FEATURE EXTRACTION FROM POINT CLOUDS

°Stefan Gumhold, *Xinlong Wang & *Rob MacLeod

° *WSI/GRIS*
University of Tübingen
stefan@gumhold.com
* *Scientific Computing and Imaging Institute*
University of Salt Lake City, Utah
wangxl@cs.utah.edu
macleod@cvrti.utah.edu

ABSTRACT

This paper describes a new method to extract feature lines directly from a surface point cloud. No surface reconstruction is needed in advance, only the inexpensive computation of a neighbor graph connecting nearby points.
The feature extraction is performed in two stages. The first stage consists of assigning a penalty weight to each point that indicates the unlikelihood that the point is part of a feature and assigning these penalty weights to the edges of a neighbor graph. Extracting a sub-graph of the neighbor graph that minimizes the edge penalty weights then produces a set of feature patterns. The second stage is especially useful for noisy data. It recovers feature lines and junctions by fitting wedges to the crease lines and corners to the junctions.
As the method works on the local neighbor graph only, it is fast and automatically adapts to the sampling resolution. This makes the approach ideal as a preprocessing step in mesh generation.

Keywords: Feature Detection, Scattered Data, Crease Recovery, Edge Linking

1. INTRODUCTION

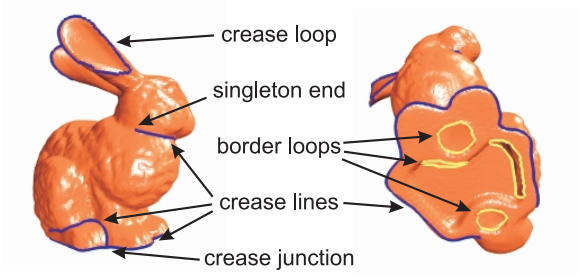


Figure 1. different elements in the crease and border patterns

In this paper we consider the feature detection and reconstruction problem for the case of the input surface being described by a point cloud. Figure 1 illustrates for the surface of the well known Stanford bunny the different types of feature elements that we want to extract. The crease pattern, shown in dark blue, consists of crease lines that either terminate in junctions or singleton ends or they close to form a loop. The border pattern consists only of border loops. Input points that lay on a crease are called crease points, points on the border loops are border points. At a junction the corresponding data point is called a corner or junction point and at singleton ends we find end points.

Feature reconstruction on a point cloud is a useful preprocessing step for surface reconstruction. The detected and reconstructed features allow to split the surface reconstruction problem into simpler sub-problems

on smooth surface patches. Another possibility to exploit reconstructed features is to enrich the point cloud around feature lines in order to ensure a sufficient sampling density. The feature extraction algorithm that we describe here is fully automated and no seed or junction points need to be marked by the user.

Figure 2 illustrates our feature extraction pipeline. The input is a point cloud a); in this case the underlying surface is a model of the human torso. In the analysis stage we construct a neighbor graph b) on the point cloud that reflects proximity and compute the local sampling density. The feature extraction stage c), d) and e) first fits ellipsoids c) to the neighborhoods of the input points, approximates the surface curvature and the maximum open angle; the latter is used to detect border points. From these vertex properties crease, corner and border penalty functions are defined that measure the unlikelihood that a specific point is on a crease or border line, respectively. From the penalty functions at the vertices penalty weights are computed for the edges of the neighbor graph. The feature line linkage d) finds a minimum spanning graph in the neighbor graph that minimize the crease or border penalty weights, respectively. The minimum spanning graph is similar to a minimum spanning tree but allows for significantly long cycles, which are needed to detect feature patterns with loops. A pruning algorithm cuts off short branches e). For noisy data the extracted crease lines are jittered because no input points lay directly on the actual crease line, in which case we apply the crease line and junction recovery stage. Here wedges, a simple crease representation consisting of two half planes meeting at a line, are fit to the crease neighborhood along the crease lines. Then the neighborhoods of junction points are fit to corners that consist of a corner point and several incident planes. The number of planes incident to a corner equals the degree of the crease junction. The jittered crease points are projected onto the wedges and corners. In a final step the feature lines and loops are converted to a spline representation f) by a least squares fitting approach.

Our feature detection and recovery approach is tailored as a preprocessing stage for surface reconstruction and optimized for fast running time. Two complexity measures influence the overall running time: the total number n of input points from the point cloud and the number m of output points on the feature patterns. As the point cloud describes a surface and the feature patterns sets of lines, m is of the order $O(\sqrt{n})$. It is obvious that we have to consider every input point as we do not know, where the features are and that therefore our algorithm has to be at least linear in n. Our approach tries to keep the computational cost per input point as low as possible by computing only some penalty functions, which reject instantly most of the edges in the neighbor graph. In this way the running time for the computation of the minimum spanning graph already depends on m and not on n.

1.1 Related Work

The feature extraction problem is closely related to surface reconstruction, which has important applications in laser range scanning, scientific computing, computer vision, medical imaging, and computer assisted surgical planning. Our algorithm addresses some of the problems that arise in surface reconstruction of datasets that contain creases and corners [19, 9, 18, 6, 3]. For these approaches it is very helpful to extract the feature lines and junctions beforehand.

Most of the reconstruction algorithms that can accommodate creases and corners do so either implicitly or in a post processing step. The graph-based surface reconstruction of Mencl and

Müller [15] handles sharp edges by maximizing for each point the sum of dihedral angles of the incident faces. The optimization is performed only locally and depends on the order in which the input points are processed. Especially in degenerate cases this approach can produce "crispy" crease lines *i.e.*, lines that are broken by notches. A problematic constellation of points that often arises at sharp edges occurs when two points on the crease form an equilateral tetrahedron with two points on different sides of the crease. There is no guarantee that the crease edge is preferred over the edge connecting the two points on the two sides. The tetrahedron described by the problematic points need not be exactly equilateral as the locations of the input points could accidentally favor constellations that cut the crease. Adamy et al. [2] described corner and crease reconstruction as a post processing step of finding a triangulation describing a valid manifold surface. The previously described case is again a problem. Both constellations for triangulating the four points on a near equilateral tetrahedron are perfectly manifold and there is no reason why the crease edge should be preferred over the crease cutting edge. The power crust algorithm of Amenta et al. [16] treats sharp edges and corners also in a post processing step. However, they reconstructed the features by extending the surfaces adjacent to the features and calculating the intersections. This idea is quite similar to our approach but their simple approach of surface extension does not tolerate noisy data. Dey and Giesen [8] propose a method to detect undersampled regions, which allows the detection of border loops, creases and corners. The surface near the features is reconstructed in a similar way as Adamy et al. proposed. Funke and Ramos [10] describe a surface reconstruction method based on a directional nearest neighbor search, that avoids the construction of a voronoi diagram and runs in near linear time. So far their algorithm can only

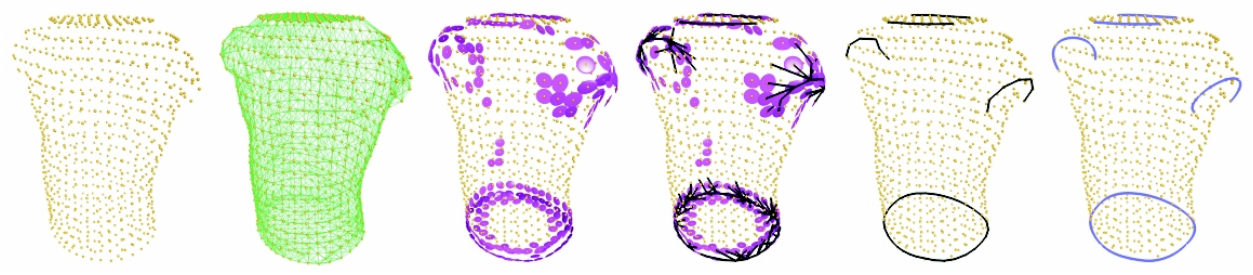


Figure 2. a) input point cloud. b) the neighbor graph. c) analysis of point neighborhoods. d) crease pattern forming. e) pruning of crease pattern. f) spline representation of crease pattern

handle surfaces without features.

