

**Detection of feature lines in a point
cloud by combination of first order
segmentation and graph theory**

Kris Demarsin

Denis Vanderstraeten

Tim Volodine

Dirk Roose

Report TW 440, October 2005



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Detection of feature lines in a point cloud by combination of first order segmentation and graph theory

Kris Demarsin

Denis Vanderstraeten

Tim Volodine

Dirk Roose

Report TW440, October 2005

Department of Computer Science, K.U.Leuven

Abstract

We present a method to find closed feature lines which indicate sharp edges in a point cloud in the context of reverse engineering. We start with a first order segmentation which results in different point clusters. A weighted graph structure is built where vertices correspond to the resulting point clusters and edges connect neighboring clusters. By choosing the weights carefully, the minimum spanning tree gives, after removal of some particular edges, a first approximation of the feature lines. This approximation has many short branches, which we remove with a pruning algorithm. Since the resulting graph consists of many unconnected pieces of feature lines, we introduce an algorithm that grows a part of a feature line and connects it to another part of the same feature line. A final clean up results in a good polygonal approximation of the feature lines.

Keywords : feature lines, point clouds, segmentation.

CR Subject Classification : I.3.5, G.2.2

AMS(MOS) Classification : Primary : 65D18, Secondary : 68R10.

Detection of feature lines in a point cloud by combination of first order segmentation and graph theory

Kris Demarsin,* Denis Vanderstraeten,† Tim Volodine, Dirk Roose

October 26, 2005

Abstract

We present a method to find closed feature lines which indicate sharp edges in a point cloud in the context of reverse engineering. We start with a first order segmentation which results in different point clusters. A weighted graph structure is built where vertices correspond to the resulting point clusters and edges connect neighboring clusters. By choosing the weights carefully, the minimum spanning tree gives, after removal of some particular edges, a first approximation of the feature lines. This approximation has many short branches, which we remove with a pruning algorithm. Since the resulting graph consists of many unconnected pieces of feature lines, we introduce an algorithm that grows a part of a feature line and connects it to another part of the same feature line. A final clean up results in a good polygonal approximation of the feature lines.

1 Introduction

Feature lines (also called *creases* or *crest lines*) can be mathematically defined via local extrema of principal curvatures along corresponding principal directions. Convex creases are called *ridges*, concave creases correspond to *ravines*. Extraction of the feature lines in a point cloud is useful in reverse engineering. After scanning a physical 3D object, a point cloud is obtained and reconstructing the surface through these points is very difficult if no information of the underlying surface is known. Knowledge about the feature lines gives additional information, it is a basis for a segmentation of the point cloud. This is very useful for fitting patches like B-splines or NURBS since the feature lines define the boundaries of the area where a patch can be fitted. Visualization is another application which makes skilful use of the information about the feature lines in a point cloud. Point clouds are visually easier to understand if the feature lines are indicated in the visualization. Shape recognition and quality control are other application areas of feature line extraction.

In this paper we define feature lines as the polygonal approximation of a region of points with high variation of surface normals instead of using the definition based on principal curvatures. Intuitively, we search for the sharp edges in a point cloud. Both definitions are used interchangeably in literature, however, the advantage of our approach is that we do not use any curvature information in our algorithm. We only rely on estimation of the normals, which is less sensitive to noise than curvature estimation.

**kris.demarsin@cs.kuleuven.be*

†Metris N.V., Interleuvenlaan 86, B-3001 Leuven, Belgium

1.1 Related work

Many feature line extraction algorithms rely on a triangular mesh as input, e.g. [4], [6], [7], [9] and [13], however, sometimes a mesh is not given or it can be hard to generate one. Additionally, most of these algorithms try to use accurate methods to estimate the discrete curvature because these approximations determine the quality of the extracted feature lines. However, good curvature estimation mostly requires costly surface fitting.

Gumhold et al. [3] extract crease lines and border loops from a point cloud. A neighbor graph of all the points is constructed and for every point a weight is calculated which represents the unlikelihood that the point is part of a feature. This classification is done using covariance analysis which gives an approximation of the surface curvature. A modified minimum spanning tree of the computed neighbor graph produces the polygonal feature lines. Pauly et al. [8] extend this method by using a multi-scale classification that allows feature analysis at multiple scales. The discrete scale parameter is the size of the local neighborhoods: a larger neighborhood gives a smaller surface variation. Hysteresis thresholding [1] is used to select the relevant feature nodes, i.e. the user needs to specify two thresholds. Both these methods use a graph structure where vertices correspond to points of the point cloud.

1.2 Paper overview

In this paper we propose an algorithm to reconstruct polygonal lines which indicate the sharp edges in a point cloud. Our prime focus is to detect closed feature lines in order to make patch fitting possible in a following step. Contrary to [4], [6], [7], [9] and [13], we use only normal information and no mesh or curvature estimation is required. The algorithm operates on point clouds and is based on graph theory, however, the graph vertices represent clusters of points and not individual points, as in [3] and [8]. We will call these point clusters also *segments*. The advantage of this approach is that we only need to process the relevant clusters instead of all the points of the point cloud. Additionally, the algorithm is less sensitive to noise at the segment level. The feature line extraction algorithm is explained in detail in the next section. In the following sections we motivate the choice of the weights of the graph and we illustrate some results of the algorithm.

