Faculty of Mathematics and Natural Sciences

Department of Mathematics and Computing Science

Geometric Simplification Algorithms for Surfaces: Enhancing KLSDecimate

René van der Zee

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

January, 1999

RuG

Enhancing KLSDecimate

René van der Zee

January 19, 1999

Contents

Chapter 1

Preface

1.1 Decimation

Throughout the years the size of datasets has increased so much that problems arise for their visualization. One of those problem areas is the ability to render 3D datasets. The complexity of the datasets increased much faster than the performance of the rendering hardware. Datasets of models used for medical or geographical applications contain up to millions of polygons. The datasets are too complex for interactive visualization and manipulation. To manipulate and visualize datasets of this complexity we have to reduce their size.

The current hardware to visualize datasets interactively is able to process big datasets, but is also very expensive. The more affordable hardware, for desktop use in offices and at home, is not able to visualize these datasets interactively. The bandwidth of the current available networks is a limiting factor too. Accessing data stored in huge databases, like CAT-scans made in hospitals, can only be done through local area networks and public telephone lines whose bandwidths are limited.

Although the speed of 3D-hardware graphics engines will probably increase by a large factor in the coming years, the size of datasets will also increase along with the advancement of technology.

Algorithms have been developed to reduce the size of datasets without loosing too much detail. These algorithms are called *decimation* algorithms. In this field, people are searching for algorithms which maintain the quality of the original as much as possible, while reducing the complexity of the dataset. In this report I will discuss a decimation algorithm which reduces the complexity of a dataset which is a representation of a surface, consisting of triangles. For example, the surface of a teapot, a dataset which stores the surface (the height at evenly spaced points) of a terrain, or the surface of a human skull generated by the marching cubes algorithm.

1.2 Previous work

The algorithm discussed in this report is a decimation algorithm by Klein, Liebich and Straßer, as discussed in [KLS96]. The algorithm reduces the complexity of the dataset by removing points from the dataset which satisfy a certain criterion. This algorithm gives better results than several other algorithms by using a special error measure called the *'Hausdorff distance'*. I will discuss the algorithm in chapter 3.

The algorithm has already been implemented by A. Noord and R.M. Aten as they have reported in [NA98]. This implementation however, can be improved at several points. I will discuss some improvements of their implementation, and describe how this changes the implementation. Both changes with respect to speed as well as functionality are presented and implemented.

Chapter 2

Background

2.1 Surfaces

Surfaces as discussed in this report can be considered as *2-manifolds*, that is, 2-dimensional surfaces embedded in a 3-dimensional Euclidean space.

2.1.1 Triangle meshes

A *triangle mesh* is an approximation of a surface consisting of triangular faces which are pasted together along their edges. The triangles consist of edges which are called the *faces* of the triangle. The edges itself also have *faces*, which are the vertices, the endpoints of the edge. These building blocks are called *simplices*, a triangle is a 2-simplex, an edge is a 1-simplex and a vertex is a 0-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 Manifolds and non-manifolds

A surface as defined above is a 2-manifold object, which means that we can travel over the surface in two dimensions. However, there are datasets that do not satisfy the condition for a 2-manifold object. These are called non-manifolds. Two examples of non-manifolds are shown in Fig 2.1.

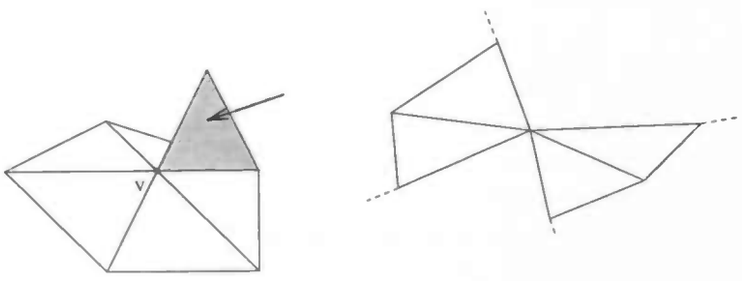

Figure 2.1: Non-manifolds.

2.1.3 Topology

When decimating a triangle mesh 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 transformation D, which means that D 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*. Furthermore the transformation D is a *topological mapping* or homeomorphism and is said to be *topology preserving*. A few examples of topology preserving and non topology preserving transformations are shown in Fig. 2.2.

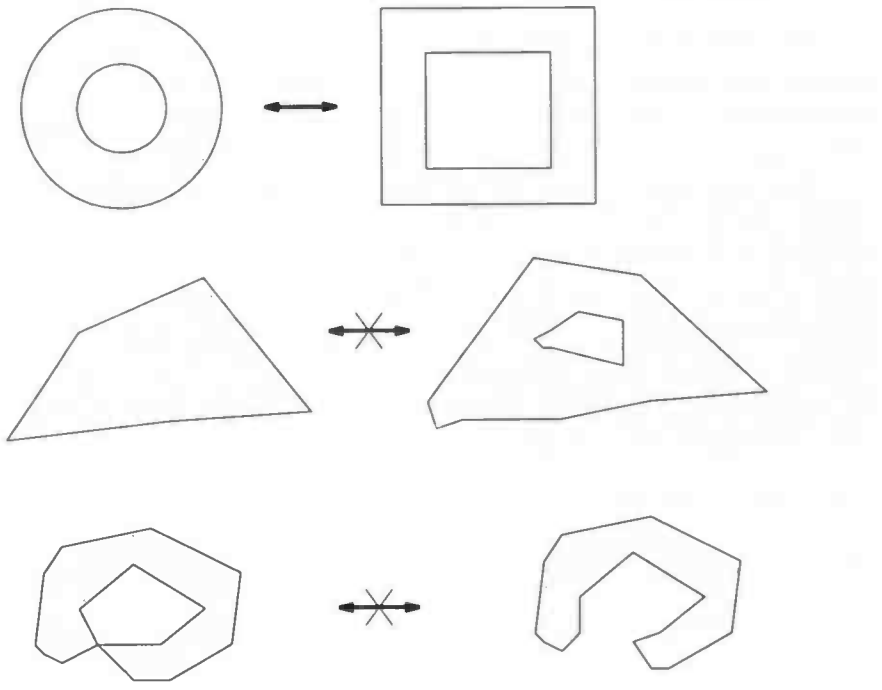

Figure 2.2: Topological equivalence and inequivalence.

2.2 Vertex classification

Schroeder, Martin and Lorensen [SML96] suggest a classification of vertices which I will use to discuss the types of vertices. This classification is very important for the implementation of the algorithm and is also used by Noord and Aten in their implementation.

We can divide the group of vertices in a number of categories. The most basic type of vertex is the *simple vertex*. When v is a *simple vertex*, all the triangles in the loop surrounding v have exactly two neighbouring triangles in that loop. A picture of such a vertex is shown in Fig. 2.3.

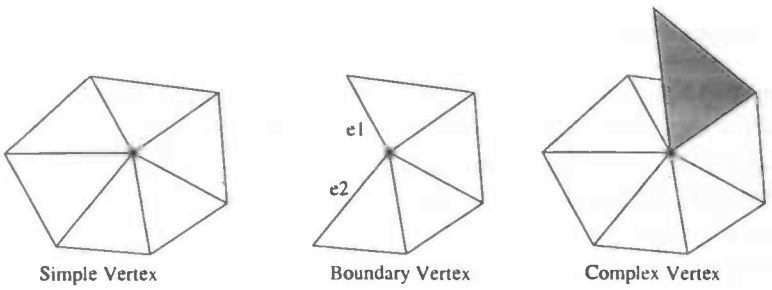

Figure 2.3: Vertex-types.

The second type of vertices are the *border vertices*. A border vertex v has exactly two triangles,which have only one neighbour in the loop of triangles surrounding v. An example of a border vertex is also shown in Fig. 2.3. The edges e_1 and e_2 are part of a border of the triangle mesh.

The third type is called the *complex vertex*, and comprises all the vertices which can not be classified as a simple vertex or a border vertex. These vertices have triangles with more than two neighbours in the loop or are part of more than one border. This is the case when the triangles are connected through the vertex and not through the edges. When such a situation occurs, the triangle mesh is not 2-manifold, but a non-manifold. In an implementation, such a complex vertex should not be removed, otherwise the topology of the triangle mesh is affected.

Chapter 3

The decimation algorithm

3.1 Introduction

The algorithm by Klein, Liebich and Straßer is designed around the Hausdorff distance calculation. The Hausdorff distance gives an error between the original mesh and the decimated mesh. Using this error calculation it is possible to give a meaningful user-defined maximum to the global error between the original and the decimated mesh. According to Klein, Liebich and Straßer this error measure is superior to the use of the L^∞-Norm. The advantages of the algorithm as proposed by Klein, Liebich and Straßer are:

- It guarantees a user defined position dependent approximation error
- It allows to generate a hierarchical geometric representation in a canonical way.
- It automatically preserves sharp edges.

The first and last item have great influence on the results of the algorithm and both are a consequence of the use of the Hausdorff distance.