In the computer vision literature there exist quite old approaches for depth images that can handle features correctly and extract the feature lines. The weak constraint optimization technique by Blake and Zisserman [7] used a non linear fitting approach. The global energy functional, which was minimized, contained a term that penalizes the generation of a crease line in order to avoid permanent creation of creases. This approach can also reconstruct the crease lines and junctions. Sinha and Schunk [17] fit spline patches to depth images and adjusted the spline parameters such that the spline patches could bend sharply around creases.

Guy and Medioni [12] described a robust algorithm to extract surfaces, feature lines and feature junctions from noisy point clouds. They discretized the space around the point cloud into a volume grid and accumulated for each cell surface votes from the data points. From the accumulated votes they defined a saliency function for junctions and a combined scalar and vector valued saliency function for the crease lines. Junctions are simply global maxima of the saliency functions and crease lines are extracted with a modified marching cubes algorithm. One of our goals was to avoid the discretization into a volume grid in order to allow for efficient handling of non uniformly sampled point clouds.

1.2 Paper Overview

The paper is partitioned into three different sections corresponding to the stages of the feature extraction pipeline. Section 2 describes the analysis stage, where the neighbor graph is constructed. In section 3 we explain the feature detection stage in detail. The recovery of crease lines and junctions is the topic of section 4. Applications and results are presented in section 5 before we end with a brief discussion in section 6.

2. ANALYSIS

The analysis phase of this approach consists of computing a neighbor graph on the input points and estimating for each point the sampling density. The neighbor graph connects points that are probably close on the underlying surface. On the one hand, the neighbor graph will be used to find the neighborhood of each data point, which allows for fast local computations. On the other hand it serves as the domain for detecting the feature line patterns.

2.1 Neighbor Graph

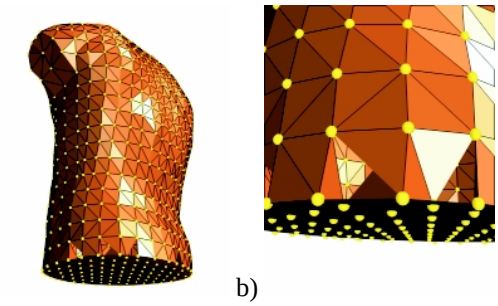
In most cases we use the Riemannian Graph as neighbor graph. The Riemannian graph contains for each data point the edges to the k nearest neighbors, where we chose k between 10 and 16. For our purposes we could compute a sufficient approximation to the Riemannian graph with the approximate nearest neighbor library ANN [4] in near linear time. For the brain dataset the Riemannian graph was computed in 7 to 10 seconds for $k = 10$ to 16. The Riemannian graph does not minimize the number of edges in the neighbor graph.

To achieve this we propose a delaunay filtering approach that has a worse asymptotic running time. It is practical for data sets with a size up to fifty thousand points. First we computed the Delaunay tetrahedralization of the input point set with the public domain software package Qhull [5]. Then we followed the approach of Adamy et al. [2] and first filtered out all triangles of the Gabriel complex. These are the triangles, which do not contain any fourth point within their minimum circumsphere. Triangles not in the Gabriel complex are most probably not part of the surface because a fourth data point is close to these triangles and the surface more likely passes through this point. The Gabriel triangles can be found by investigating the incident tetrahedra. If there is only one incident tetrahedron, we check if the center of its circumsphere is

on the same side of the triangle as the fourth point of the tetrahedron. In case of two incident tetrahedra, we check if their circumsphere centers lay on different sides of the triangle.

The second step of the algorithm is to filter out a reasonable subset of the Gabriel complex, which describes the surface. We used two triangle filters. The first was a simplified version of Adamy et al.'s umbrella filter, which ensures that the triangles incident on each data point form at least one closed fan. To find a good guess for this fan, the Gabriel triangles incident on a point are sorted according to a parameter, which gives a reasonable measure of whether the triangle should be part of the surface. For each data point, triangles are extracted from the sorted list, until the first fan is completed at which point these triangles are validated as part of the output of the filter. We implemented the fan detection and identification with a modified version of a disjoint set data structure that did not prune the trees of the connected sets.

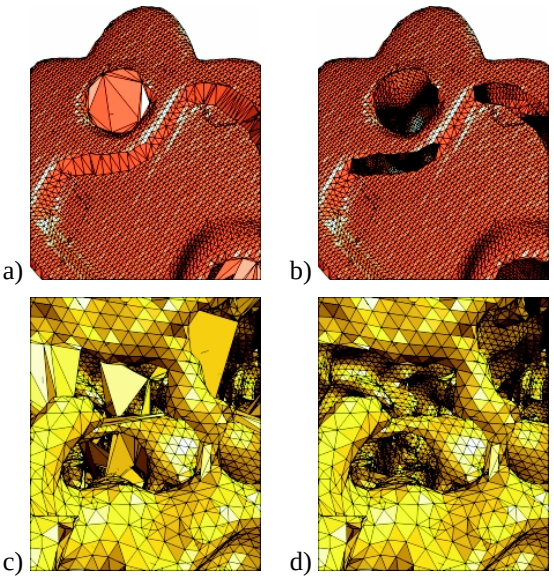
The simplified umbrella filter closes the holes of the model and connects different surface parts (see figure 4 a) and c)). The aim of the second filter was to recover the holes and to disconnect distant surface parts simply by eliminating triangles that have extraordinarily long edges. For this, we first computed the average edge length at each data point. Then we removed all triangles which contained an edge that was three times longer than the average edge length of the two incident vertices. In this way we recovered holes and disconnected nearby but disjoint surface parts (see figure 4 b) and d)).



a) b)

Figure 3. a) triangles recovered by Delaunay filtering, b) problems at creases

The Delaunay filtering has the advantage that it helps to disconnect data points that lay on different surface parts. Our simplified filter did not serve for surface reconstruction, in cases in which the models contained sharp edges as figures 3 a) and b) illustrate. But this is not important for our purpose as we detect the creases later on and only need the neighbor information contained in the produced triangles. The two Delaunay filters are relatively fast and consumed for the brain dataset of figure 13 with 84,000 points about 10 seconds, which is about one third of the time consumed by Qhull.



a) b) c) d)

Figure 4. Effects of filtering long triangles: a) closed holes before filter, b) holes recovered, c) inter-surface connections before filter, d) surface separated

