to stop subdividing the triangles should be sharpened a lot. The criterion could be made dependent on the geometrical size of the dataset.

Bibliography

[CLR90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to algorithms*, The MIT Press, 1990. ISBN 0-262-03141-8.

[KG98] R. Klein, S. Gumhold, "Data Compression of Multiresolution Surfaces", *Visualization in Scientific Computing '98*, pp. 13-24, 1998.

[KK97] R. Klein, J. Krämer, "Building Multiresolution Models for Fast Interactive Visualization". *Proceedings of the SCCG '97*, 1997.

[KLS96] R. Klein, G. Liebich, W. Straßer, "Mesh Reduction with Error Control". *IEEE Visualization '96*, pp. 311-318, 1996.

[NA98] A. Noord, R.M. Aten, "Geometric Simplification Algorithms for Surfaces", *Master's thesis, Dept. of Mathematics and Computing Science, University of Groningen*, 1998.

[PS97] E. Puppo, R. Scopigno, "Simplification, LOD and Multiresolution Principles and Applications". *Proceedings Eurographics '97*, Vol. 16, nr. 3, 1997.

[Pup96] E. Puppo, "Variable Resolution of Terrain Surfaces". *Proceedings Eighth Canadian Conference on Computational Geometry*, August 1996.

[RvdZ99] R. van der Zee, "Enhancing KLSDecimate", *Master's thesis, Dept. of Mathematics and Computing Science, University of Groningen*, 1999.

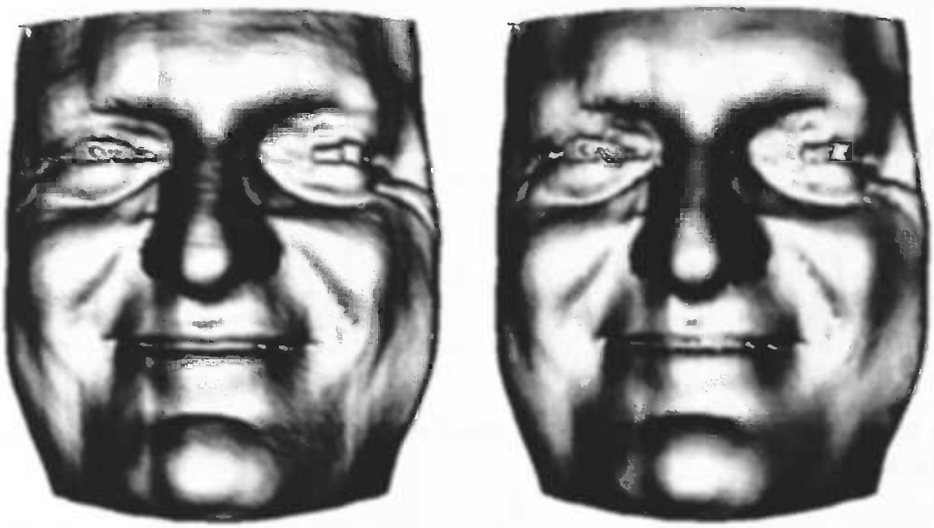
[SL96] M. Soucy, D. Laurendeau, "Multiresolution Surface Modeling Based on Hierarchical Triangulation". *Computer Vision and Image Understanding '96*, Vol. 63, No. 1, January, pp 1-14, 1996.

[VTK98] W.J. Schroeder, K.W. Martin, W.E. Lorensen, *The Visualization Toolkit*, 2nd edition, Prentice Hall, 1998. ISBN 0-13-954694-4.

Appendix A

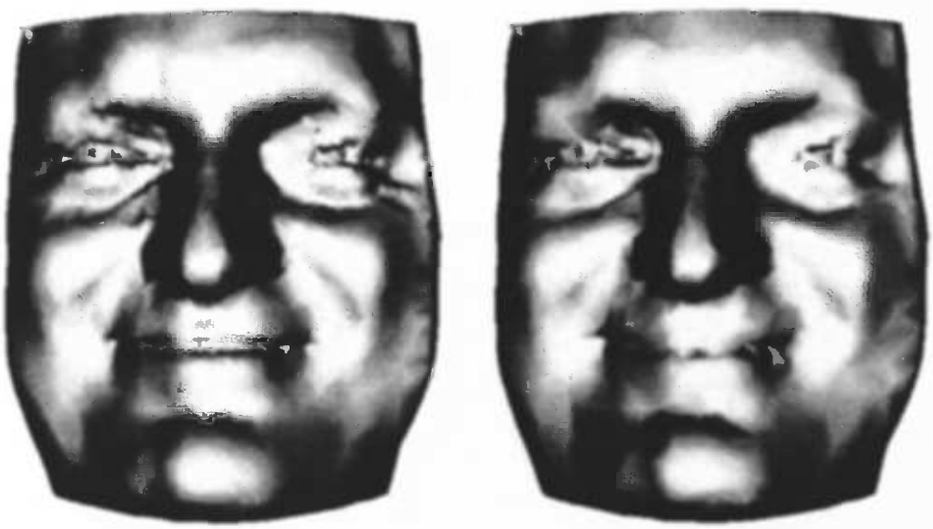
Pictures

In the following pages examples are given of the output datasets of KLSDecimate.