3.2 The Hausdorff distance

We define the Euclidean distance between a point t and a set S (surface) as

$$d(t, S) = \min_{s \in S} d(t, s)$$

where $d(t, s)$ is the Euclidean distance between two points $t, s \in R^n$

Assume T is the original mesh and S is the decimated mesh. Than the one-sided Hausdorff-distance is defined as:

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

If the one-sided Hausdorff distance $d_E(X,Y)$ is smaller than a certain ϵ then

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

where x and y are points on the surface of the meshes X and Y, respectively.

Because the *one-sided Hausdorff distance* is not symmetric, there are some problems near borders and interior edges. By using the *two-sided Hausdorff distance* in these cases these problems are solved. The two-sided Hausdorff distance $d_H(X,Y)$ is defined as:

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

The two-sided Hausdorff distance is symmetric. If this two-sided distance $d_H(X,Y)$ is zero then X = Y.

If $d_H(X,Y) < \epsilon$ then

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

and because it is, unlike the one-sided Hausdorff distance, symmetric

$$\forall y \in Y \text{ there is an } x \in X \text{ with } d(x,y) < \epsilon$$

3.3 The algorithm

3.3.1 Potential error calculation

The decimation algorithm removes vertices from the mesh as long as the error introduced by the removal is smaller than the maximum user-defined global error. The order in which the vertices are removed is determined by the order in which the vertices are stored in a priority queue. The priority of a vertex is determined by its *potential error*. The priority queue is sorted in ascending order. The potential error of a vertex is the error which would be introduced when that vertex was removed from the triangle mesh. To construct the priority queue, the potential errors of all the vertices need to be known. Therefore the algorithm starts with the calculation of the potential error of the vertices. The Hausdorff distance is used as the error measure throughout the algorithm. Because the potential error of a vertex v is the error (Haussdorf distance) that would be introduced by the removal of v, the Haussdorf distance needs to be calculated between the triangle mesh X and the same triangle mesh with v removed.

$$E_p = d_H(X, X \setminus \{v\})$$

Whenever a vertex v is removed from the triangle mesh, the potential errors of the surrounding vertices may change. A triangle which was previously used in an error calculation might be removed in the retriangulation (See section 3.3.2) process. Therefore the potential errors of the vertices surrounding v have to be updated. To update the potential error, we again have to compute the distance between the original and the mesh decimated so far.

When the algorithm starts it is clear which distance needs to be calculated. As long as none of the neighbouring vertices of a given vertex v have been removed from the original mesh then the distance from the new created mesh to the old mesh can be computed as follows:

Let v be the removed vertex, let $\{t_i\}_{1,\ldots n}$ be the set of removed triangles and let $\{s_j\}_{j=1,\ldots,m}$ be the set of inserted triangles; $m = n-1$ if the removed vertex was a border vertex, otherwise $m = n-2$. Then it is sufficient to compute

$$d_E(\{t_i\}_{i=1,\ldots,n}, \{s_j\}_{j=1,\ldots,m}) = \max_{j=1,\ldots,m}(d(v, \{s_j\}))$$

When there are triangles in the original mesh with vertices that no longer belong to the simplified mesh, thus they do not share a vertex with the newly created mesh, we need a system to determine which distance we need to calculate. For this purpose we need to know the *correspondence*. The correspondence gives for a triangle in the original mesh, the triangles to which the distance has to be calculated.

For each already removed vertex v of the original mesh X, the triangle $y \in Y$ that has the smallest distance to v is stored. Furthermore, for each triangle $y \in Y$ a subset of vertices $X_y \subseteq X$ for which y is the vertex with the smallest distance is stored. This is the correspondence information. This correspondence information has to be updated with every removal of a vertex from the triangle mesh.

3.3.2 Retriangulation

When a vertex v is removed from the mesh the triangles surrounding v are removed. Therefore we have to fill the hole with new triangles. The process of refilling the hole with triangles is called *retriangulation*. The new triangle mesh will not have the same shape as the mesh before the removal of v, because v can not be used as a corner of a triangle anymore.

3.3.3 Distance calculation

Computing the distance from a triangle to a triangle mesh is not as straightforward as might seem. When we would like to know the distance between

a triangle and a mesh, we cannot take the maximum of the distances from the triangle's vertices to the mesh. An illustration of this situation is shown in Fig. 3.1.

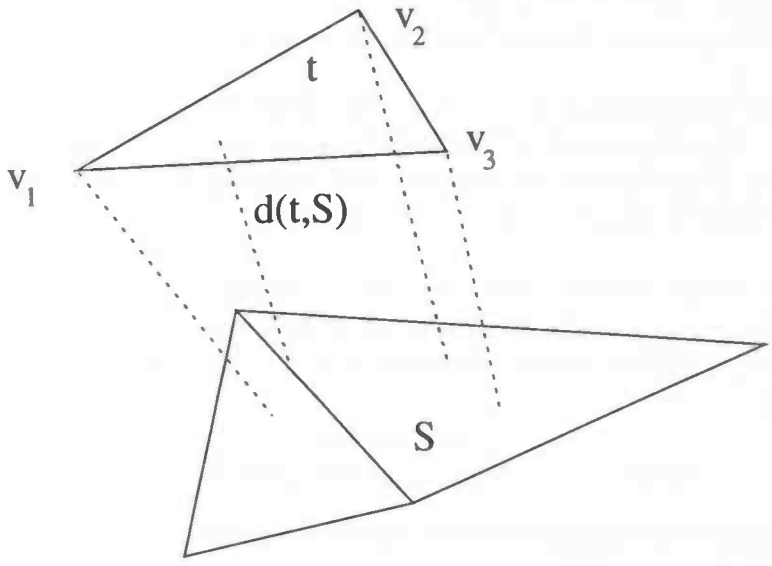

Figure 3.1: $d(t, S) > \max(d(v_1, S), d(v_2, S), d(v_3, S))$.

We consider the following cases:

- 1 Triangle t of the original mesh has no vertex in common with the simplified triangle mesh S.
  - 1.1 All three vertices are nearest to the same triangle $s \in S$.
  - 1.2 The three vertices are nearest to two triangles $s_1, s_2 \in S$ that share an edge
  - 1.3 All other cases.
- 2 Triangle t of the original triangle mesh has one or two vertices in common with the simplified triangle mesh S.

Sub-case 1.1 is solved by simply calculating

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

Subcase 2.2 can be solved by creating a *half-angle plane* between the two triangles s_1 and s_2. By intersecting this half-angle plane with the edges of triangle t having endpoints nearest to different triangles, two points p_1 and p_2 can be found. The distance can now be calculated:

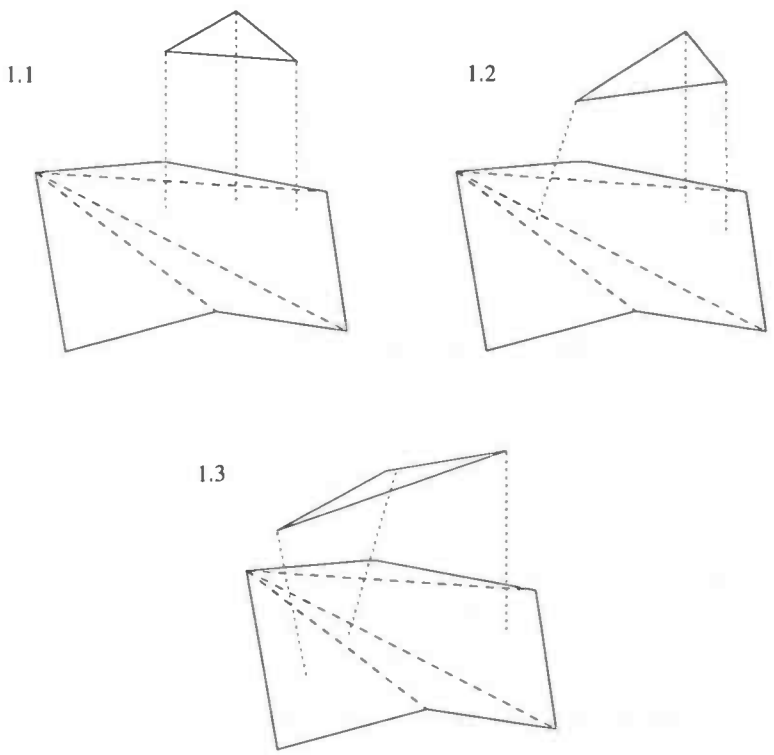

Figure 3.2: A few examples of the subcases.

$$d(t, S) = \max(d(p_1, S), d(p_2, S))$$

The creation of a half-angle plane is illustrated in Fig. 3.3.

Subcase 1.3 is solved by subdividing the triangle t and recursively applying the distance calculation to the three sub-triangles. The subdivision of a triangle is shown in Fig. 3.4

There is a maximum bound to the subdivision. When the longest edge of a subtriangle is smaller than a predefined error tolerance ϵ the subdivision terminates. The distance is calculated as

$$d(t, S) = \max(d(t_{sub1}, S), d(t_{sub2}, S), d(t_{sub3}))$$

In case 2 the upper bound of the maximum distance is also computed using the half-angle plane. By subdivision the problem is reduced to subcases of case 1.