2 Feature line extraction algorithm

2.1 Algorithm overview

Starting from a point cloud we extract polygonal lines indicating the sharp edges. Algorithm 1 gives the different steps of the algorithm which will be explained in this section. We apply the algorithm to a point cloud of 8636 points, representing a detail of a mobile phone. The results of each step are illustrated in figures 1 to 6. The red line is the boundary of the point cloud, the black lines approximate the feature lines.

2.2 First order segmentation of point cloud

Normal estimation and neighborhood selection The first step of the algorithm divides a point cloud in different clusters of points. For that purpose, we need to know the neighbors of every point as well as an estimation for the normal vector. In order to detect the transition from a smooth area to a sharp edge accurately, we estimate the normal as local as possible. For this purpose we use the 1-ring neighborhood to represent

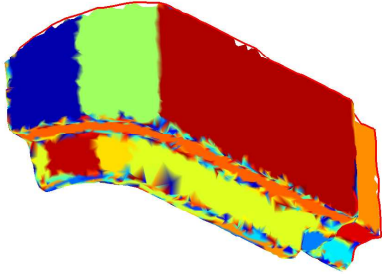


Figure 1: First order segmentation of a detail of a mobile phone.

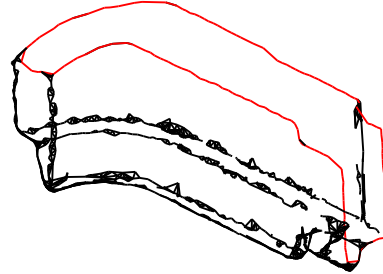


Figure 2: Part of G_{all} : edges involving two small segments.

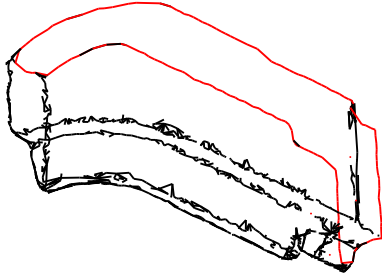


Figure 3: G_{pruned_mst} : graph after building the pruned minimum spanning tree of G_{all} .

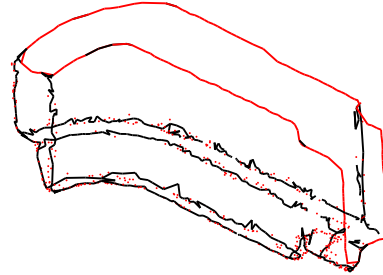


Figure 4: $G_{pruned_branches}$: graph after pruning short branches in G_{pruned_mst} .

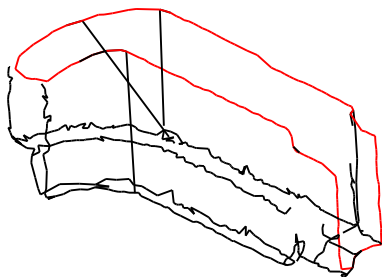


Figure 5: G_{grown} : graph after growing and connecting the feature lines in $G_{pruned_branches}$.

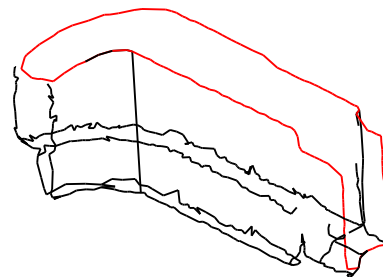


Figure 6: G_{clean} : resulting graph after cleaning up G_{grown} .

Algorithm 1 High level description of the feature line extraction algorithm.

1. First order segmentation of point cloud
⇒ point clusters (*segments*) (section 2.2, figure 1)
 2. Build graph G_{all} of all segments (section 2.3, figure 2)
 3. Build the pruned minimum spanning tree of G_{all}
⇒ G_{pruned_mst} (section 2.4, figure 3)
 4. Prune short branches in G_{pruned_mst}
⇒ $G_{pruned_branches}$ (section 2.5, figure 4)
 5. Grow the pieces of the feature lines in $G_{pruned_branches}$ and connect them
⇒ G_{grown} (section 2.6, figure 5)
 6. Clean up G_{grown} to get a good polygonal approximation G_{clean} (section 2.7, figure 6)
-

the neighbors of a point p , approximated by the Delaunay neighborhood [2]. To construct this neighborhood of p we build a local mesh. We determine the k nearest neighbors of p , we construct the least squares plane through these points and we project the points on this plane. We then compute the Delaunay-triangulation of these projected points and only the points that share an edge with p in this triangulation, constitute the Delaunay neighborhood of p . The normal vectors are estimated by the PCA analysis of the 1-ring neighbors, as explained in [5].

Region growing The method we use to divide a point cloud in segments is based on the first order segmentation described by Vanco et al. ([10], [11] and [12]). This segmentation method uses a region growing technique: based on the variation of the normal from a point to a neighbor, different clusters of points arise. For this purpose, a threshold angle α is introduced, which specifies the maximum acceptable angle between two adjacent normals in one segment. An additional threshold angle β controls the angle between the normal of a candidate point to be added to a segment and the reference normal of that segment. The threshold β insures that it is impossible to grow over a sharp edge. More details can be found in [10], [11] and [12]. Because of the usage of the angles α and β , there are large segments with low variation of the normals and small segments with high normal variation.

Indeed, at a sharp edge, the normal estimation depends heavily on the computed 1-ring neighborhood, since this neighborhood is very local and since these neighbors are located on both sides of the sharp edge. This means that the normal variation along a sharp edge is high, which results in many small segments at the sharp edges after the segmentation. Consequently, we build a graph at segment-level in the next step of the algorithm.

