数理解析研究所講究録
1185巻 2001年 36-52

Detecting Undersampling in Surface Reconstruction

Tamal K. Dey Joachim Giesen *

Abstract

Current surface reconstruction algorithms perform satisfactorily on well-sampled, smooth surfaces without boundaries. However, these algorithms have severe problems with undersampling. Cases of undersampling are prevalent in real data since often they sample a part of the boundary of an object, or are derived from a surfaces with high curvature. In this paper we present an algorithm to detect the regions of undersampling. This information can be used to reconstruct surfaces with boundaries, and also to localize sharp features in nonsmooth surfaces. We report the effectiveness of the algorithm with a number of experimental results. Theoretically, we justify the algorithm with some mild assumptions that are valid for most practical data.

Keywords: Reconstruction, surface, sampling, boundary, Voronoi diagram, triangulation

1 Introduction

Many applications in CAD, computer graphics and scientific computations involve approximating a surface from its samples. A piecewise linear approximation to the surface which is sought in *surface reconstruction* is often appropriate for visual aids. They also form the control net for generating limit surfaces with higher continuity using subdivision methods [23].

The two dimensional version of the problem, namely *curve reconstruction* has been well studied in the literature [2, 5, 11, 12, 13, 18, 19, 22]. Among the algorithms proposed in the literature for surface reconstruction, the earlier ones [6, 7, 9, 15, 21] concentrated on the empirical results and did not focus so much on the theoretical guarantees. Edelsbrunner [14] reports the development of a commercial software under propriety rights which is based on the ideas of α-shapes. Very recently, starting with the algorithm of [1] three algorithms have been proposed with the guarantee that the output surface is homeomorphic and geometrically close to the sampled surface. They are the CRUST algorithm by Amenta and Bern [1], the CO-CONE algorithm by Amenta, Choi, Dey and Leekha [4], and the natural neighbor algorithm by Boissonat and Cazals [8]. The theoretical guarantee provided by these algorithms requires that the given data sample a surface densely. All these algorithms run into serious trouble if this condition is not met. In practice, the data may sample only part of a surface densely. This may be intentional for introducing boundaries, may be accidental for high curvature regions, or may be unavoidable due to non-smoothness.

In this paper we present an algorithm that detects the regions of undersampling under some assumptions that are reasonable for most practical data. This information is used in the co-cone algorithm [4] to reconstruct the surface patches that are well sampled. The main idea of detecting the locality of undersampled regions is to consider the structure of the Voronoi cells as indicated in [3]. We mature this idea with new definitions and assumptions and present a proof that the algorithm finds undersampled regions. Our algorithm finds immediate application

*Department of CIS, Ohio State University, Columbus, OH 43210.
e-mail: {tamaldey,giesen}@cis.ohio-state.edu

in reconstructing surfaces with boundaries and also in detecting sharp features in non-smooth surfaces. To our knowledge, boundaries and non-smoothness pose serious difficulty in surface reconstruction, and none of the known algorithms can handle them. We present experimental results with several data sets. The algorithm detects the boundaries in these test cases. It also points out the regions of high curvature where the surface is undersampled. This information can be used in a repair phase that fills up the "holes".

2 Definitions

2.1 Sampling

We base our algorithm on the notion of ε-sampling of smooth surfaces as introduced in [1]. The *medial axis* of a smooth surface $F \subset \mathbb{R}^3$ is the closure of the set of points that have more than one closest point on F. The *local feature size* $f(p)$ at a point $p \in F$ is the least distance of p to the medial axis. The *medial balls* at p are defined as the balls that touch F tangentially at p and are centered on the medial axis. A point set $P \subset F$ is called an ε-sample of a surface F if every point $p \in F$ has a sample within distance of $\varepsilon f(p)$.

Well-sampled patch: Let $S \subseteq F$ be a surface patch such that each point $x \in S$ has a sample within $\varepsilon f(x)$ distance and S is maximal in the sense that no other point $y \notin S$ has this property. Notice that S may have several components. Boundaries of S coincide with the boundaries of undersampled regions in F. Our goal is to recognize the boundaries in S and reconstruct it from sample P.