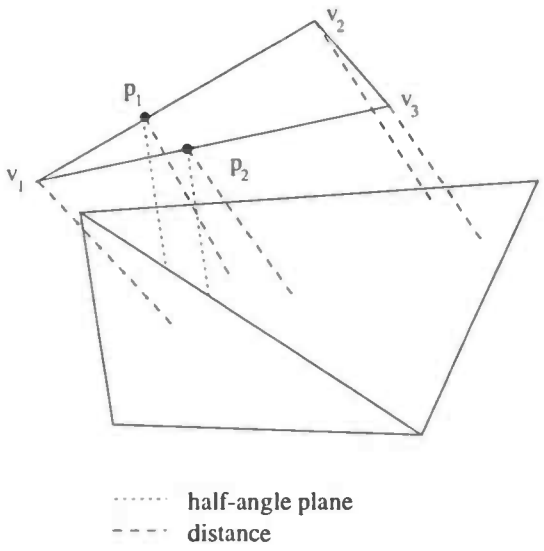

Figure 3.3: An example of a half-angle plane.

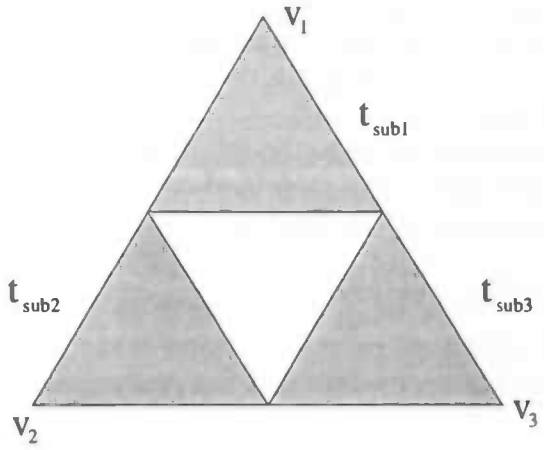

Figure 3.4: The subdivision of a triangle, as required in subcase 1-3.

Chapter 4

Implementation issues

4.1 Introduction

The algorithm of Klein, Liebich and Straßer has been implemented by A. Noord and R.M. Aten [NA98] using the VTK toolkit. The result was a working decimation routine, which lacks some desired features and speed. It was their goal to build a working version which could be used to test the algorithm presented by Klein, Liebich and Straßer, not a full-featured one. In their report they conclude that the algorithm gives a good result and it is a decimation algorithm that has a lot of potential. Furthermore, because there is only one parameter which needs to be set by the user, the program is easy to use. The parameter that can be changed by the user is the maximum error between the original and decimated surface. No fine-tuning is needed to get the optimal decimation from the implementation.

4.2 Data structures

There are two data structures used in the program that are important to the algorithm, the Sorted Vertex List and the Correspondence List. The Sorted Vertex List serves as the priority queue with *potential errors* (See Ch. 3) needed for the main loop to determine which vertex should be removed first. The Correspondence List is used to store the correspondences between triangles and vertices, as described earlier.

Because the implementation was intended to be a working version rather than a full-featured one, the data structures in the implementation are not optimized for speed, but rather for simplicity. The authors used double linked lists for the implementation. Double linked lists work very well for small datasets, say, less than a thousand vertices, but performance quickly slows down for bigger datasets. This is because the complexity of accessing an element in a linked list is inferior to other data structures better suited for the job. As I will show later, it is not trivial to choose a certain data

structure as a replacement, because the program accesses the data structures in different ways.

4.3 Triangulation

4.3.1 Recursive loop splitting

The triangulation routine used in KLSDecimate uses *recursive loop splitting* to triangulate the hole introduced by the removal of a vertex. This algorithm divides the loop which has to be triangulated in two loops along a split line between two non-neighbouring vertices. This process is then recursively applied to the two loops until the loops contain only three vertices, which then form a triangle. It is possible to split a loop along more than one split line, therefore the best possibility is selected. A split plane (through the split line) is used to check that the two newly created loops do not overlap. The selection of the split line is done by calculating the *aspect ratio*. The aspect ratio is the minimal distance of the loop vertices to the split plane divided by the length of the split line. The split line with the smallest aspect ratio is chosen. An extra check is added to prevent the collapse of structures in an already existing triangle. When such a situation occurs, it would mean a change in topology. When such a situation is found, the triangulation is stopped and the vertex is not removed. When using this check topology changes are avoided.

4.3.2 Other triangulation algorithms

In the article by Klein, Liebich and Straßer the use of a *constrained Delaunay* triangulation is suggested. The use of such an algorithm is however not trivial, because it is an algorithm that calculates the best triangulation in only two dimensions. Although a surface is a two dimensional object, we still have three-dimensional coordinates. If we would like to use a 2-D algorithm like Delaunay, we would need to map the 3-D coordinates onto 2-D parametric space. We also need to make sure that the triangles do not overlap in 2-D space. These problems do not arise with an algorithm like recursive loop splitting. The best 2-dimensional triangulation is however not always the best triangulation in 3-D, because information is lost in the process of mapping 3-D coordinates to 2-D coordinates. In Ch. 8, I will discuss the matter of choosing a triangulation algorithm in more detail.

4.4 The VTK toolkit

The existing implementation was based on the VTK toolkit, more exactly version 1.3 of the toolkit. The toolkit has some data structures which were used in the implementation. On these data structures a huge amount of

operations have been defined. The toolkit is written in C++, a language based on objects and classes. For this reason the implementation was also written in C++. Some of the classes from VTK used by the implementation are:

- vtkPolyData – A data structure which is able to store polygons and its vertices. Highly suited to store/manipulate triangle meshes.
- vtkIdList – A data structure used for storing identifiers like vertex-identifiers.
- vtkTriangle – A triangle
- vtkPolyDataReader/Writer – Objects to stream polygon data from/to a file

vtkPolyData is the most important data structure used by the implementation. Every element in a vtkPolyData is called a *cell*, and all the relations between cells are stored. This made it really interesting to use, because these relations are needed by the program. For example, when a vertex v is removed, then we need to know the triangles which surrounded v. We can then use the

```
void GetPointCells(int ptId, vtkIdList& cellIds);
```

method to get all the cells surrounding a certain vertex (point) with identifier ptId from a vtkPolyData object. Another method is the

```
void GetCellPoints(int cellId, vtkIdList& ptIds);
```

method. This method returns all the identifiers of the points/vertices used by a certain cell.

4.4.1 VTK 2.1

One of the things done first was to make the implementation work with version 2.1 of the VTK toolkit. With version 2.0 came a new naming convention, and thus a lot of names of classes and methods changed. At the moment of writing even a beta of version 2.2 is already available.

Not only the names changed, a lot of bugs were fixed and there were a few functional changes. These small changes in the new version broke a few routines in KLSDecimate. It took some time to find the exact locations of these problems and fix them.

Chapter 5

Implementation for non-manifolds

5.1 Introduction

The implementation as written by A. Noord and R.M. Aten is only suited to take manifold meshes as an input. They did not take complex vertices into account. By adding checks for these complex vertices it was possible for the program to take non-manifolds as an input. They use one routine in their implementation to determine the type of the vertex and the vertices surrounding the vertex. Therefore, apart from some other small changes, only modifications in this routine had to be made to make the implementation work with complex vertices.

5.2 Implementation

The routine that classifies the vertices is

```
int CreateLoop(vtkPolyData *pd, int id, int &numpts, vtkIdList &loop)
```

This routine creates a loop of vertices around a vertex with identifier *id* and in the process it determines whether it is a simple or border vertex.

This routine was modified so it could also determine whether it was a complex vertex or not. Now the return values of *CreateLoop* are:

- KLS_SIMPLE_VERTEX
- KLS_BORDER_VERTEX
- KLS_COMPLEX_VERTEX

The routine walks over the vertices surrounding *id*, determining the next vertex nv by testing whether (id, nv) is a edge. As long as such an edge is

part of two triangles, *id* is a simple vertex. This number of triangles minus 1 is called 'neighbours' in the implementation. The main loop in the implementation contains:

```
while (nv != start && neighbours == 1) {
...
}
```

When more than 1 neighbour is found, thus the edge (id, nv) is part of 3 or more triangles, a situation as seen in Fig. 5.1 is detected. Now vertex *id* is also classified as a complex vertex and the new version of CreateLoop returns KLS_COMPLEX_VERTEX

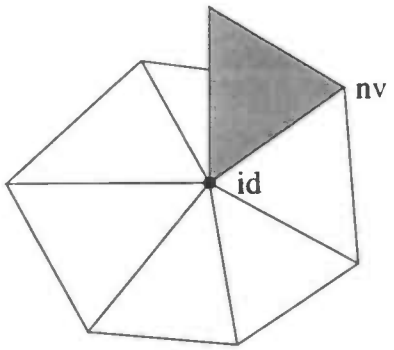

Figure 5.1: An example of a complex vertex.

When the number of neighbours is zero, a border vertex is found, unless there are more triangles to come, for example as shown in figure 5.2

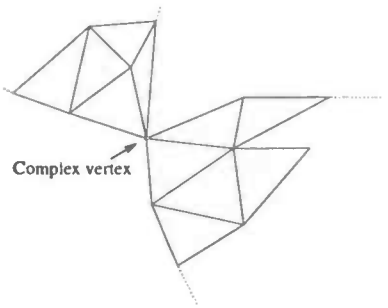

Figure 5.2: An example of a vertex that is NOT a border vertex.