Figure 7 gives a close-up view of the first order segmentation, applied to the mobile phone point cloud. After the region growing, every point belongs to a certain segment, and thus a point can be colored with respect to the segment it belongs to. In that way, the different colors represent the different clusters. Since point clouds are hard to visualize, we use a triangular mesh for the visualization, but we do not use this mesh information in the algorithm. The result for the whole point cloud is shown in figure 1.

We implemented the region growing method in a breadth-first way: first all neighbors of a point are treated (added or not to the current segment) before going to their neighbors. When no more points can be added, we choose a random seed from the points which do not belong to a segment yet.

Perfectly aligned points Since the high normal variation is caused by differences in the 1-ring neighborhood, we have to be careful with point clouds where the points are

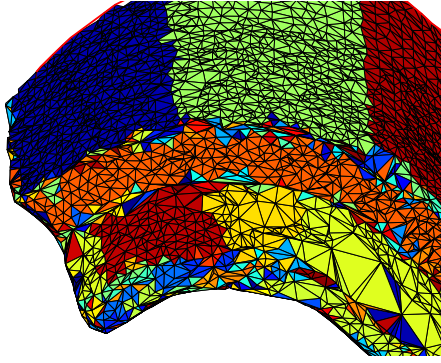


Figure 7: Close-up view of the first order segmentation of the mobile phone point cloud.

perfectly aligned. Suppose we have a point cloud with uniformly distributed points on two planes, connected to each other in a right angle, as illustrated in figure 8. We will show that, even in this special case, there are many small segments at the sharp edge between the two planes after the segmentation. This is due to the high normal variation at the sharp edge, which has the following two causes.

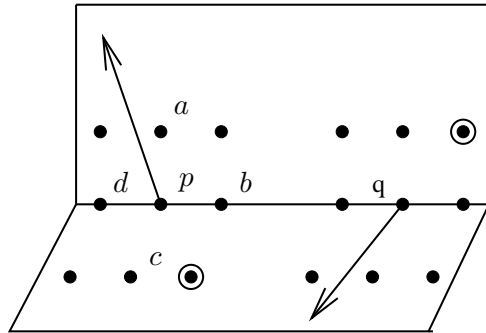


Figure 8: Piece of a point cloud with uniformly distributed points on two perpendicular planes. The two arrows are the normals of p and q . The two circles indicate the chosen fifth point.

The first reason is the computation of the k nearest neighbors. Suppose we want to compute the 5 nearest neighbors of the point p on the sharp edge in figure 8. After selecting the points a , b , c and d , which are located closest to p , we make an arbitrary choice for the fifth point between the four other points close to p , since they are on the same distance from p . This means that two distinctive points on the sharp edge, e.g. p and q , can have unsimilar k nearest neighbors and consequently, the computed Delaunay neighborhoods are incomparable. A small difference in this neighborhood results in a large difference in the computed normal.

The second reason is as follows: suppose we do select equivalent k nearest neighbors, then different Delaunay-triangulations might be possible, which results again in different 1-ring neighborhoods and corresponding normals.

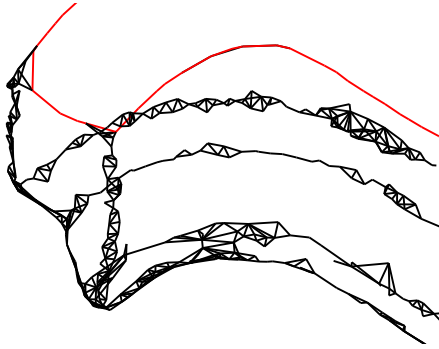


Figure 9: Close-up view of a part of G_{all} : edges involving two small segments.

2.3 Building graph G_{all} of all segments

Gumhold et al. [3] and Pauly et al. [8] already introduced a graph approach to extract feature lines, but at the level of individual points. Since the segmentation gives a strong indication of the location of the feature lines, we base the algorithm on a graph structure *at the level of segments*, which yields cost reduction and less sensitivity to noise. Additionally, we keep information about neighboring segments and, contrary to [3] and [8], our prime focus is to detect closed feature lines to make patch fitting possible later on.

We construct the connected graph G_{all} , where each vertex represents a segment and each edge connects two segments that contain at least one point with overlapping 1-ring neighborhood. For every vertex of the graph we keep a representation point of the segment, i.e. the average of all the points of the segment, and information about the size of the segment. The result can be seen in figures 2 and 9. Note that these figures only plot the edges between the small segments, which give us a first idea of the location of the feature lines. For visualization reasons the edges which involve a large segment are left out. From this point, we only process the graph and the point cloud is not needed anymore.

2.4 Building G_{pruned_mst} , the pruned minimum spanning tree of G_{all}

The edges of G_{all} involving two small segments, see figure 9, give us an idea of the location of the feature lines. However, there are many cycles and therefore, we construct the minimum spanning tree (MST) of G_{all} . For this purpose, weights of edges between small segments are calculated as the inverse of the distance between the representation points of the segments. Later on, we will explain why we choose the weights in this way. Additionally, we attach large weights (larger than the weights between small segments) to edges involving a large segment. Building the MST of G_{all} with these weights results in a graph with a reduced number of edges of which only a limited number of edges involve a large segment. We remove the latter edges and also the vertices which represent large segments. The result is illustrated in figure 3.