2.2 Boundaries

For any compact surface S we can distinguish interior points from the boundary points. An interior point has a neighborhood homeomorphic to the open disc $\mathbb{D}^2 = \{x \in \mathbb{R}^2 : |x| < 1\}$. A boundary point, on the other hand, has a neighborhood homeomorphic to the halfdisc $\mathbb{D}^2 \cap \mathbb{H}^2_+$ which is open in the halfspace $\mathbb{H}^2_+ = \{(x, y) \in \mathbb{R}^2 : x \geq 0\}$.

For the reconstruction of S only the finite set of sample points P is given that well samples S. Even though all points in P may be interior points of S, the existence of a non empty boundary should reflect itself in the sample points. We are looking for a definition of interior and boundary points for a finite sample from a surface that captures the intuitive difference between interior and boundary points. We base our definition on the restricted Voronoi diagram as introduced in [16].

Restricted Voronoi diagram: Let P be a sample of a compact surface S with or without boundary embedded in $\mathbb{R}^3$. Denote the Voronoi diagram of P by V_P. The restriction of V_P on the surface S defines the *restricted Voronoi diagram* containing the *restricted Voronoi cells* $V_{p,S} = V_p \cap S$.

The dual of the restricted Voronoi diagram defines the *restricted Delaunay triangulation* $D_{P,S}$. Specifically, an edge pq is in $D_{P,S}$ iff $V_{p,S} \cap V_{q,S} = \emptyset$; a triangle pqr is in $D_{P,S}$ iff $V_{p,S} \cap V_{q,S} \cap V_{r,S} = \emptyset$ and a tetrahedron $pqrs$ is in $D_{P,S}$ iff $V_{p,S} \cap V_{q,S} \cap V_{r,S} \cap V_{s,S} = \emptyset$. Edelsbrunner and Shah [16] showed that if each restricted Voronoi cell (defined recursively with dimensions) is a closed ball, and the same condition holds for the boundary of S, then S is homeomorphic to $D_{P,S}$. This leads us to define the *neighborhood* of a sample point p as $V_{p,S}$.

Using this definition of neighborhood we define interior vertices. All vertices that are not interior are called *boundary vertices.*

Definition 1 (Interior and boundary vertices) *A sample point p from a sample P of S is called interior vertex if $V_{p,S}$ does not contain a boundary point of S. Sample points that are not interior are called boundary vertices.*

Arguments of [1] can be used to show that the neighborhoods of interior vertices are homeomorphic to disks if P is an ε-sample of S with $\varepsilon < 0.1$.

2.3 Flat vertices

Our goal is to detect boundary vertices algorithmically and to exploit this information for surface reconstruction. Given only a finite sample P from a surface S embedded in $\mathbb{R}^3$ we cannot construct the restricted Voronoi diagram $V_{P,S}$, because the surface S is unknown to us. Therefore we cannot exploit our definition of interior and boundary vertices algorithmically.

To cope with this problem we define a *flatness* condition and show that boundary vertices cannot be flat whereas interior vertices with well sampled neighborhoods are flat. This definition uses the definitions of *poles* and the *width* and *heights* of Voronoi cells. The benefit we get from the definition of flat vertices is that it can be exploited algorithmically.

Ray: In what follows we will denote the ray from p to y with $\vec{y}$ for any sample point p and any point $y \in V_p$.

Angles: We use the notation $\angle(\vec{v_1}, \vec{v_2})$ to denote the acute angle between the lines supporting two vectors $\vec{v_1}$ and $\vec{v_2}$.

Poles: Given a finite point set $P \subset \mathbb{R}^3$, let V_p be the Voronoi cell of a point $p \in P$. We borrow the definition of poles from [1]. The vertex v_p^+ of the Voronoi cell V_p which is farthest from the point p is called the positive *pole* of V_p. Assuming that P does not lie on a plane, v_p^+ always exists. The negative pole of V_p is the farthest point $v_p^- \in V_p$ from p such that $\vec{v_p^-} \cdot \vec{v_p^+} < 0$. Basically, negative pole is the farthest point in V_p in the "opposite direction" of the positive pole.

Co-cone: The set $C_p = \{y \in V_p : \angle(\vec{y}, \vec{v_p^+}) \geq \frac{3\pi}{8}\}$ is called the co-cone of p. Basically, C_p is the complement of a double cone (clipped within V_p) centered at p with an opening angle $\frac{3\pi}{8}$. See Figure 1 for an example of a co-cone.