(a) The original dataset with 26460 vertices and 52260 triangles.

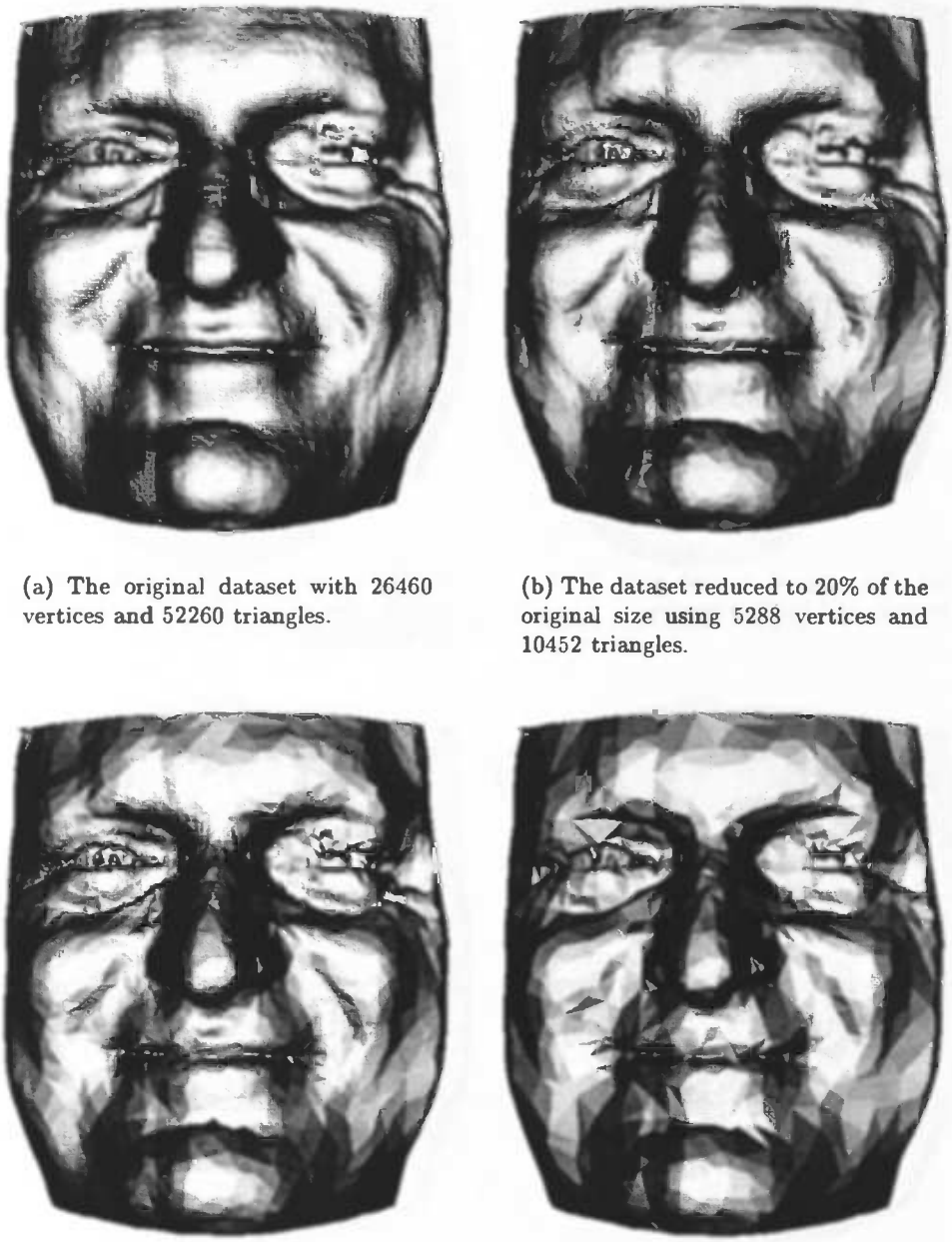
(b) The dataset reduced to 20% of the original size using 5288 vertices and 10452 triangles.