2.2 Sampling Density

In the following computations it was often important to relate the computed quantities to the sampling density. Most often we needed the sampling density in units of length. Therefore we defined the average distance μ_i of a data point p_i with the set N_i of direct neighbor points in the neighbor graph as

$$\mu_i : \frac{1}{|N_i|} \sum_{q \in N_i} |p_i - q| \qquad (1)$$

3. FEATURE DETECTION

In this section we describe how to find the crease and border patterns in three steps as illustrated in figure 5:

1. compute point penalty functions

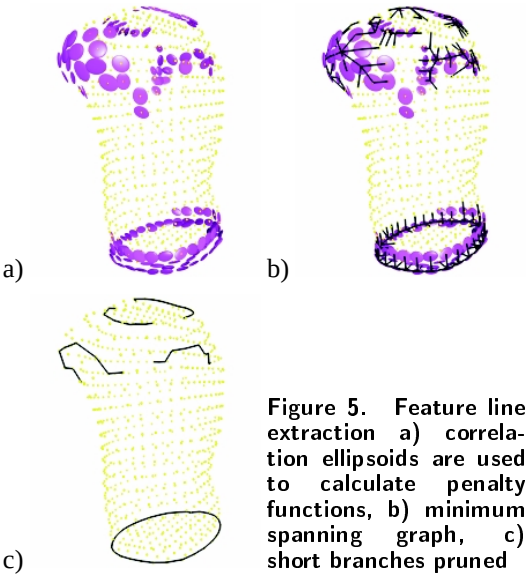


Figure 5. Feature line extraction a) correlation ellipsoids are used to calculate penalty functions, b) minimum spanning graph, c) short branches pruned

2. compute minimum spanning graph
3. prune pattern

The first step (figure 5 a)) analyzes the neighborhood of each data point. The points are not explicitly classified into surface, crease, corner or border points, but penalty functions are computed that describe how far the point is off a crease, corner or border point. The second step (figure 5 b)) transfers the point penalty functions to the edges of the neighbor graph and combines the edge penalty functions with the edge lengths to edge weights. These weights are used to compute a modified minimum spanning tree – the so called *minimum spanning graph (MSG)*. The MSG allows the creation of cycles if they are sufficiently long. This is an essential ingredient to be able to extract feature patterns with loops. In the third step (figure 5 c)) the short branches of the MSG are pruned away.

3.1 Point Penalty Functions Calculation

For each data point p_i we want to exploit the information in the nearest neighbors to compute penalty functions that estimate how unprobable it is that the point is close to a feature. The extend of the considered neighborhood depends on the noise level of the dataset. For data with low or no noise it is sufficient to gather the direct neighbors of the data points while for highly noisy data one must consider all neighbors with a graph distance less than three, four or five edges depending on the noise level.

Let us denote the set of neighbors by N_i. We first compute the center location c_i and the correlation matrix C_i of the neighborhood given by

$$\mathbb{R}^3 \ni c_i \stackrel{\text{def}}{=} \frac{1}{|N_i|} \sum_{q \in N_i} q \tag{2}$$

$$\mathbb{R}^{3\times 3} \ni C_i \stackrel{\text{def}}{=} \frac{1}{|N_i|} \sum_{q \in N_i} (q - c_i)(q - c_i)^t, \tag{3}$$

before we calculate the different penalty functions for each point. The eigenvectors $\{e_0, e_1, e_2\}$ of the correlation matrix together with the corresponding eigenvalues $\{\lambda_0, \lambda_1, \lambda_2\}$, where $\lambda_0 \leq \lambda_1 \leq \lambda_2$, define the *correlation ellipsoid* that adopts the general form of the neighbor points. Hoppe [13] used the smallest eigenvalue of the correlation matrix to determine the normal direction of the surface. Similar to Guy and Medioni [12] we use the eigenvalues to measure the probability of feature lines. Figure 6 shows the different shapes, that we are interested in. It shows in green color the surface type at the current data point: plane, crease wedge, border half plane and corner. The data point is the yellow ball and the correlation ellipsoid is depicted around the centroid c_i in violet.

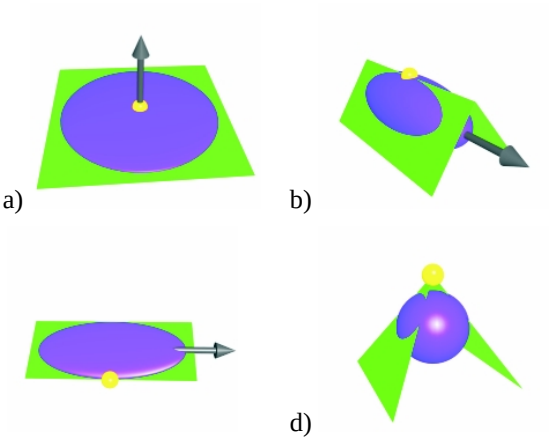


Figure 6. shape of the correlation ellipsoid around different types of data points: a) surface point, b) crease point, c) border point, d) corner point

For a point on a flat surface the ellipsoid degenerates to a pancake with $\lambda_1 \approx \lambda_2$ and $\lambda_0 \approx 0$. The unit vector e_0 points in normal direction, illustrated by the arrow in figure 6 a). The normal derived in this way is actually the surface normal of a plane fitted to the set of neighbors, that minimizes the squared distance to the neighbors. The optimal plane must go through the centroid c_i. The equivalence of the different normal estimations can be seen easily by looking briefly

at the least squares problem. Suppose $\hat{n}$ is the normal of the optimal plane. Then we want to minimize $\sum_{q \in N_i} (\hat{n}^t (q - c_i))^2$ under the constraint $\hat{n}^t \hat{n} = 1$. By the method of the Lagrange multipliers we need to minimize

$$\begin{aligned} \sum_{q \in N_i} \hat{n}^t (q - c_i)(q - c_i)^t \hat{n} \quad &+ \quad \lambda(1 - \hat{n}^t \hat{n}) \\ &= \quad \hat{n}^t (C_i - \lambda I) \hat{n} + \lambda \end{aligned}$$

with the Lagrange multiplier λ. Notice that vv^t denotes the outer vector product, which results in a symmetric matrix, whereas $v^t v$ denotes the dot product. Setting the gradient for $\hat{n}$ of this equation to zero yields exactly the eigenvalue equation $C_i \hat{n} = \lambda \hat{n}$. The solution with the minimum residual is given by the eigenvector corresponding to the smallest eigenvalue.