Co-cone neighbors: Given a finite sample P, the set $N_p = \{q \in P : C_p \cap V_q \neq \emptyset\}$ is called the set of co-cone neighbors of p.

Width and height of Voronoi cells: The *width* $w(V_p)$ of the Voronoi cell V_p of p is defined as the radius of C_p, i.e., $w(V_p) = \max\{|py| : y \in C_p\}$, see also Figure 1. Let $h^+(V_p)$ and $h^-(V_p)$ denote the lengths $|pv_p^+|$ of $\vec{v_p^+}$ and $|pv_p^-|$ of $\vec{v_p^-}$ respectively. Define the *height* $h(V_p)$ as $\min(h^+(V_p), h^-(V_p))$.

Now, we are prepared to define flatness, which depends on two parameters ρ and θ.

Definition 2 (Flatness) *A sample point $p \in P$ is called flat if the following two conditions hold:*

1. *Ratio condition:* $\rho w(V_p) \leq h(V_p)$
2. *Normal condition:* $\forall q$ *with* $p \in N_q$, $\angle(v_p^+, v_q^+) \leq \theta$.

Ratio condition imposes that the Voronoi cell V_p is long and thin in the directions of the pole vectors $\vec{v_p^+}$ and $\vec{v_p^-}$. The normal condition stipulates that the direction of elongation of V_p matches

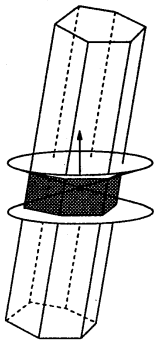


Figure 1: A Voronoi cell together with the normalized pole vector and the co-cone (shaded).

with that of any vertex whose co-cone neighbor is p. In the proof we use $\rho = \frac{1}{1.3\varepsilon}$ and $\theta = 0.14$ radians.

3 Assumptions and Theorems

Our goal is to exploit the definition of flat vertices in a boundary detection algorithm. In Theorem 1 we prove that interior vertices with well sampled neighborhoods are flat. In Theorem 2 we prove that the boundary vertices cannot be flat. These two theorems form the basis of our boundary detection algorithm. We omit the proofs of these two theorems in this version. See [10] for details.

In the proofs we assume $\varepsilon \leq 0.01$, $\rho = \frac{1}{1.3\varepsilon}$ and $\theta = 0.14$ radians. With these values we can show that interior vertices satisfy the ratio condition. However, the normal condition may not hold for all of them. Nevertheless, we can show that a subset of interior vertices that have well sampled neighborhoods satisfy the normal condition. To be precise we introduce the following definition.

Definition 3 (Deep interior vertex:) *An interior vertex p is called deep if it does not have any boundary vertex as Voronoi neighbor.*

Theorem 1 *All deep interior vertices are flat.*

For Theorem 2 we need some boundary assumptions. The first assumption (i) says that boundary vertices remain as boundary even if S is expanded with a small collar around its boundary, and the assumption (ii) stipulates that the boundaries be "well separated".

Assumption 1 (Boundary assumption)

i. The restricted Voronoi cell of a boundary vertex is a disk. The surface patch $S' \supset S$ with the condition that any $x \in S'$ has a sample within $\delta f(x)$ distance for some $\delta > \varepsilon$ defines the same set of boundary vertices as S does. We will need $\delta = 1.3\Delta\varepsilon$ for our proofs, where Δ is defined as $\max\{\frac{h(V_p)}{f(p)}\}$.

ii. Neighborhood of each boundary vertex intersects the neighborhood of at least an interior vertex.

Theorem 2 *Boundary vertices cannot be flat.*

Remark. It is interesting to note that one can assume the surface F to have small features around any sample point to render it as a boundary point in our definition. After all, the surface F is unknown to us. This should reflect in our assumptions and proofs. Observe that, the smaller the feature size gets, the larger gets Δ as $\Delta = \max\{\frac{h_p}{f(p)}\}$. In order for boundary assumption to be valid, we need δ to be fixed even though Δ increases. This requires ε to decrease as $\delta = 1.3\Delta\varepsilon$. But, decreasing ε would require a larger value of ρ. Thus, indeed larger value of ρ are needed to detect more vertices as boundary.