(c) The dataset reduced to 10% of the original size using 2652 vertices and 5226 triangles.

(d) The dataset reduced to 5% of the original size using 1339 vertices and 2612 triangles.

Figure A.1: *Results of decimating the 'fran' dataset with various reduction rates, represented using Gouraud shading.*



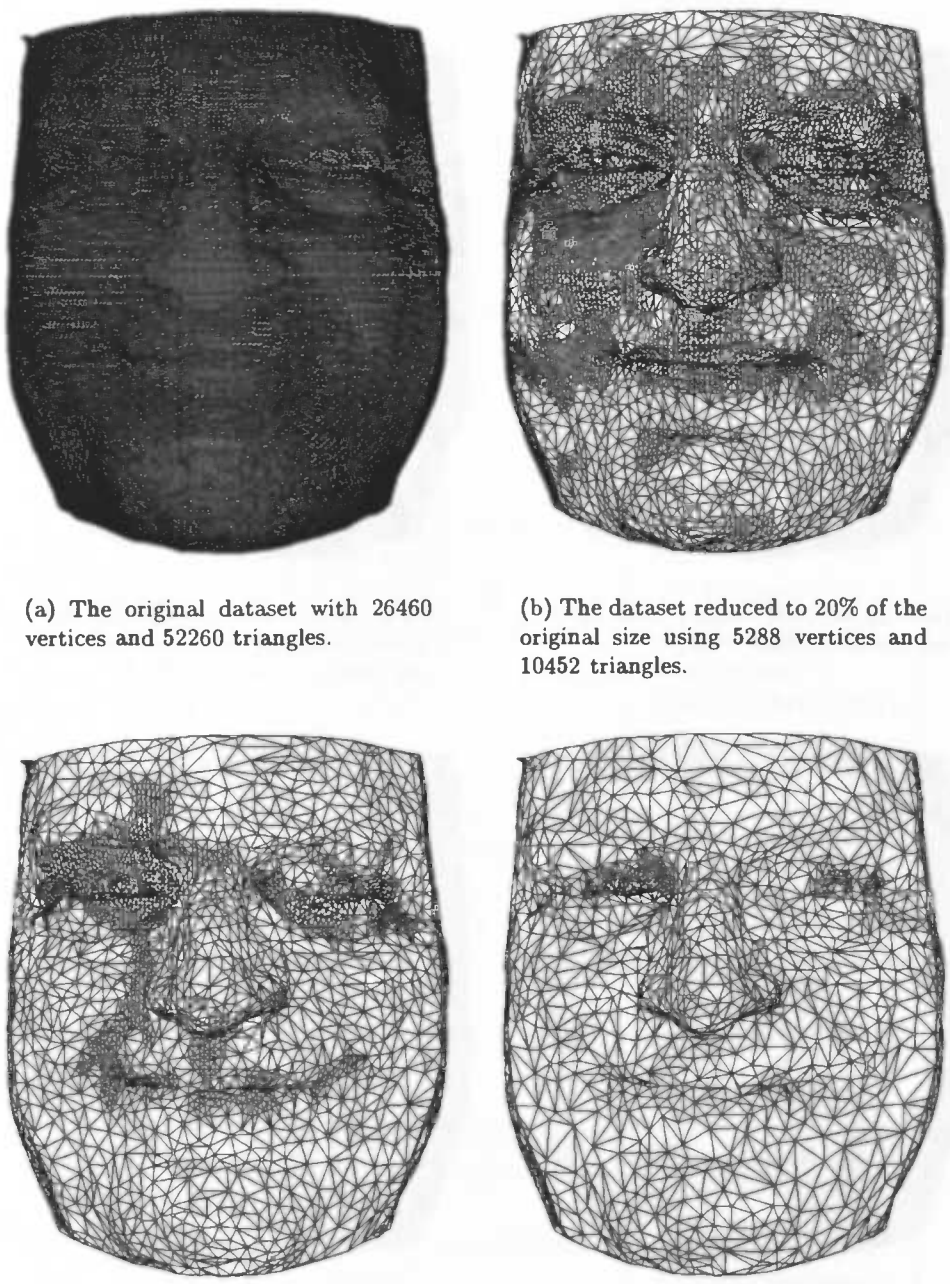
(a) The original dataset with 26460 vertices and 52260 triangles.

(b) The dataset reduced to 20% of the original size using 5288 vertices and 10452 triangles.

(c) The dataset reduced to 10% of the original size using 2652 vertices and 5226 triangles.

(d) The dataset reduced to 5% of the original size using 1339 vertices and 2612 triangles.