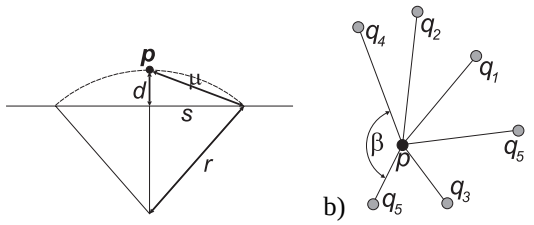


Figure 7. a) curvature estimation from fitted plane and average neighbor distance μ, b) determination of maximum open angle

For data with low noise we can estimate the average curvature from the least square fitted plane given by (e_0, c_i). Figure 7 a) shows a two dimensional projection of the data point p with its tangent plane degenerated to the horizontal line. The point has the distance $d = |e_0^t (p - c_i)|$ from its tangent plane. The curvature radius, shown as a dashed line in the figure intersects the tangent plane at approximately the average neighbor distance μ. From $s^2 = \mu^2 - d^2$ and $s^2 = r^2 - (r - d)^2$, the curvature $\kappa = 1/r$ computes to $2d/\mu^2$ and is a good criterion for detecting crease and corner points. We define the

curvature estimate κ_i for each data point p_i with distance d_i from the fitted plane as

$$\kappa_i \stackrel{\text{def}}{=} \frac{2 d_i}{\mu_i^2}; \tag{4}$$

By computing the maximum curvature estimation $\kappa_{\max}$ of all data points, we can define the curvature penalty function ω_κ as

$$\omega_\kappa(p) \stackrel{\text{def}}{=} 1 - \frac{\kappa(p)}{\kappa_{\max}}. \tag{5}$$

The correlation ellipsoid tells us more than the approximated normal direction. Figure 6 b) shows that in the case of a crease point the correlation ellipsoid is stretched in the direction of the crease and the eigenvalues obey $\lambda_0 \approx \lambda_1$ and $\lambda_0 + \lambda_1 \approx \lambda_2$. We encapsulate the information of how well the eigenvalues fit to the crease case together with the primary direction of the ellipsoid into a vector valued penalty function ω_{cr}

$$\vec{\omega}_{cr}(p) \stackrel{\text{def}}{=} \frac{\max\{\lambda_1 - \lambda_0, |\lambda_2 - (\lambda_0 + \lambda_1)|\}}{\lambda_2} \cdot e_2. \tag{6}$$

In the case of a border point, the ellipsoid degenerates to an ellipse. The smallest eigenvalue is still approximately zero and the corresponding eigenvector gives a good approximation of the surface normal. The other two eigenvalues obey $2\lambda_1 \approx \lambda_2$. The eigenvector e_2 of the biggest eigenvalue gives again the approximate direction of the border loop. This yields the vector valued border penalty function $\vec{\omega}_{b1}$

$$\vec{\omega}_{b1}(p) \stackrel{\text{def}}{=} \frac{|\lambda_2(p) - 2\lambda_1(p)|}{\lambda_2(p)} e_2(p). \tag{7}$$

With the help of the normal direction we can derive a second penalty function for the border. For this we project the point p together with its neighbors into an xy-coordinate system orthogonal to the normal direction, where p is the origin as shown in figure 7 b). Then we sort the neighbors according to their angle around p. Let β be the maximum angle interval, that does not contain any neighbor point. The bigger β is, the higher is the probability that p is a border point. The corresponding penalty ω_{b2} function is given by

$$\omega_{b2}(p) \stackrel{\text{def}}{=} 1 - \frac{\beta(p)}{2\pi}. \tag{8}$$

Finally, at a corner (see figure 6 d)) the correlation ellipsoid has no preference direction and all eigenvalues should be approximately the same. The corner penalty function ω_{co} is defined as

$$\omega_{co}(p) \stackrel{\text{def}}{=} \frac{\lambda_2(p) - \lambda_0(p),}{\lambda_2(p)}. \tag{9}$$

3.2 Minimum Spanning Graph Computation

In the previous section we defined several penalty functions for creases, corners and border loops. Now we exploit them to compute the minimum spanning graph, which contains a superset of the edges in the final feature patterns. In the case of crease pattern extraction we define the weight $w_c(e)$ of an edge $e = (p, q)$ with length $|e| = |q - p|$ and direction $\vec{e} = (q - p)/|e|$ as

$$\begin{aligned} w_c(e) \stackrel{\text{def}}{=} & \alpha(\omega_\kappa(p) + \omega_\kappa(q)) + \\ & (1-\alpha)\left(\min\left\{|\vec{\omega}_{cr}(p)^t\vec{e}|, = \omega_{co}(p)\right\} + \right. \\ & \left. \min\left\{|\vec{\omega}_{cr}(q)^t\vec{e}|, \omega_{co}(q)\right\}\right) + \\ & \frac{2|e|}{\mu(p)+\mu(q)}. \end{aligned}$$

We take the minimum of the directional penalty function $\vec{\omega}_{cr}$ and the corner penalty function ω_{co} in order to incorporate the corner penalty function what improves the locations of the detected crease corners. The factor α depends on the amount of noise in the data. For data sets with nearly no noise we choose $\alpha = 1/2$. In cases of high noise levels we reduce α to 0.2, as the curvature estimate is no longer very reliable. The edge length term that favors short edges is added in order to straighten the feature lines. The scaling of the edge length term does not significantly influence the output of the algorithm. Therefore, we refrained from adding another weighting parameter.

To detect border loops we define the edge weight $w_b(e)$ from the corresponding penalty weights of the incident points

$$\begin{aligned} w_b(e) \stackrel{\text{def}}{=} & \gamma(\omega_{b2}(p) + \omega_{b2}(q)) + \\ & (1-\gamma)\left(|\vec{\omega}_{b1}(p)^t\vec{e}| + |\vec{\omega}_{b1}(q)^t\vec{e}|\right) + \\ & \frac{2|e|}{\mu(p)+\mu(q)}. \end{aligned}$$

We chose the factor $\gamma = 1/2$, which produced closed border loops in all cases that we tested.

For the extraction of the MSG as shown in figure 5 b) we first compute the weights for all edges in the neighbor graph. Edges are not considered at all if the portion of the penalty derived from the vertex penalty functions exceeds a tolerance τ. The tolerance τ is a measure of the sensitivity of the crease detection. As all the penalty functions are normalized to the interval $[0, 1]$, it turns out that a value of $\tau = 1$ is appropriate. If the dataset contains blunt creases, τ will require some fine tuning by the user.

The remaining edges are entered into a queue. The MSG is built by considering the edge with the smallest remaining weight. A disjoint set data structure is used to determine whether this edge produces a cycle. If not, the edge is added to the crease pattern. In the case in which the edge under consideration produces a cycle in the crease pattern, we check if the produced cycle is longer than a user defined constant ρ. In that case this edge also becomes part of the crease pattern. Given the number of input points n a good guess for ρ is $\sqrt{n}/2$, which reflects the assumption that a cyclic crease should at least have the length of half of the diameter of the described object. The pseudo code for the first part of the feature line detection looks like this:

```
{ compute MSG }

iterate all edges e in neighbor graph
      if weight(e).penalty < τ then
            queue.insert(e, weight(e))

while not queue.empty() do
e = (p, q) := queue.extractMinimum()
      if disjointSet.find(p, q) or
         pattern.pathLonger(p, q, ρ) then
            pattern.addEdge(p, q)
```

The method `find(`p, q`)` in the disjoint set checks if the edge produces a cycle. If not, it joins the two connected components incident to p and q and returns **true**. Otherwise it returns **false** and the method `pathLonger(`p, q, ρ`)` of the graph data structure checks if the cycle is long enough. The latter method is time critical for large values of ρ. We implemented it with a breadth first search through the graph, in which we kept track of the visited vertices. We used counters instead of flags for the tracking of visited vertices in order to avoid expensive initializations for each breadth first search. In this way the search cost was $O(\rho^2)$ in the worst case. But the vast majority of edges are added to graph components

with diameter much smaller than ρ. In practice there is hardly any dependence of the run time on ρ. Even for the brain model with 84,000 vertices and a ρ of 150 the pattern was extracted after two seconds.