4 Boundary detection

The algorithm for boundary detection first computes the set of interior vertices, R, that are flat. It uses two parameters ρ and θ to check the ratio and normal conditions. In theory, we require $\rho = \frac{1}{1.3\varepsilon}$ and $\theta = 0.14$. However, in our implementation we obtain good results for smaller ρ and larger θ. Theorem 1 and Assumption 2 guarantee that R is not empty. In a subsequent phase R is expanded to include all interior vertices in an iterative procedure. A generic iteration proceeds as follows. Let p be any co-cone neighbor of a vertex $q \in R$ so that $p \notin R$ and V_p satisfies the ratio condition. If v_p^+ and v_q^+ make small angle up to orientation, i.e., if $\angle(\vec{v_p}^+, \vec{v_q}^+) \leq \theta$, we include p in R. If no such vertex can be found, the iteration stops. We argue that R includes only and all interior vertices at the end. The rest of the vertices are detected as boundary ones.

The following routine ISFLAT checks the conditions stated in Definition 2 to detect flat vertices. The input is a sample point $p \in P$ with a parameter ρ that measures the height vs. width ratio for V_p. The return value is `true` if p is a flat vertex, and `false` otherwise. The routine BOUNDARY uses ISFLAT to detect the boundary vertices.

```
ISFLAT (p ∈ P, θ, ρ)
1  compute the width w(V_p) and the height h(V_p)
2  if ρw(V_p) ≤ h(V_p)
3      if ∀q with p ∈ N_q : ∠(v_p^+, v_q^+) ≤ θ
4          return true
5  return false

BOUNDARY (P, θ, ρ)
1  R := ∅
2  for all p ∈ P
3     if ISFLAT(p) R := R ∪ p
4  while ∃p ∉ R and ∃q ∈ R with p ∈ N_q,
       and ρw(V_p) ≤ h(V_p) and ∠(v_p^+, v_q^+) ≤ θ
5     R := R ∪ p
6  return P \ R
```

4.1 Justification

We can show that BOUNDARY computes all and only boundary vertices if we make the following assumption. Notice that this assumption is valid for most practical data. The proof of the claim is available in [10].

Assumption 2 (Interior assumption) *Each interior vertex is path connected to a deep interior vertex in the restricted Delaunay triangulation of S.*

Figure 2 shows two triangulations of the dataset FOOT. The first triangulation is computed with the CO-CONE algorithm [4] without boundary detection. The second triangulation is computed with the modified CO-CONE algorithm that uses BOUNDARY to detect boundary vertices. Triangles incident to the boundary vertices are shaded darker.

(a) (b)

Figure 2: Triangulation of the dataset FOOT (a) without boundary detection; the big hole above the ankle is covered with triangles, (b) with boundary detection using the algorithm BOUNDARY; the hole above the ankle is well detected.

Non-smoothness

Recall that our theory is based on the assumption that the sampled surface is smooth. However, we observe that the boundary detection algorithm also detects undersampling in non-smooth surfaces, see Figure 3 and also Figure 7, 8 and 10. The ability to handle non-smooth surfaces owes to the fact that non-smooth surfaces may be approximated with a smooth one that interpolates the samples. For example, one can resort to the implicit surface that is C^1-smooth and interpolates the sample points using natural co-ordinates as explained in [8]. For higher order continuity results of [20] can be called upon. These smooth surfaces have high curvatures near the sharp features of the original non-smooth surface. Our theory can be applied to the approximating smooth surface to ascertain that the samples in the vicinity of sharp features act as boundary vertices in the vicinity of high curvatures for the smooth surface.

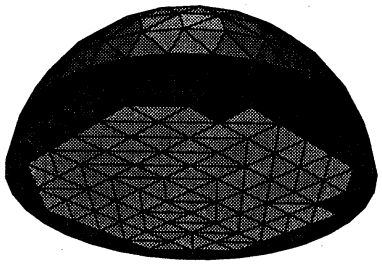


Figure 3: Boundary vertices are detected in the vicinity of the sharp edge in HALFSPHERE, triangles incident to boundary vertices are shaded darker.

5 Surface reconstruction

Our goal is to reconstruct a smooth compact surface $S \subset \mathbb{R}^3$ which may have a non-empty boundary from a finite sample $P \subset S$. For this purpose we modify the CO-CONE algorithm [4] such that it is capable of reconstructing surfaces with boundary. Like the CRUST the CO-CONE algorithm consists of three steps:

(1) CANDIDATETRIANGLEEXTRACTION. In this step a set of candidate triangles for the reconstruction is extracted from the Delaunay triangulation of the sample points. In general the underlying space of these triangles is not a manifold, but a manifold with boundary can be extracted in the next two steps.