Since we have to distinguish between small and large segments, we could use a user-defined threshold. However, in practice, there are many small segments and less large segments with the large segments much larger than the small segments and thus taking the average of all sizes of segments is a good heuristic to separate the small segments from the larger ones.

At this point, isolated vertices, i.e. vertices with no neighbors, may exist in the new graph G_{pruned_mst} , see the red points in figure 3. They correspond to small segments which

were part of a removed edge, but since they represent small segments, they are probably part of a feature line and we do not remove them.

As already mentioned, our goal is to extract closed feature lines, however, we see that graph G_{pruned_mst} has many unwanted 'gaps'. These gaps can be the result of constructing the minimum spanning tree or due to the removal of edges involving a large segment. Later on, we will see how we can 'close' these gaps.

2.5 Construction of $G_{pruned_branches}$ by pruning short branches in G_{pruned_mst}

Although constructing the pruned minimum spanning tree gives an initial reconstruction of the feature lines, the graph G_{pruned_mst} contains many short branches, as can be seen in figure 3. The next step of the algorithm, explained in section 2.6, reconstructs closed feature lines by growing an endpoint in the graph, i.e. a point with exactly one incident edge, through suited isolated vertices and connect it with another endpoint. For that reason, we prune the graph to remove unnecessary branches, such as the branches ending in e_0 and e_1 in figure 10. By removing the edges of the unnecessary branches, but not the vertices, vertices representing unnecessary endpoints become isolated vertices, such as e_0 and e_1 in figure 11. Some of the isolated vertices are useful in the grow and connect algorithm, e.g. in figure 11 it is now possible to grow from endpoint e_2 through a and b and to connect with endpoint e_3 .

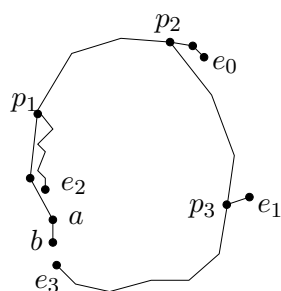


Figure 10: Graph before pruning.

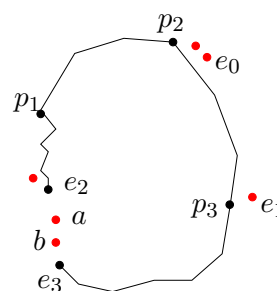


Figure 11: Graph after pruning.

We use algorithm 2 to prune, which is a similar method as in [3]. Note that only the points with more than two incident branches are treated by the algorithm and no input of the user is needed to fine-tune the *LONG_BRANCH*-parameter dependent on the point cloud. Setting the value of this parameter to 5 gave good results for all the tested point clouds. The resulting graph $G_{pruned_branches}$ for the mobile phone point cloud can be seen in figure 4.

2.6 Growing and connecting in $G_{pruned_branches}$ yielding G_{grown}

$G_{pruned_branches}$ (figure 4) consists of unconnected pieces of feature lines and many isolated vertices. These isolated vertices are due to the removal of the edges involving a large segment in the graph G_{all} and due to the pruning of short branches in G_{pruned_mst} by algorithm 2. These vertices represent small segments that might perhaps be part of a feature line. We introduce a 'grow and connect' algorithm to grow every endpoint to an isolated vertex, if possible, and to check if endpoints can be connected after each growing. This is repeated as long as growing is possible and we alternate growing and connecting to avoid that a

Algorithm 2 Pruning algorithm.

 $LONG_BRANCH = 5$ **for all** points p in the graph G_{pruned_mst} with *more than 2* incident edges **do** **if** minimum 2 incident branches have depth $\geq LONG_BRANCH$ **then** remove all incident branches with depth $< LONG_BRANCH$ **end if** **if** (exactly 1 incident branch has depth $\geq LONG_BRANCH$) **and** (minimum 2 incident branches have depth $< LONG_BRANCH$) **then**

from the short branches, keep only the one with the largest depth

end if**end for**

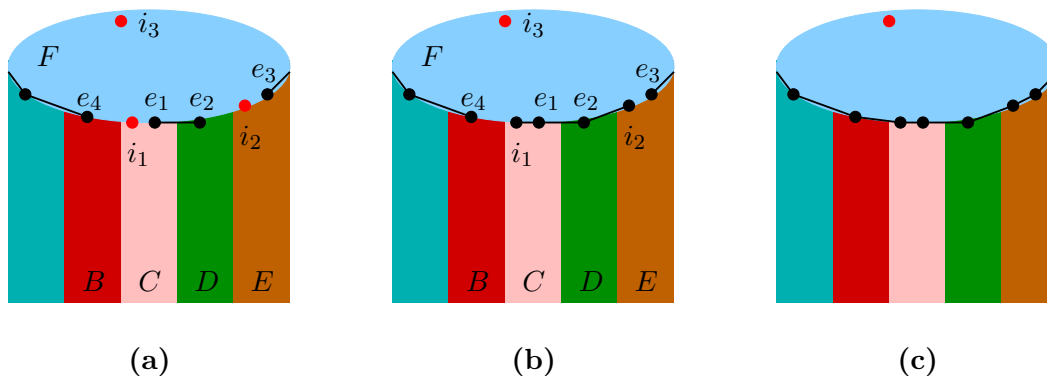


Figure 12: First order segmentation of a cylinder with the black lines representing $G_{pruned_branches}$ and the red points represent the isolated vertices. **(a)** Unconnected pieces in $G_{pruned_branches}$. **(b)** After growing from e_1 to i_1 and from e_2 to i_2 . **(c)** After adding edges (i_1, e_4) and (i_2, e_3) in a connect step.

feature line keeps on growing and 'misses' the closure. The algorithm starts with figure 12 (a), which represents the first order segmentation of a cylinder with the black lines representing $G_{pruned_branches}$. To make it visually easier to understand, the small segments corresponding to the vertices of $G_{pruned_branches}$ are left out. The result of the grow and connect algorithm can be seen in figure 12 (c): the pieces of feature lines are connected into one closed feature line.