3.3 Pattern Pruning

At this point in the process, the extracted pattern looks like the one in figure 5 b). There are a lot of short branches that we have to remove. The basic idea is to remove all branches shorter than $\rho/2$ while still managing to take care of singleton ends, which we do not want to cut short. Thus in a first iteration over these points in the pattern that have more than

two incident edges, we check at each incident edge whether the incident branch is a sub-tree with maximum depth less than $\rho/2$. The check is performed with a similar breadth first searching algorithm as we used for `pathLonger`(p, g, ρ). If there are at least two branches with depth larger than $\rho/2$, we can safely remove all incident branches

with smaller depth without affecting singleton ends. The second step takes care of the singleton ends and finds for each tree at a singleton end the longest path with minimum edge weight. All other branches at the singleton end are removed. The final result of this process can be seen in figure 5 c).

4. FEATURE RECOVERY

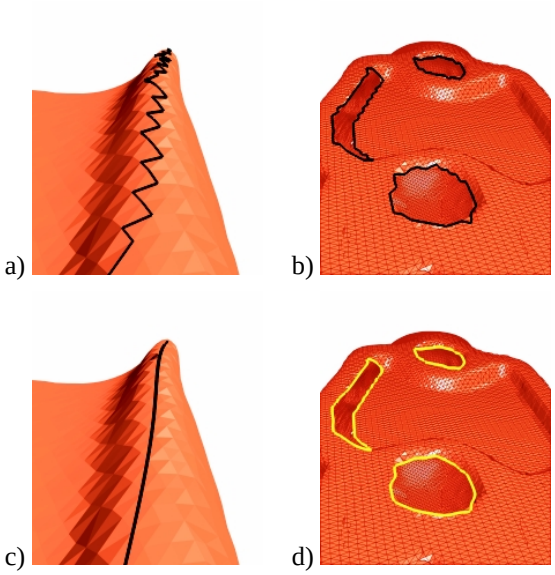


Figure 8. a) zigzag crease line, b) zigzag border line, c) smoothed crease, d) smoothed border loops

For datasets in which the data points lie directly on the feature lines, the above algorithm robustly extracts the crease and border loop patterns. For datasets scanned from real data with for example a laser range scanner even without any noise the probability that a data point lies on a feature line is extremely low. If the laser beam of the scanning device directly hits a feature line, the beam is dispersed and the measurement impossible. Thus we rather have to expect that there are no points directly on the feature lines. As a result of the lack of feature points, the output of the feature line detection algorithm is a zigzag line strip around the actual feature line. Figure 8 a) and b) give two examples of this behavior. A simple solution is sufficient for many applications. We just smooth the zigzag away by fitting a low degree spline to the feature lines. An example of the result from such an approach is shown in figures 8 c) and d). As the points on the crease lines are ordered, we can use a standard spline fitting procedure, which estimates the spline parameter values of the input points via a chordal length parameterization and then solves the resulting least squares problem. For this fitting, we have used the Nurbs++ library.

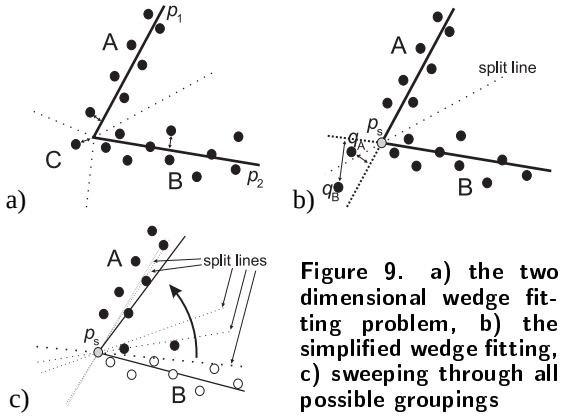


Figure 9. a) the two dimensional wedge fitting problem, b) the simplified wedge fitting, c) sweeping through all possible groupings

In the case of crease line reconstruction, we propose a better solution than spline fitting. The basic idea is to reconstruct the crease from a sufficiently large neighborhood and then move the points on the zigzag pattern line onto the reconstructed crease. The question is how to reconstruct the crease lines and corner locations at the crease junctions. Given a point close to a crease line and its neighborhood one would like to fit a wedge to the neighbor points. Figure 9 a) illustrates the fitting in two dimensions. The wedge is defined by two planes p_1 and p_2. The dotted line shows how the wedge splits the space into three regions A, B and C. All points in region A are closest to plane p_1, the ones in region B to p_2, and the points in region C are closest to the intersection line (or in two dimensions, the intersection point) of the two planes. This split into three regions makes a least squares fitting approach extremely difficult, as it is not known in advance which point will fall into which region after the fitting process. Thus the primary question is how to split the input points into regions.

The approach described in the next subsection splits the difficult three-dimensional problem into a number of simplified two-dimensional problems, where the grouping problem can be solved efficiently. A robust approximate solution to the three-dimensional grouping problem is then composed from the two-dimensional solutions via a voting process. In this way

we can exploit the information available from the feature detection stage to simplify the feature recovery. After we have derived a reliable grouping we can fit the wedges to the points in the crease neighborhood and project the potential crease points directly onto the crease.

This crease line recovery approach breaks down close to crease junctions as too many data points are incorporated into the fitting process. For this reason we first determine the point grouping around corners. As we exploit the solution strategy of the line recovery but in a more complicated manner, we describe the line recovery process first.

4.1 Feature Line Recovery

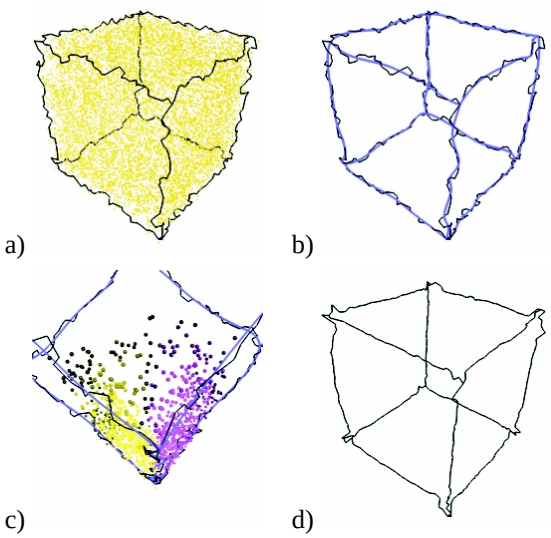


Figure 10. Crease Recovery a) detected crease pattern, b) smoothing through spline fitting, c) summed grouping weights along crease strip, d) points moved onto recovered crease

Figure 10 demonstrates the different steps of crease line recovery on a very noisy dataset. We generated this dataset by randomly distribution 10,000 points in a volume defined as the solid difference between a cube with edge length 2 and one with edge length 1.6, both centered around the origin. The result was a cube surface with 20% noise in all directions.