(2) PRUNING. The candidate triangles are already close to a manifold for sufficiently dense samples. We want to extract a manifold from this set by walking on the outside or inside of this set. During the walk we may encounter the problem of entering a triangle t which has a bare edge, i.e., an edge incident to a single triangle, namely t. The purpose of this step is to get rid of these edges with their incident triangles.

(3) WALK. We walk on the out- or inside of the set of triangles that remained after pruning and report the triangles walked over.

We modify the first step of CO-CONE algorithm for boundary detection. In the original co-cone algorithm each sample chooses a set of candidate triangles incident to it. In the modified algorithm, only the interior vertices are allowed to choose the candidate triangles.

CANDIDATETRIANGLEEXTRACTION ($P \subset \mathbb{R}^3$, $\theta \in (0, \pi/2)$, ρ)

```
1  compute the Voronoi diagram V_P
2  CandidateTriangles := DelaunayTriangles
3  B :=BOUNDARY(P, θ, ρ)
4  for each sample p ∈ P
5    if p ∉ B
6      for each Voronoi edge e ∈ V_p
7        if C_p ∩ e ≠ ∅
8          e.Mark.insert(p)
9  for each Voronoi edge e ∈ V_P
10   for each p ∈ DUALTRIANGLE(e) ∩ P
11     if (p ∉ B) and (p ∉ e.Mark))
12       CandidateTriangles.delete(DUALTRIANGLE(e))
13 return CandidateTriangles
```

First the set *CandidateTriangles* is initialized to the set of all Delaunay triangles (line 2). This set gets filtered subsequently. Each Voronoi edge e has a field $Mark$ that collects the samples whose co-cones intersect e (line 7). This check is done only if the sample is not a boundary vertex. In other words, only interior vertices mark the Voronoi edges that intersect their co-cones. Next, we look at the markings of the Voronoi edges and delete their dual triangles from the candidate set if they are not marked by an adjacent sample which is interior (lines 10-12). In essence, we rely only on the triangles that are chosen by interior vertices and a triangle is in the candidate set only if all of its interior vertices have chosen it.

The second step PRUNING removes triangles incident to sharp edges in a cascaded manner as was originally suggested in [1]. An edge e is called *sharp* if there are two consecutive triangles incident to e such that the angle between them is more than $3\pi/2$. An edge is also called sharp

if it has only one incident triangle. We have to be careful in this step not to trigger the cascaded deletion of the desired surface by deleting a triangle incident on a boundary vertex which has a bare edge. So, we remove a triangle only if it is not incident on a boundary vertex.

Let K be a two dimensional simplicial complex.

```
PRUNING (K)
 1  Pending := ∅
 2  for each edge e ∈ K
 3    Pending.push(e)
 4  while Pending ≠ ∅
 5    e := Pending.pop()
 6    if ISSHARP(e) = true
 7      for each t ∈ e.Triangles
 8        if ∀(p ∈ t ∩ P) p ∉ B
 9          K := K\{t}
10          for each e' ∈ t.Edges\{e}
11            Pending.push(e')
12  return K
```

First a stack *Pending* is initialized empty (line 1). Then all edges in the complex K are pushed onto this stack (lines 2 and 3). Together with every triangle t we store a list *Edges* of its edges. With each edge e we store a set *Triangles* of triangles incident to e. We assume that we have a function ISSHARP available that requires an edge e as input and returns `true` if e is sharp and `false` if e is not sharp. As long as the stack *Pending* is not empty, an edge e is popped from the stack. If this edge is sharp, all those incident triangles are removed from the complex K that are not incident to a boundary vertex. All edges other than e that are incident to the deleted triangle t are pushed onto the stack *Pending* (lines $4-11$). These edges may become sharp due to the deletion of the triangle t. Observe that during deleting triangles we not only check if it is incident to a sharp edge, but also check if it is incident to any boundary vertex. Finally, the reduced complex K is returned (line 12).