2.6.1 Growing

The goal of this step is to grow an endpoint in $G_{pruned_branches}$ to a correct isolated vertex:

Case 1 In figure 12 (a) we see that i_1 is a good choice to grow to from e_1 , since the edge (e_1, i_1) forms the border between two large segments, i.e. C and F , or, in other words, this edge is located on a sharp edge or feature line. We say that the segments represented by e_1 and i_1 have two common *large neighboring segments* or, shortly, e_1 and i_1 have two common large neighboring segments. Using this criterion, we will never grow to isolated vertex i_2 or i_3 from e_1 .

Case 2 Growing from e_2 to i_2 could be motivated in a similar way: e_2 and i_2 form the border between one large segment and two other large segments which are neighbors.

In other words, they have one common large neighboring segment F and another large neighboring segment of e_2 , namely D , is a neighbor of another large neighboring segment of i_2 , namely E . This second case is necessary when we have a cylinder or a similar surface type.

The result of growing in both cases is illustrated in figure 12 (b).

2.6.2 Connecting

We will connect two endpoints p and q when the following conditions are satisfied:

Case 1 p and q have two common large neighboring segments e.g. i_2 and e_3 have the large segments F and E as neighbors.

Case 2 The edge (p,q) separates a large segment from two other large segments which are neighbors, e.g. the edge (e_4,i_1) separates F from B and C and segments B and C are neighbors.

The result of connecting in both cases can be seen in figure 12 (c).

After the grow and connect algorithm, the isolated vertices that still exist, can be safely removed. The resulting graph G_{grown} of the mobile phone point cloud can be seen in figure 5. We note that three long edges appear which we can explain as follows: suppose we have a cylinder and we connect two points with the same large neighbors, then it is possible that the two points are located on a different circular surface of the cylinder. The result is that we get a feature line quasi parallel with the axis of the cylinder, which is the case with two of the three long edges in figure 5. These two edges separate a planar surface from a cylindrical surface. However, we also have one long undesirable edge in figure 5 that is not parallel with the axis, which happens when we connect two points similar to case 2 of the grow or connect step.

2.6.3 Preparation of the grow and connect algorithm

An important thing to note is that the pruning step does not prune edges like e_0 and e_1 in figure 13, which represents a piece of the graph $G_{pruned_branches}$ with the red dots the isolated vertices. This is because p has only two incident edges and the pruning algorithm only treats points with more than two incident edges. We call these edges 'wrong endings' because the grow step will grow from w completely in the wrong direction instead of connecting p and r . Algorithm 3 gives a method to remove such edges. Running the grow and connect algorithm *after* these removals will connect endpoints p and r . It is a small but important step before we start growing and connecting.

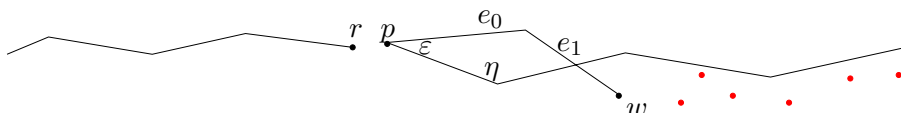


Figure 13: Wrong endings e_0 and e_1 in $G_{pruned_branches}$.

Algorithm 3 Algorithm to remove wrong endings.

$LONG_BRANCH = 5$

for all points p in the graph $G_{pruned_branches}$ with *exactly* 2 incident edges **do**
 ε , η and e_i with $i = 0 \dots n - 1$ as in figure 13
if ($n < LONG_BRANCH$) and ($\varepsilon < PI/2$) **and** ($\eta \geq PI/2$) **then**
remove edges e_i with $i = 0 \dots n - 1$
end if
if (depth of the 2 incident branches of p is exactly 1) **and** (these 2 edges make a sharp angle) **then**
remove one of these two edges
end if
end for

2.6.4 In-depth study of some aspects of the grow and connect algorithm

Finding the large neighboring segments of a small segment The grow and connect algorithm we presented here needs information about the large neighboring segments of the small segments representing an endpoint or an isolated vertex in $G_{pruned_branches}$. Since this information is not available in $G_{pruned_branches}$, we build, immediately after the construction of G_{all} , another graph $G_{large_neighbors}$ which connects each small segment with large neighboring segments, found by recursively searching the neighboring segments. As with G_{pruned_mst} , only the representation points of the segments are kept in $G_{large_neighbors}$.

For example, to find the large neighboring segments of the segment represented by s in the graph G_{all} , illustrated in figure 14 (a), we recursively search the neighbors of the surrounding small segments of s and we find the segments corresponding to n_1 and n_2 . A special case exists when a small segment is completely surrounded by one large segment, like the segment represented by i_2 . In that case, only one large neighbor is found. A piece of the constructed graph $G_{large_neighbors}$ is illustrated in figure 14 (b).