Figure A.2: *Results of decimating the 'fran' dataset with various reduction rates, represented using flat shading.*



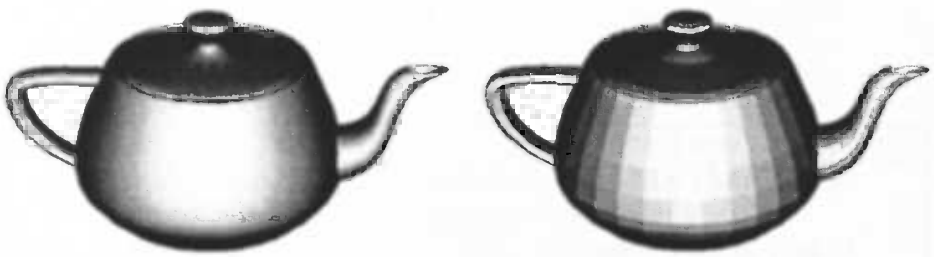
(a) The original dataset with 26460 vertices and 52260 triangles.

(b) The dataset reduced to 20% of the original size using 5288 vertices and 10452 triangles.

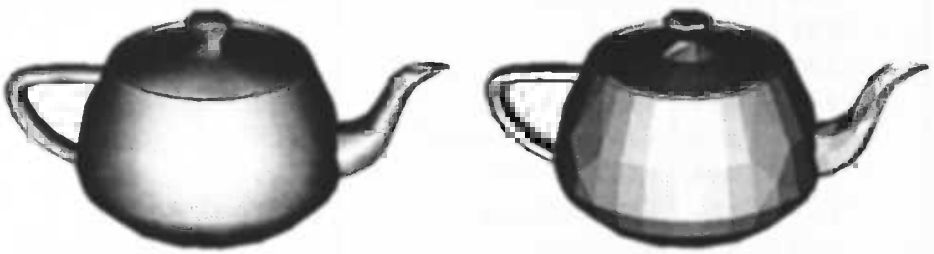
(c) The dataset reduced to 10% of the original size using 2652 vertices and 5226 triangles.

(d) The dataset reduced to 5% of the original size using 1339 vertices and 2612 triangles.

Figure A.3: *Results of decimating the 'fran' dataset with various reduction rates, represented using wireframe.*



(a) The original dataset with 1976 vertices and 3751 triangles.



(b) The dataset reduced to 20% of the original size using 414 vertices and 751 triangles.

Figure A.4: *Result of decimating the 'teapot' dataset, using Gouraud shading on the left and flat shading on the right.*

Appendix B

Profiling results

cycles(%)	secs	procedure	(file)
29.55	44.53	Distance2Between-PointAndTriangle	(KLSDecimate.cc)
14.08	21.22	PointInTriangle	(vtkTriangle.cxx)
13.14	19.81	CalculateSpecialCase1	(KLSDecimate.cc)
8.76	13.21	__sqrt	(libm.so:sqrt.s)
7.50	11.31	GetTuple	(vtkFloatArray.cxx)
6.51	9.81	Cross	(vtkMath.cxx)
3.52	5.30	GetPoint	(vtkPointSet.h)
3.41	5.14	CanSplitLoop	(KLSDecimate.cc)
2.15	3.25	CalculatePotentialError	(KLSDecimate.cc)
1.91	2.88	IsInTriArray	(KLSDecimate.cc)
1.77	2.67	GetCellPoints	(vtkPolyData.cxx)
1.63	2.45	SplitLoop	(KLSDecimate.cc)
0.70	1.06	Triangulate	(KLSDecimate.cc)
0.64	0.96	realfree	(libc.so.1:malloc.c)
0.57	0.85	__malloc	(libc.so.1:malloc.c)
0.32	0.48	cleanfree	(libc.so.1:malloc.c)
0.29	0.43	GetCellEdgeNeighbors	(vtkPolyData.cxx)
0.23	0.35	CalculateLoopNormal	(KLSDecimate.cc)
0.22	0.33	IsInIntList	(KLSDecimate.cc)
0.19	0.29	CalcOrigToSimplified	(KLSDecimate.cc)
0.18	0.28	_malloc	(libc.so.1:malloc.c)
0.18	0.27	__free	(libc.so.1:malloc.c)
0.18	0.27	_free	(libc.so.1:malloc.c)
0.17	0.26	t_delete	(libc.so.1:malloc.c)
0.16	0.24	t_splay	(libc.so.1:malloc.c)
0.16	0.23	__nw	(libC.so:_new.c++)
0.14	0.22	RemoveVertex	(KLSDecimate.cc)