In order to project the wedge fitting problem for each crease point to the two dimensions case as shown in figure 9 a), we need to know the crease direction or a good approximation to it. As the detected feature lines are noisy, we use the spline approximation. From the spline fitting process we know for each crease point, p, the parameter value t_p of its counterpart on the spline $s(t)$. Thus we can compute the smoothed location $p_s \stackrel{\text{def}}{=} s(t_p)$ and crease direction $d_s \stackrel{\text{def}}{=} \dot{s}(t_p)/|\dot{s}(t_p)|$. We extend d_s to an orthogonal coordinate system with an xy plane orthogonal to d_s and project neighbor points of p to the xy plane arriving at the configuration described in figure 9 b).

The wedge fitting problem in two dimensions is still very difficult as we do not know the zenith location of the wedge. Thus we simplify the problem be assuming that p_s is the zenith of the wedge. Furthermore we split the space at the *split line* into only two regions A and B, as depicted in figure 9 b). The distances from the points in the previously defined region C to the wedge are underestimated (see neighbors q_A and q_B in figure 9 b) for two examples). This is not a serious problem for our application as these points are rare and their existence either implies that we detected the crease point wrong or that there is noise in the data.

The helpful property of the simplified wedge fitting problem is that no matter what the solution is, the points are split into two groups at a split line that goes through p_s. There are only n of these groupings, where n is the number of points in the neighborhood. We can enumerate them by sweeping the split line in a rotational motion through the neighbor points as illustrated in figure 9 c). The final solution is the one with the smallest residual.

Given the set of neighbor points N and a grouping $N_1 \dot{\cap} N_2 = N$, the solution of the simplified wedge fitting problem is found by solving two least square line fitting problems. To each neighbor group N_1 and N_2 we fit a line through the origin p_s. To avoid the $O(n)$ floating point operations per grouping, we prepare the input matrices

$$\mathbb{R}^{2\times 2} \ni m_j \stackrel{\text{def}}{=} \sum_{q \in N_j} qq^t \qquad j \in \{1,2\}$$

incrementally during the rotation of the split line. For the first split line location we compute the matrices explicitly. Each time the split line rotates one step further, only one point q changes from one group to the next. We subtract qq^t from the matrix of the group that q leaves and add it to the other matrix. As the matrices are symmetric and we are in two dimensions, the cost is only three multiplications and six additions. Given the matrix m_i, the normal to the optimal line is given by the eigenvector of m_i corresponding to the smallest eigenvalue. The following algorithm computes this eigenvector of a matrix m with three square root operations, one division, four multiplications, five additions and some sign and binary shift operations.

$a := (m_{11} - m_{22})/2;$
if $a = 0$ **and** $m_{12} = 0$ **then return** $(1,0)$
if $m_{12} = 0$ **then**
  if $a > 0$ **then return** $(0,1)$
    else return $(1,0)$
if $a = 0$ **then**
  if $m_{12} > 0$ **then return** $(\sqrt{2},-\sqrt{2})/2$
    else return $(\sqrt{2},\sqrt{2})/2$
$f := a/(2\sqrt{a^2+m_{12}^2})$
$c := \sqrt{1/2+f}$
$d := \mathrm{sgn}(m_{12})\sqrt{1/2-f}$
if $\mathrm{sgn}(a) = \mathrm{sgn}(d^2-c^2)$ **then return** (c,d)
  else return $(d,-c)$

Now we are in the position to efficiently group the neighborhoods of each crease point optimally according to the simplified wedge fitting problem. However, each single grouping might contain a few misgrouped data points, for example if the crease has a sharp turn. We want to combine as many groupings of adjacent crease points as possible. The maximum sub-graph of a crease pattern, where the neighboring points split into exactly two groups, is either a closed loop or a path connecting two crease points of degree unequal two, i.e. crease junctions and singleton ends. We call these maximum crease paths *crease strips*. For each crease strip we attach a weight field to each of the data point that is adjacent to at least one point on the crease strip. After initializing the weights to zero, we start a voting process at one end of the strip. We build the xy coordinate system orthogonal to the spline direction and find the optimal grouping of the neighbors that solves the simplified wedge fitting problem. The weights of neighbors are incremented for members of group A and decremented for group B members. The amount of the increment decreases with increasing distance from the wedge zenith. When

we move on to the next crease point on the strip, we choose x- and y-axes as close as possible to the previous axes, such that we can identify the groups A and B of the new grouping with the previous groups. In this way we sweep through the strip and increment and decrement weights for the neighbors. A sample weighting along one crease strip in the cube model is shown in figure 10 c). After all crease points have contributed to the weights, we classify the neighbor points into group A if their final weight is positive and into group B if the resulting weight is negative. If a weight is very close to zero we do not consider the corresponding neighbor point in the final recovery stage.

In the end we derive a grouping of the crease neighbors that exploits all the information we get from the feature detection stage. A final iteration through the points on the crease strip recovers the origin crease locations. The neighborhood of each point on the strip is now reliably split into two groups. Thus we can fit two planes to the two groups in the least square sense by computing and analyzing the correlation matrix for each group relative to the centroid of the group. The local crease approximation is the intersection of the resulting planes. We project the crease point orthogonal onto the crease approximation. Figure 10 d) shows the result, a more or less straight line for the cube dataset with 20% noise. There are still problems near the crease junctions because we do not exclude neighbors in opposite faces of the creases. The solution to this problem is described in the next section.

4.2 Feature Junction Recovery

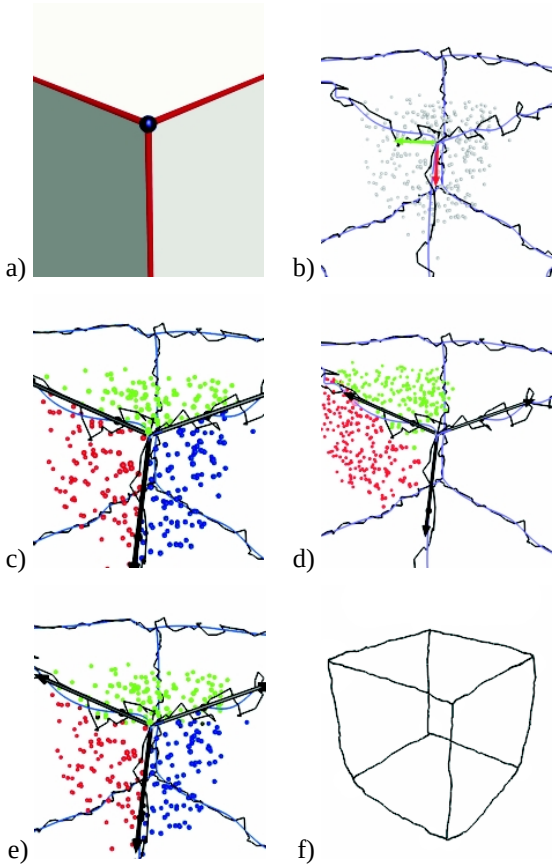


Figure 11. Junction Recovery a) a corner is given by a set of planes, b) finding a suitable 2d projection, c) initial grouping, d) grouping along crease strip, e) summed weights after strip grouping, f) corners and creases reconstructed

In order to find realistic crease lines and their intersection at crease line junctions, we have to solve the *corner fitting problem*. Suppose we have a junction

of k intersecting crease lines. The surface is split into k regions, which are planar near the junction as depicted in figure 11 a). The corner, which we want to fit to the data points, is given by the piecewise planar approximation of the surface at the junction. It is defined by the corner point and k plane normals. For a least squares fit of a corner to a set of data points we have to split the data points into k groups, such that the total residual of square distances is minimized, when k planes are fit to the k groups. Figure 11 c) shows such a grouping. This problem is more difficult than the wedge fitting problem and the number of different groupings increases exponentially with k, even if we can arrange the vertices in a linear order.