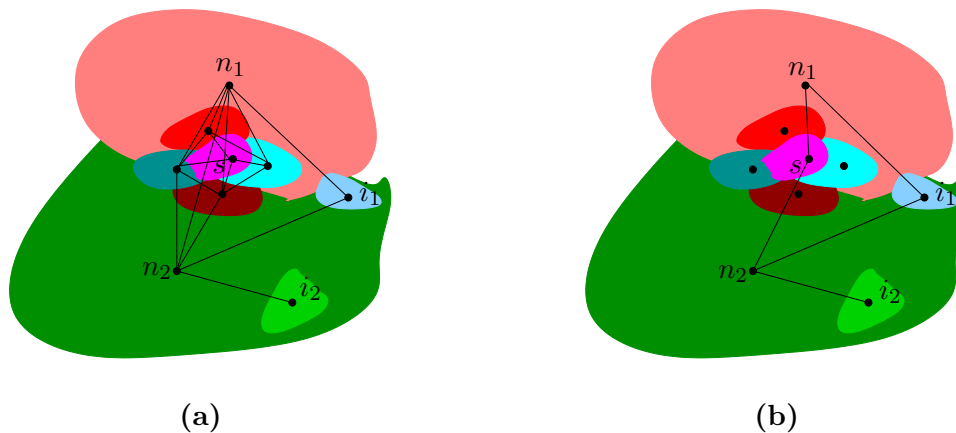


Figure 14: Segmentation where n_1 and n_2 are representation points of large segments; the other points represent small segments; s is an endpoint and i_1 and i_2 are isolated vertices in $G_{pruned_branches}$. **(a)** G_{all} , where each segment is connected to the neighboring segments. **(b)** Piece of $G_{large_neighbors}$, where each small segment is connected to large neighbors.

Efficient implementation of the grow step To find a suited isolated vertex for every endpoint, we could check *all* the isolated vertices for every endpoint in $G_{pruned_branches}$, however, we use a more efficient approach. Suppose we want to grow from endpoint s in $G_{pruned_branches}$. Because of the conditions in case 1 and 2 of the grow step, we can limit the set of candidate isolated vertices by constructing a set V of isolated vertices sharing a large neighboring segment with s . For this purpose we search the neighbors of the large neighboring segments of s in the graph $G_{large_neighbors}$, see figure 14 (b), which gives us $V = \{i_1, i_2\}$. We first check if there are isolated vertices in V sharing at least *two* common large neighboring segments with s , and if there are different possibilities, we select the vertex which maximizes the angle in s between the two incident edges in $G_{pruned_branches}$. If no vertex in V shares two large neighboring segments with s , we check if there are elements in V similar to case 2 of the grow step described above and we choose between different vertices with the same angle criterion. Since in figure 14 (b) only i_1 has two common large neighboring segments with s , we grow from s to i_1 in $G_{pruned_branches}$.

2.7 Cleaning up G_{grown} to get a good polygonal approximation G_{clean}

Although the grow and connect step connects as many components as possible, it is possible that afterwards there still exist small components in the graph G_{grown} , which we consider as noise and can be removed.

As a result of the grow and connect step, it is likely that even more branches can be pruned. An additional pruning step then results in a better approximation of the feature lines. The final graph G_{clean} can be seen in figure 6.

3 Choice of the weights

The grow and connect step is an important step in the algorithm that grows the feature lines through the gaps in $G_{pruned_branches}$ and attach them to each other. However, the size of these gaps determines the quality of this step: to obtain good results, the gaps must be as small as possible. This is illustrated in figure 15, which gives two different possibilities for $G_{pruned_branches}$: (a) has a small gap between endpoints b and c and the connect step will connect them, however, the gap in (b) is too large to be connected, because n_a^2 and n_b^2 are no neighboring segments.

Hence, during the construction of the MST of G_{all} the gaps in G_{pruned_mst} (and thus also in $G_{pruned_branches}$) must be kept small. Prim's algorithm minimizes the total sum of weights of all edges and thus chooses every time the edge with the smallest weight which connects a point that is already in the tree with a point which is not yet in the tree. This will result in small gaps if we assign weights to the edges inversely related to the Euclidean distance between the corresponding vertices. Figure 15 (a) and (b) are the results of using the reciprocal respectively the Euclidean distance weights.

Figures 16 and 17 compare the results of the mobile phone point cloud when using the reciprocal respectively the Euclidean distance weights. Using the Euclidean distance weights results in a worse approximation of the feature lines: many gaps are not closed with the grow and connect algorithm, since they are too large.

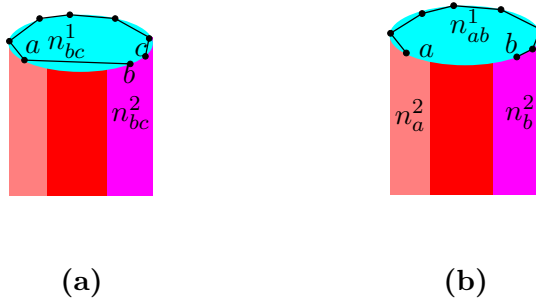


Figure 15: Two possibilities for $G_{pruned_branches}$ according to the choice of weights. **(a)** Reciprocal distance weights. **(b)** Euclidean distance weights.