To classify a situation as in Fig. 5.2 (this is not a border vertex) we try to continue walking around the loop, after skipping the border. If we now find another border, this is a complex vertex. This is exactly the situation as seen in Fig. 5.2. If there is not another border, it is a border vertex. In

this case KLS_BORDER_VERTEX is returned. The old version of CreateLoop gave an error-message (and continue) if it found a complex-vertex here, the new version returns KLS_COMPLEX_VERTEX.

Chapter 6

Complexity

By looking at the expected complexity and the running time of the program, it is possible to determine if a too large complexity (e.g. because of suboptimal data structures) of the implementation is the cause of the long running-times. In this chapter I will compare these and show that `KLSdecimate` does not have the complexity as would be expected from theoretical complexity calculations.

In an article by Sourcy and Laurendeau [SL96] 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 have the same complexity. By comparing the complexity of KLSDecimate with the complexity as calculated by Sourcy and Laurendeau, we can determine if there are any routines in the program which were not efficiently implemented.

6.1 Calculation of the complexity

Let t_{dpt} be the time needed to calculate the distance from a vertex to a triangle. When a vertex v is removed the time needed to recalculate the error is N_r times t_{dpt}, where N_r is the number of triangles needed for the retriangulation. Let N be the number of vertices in the initial mesh and let n be the number of removed vertices, then the number of already removed vertices N_{rv} in the area of a retriangulation, can be estimated by

$$N_{rv} = \frac{N}{N-n}$$

These vertices are to be used in the calculation of the retriangulation error.

To estimate the time needed to compute a retriangulation error for a given number of removed vertices n is then

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

In this formula it can be seen that the time needed to calculate a potential error increases with the number of already removed vertices. This is what already was expected after runs from our implementation since the implementation starts removing vertices fast and slows down when the number of removed vertices increases.

When we focus on the error calculation and all the other computation times are neglected, a vertex can be removed in the following time

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

where N_t is the estimated number of vertices for which the potential error needs to be recalculated.

Now it is possible to give an estimation for the running time $t(n)$ of the algorithm, the total running time of the algorithm needed to remove n vertices from the mesh:

$$t_{[SL96]}(n) = \sum_{k=0}^{n} t_k \approx \int_0^n t_k dk = N_t N_r t_{dpt} N \log(\frac{N}{N-n})$$

6.2 Experimental results

To see whether the complexity of the original KLSDecimate implementation complies with the theoretical complexity, I collected some run-time information. I used the 'fran' dataset which is supplied with VTK. This dataset consists of 26460 vertices and 52260 triangles. After every 100 removed vertices we recorded the total running time in milliseconds. A subset (after every 1000th vertex) of the results is shown in Table 6.2.

Vertex	Time	Vertex	Time
1000	179610	13000	3632819
2000	397461	14000	4015689
3000	621398	15000	4418219
4000	858541	16000	4834543
5000	1128819	17000	5286594
6000	1386299	18000	5756829
7000	1659847	19000	6268025
8000	1945681	20000	6829794
9000	2250914	21000	7439121
10000	2576323	21700	7900062
11000	2908442	22000	8117687
12000	3262402	23000	8867284

Table 6.1: Experimental results of the unmodified implementation.

The first column is the vertex after which the time was written and the second column is the time in milliseconds. The same results, in graphical form, including a theoretical curve are shown in Fig. 6.1.

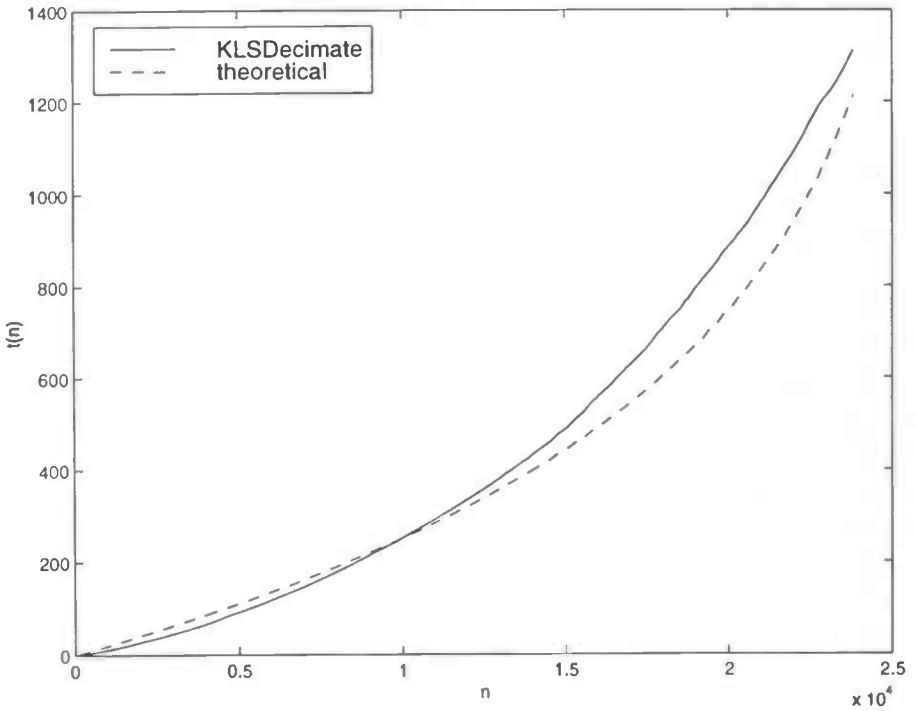

Figure 6.1: The timing results using a dataset of about 26460 vertices.

The parameters used to draw the theoretical curve were:

- $N = 26460$
- $N_t N_r t_{dpt} = 155.0$

Of course the second item is experimentally determined, but the theoretical and experimental curves do not have the same shape, and this is what counts when we do a comparison between complexities. The implementation is slower in the beginning than would be expected from the curve of the theoretical complexity.

Another interesting quantity to look at is

$$\frac{t(n)}{t_{[SL96]}(n)}$$

The result from this formula shows us the factor between the experimental and theoretical timing-values.

If the plot of this formula is a straight horizontal line, there is no difference between the complexity of the program and the expected (theoretical) complexity. As we already expected the complexities are not the same.

This plot is shown in Fig. 6.2.

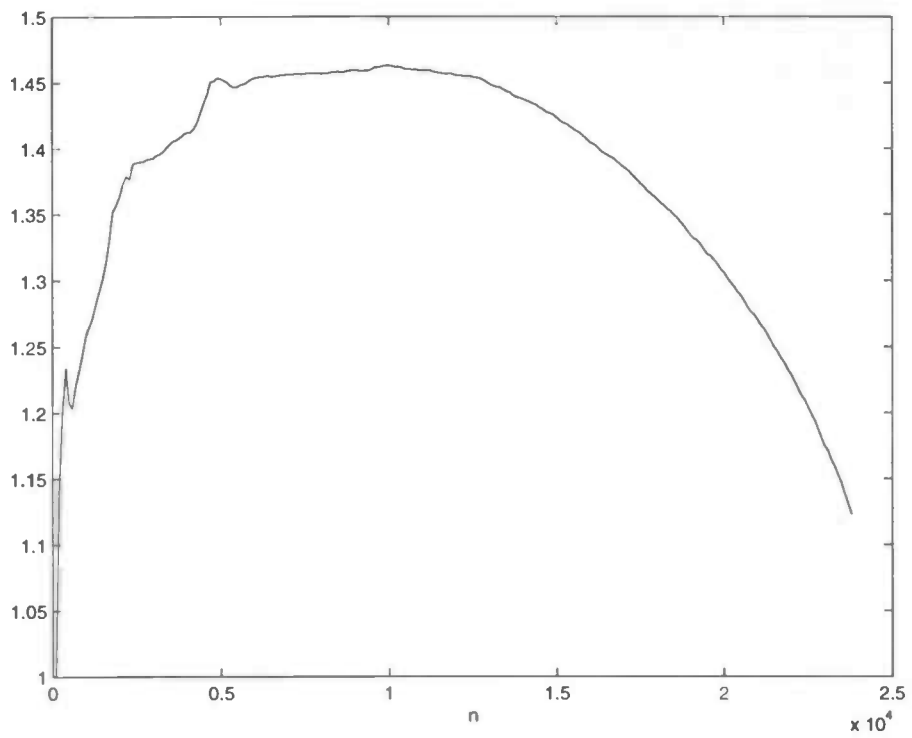

Figure 6.2: The timing of KLSDecimate divided by the expected timing.

6.3 Conclusion

As shown, the theoretical and the experimental curves do not have the same shape, thus the theoretical complexity of the algorithm is not the same as of the implementation. This is due to the fact that the data structures in KLSDecimate are not really efficient. The plot shown in figure 6.2 shows us there is a routine in the program which slows down when n increases, while for very large n the running time is close to expected. This is probably the slow handling of the *potential error* list. This priority queue contains all the vertices when the program starts. The vertices are removed from this queue when they are removed from the mesh. As the list gets smaller the speed increases. This is the case, because a linear search is used to find vertices when their potential error needs to be updated.