Figure 11 illustrates our approach to an approximate solution of the corner fitting problem. Suppose we start of with a crease junction point p_c detected in the feature detection stage. First we find a two dimensional projection of the problem, in which the points in the neighborhood N_c of p_c can be sorted around p_c according to their angle relative to an arbitrary x-axis in the projected space (figure 11 b)). We can imagine different approaches to do this. First we could fit an ellipsoid to the point set [1] and use the direction from the center to the junction point as the projection direction. A second approach would be to trace the normal direction, given from the correlation tensor, from the surrounding points to the junction. We used the following approach, which works very well in all test cases we evaluated. We selected the direction from the centroid of the neighborhood to the junction point as the projection direction.

An initial group of the neighbor points is derived from the spline approximation of the incident crease lines. We partition the two dimensional space into k angular partitions, which split the neighbor points from N_c into k groups as shown in 11 c).

To improve the grouping, we use the weighting idea described in the previous section. We trace along each of the k incident crease strips and solve the simplified wedge fitting problem without considering data points in the groups not incident to the strip (see figure 11 d)). Each data point in N_c accumulates distance-weighted group votes in k weight variables. Each strip is traced until no more votes are added to the points in N_c or the end of the strip is reached. After all incident strips have been traced, a new grouping is derived from the weights, an example of these steps is shown in figure 11 e). Each data point in N_c is assigned to the group with the maximum weight. Then we start the whole strip tracing and voting process over and over again until no data point changes the group anymore or a maximum number of iterations is reached. In all our examples the grouping terminated after two or three iterations.

Given the optimal grouping, we can fit k planes to each of the k groups and reconstruct the corner. For $k > 3$ the planes do not always intersect in a single point. Therefore, we again use a least squares approach to reconstruct the corner location. We minimize the square distance to all of the k planes. The tool to solve this minimization problem is also known as quadric error metrics, originally introduced by Garland and Heckbert [11] in mesh simplification to find optimal locations for the mesh vertices after an edge collapse operation. With the knowledge of the vertex grouping around the junction point, we can finally recover the crease close to the junction by not considering any of the data points of N_c that do not belong to the two groups of the crease strip.

Let us summarize. The feature recovery stage first groups the neighborhoods around crease junctions into as many groups as there are incident faces. From the grouping the corner locations are recovered. Then the crease lines close to the junctions are reconstructed with the knowledge about grouping near the junctions. Finally, the remaining parts of the crease lines are recovered with the method described in the previous section. Figure 11 f) shows the final result of the noisy cube.

5. APPLICATIONS & RESULTS

5.1 Surface Meshing

Our primary goal in developing the feature extraction algorithm was as the first stage of a point cloud mesh generation tool. The basic idea behind the point cloud mesh generator is to locally fit polynomial patches to the point cloud and directly generate a mesh with element sizes that are adapted to curvature and to an anisotropic sizing function. The mesh is generated with an advancing front algorithm. While the front advances a new local approximation is computed, whenever the front leaves the domain of the current polynomial patch. An important ingredient is again a neighbor graph that reflects the proximity along the surface. Problems arise as always at the crease lines, where a naively computed neighbor graph connects the different surface parts incident to the crease. This will diminish the quality of the fitted patch near the crease lines significantly. With the help of point grouping along the edges, as described in section 4.1 and 4.2, we can easily remove all edges that cut the creases and corners. Thus our feature extraction stage not only produces the feature lines for the point cloud mesher, but also a sophisticated neighbor graph.

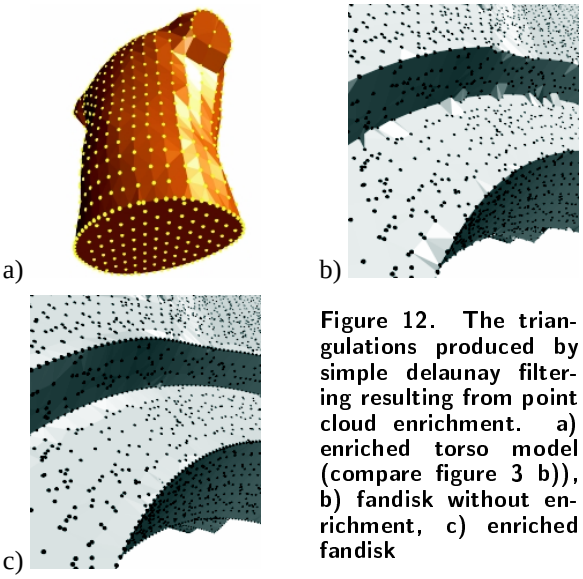


Figure 12. The triangulations produced by simple delaunay filtering resulting from point cloud enrichment. a) enriched torso model (compare figure 3 b)), b) fandisk without enrichment, c) enriched fandisk

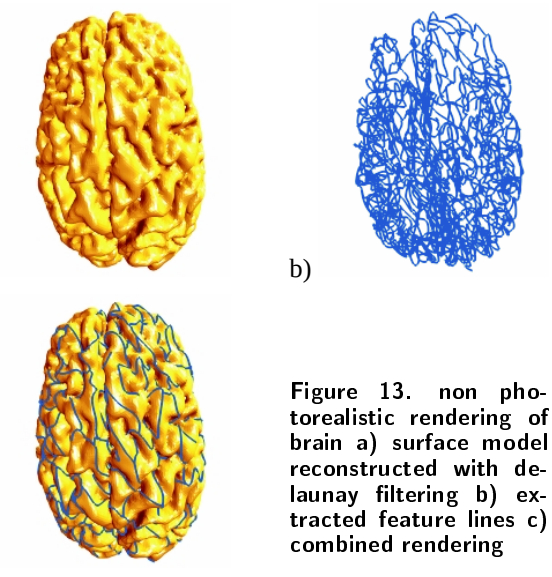


Figure 13. non photorealistic rendering of brain a) surface model reconstructed with delaunay filtering b) extracted feature lines c) combined rendering

5.2 Point Cloud Enrichment

A very simple but useful application of the feature extraction algorithm is point cloud enrichment. The goal here is to improve the resolution of feature lines by resampling the initial point set so that standard algorithms can now handle the features. Figure 12 shows two examples, the torso model and the fandisk. The torso is reconstructed correctly with the simplified delaunay filter. On the fan disk there are problems with a few triangles that connect two different sides of a crease. An example can be seen in the middle of the concave crease. As we group the vertices around creases during the feature recovery stage, we could add a third filter for the simplified surface reconstruction that removes triangles, that interconnected different sides of a crease. This was not implemented yet.

5.3 Non Photorealistic Rendering

In non photorealistic rendering [14] surface display is often enhanced along feature lines and lines of high curvature. Our first example of surface enhancement was the bunny model in figure 1, in which we singled out the crease and border lines. We did the same for the brain model as shown in figure 13.