The third step WALK extracts a manifold.

```
WALK (K)
 1  Surface := ∅
 2  choose arbitrary oriented t ∈ K
 3  Surface.insert(t)
 4  Pending := ∅
 5  for each ē ∈ t.Edges
 6    Pending.push((ē, t))
 7  while Pending ≠ ∅
 8    (ē, t) := Pending.pop()
 9    if e.processed = false
10      e.processed := true
11      t' := SURFACENEIGHBOR(K, ē, t)
12      if t' ≠ ∅
13        Surface.insert(t')
14        for each ē' ∈ t'.Edges\{ē}
15          Pending.push((ē', t'))
16  return Surface
```

First we initialize the set *Surface* empty (line 1). Then we choose an arbitrary triangle t from the complex K, orient it and insert it in the the set *Surface* (lines 2 and 3). Next we initialize the stack *Pending* empty (line 4). From the orientation of the chosen triangle t we derive an orientation for all edges incident to t. These edges are stored in a field *Edges* associated with every triangle. We denote an oriented edge e by $\vec{e}$. For each oriented edge e of triangle t, we push the pair $(\vec{e}, t)$ onto the stack (lines 5 and 6). As long as the stack *Pending* is not empty, we pop its top element $(\vec{e}, t)$ (line 8). If the edge e is not processed so far we use the field *processed* to mark it processed and compute the surface neighbor of $(\vec{e}, t)$, i.e. the triangle t' incident to e that 'best fits' t (lines 9-11). If t' does not exist, the triangle t is incident to the boundary. We insert t' in the set *Surface* (lines 12 and 13). We assume that the function SURFACENEIGHBOR orients t' such that its orientation matches the orientation of t and push all the pairs of oriented edges $\vec{e}'$ besides $\vec{e}$ incident to t' together with the triangle t' itself onto the stack *Pending* (lines 14 and 15). Finally we return the surface *Surface* (line 16). The above method works under the assumption that the surface is orientable, i.e., surfaces like Möbius strip are not allowed.

The return value t' of SURFACENEIGHBOR$(C, \vec{e}, t)$ is the topmost triangle triangle among the set of triangles incident to e whose normals (oriented according to the orientation of t) make an angle smaller than $\frac{\pi}{2}$ with the normal of t, see Figure 4.

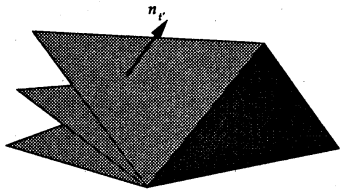


Figure 4: The surface neighbor of the dark grey shaded triangle is the topmost triangle among the light grey shaded triangles.

Putting the three steps CANDIDATETRIANGLEEXTRACTION, PRUNING and WALK together gives the modified CO-CONE algorithm.

CO-CONE ($P \subset \mathbb{R}^3$, $\theta \in (0, \pi/2)$, ρ)
1 K:=CANDIDATETRIANGLEEXTRACTION (P, θ, ρ)
2 K':=PRUNING (K)
3 K'':=WALK (K')
4 **return** K''

6 Implementation and results

As with many other geometric algorithms we faced the numerical robustness as an important issue in implementing CO-CONE. There are two main sources of problems. First, algorithms and data structures defined under a *general position* assumption do not work for practical data sets since such degenerate situations do occur. Second, we need to evaluate geometric predicates which are difficult to compute exactly (especially if the situation is close to a degenerate one). To cope with this problem we resort to the robust library of geometric algorithms called CGAL [24] developed by an European consortium of seven universities. The library consists of C++ template

code which makes it easy to use different number types and implementations of predicates for the computation.

For our implementation of the CO-CONE algorithm we used the Delaunay triangulation from the CGAL library together with filtered predicates. Computation of filtered predicates simulates exact evaluation on a demand basis and thus runs faster than predicate evaluations with exact arithmetic. Instead of filtered predicates if we use floating point arithmetic the running time decreases roughly by a factor of two, see Table 2. However, results may not be reliable.

Besides robustness we encountered a problem which is more inherent to our algorithm: Some data contain noise beyond the tolerance of the CO-CONE algorithm. This may turn some interior vertices as "false boundary vertices". These points are detected as interior by BOUNDARY, but their incident candidate triangles after the CO-CONE step do not form a flat disk. Thus, the pruning step runs the risk of deleting the desired output since it does not recognize these "false boundary vertices". That means, it can happen that during the PRUNING step all triangles are removes that are not incident to a boundary vertex. To cope with this problem, we employ a safety check HASUMBRELLA to each vertex.