Chapter 7

Optimization

As already discussed in the previous chapter, the implementation is at the moment rather slow. As we already saw in the previous chapter the complexities of the program and the theoretical complexity differ. By optimizing the complexity of the routines in the program we can try to move the complexity towards the theoretical complexity. When all the routines which influence the complexity have been improved, we can only optimize the program with changes which speed up the program by a constant factor.

7.1 Sorted Vertex List

In the implementation as discussed in [NA98] the priority queue of vertices is implemented using a sorted doubly linked list. The sorting key is of course the potential error value. In the sorted vertex list the element with the smallest potential error is at the front. However, sometimes we need to search a vertex using its 'id'. Because the list is sorted with respect to potential error we need to traverse the list from its beginning to find the vertex.

By replacing the list with a more efficient data structure, operations performed on this data structure will be faster. On the other hand, the simplicity of the linked-list gives good performance for meshes with small numbers of vertices. When the size of the mesh and thus the number of vertices increases, the operations on the sorted vertex list slow down rapidly. One of the operations performed on the list is the *update* operation: the potential errors of some vertices are changed, and thus the potential errors of the changed vertices have to be moved in the list to maintain the sorting order. Especially this operation is slow, for the element has to be deleted and reinserted. Both these actions are of order $O(n)$, where n is the size of the list.

7.1.1 Heap or hash

Heap

A data structure designed to have the smallest element (or the largest depending on the criterion), available in $O(1)$ time is a *heap*. A *heap* is a 'binary tree' where the elements are ordered according to a certain criterion. Let us 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 larger than the key-value of its children. An example of such a heap is shown in Fig. 7.1.

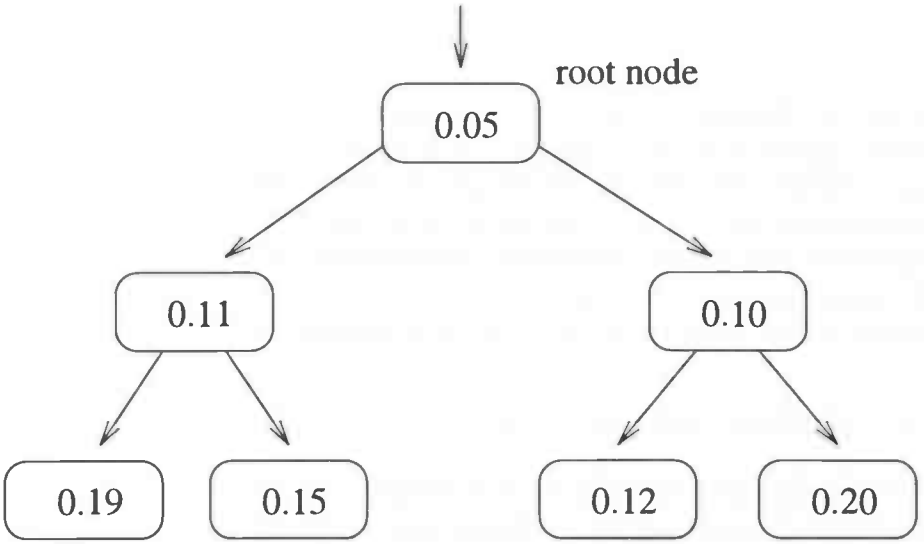

Figure 7.1: An example of a heap.

Although this data structure returns the smallest element in $O(1)$ time, it has not got any searching capabilities, which are needed for our potential error updates. Searching in a heap takes $O(n)$ time.

Hash table

When we would like to search efficiently, we need a data structure like a binary tree or a hash table. When we would like to use a binary tree we would need a balanced binary tree, but this gives too much overhead for our application. This is because we are constantly deleting elements (the smallest) from the tree. A hash table would be more suited for the implementation. However, both these data structures can not find the element with the smallest potential error in $O(1)$ time. Instead, it would take $O(n)$ time to find the smallest element.

Both a hash table and a heap

A heap is good at finding the smallest element in $O(1)$ time and a hash table is good in finding elements with a certain 'id'. A solution to the problem is using both of these structures and using links with pointers between them to find the corresponding elements. The result is a data structure that is good in both finding the element with the smallest error-value and deleting elements with a certain 'id'.

A data structure with both a heap and a hash table was implemented and added to KLSDecimate. When small datasets were used, there was no noticeable increase in speed. For bigger datasets there was a noticeable speedup in KLSDecimate. Huge datasets were not tested, mainly because KLSDecimate is still too slow to use these datasets.

	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 7.1: The complexity of the data structures, where m is the average depth of a node from the hash table.

7.1.2 Implementation of a combined heap and hash table

To combine the capabilities of a hash table and a heap, a combination of the two was implemented. The resulting structure contains a separate heap and a hash table, which are linked through pointer structures. When an operation is performed on the data structure the fasted method to access the structure is chosen. This can be through the hash table and/or through the heap. I will discuss the possible operations separately below.

The interface to the data structure is almost the same as the interface to the old sorted vertex list. As a result, replacing the old data structure with the new data structure in the implementation was not difficult. In the end some minor modifications where done to improve efficiency of the new data structure in combination with KLSDecimate.

Insertion

```
void  Insert (int ID, float pot_err);
```

When an element $(id, error)$ is inserted, it is separately inserted in both the hash table and the heap. When they are inserted, the pointer in the hash table node which should point to heap node is set. The pointer in the

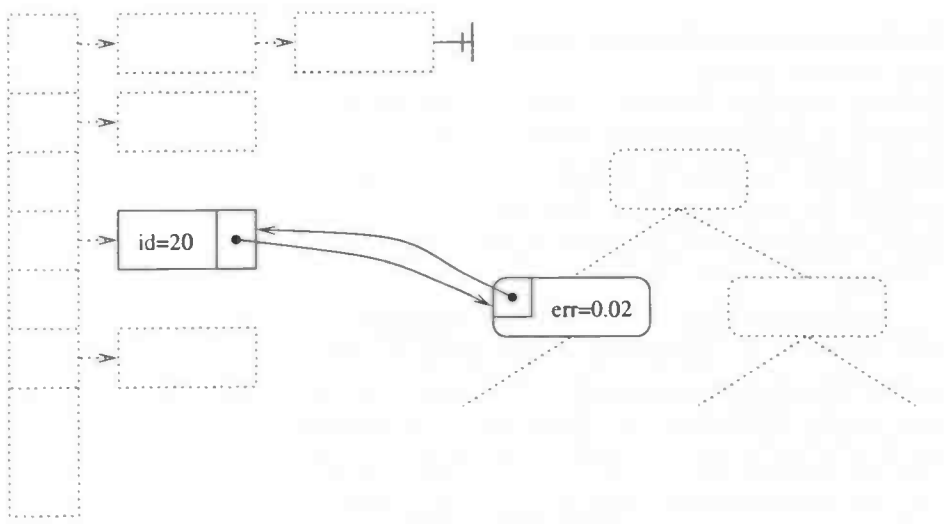

Figure 7.2: An example of a combined heap and hash table.

node in the heap is also set to point to the corresponding node in the hash table.

Searching

```
float GetPotentialError (int ID);
```

To find an element with identifier 'id', the element with identifier 'id' is searched in the hash table. This operation can be performed in $O(m)$ time. This is the average time needed to search in the element list of a hash table bucket. m is the number of elements in the hash table divided by the number of buckets. In KLSDecimate the number of buckets is calculated by dividing the initial number of vertices by 20. Thus initially m is 20 in KLSDecimate. The GetPotentialError() method returns the error value associated with 'id'. When the element with identifier 'id' is not found an error is returned.

Deletion

```
int   Delete (int ID);
```

To delete an element with a certain 'id', first the element is searched in the hash table. Through the pointer which points to the corresponding element in the heap, we now know both the position in the heap and the hash table. To delete the elements, the normal deletion for a heap and for a hash table are used.

Finding the smallest element

```
float GetFirstPotentialError(int &id);
```

To find the smallest element in the structure it is simply a matter of looking at the element which is at the top of the heap. The GetFirstPotentialError() function returns the error of this element and stores its identifier in 'id'. When the data structure is empty it returns identifier -1.

An additional method – SetPotentialError

One other method was added to improve the efficiency of the data structure, the SetPotentialError() method. This method updates the error of an element in the data structure. In KLSDecimate this method is used when the potential errors of vertices are recalculated, for example in the neighbourhood of a retriangulation. Because the identifier of the element remains the same it is not necessary to delete and reinsert the element in the hash table. It only has to be moved in the heap. This saves the $O(m)$ delete-operation and the allocation of memory for a new element.

7.1.3 Profiling and square roots

With a profiler we can determine the time spend in functions in the program. The result given by the profiler was really interesting, because the function that took most of the time was the system library function 'sqrt' , not a self-implemented function. The square root is used on a lot of places in the implementation for the distance measurement. Remember that the distance between two points is given by

$$\sqrt{(\Delta x^2 + \Delta y^2 + \Delta z^2)}$$

We can remove square roots from the program by using the square of the distance instead of the distance itself as the error measure. This small change speeds the implementation up by about 50%, but after the improvement the sqrt-function still remained to be the function that took the largest part of the time in the program. Some investigation showed that a VTK function to normalize the length of the vector used sqrt, which is pretty obvious. This function was used throughout the program, even at points where it was not necessary. Therefore an extra function to calculate a non-normalized normal was added. This function was used at some points instead of the function that always calculated the normalized normal.