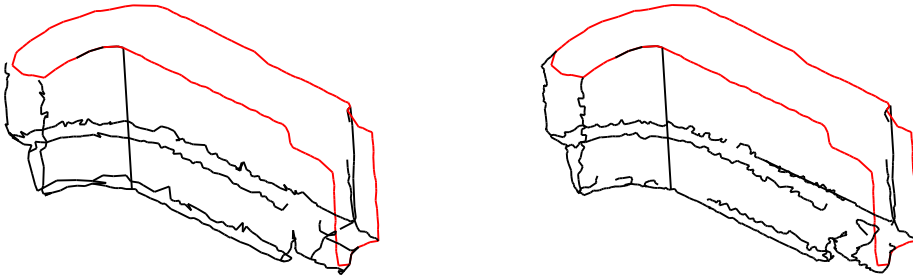


Figure 16: Approximation of the feature lines using reciprocal distance weights. Figure 17: Approximation of the feature lines using Euclidean distance weights.

4 Results

We have used the algorithm on three point clouds: 'phone' (8636 points, figure 6), 'pump' (7587 points, figure 18) and 'rocker arm' (30132 points, figure 19). These results illustrate how different pieces of feature lines grow through the isolated vertices and attach to each other, especially the circular feature lines illustrate this. Additionally, we see an unwanted effect of the algorithm: sometimes a gap does not meet the connect conditions although we would like to close it. If the grow conditions are satisfied, feature lines might grow 'over' each other, e.g. in some circular feature lines the two endpoints grew too far instead of connecting to each other. Otherwise, the grow and connect algorithm will stop and the end result has unwanted gaps.

Table 1 gives for every step of the algorithm the number of vertices and edges of the corresponding graph. In the case of the mobile phone point cloud, we start with a point cloud of 8636 points and then we build a graph G_{all} of 844 vertices and 2584 edges. From this point, every step reduces the memory consumption of the graph: a huge reduction in the number of edges happens when building G_{pruned_mst} . Pruning and removal of wrong endings further decrease the number of edges, while the number of vertices stays the same. After the grow and connect step, the isolated vertices can be thrown away and thus the number of vertices decreases. The results for the pump and rocker arm point cloud can also be found in the table.

Table 2 illustrates the time consumption of the algorithm. The segmentation step requires much more time compared to the other steps of the algorithm, because this step

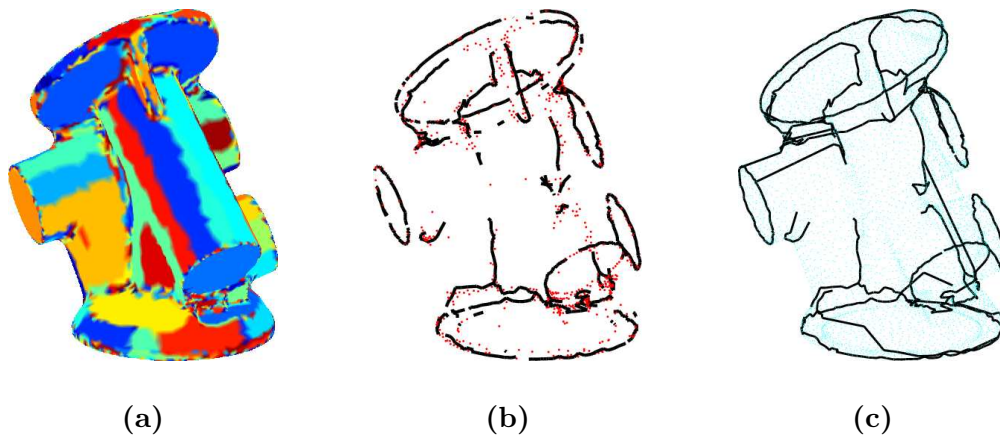


Figure 18: The pump point cloud. (a) Segmentation. (b) Approximation of the feature lines before the grow and connect algorithm. (c) Final approximation of the feature lines plotted on the point cloud.

has to grow through all the points of the point cloud and the normal for each point needs to be estimated. Note however that we did not optimize the implementation for execution speed. There are two other steps which require more time than the rest of the algorithm. The first one is the construction of the graph G_{all} , since for each segment the neighboring segments need to be computed and therefore, all the points of each segment are used. The second one is the construction of the graph G_{grown} . In the table, two times are mentioned: the first one is the time required for the removal of wrong endings and the second one is the time consumption of the grow and connect algorithm itself. The latter time depends on the initial reconstruction of the feature lines and on the number of iterations needed in the grow and connect algorithm. For example, the rocker arm point cloud is much larger than the pump point cloud, but requires less time to grow and connect.

	Size of point cloud						
<i>Phone</i>	8636						
<i>Pump</i>	7587						
<i>Rocker arm</i>	30132						
Edges	G_{all}	$G_{large_neighbors}$	G_{pruned_mst}	$G_{pruned_branches}$	G_{grown}^1	G_{grown}^2	G_{clean}
<i>Phone</i>	2584	1213	799	419	378	394	378
<i>Pump</i>	5002	2959	1364	847	732	859	818
<i>Rocker arm</i>	6702	3409	1960	956	849	941	911
Vertices							
<i>Phone</i>	844	844	816	816	816	400	381
<i>Pump</i>	1666	1666	1477	1477	1477	878	831
<i>Rocker arm</i>	2253	2253	2098	2098	2098	957	927

Table 1: Size of the graph structure in the different steps of the feature line extraction algorithm. G_{grown}^1 is the graph after removal of wrong endings and G_{grown}^2 is the graph after the grow and connect algorithm.