5.4 Results

Figure 14 shows the extracted crease patterns of the torso, the fandisk and the bunny models. The crease patterns of the random cube and the brain can be seen in figure 11 f) and 13 c). The surface triangulations were generated with the Delaunay filtering approach described in section 2.1.

Table 1 gathers more information on the used parameters and the run time consumption of the different stages. All times were measured in seconds on a Pentium III 600 MHz. The capitalized columns in table 1 contain timing measurements. The second column contains the number of input data points. The next three columns document the analysis stage. We measured the time consumed by the Delaunay filtering for comparison only. The time is split into the run time of Qhull and the run time of the filters. For all models the Riemannian graph with the tabulated k was used

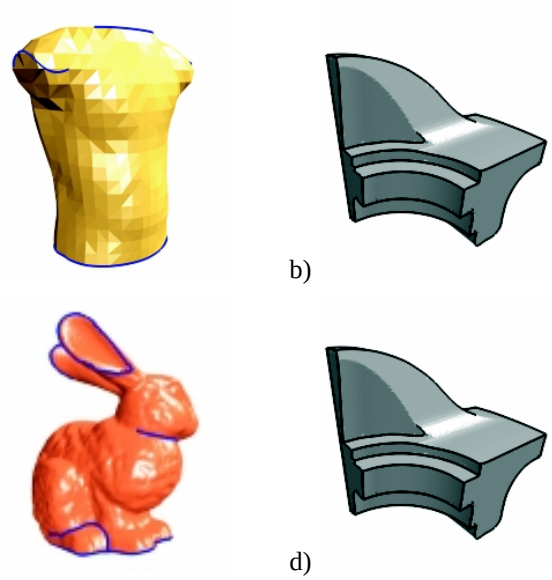


Figure 14. Crease patterns on torso, fandisk, bunny and frog model

Model		Analysis			Feature Detection				Feature Recovery		
name	vertices	DELAUNAY	RIEM.	k	α	γ	ρ	TIME	dist	COR.	STR.
torso	771	$0.15 + 0.05$	0.06	13	0.5	0.5	15	0.05	2	0	0.07
random cube	10,000	$2.5 + 1.2$	2	16	0.25	0.5	40	$10.2 + 1.4$	4	0.4	1.4
fandisk	16,477	$6.4 + 2.0$	3.13	13	0.667	0.5	40	2.9	2	1.3	1.8
bunny	34,835	$9.5 + 5.1$	2.7	10	0.667	0.5	40	$3.5 + 0.8$	2	0.8	2.3
brain	83,835	$26 + 10$	5.2	13	0.5	0.5	80	$10 + 3.5$	–	–	–

Table 1. Parameter values and run time measurements for the different feature extraction stages measured on different models. The times were measured in seconds on a Pentium III with 600 MHz.

as neighbor graph. The running time was for the larger models significantly less than for the Delaunay filtering. The next four columns contain the parameter values used for feature detection and the consumed time, which is split into the time consumed by the calculation of the point correlation matrix and penalty functions and the time consumed by the minimum pattern extraction and pruning. The calculation of the correlation matrices consumed most time for the cube dataset with a lot of noise as we used all neighbor points with graph distance up to three. The last three columns tabulate the feature recovery stage. First we specify the graph distance in which we analyzed the neighborhood around corners and crease strips. The last two columns contain the run times for the corner and strip recovery, respectively.

6. CONCLUSION

6.1 Summary

We have presented a new approach for feature extraction of crease and border patterns from point clouds. The proposed method is reasonably fast, robust against noise, and adapts easily to non uniformly sampled point clouds. For noisy data or data that is undersampled on the crease lines, a least square based feature recovery algorithm allows us to project data points that are close to the actual crease junctions and lines onto the features. These properties make our approach an excellent preprocessing step for surface reconstruction algorithms. Other applications like point cloud enrichment and non photorealistic rendering have been described.

6.2 Discussion & Future Work

We presented efficient solutions to the wedge and corner fitting problem. The information from the feature detection stage was exploited to simplify the problems. In future work we plan to determine whether the feature detection and recovery stages can be combined to make the algorithm even more robust.

Some datasets contain isolated peeks without any creases. For example, the well known cow dataset has two peaks at the horns. These peaks are singleton points in the crease pattern and not detected by our algorithm. Their neighborhood is also quite difficult to mesh with polygonal faces. We want to investigate how to fit cones to the peaks and how to incorporate them into polygonal mesh representations.

ACKNOWLEDGEMENTS

This work was supported by the National Institutes of Health [NCRR Grant #5-P41-RR12553-02].

REFERENCES

[1] R. Fisher A. Fitzgibbon, M. Pilu. Direct least-square fitting of ellipses. In *International Conference on Pattern Recognition*, August 1996.

[2] U. Adamy, J. Giesen, and M. John. New techniques for topologically correct surface reconstruction. In *Proceedings Visualization 2000*, pages 373–380, 2000.

[3] M.-E. Algorri and F. Schmitt. Surface reconstruction from unstructured 3D data. *Computer Graphics Forum*, 15(1):47–60, March 1996.

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *JACM: Journal of the ACM*, 45, 1998.

[5] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, December 1996.

[6] E. Bittar, N. Tsingos, and M.-P. Gascuel. Automatic reconstruction of unstructured 3D data: Combining medial axis and implicit surfaces. *Computer Graphics Forum*, 14(3):C/457–C/468, September 1995.

[7] A. Blake and A. Zisserman. Invariant surface reconstruction using weak continuity constraints. In *Proc.*

IEEE Computer Vision and Pattern Recognition, July 1985.

[8] T. K. Dey and J. Giesen. Detecting undersampling in surface reconstruction. *Proc. 17th ACM Symposium Computational Geometry*, pages 257–263, June 2001.

[9] P. Fua and P. Sander. Reconstructing surfaces from unstructured 3d points. In *Image Understanding Workshop*, January 1992. San Diego, California.

[10] S. Funke and E. A. Ramos. Surface reconstruction in almost linear time under locally uniform sampling. *Proc. European Workshop on Computational Geometry*, March 2001.

[11] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH'97 Conference Proceedings*, pages 209–216, 1997.

[12] G. Guy and G. Medioni. Inference of surfaces, 3d curves and junctions from sparse, noisy 3d data, 1997.

[13] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):71–78, July 1992.

[14] J. Lansdown and S. Schofield. Expressive rendering: a review of nonphotorealistic techniques. *IEEE Computer Graphics and Applications*, 15(3):29–37, May 1995.

[15] R. Mencl and H. Müller. Graph-based surface reconstruction using structures in scattered point sets. In *Proceedings of the Conference on Computer Graphics International 1998 (CGI-98)*, pages 298–311, Los Alamitos, California, June 22–26 1998. IEEE Computer Society.

[16] S. Choi N. Amenta and R. Kolluri. The power crust. In *to appear in ACM Symposium on Solid Modeling and Applications*, 2001.

[17] S. S. Sinha and B. G. Schunck. A two-stage algorithm for discontinuity-preserving surface reconstruction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):36–55, January 1992.

[18] R. Szeliski and D. Tonnesen. Surface modeling with oriented particle systems. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):185–194, July 1992.

[19] D. Terzopoulos and D. Metaxas. Dynamic 3D models with local and global deformations: Deformable superquadrics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-13(7):703–714, July 1991.