After profiling the calling of the sqrt-function there was still one function where the normalization calls could not be removed, LineFacet(), which determines if a line intersects a triangle (facet) in 3D-space, and if so, in which point it intersects. The routine does this by calculating the intersection point between the line and the plane defined by the triangle, and looking

at the position of the intersection point. If it is inside the triangle, then there is an intersection, otherwise there is none. To determine whether the intersection point is inside the triangle a rather complicated method is used, which needs three normalized vectors. There is however a function in VTK which can determine if a point is inside or outside a triangle. After some minor modifications this routine could be fitted inside LineFacet(). Again three normalizations where removed and the percentage of time used by *sqrt* decreased.

After all the performance-modifications above, a really large improvement in speed was gained, as will be discussed in chapter 9.

The profiling report of the modified implementation is shown below. This table shows us which functions are now the slowest in the program.

It also shows us the function which now takes most of the time in the program, the distance measurement routine Distance2BetweenPointAndTriangle. This routine calculates the square distance between a point and a triangle. We can now conclude that the distance calculation is the next routine which should be optimized. The optimizations proposed in [KK97] will (now other slow routines have been removed from the program) have a positive influence on the speed of the implementation. Speeding up other routines will not have such a large influence on the speed of the program as optimizing the distance calculation.

cycles%	secs	procedure	(file)
46.67	2.66	Distance2Between-PointAndTriangle	(KLS:KLSDecimate.cc)
15.06	0.86	vtkTriangle::PointInTriangle	(VTK:vtkTriangle.cxx)
9.25	0.53	__sqrt	(libm.so:sqrt.s)
6.87	0.39	vtkMathCross	(VTK:vtkMath.cxx)
3.92	0.22	vtkFloatArray::GetTuple	(VTK:vtkFloatArray.cxx)
2.26	0.13	CanSplitLoop	(KLS:KLSDecimate.cc)
2.24	0.13	CalculateCase1	(KLS:KLSDecimate.cc)
1.84	0.10	vtkPointSet::GetPoint	(KLS:vtkPointSet.h)
1.40	0.08	vtkPolyData::GetCellPoints	(VTK:vtkPolyData.cxx)
1.14	0.06	SplitLoop	(KLS:KLSDecimate.cc)
1.13	0.06	IsInTriArray	(KLS:KLSDecimate.cc)
0.78	0.04	realfree	(libc.so.1:malloc.c)
0.70	0.04	CorrespondenceList::GetTriangleCorr	(KLS:CorrespondenceList.cc)
0.64	0.04	Triangulate	(KLS:KLSDecimate.cc)
0.63	0.04	__malloc	(libc.so.1:malloc.c)
0.61	0.03	CalculatePotentialError	(KLS:KLSDecimate.cc)
0.38	0.02	t_splay	(libc.so.1:malloc.c)
0.33	0.02	cleanfree	(libc.so.1:malloc.c)
0.32	0.02	t_delete	(libc.so.1:malloc.c)
0.29	0.02	vtkPolyData::GetCellEdgeNeighbors	(VTK:vtkPolyData.cxx)
0.27	0.02	CalculateLoopNormal	(KLS:KLSDecimate.cc)
0.19	0.01	_malloc	(libc.so.1:malloc.c)
0.19	0.01	__free	(libc.so.1:malloc.c)
0.18	0.01	_free	(libc.so.1:malloc.c)
0.18	0.01	RemoveVertex	(KLS:KLSDecimate.cc)
0.18	0.01	IsInIntList	(KLS:KLSDecimate.cc)
0.16	0.01	__nw	(libC.so:_new.c++)
0.14	0.01	vtkIntArray::__dt	(VTK:vtkIntArray.cxx)
0.12	0.01	__dl	(libC.so:_delete.c++)
0.12	0.01	vtkObject::__ct	(VTK:vtkObject.cxx)
0.10	0.01	CreateLoop	(KLS:KLSDecimate.cc)
0.10	0.01	vtkIntArray::Allocate	(VTK:vtkIntArray.cxx)
0.10	0.01	__dt::vtkObject	(VTK:vtkObject.cxx)
0.10	0.01	vtkIntArray::__ct	(VTK:vtkIntArray.cxx)
0.09	0.01	vtkReferenceCount::UnRegister	(VTK:vtkReferenceCount.cxx)
0.09	0.01	vtkIdList::__ct	(VTK:vtkIdList.cxx)
0.08	0.00	_smalloc	(libc.so.1:malloc.c)
0.08	0.00	CorrespondenceList::GetVertexCorr	(KLS:CorrespondenceList.cc)
0.08	0.00	vtkIdList::__dt	(VTK:vtkIdList.cxx)
0.06	0.00	vtkReferenceCount::__dt	(VTK:vtkReferenceCount.cxx)

Chapter 8

Triangulation

8.1 Introduction

After a vertex is removed from the triangle mesh, the hole needs to be filled with new triangles. This 'filling' is done with a *triangulation routine*, which fills the hole with triangles, as efficient as possible. The area to be filled is constrained by a circle of vertices which used to be connected to the removed vertex by an edge. These vertices form a *polygon*, this is the reason why the triangulation needed for the program needs to be a *polygon triangulation algorithm*. It is up to the triangulation algorithm to fill the area inside this polygon with triangles in such a way that as few triangles as possible are used and there are no holes in the result. If it even tries to adapt the resulting triangle mesh to the shape of the polygon, it would be even better.

8.2 Polygon triangulation

As already said in the introduction, the area to be triangulated is constrained by a polygon. We also know that all the vertices in that area are also in the polygon, in other words, there are no vertices lying around in the polygon. Algorithms which triangulate polygons which have no other vertices in the area, are called *polygon triangulation algorithms*.

An example of polygon triangulation is given in Fig. 8.1. In the left picture a triangle mesh is shown where the vertex v is removed. The resulting hole is constrained by polygon P. A possible triangulation is shown in the right picture. For the triangulation only 5 triangles were used, where the original used 7 triangles.

The polygon triangulation algorithm used by the implementation as proposed by A. Noord and R.M. Aten [NA98], is a recursive loop splitting algorithm. It splits the loop of vertices (polygon) in two parts along a *split line* and recursively applies the algorithm to the two resulting loops. This process repeats until the loops contain exactly three vertices, which form a

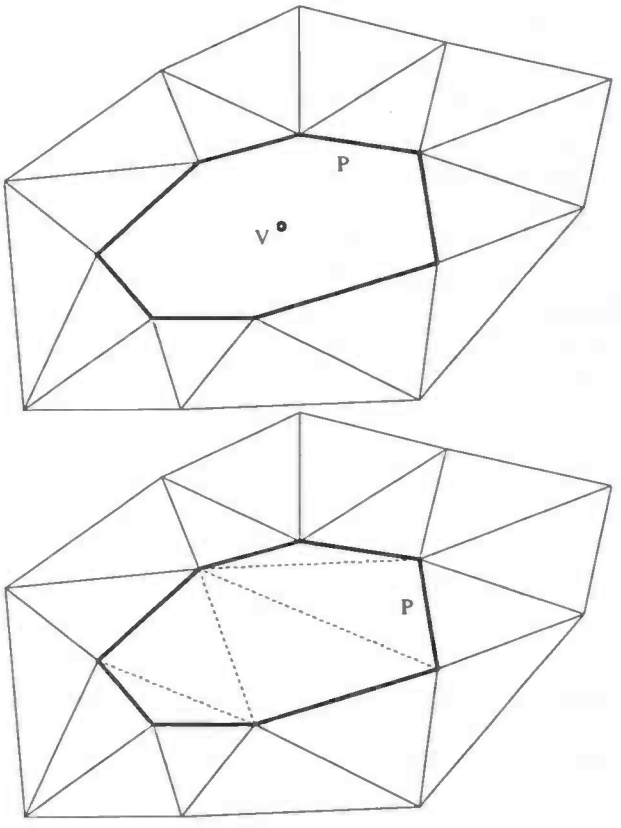

Figure 8.1: An example of polygon retriangulation.

triangle. The algorithm tries to split the loop at different split lines, and chooses the split line that would lead to the best result. In uses an *aspect ratio* to determine the best split line (the one with the largest aspect ratio). To determine the *aspect ratio* a split plane is used. This is done because the loop may be non-planar. The split plane runs through the split line and is orthogonal to the average plane of the loop. An example of a split plane is shown figure 8.2. The *aspect ratio* is the minimum distance of the vertices to the *splitting plane*, divided by the length of the splitting line.

8.3 Constrained Delaunay triangulation

We tried to replace the loop-splitting triangulation algorithm with a more common triangulation algorithm, the constrained Delaunay algorithm. However, there are some problems which make it difficult to insert an implementation of the Delaunay algorithm in the program. At first sight, it looked rather easy, because there was already an implementation of the Delaunay algorithm in the VTK toolkit.