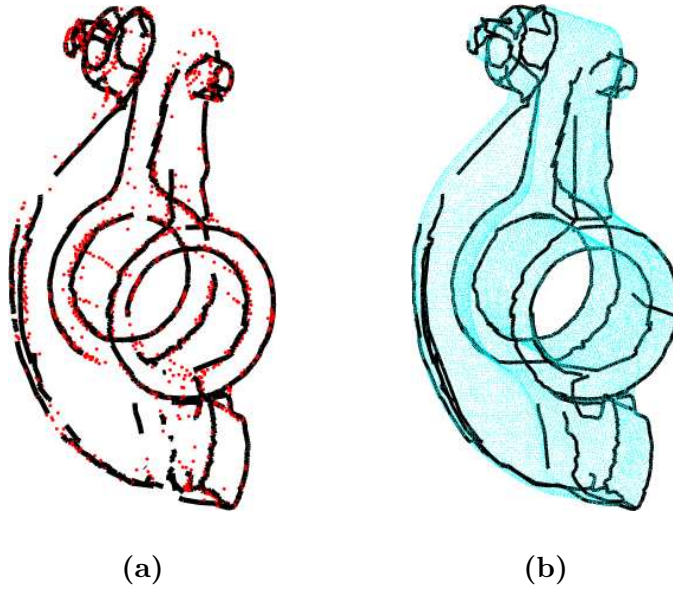


Figure 19: The rocker arm point cloud. (a) Approximation of the feature lines before the grow and connect algorithm. (b) Final approximation of the feature lines plotted on the point cloud.

Time(s)	Segment	G_{all}	G_{pruned_mst}	$G_{pruned_branches}$	G_{grown}	G_{clean}
<i>Phone</i>	3.32	0.48	0.06	0.01	0.05 + 0.21	0.02
<i>Pump</i>	2.95	0.53	0.11	0.01	0.02 + 2.67	0.06
<i>Rocker arm</i>	26.95	1.55	0.19	0.02	0.06 + 2.3	0.05

Table 2: Time consumption of the different steps of the feature line extraction algorithm on an Intel Pentium 4, 3.20 GHz. All times are in seconds.

5 Conclusion

We presented an algorithm to extract feature lines from a point cloud, with the aim to obtain closed feature lines to make patch fitting possible later on. We start with a first order segmentation which gives an initial idea of the location of the feature lines. Since the region growing method has to grow through *all* the points, it is a dominating step in the algorithm, however, we limit this cost by using only normal information. Afterwards, we build and manipulate a graph of the segments G_{all} . Using a graph structure at the level of segments yields faster execution times and less memory consumption. In addition, at the segment level the algorithm is less sensitive to noise. Once we build the graph G_{all} , the point cloud is not needed anymore and we only need to process the graph in following steps.

The algorithm extracts the feature lines from a point cloud without estimating the curvature and without triangulating the point cloud.

A few parameters must be set, e.g. the threshold angles of the segmentation step and the parameter *LONG_BRANCH*, used in the pruning step. Since the proposed values for the parameters work for a broad range of point clouds, we can state that the algorithm is fully automated.

6 Acknowledgements

The 'mobile phone' point cloud that was used to illustrate the different steps in the algorithm is courtesy of Metris N.V. Belgium. The 'pump' point cloud and the 'rocker arm' point cloud are courtesy of respectively Materialise N.V. Belgium and Cyberware.

References

- [1] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986.
- [2] M. S. Floater and Martin Reimers. Meshless parameterization and surface reconstruction. *Computer Aided Geometric Design*, 18(2):77–92, 2001.
- [3] S. Gumhold, X. Wang, and R. MacLeod. Feature extraction from point clouds. *Proceedings of the 10th International Meshing Roundtable*, pages 293–305, 2001.
- [4] Klaus Hildebrandt, Konrad Polthier, and Max Wardetzky. Smooth feature lines on surface meshes. *Symposium on Geometry Processing*, pages 85–90, 2005.
- [5] K. Hormann. *Theory and Applications of Parameterizing Triangulations*. PhD thesis, Department of Computer Science, University of Erlangen, November 2001.
- [6] Yutaka Ohtake and Alexander Belyaev. Automatic detection of geodesic ridges and ravines on polygonal surfaces. *The Journal of Three Dimensional Images*, 15(1):127–132, 2001.
- [7] Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. Ridge-valley lines on meshes via implicit surface fitting. *SIGGRAPH*, pages 609–612, 2004.
- [8] Mark Pauly, Richard Keiser, and Markus H. Gross. Multi-scale feature extraction on point-sampled surfaces. *Comput. Graph. Forum*, 22(3):281–290, 2003.
- [9] G. Stylianou and G. Farin. Crest lines extraction from 3D triangulated meshes. *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 269–281, 2003.
- [10] M. Vanco, G. Brunnett, and Th. Schreiber. A direct approach towards automatic surface segmentation of unorganized 3D points. *Proceedings Spring Conference on Computer Graphics*, pages 185–194, 2000.
- [11] Marek Vanco and Guido Brunnett. Direct segmentation for reverse engineering. In *Proceedings International Symposium on Cyber Worlds*, pages 24–37, 2002.
- [12] Marek Vanco and Guido Brunnett. Direct segmentation of algebraic models for reverse engineering. *Computing*, 72(1-2):207–220, 2004.
- [13] Kouki Watanabe and Alexander G. Belyaev. Detection of salient curvature features on polygonal surfaces. *Computer Graphics Forum*, 20(3):385–392, 2001.