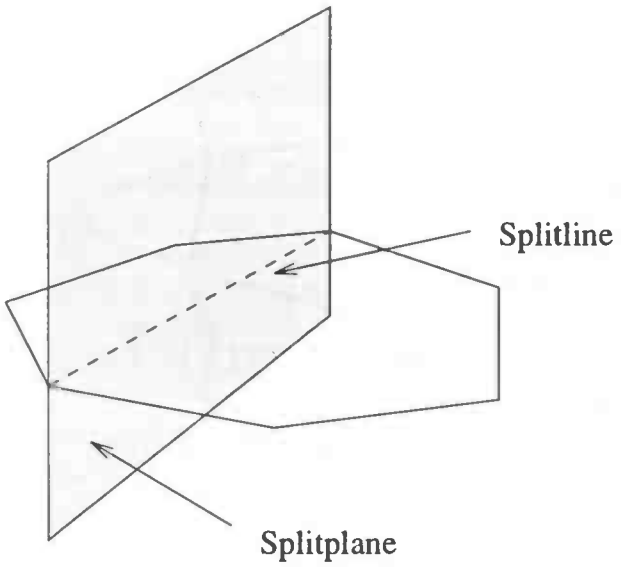

Figure 8.2: A split plane.

Some problems which arise when we would use a Delaunay algorithm:

- the Delaunay in VTK is not a constrained Delaunay.
- the Delaunay algorithm only works in 2-D
- the Delaunay algorithm is not specific for polygon triangulation

The first item is a problem because the 'normal' Delaunay algorithm does not detect non-convex polygons. We can probably overcome this problem by afterwards removing all the triangles outside the polygon. The second problem is a more serious one. Our triangle meshes are in 3-D. Although surfaces itself are 2-D objects, we still have 3-D coordinates. When we would need to use the Delaunay algorithm we would have to transform the 3-D coordinates to 2-D coordinates by projecting the 3-D surface on a 2-D plane. This sounds rather trivial, but it is not. According to the authors of [BDE96] it requires an $O(n^4)$ algorithm to project the triangle mesh on a plane. This is because we need a projection, for which the polygon in 2-D is not self-intersecting. A selfintersecting polygon can not be triangulated by the Constrained Delaunay algorithm. When projecting the polygon on a plane we loose the 3-D information like the three-dimensional shape of the polygon. For the Delaunay algorithm this is not a problem, because it does not (can not) use this information. This also means that the Delaunay algorithm does not adapt the generated triangle mesh to the 3-D shape of the polygon. This brings us to the last point, the Delaunay algorithm is not designed to triangulate polygons but to triangulate 'clouds of points'.

It can be used to generate a triangle mesh from height samples of a terrain, but it is not designed to triangulate polygons in 3-D, not even polygons in 2-D. It is possible to triangulate polygons with it by only inserting the points from the polygon, but it will not necessarily generate an optimal triangulation. In [KK97], the Delaunay algorithm is used in conjunction with the algorithm from Klein, Liebich and Straßer to decimate triangle meshes. In the article the authors show that the Delaunay algorithm in certain situations does not return a good result. Another approach is used by the authors of [SL96], who solve the problem by optimizing the results from a 2-D Delaunay triangulation in 3-D. In their article they speak of a 2.5-D triangulation.

8.4 Conclusion

Replacing the recursive loop splitting algorithm would be a waste of time, as this algorithm gives good results in 3-D. This is the result of the calculation and the use of the aspect ratio. The Delaunay triangulation algorithm is a 2-D triangulation algorithm, not designed to triangulate polygons in 3-D. Furthermore, it is difficult to convert the algorithm to a 3-D triangulation algorithm. As already shown in section 7.1.3, the speed of the recursive loop splitting is not a limiting factor. Replacing the recursive loop splitting routine is not necessary.

Chapter 9

Results of the new implementation

9.1 Complexity of the new implementation

The same measurements as already described in chapter 6 were done for the new implementation. The results in both tabular and graphical form are shown below in table 9.1 and Fig. 9.1.

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 9.1: Experimental results of the modified implementation.

In this instance the parameters used to draw the theoretical curve were:

- $N = 26460$
- $N_t N_r t_{dpt} = 155.0$

Again we also compute the $\frac{t(n)}{t_{[SL96]}(n)}$ ratio. The results are shown in Fig. 9.2.

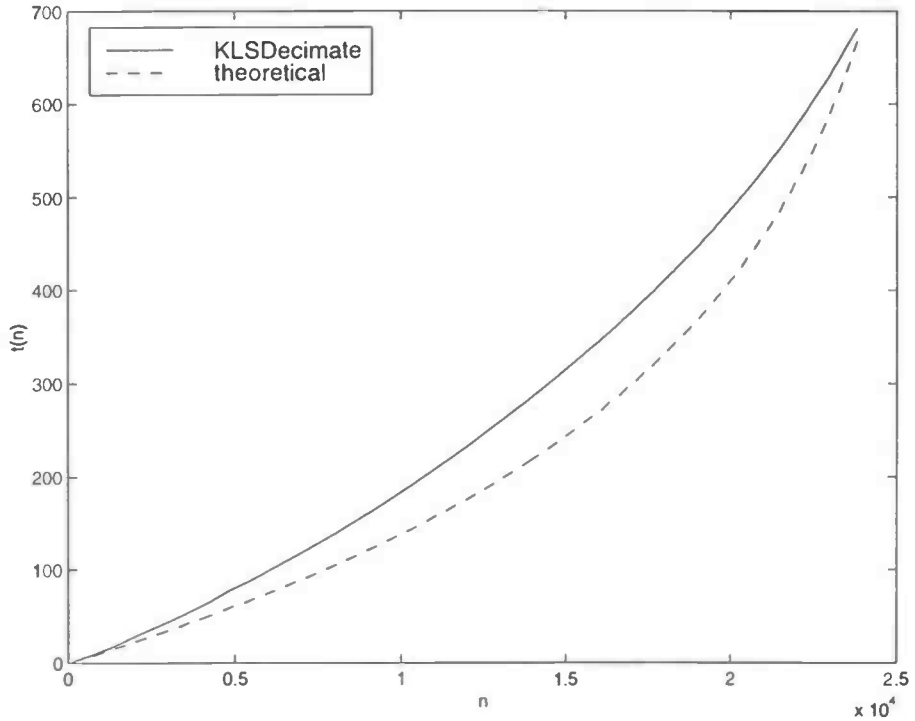

Figure 9.1: The timing results using a dataset of about 26460 vertices.

The figure is not a straight horizontal line. This means that the complexity is still not equal to the theoretical complexity of a decimation program. On the other hand, the results are now bounded in a smaller range than it was the case with the old implementation.

When we compare this graph with the graph in Fig. 6.2, we see that the 'bump' in the graph has moved more towards the end. This is caused by the replacement of the inefficient potential error list. The presence of the 'bump' is probably due to the inefficient correspondence list which is still present in the implementation. This correspondence list is also based on linked lists. When the program removes the first vertices, this correspondence list is not used, but when the number of removed vertices increases it is used more often. In the end, the number of vertices is small and thus the number of entries in the correspondence list is small and the the inefficient correspondence list has less influence on the complexity.

Now the theoretical curve almost fits the experimental curve. We can now safely conclude that the complexity has improved over the original implementation. The only change made that could influence complexity is the replacement of the *Sorted Vertex List*. Although the program is still not fast we can now state that the complexity of the program has improved quite a bit. The inefficient correspondence list is probably the only structure

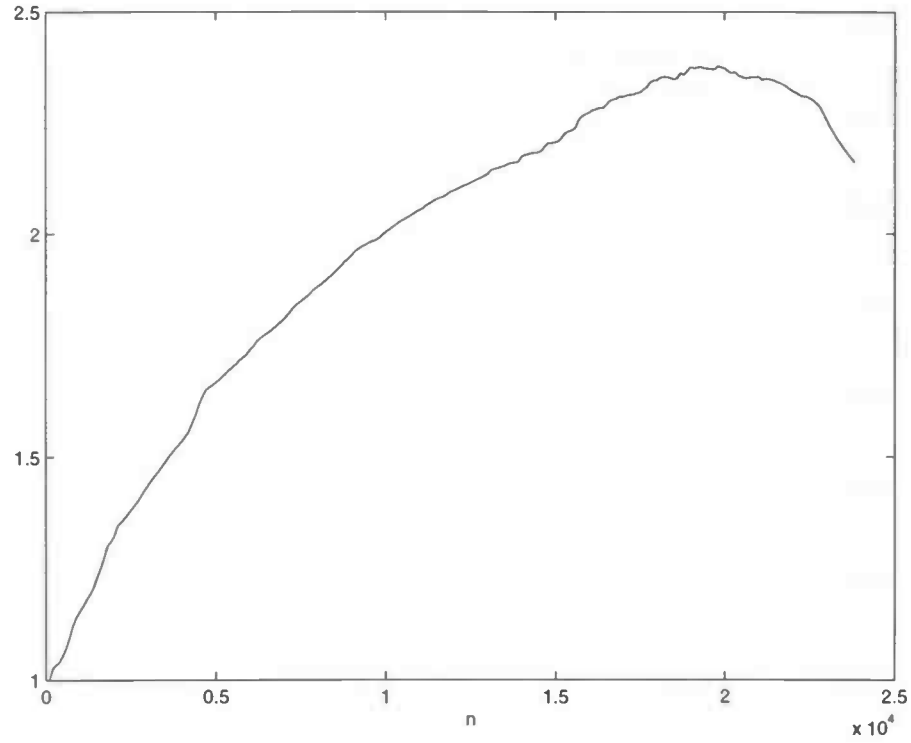

Figure 9.2: The $\frac{t(n)}{t_{[SL96]}(n)}$ for the optimized implementation.

which still influences the complexity in such a way that we still do not have the complexity as calculated in chapter 6.

9.2 Complexity according to [KK97]

In [KK97] a different formula is given to calculate the running time of the algorithm for the removal of n vertices out of a dataset of N vertices:

$$t_{[KK97]}(n) = 180\left(n + \sum_{m=1}^{n} \frac{m-1}{2(N-m)-4}\right)$$

To compare this formula with the formula given in [SL96], both formulas were plotted. For N, the number of vertices of the 'fran'-dataset(26460) was used. This is shown in figure 9.3.

The graphs are almost the same. To compare the two formulas, one was divided by the other and the result was plotted. This can be seen in figure 9.4. From this graph we can see a small difference between the two theoretically calculated complexities.

Although the formulas differ, the graphs only show a small difference. To compare the two formulas with the timing data from KLSDecimate, Fig.

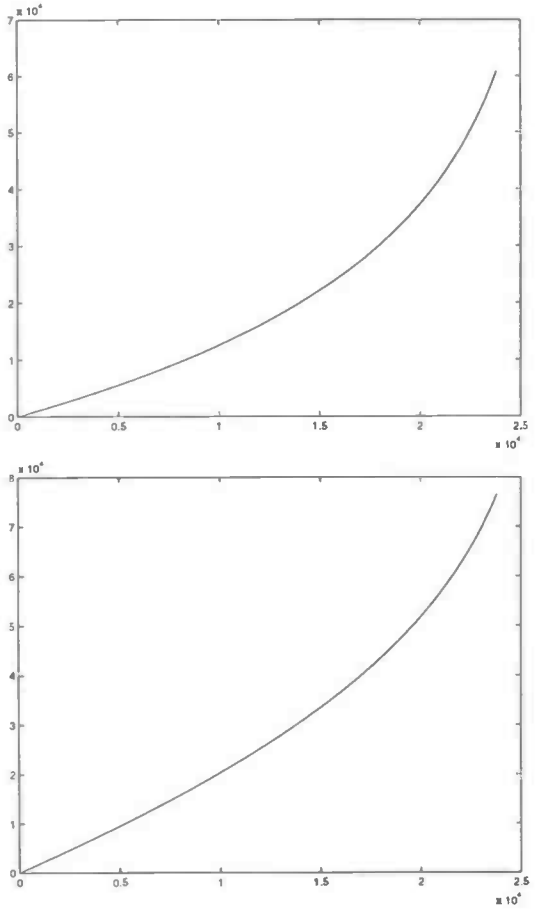

Figure 9.3: On the left the graph of the formula from [SL96], on the right [KK97].

9.5 was generated. This figure shows us $\frac{t(n)}{t_{expected}(n)}$ for both the formula from [KK97] and the formula from [SL96]. Again the experimental data from the 'fran'-dataset was used as the experimental timing data $t(n)$.

9.3 Measurements

Table 9.2 contains measurements done on some datasets, with both the running time of the (almost) original version (only support for complex vertices was added) and the improved version with all the optimizations.

Table 9.2 shows us an improvement factor of about 2.5 to 3 from the original to the new version. Although this is a large factor, it has to be said that it is still not a fast program for decimating triangle meshes.

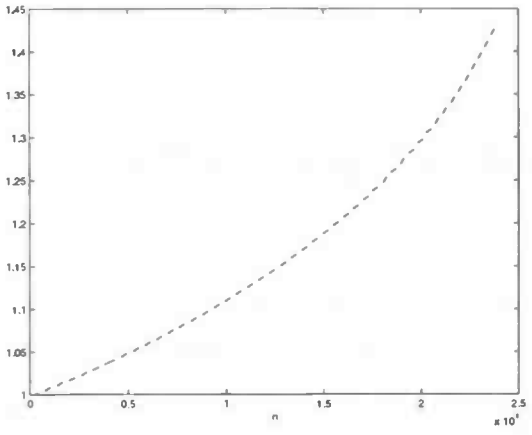

Figure 9.4: $t_{[SL96]}(n)/t_{[KK97]}(n)$.

dataset	#vertices	#triangles	running-time non-optimized	running-time optimized
teapot90.vtk	216	375	0:22.64	0:07.43
teapot.vtk	1976	3751	6:12.17	2:05.51
fohe.vtk	4005	8038	20:33.97	6:27.39
fran.vtk	26460	52260	2:41:22.31	1:07:15.99

Table 9.2: The running-times of both the non-optimized and optimized implementations.

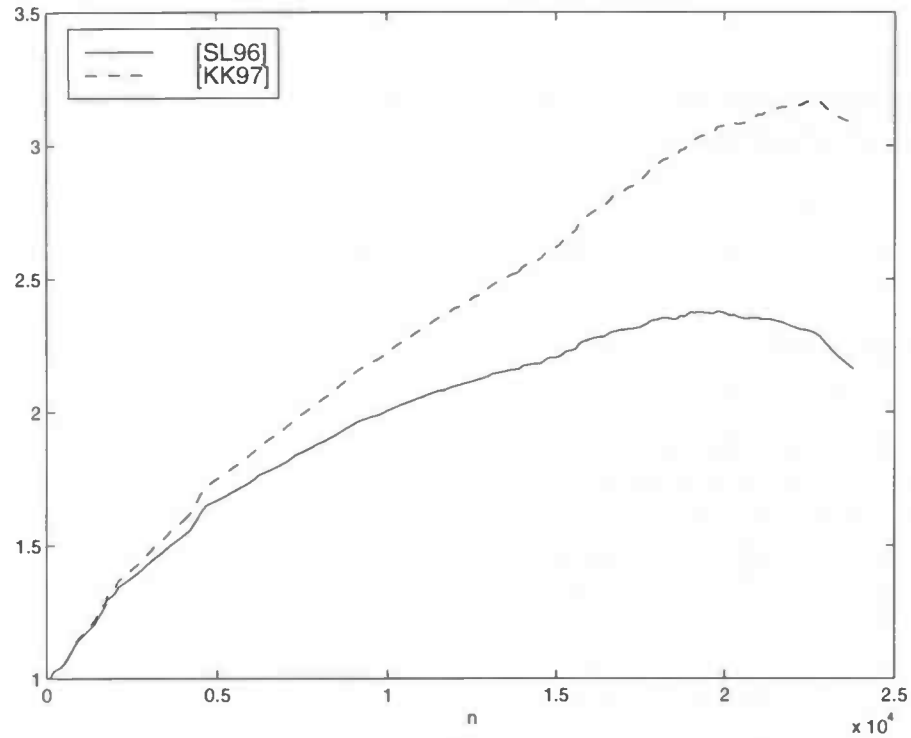

Figure 9.5: $t(n)/t_{expected}(n)$ for [KK97] and [SL96].

Chapter 10

Summary and conclusions

As a continuation of the project of R.M. Aten and A. Noord, a number of suggested improvements were implemented or analyzed:

- Adapting the routine to work with the newest version with VTK.
- Implementation for complex vertices, which would make decimation of arbitrary 3D objects possible.
- Replacing the recursive loop splitting triangulation algorithm was considered, but was not necessary.

In the end, the implementation now runs faster, is able to decimate non-manifolds and is implemented using vtk version 2.1. It is however still not a fast decimation program. Although major speedups were implemented, the programs is still not fast enough for practical purposes. The results however are qualitatively much better than the results from some faster decimation algorithms.

Suggested improvements

There are still things in the program that can be improved:

- The *correspondence list* – The correspondence list is still a doubly linked list and is used very often. It gets both read and updated. A doubly linked list is not the right data structure for this purpose. Replacing it with another, faster data structure would probably be a major improvement for the program. It is however a very complex data structure and designing/implementing a faster data structure would probably cost a lot of time.
- Optimizing the distance calculation – The distance calculation is now the slowest routine in the program (See section 7.1.3). Optimizing this routine will speed up the complete program.

Chapter 11

References

[BDE96] G. Barequet, M. Dickerson, D.Eppstein, "On Triangulating Three-Dimensional Polygons", *Proc. 12th Annu. ACM Sympos. Comput. Geom*, pp. 38-47, June 1996

[ECGP] J. Erickson, "Computational Geometry Pages", *http://compgeom.cs.uiuc.edu/ jeffe/compgeom/*

[H98] M. Held, "FIST: First Industrial-Strength Triangulation of Polygons" , *Algorithmica*, 1998, submitted.

[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, "Meshreduction with error control", *IEEE Visualisation '96*, pp. 311-318, 1996

[NA98] A. Noord, R.M. Aten, "Geometric Simplification Algorithms for surfaces", 1998

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

[VTK] W.J. Schroeder, K. Martin, W.E. Lorensen, "The Visualisation Toolkit", *Prentice Hall*, ISBN 0-130199837-4, 1996

[VTKH] "The Visualization Toolkit (vtk) Home Page", *http://www.kitware.com/vtk.html*