6.1 Umbrella check

An *umbrella* incident to a vertex $p \in K$ is a set of triangles incident to p which form a topological closed disc $\mathbb{D}^2$. An umbrella is called sharp if it contains a sharp edge. HASUMBRELLA deletes the sharp edges and their incident triangles in a cascaded manner. But, unlike pruning this cascaded deletion is applied only to edges and triangles incident to the vertex being checked. If the vertex retains a triangle after this cascaded deletion, HASUMBRELLA returns `true`. To avoid misunderstandings note that HASUMBRELLA removes the triangles only virtually, i.e. after applying HASUMBRELLA to a point p of a complex K the set of triangles incident to p is exactly the same as it was before. That means, HASUMBRELLA is only used to check if a point has a non sharp umbrella but not to alter the complex K.

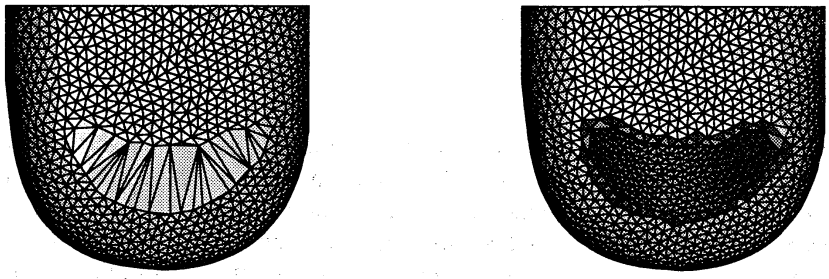
```
HASUMBRELLA (K, p ∈ K)
 1  Umbrella := p.Triangles
 2  Pending := ∅
 3  for every edge e ∈ p.Edges
 4    Pending.push(e)
 5  while Pending ≠ ∅
 6    e := Pending.pop()
 7    if ISSHARP(e) = true
 8      for each t ∈ e.Triangles
 9        Pending.push(t ∩ (p.Edges\e))
10        Umbrella.delete(t)
11  if Umbrella = ∅
12    return false
13  return true
```

First the set *Umbrella* is initialized to the set of triangles incident to p (line 1). Next we initialize the stack *Pending* empty (line 2). Then we push all edges e incident to p onto the stack *Pending* (lines 3 and 4). We assume that the edges incident to p are stored in the set *Edges* and the triangles incident to p are stored in the set *Triangles*. As long as the stack *Pending* is not empty we pop its top element. If this element is a sharp edge e, we delete all triangles incident to e in the complex K from the set *Umbrella*, and push all edges incident to such a triangle and the point p besides the edge e onto the stack *Pending* (lines 5-11). Finally we return `false` if the set *Umbrella* is empty and `true` otherwise.

The boundary vertices detected by our implementation are the union of the boundary vertices detected by the algorithm BOUNDARY and the vertices not detected by the algorithm HASUMBRELLA. We observe that PRUNING with this enlarged set of boundary vertices is safe. For samples from smooth surfaces that fulfill the sampling condition we observe that UMBRELLACHECK is not necessary.

6.2 Parameters and precision

In the proofs we choose the ratio ρ of the height to the width of a Voronoi cell to be more than $\frac{1}{1.3\epsilon}$. In practice a value between 1.1 and 1.7 gives good results. Increasing this ratio detects more points as boundary as can be observed in Figure 5.



(a) (b)

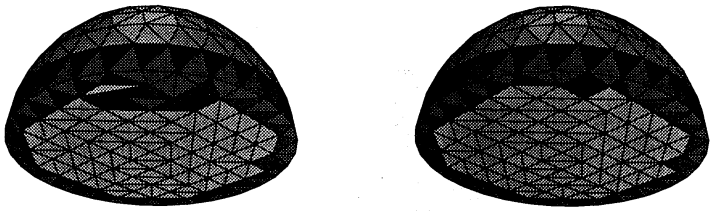
Figure 5: Both figures show the reconstructed heel of the dataset FOOT for different values of ρ. In (a) this ratio is 1.5 and in (b) it is 4.3.

Although θ is 0.14 radians in theory, a value as large as $\frac{\pi}{6}$ gives good results. We observed that the output is quite sensitive to the opening angle of the double cones that defines co-cones. If this angle is decreased, the width increases with the inverse of the sin of this angle. Thus, the height vs. width ratio deteriorates quite sharply forcing the ratio condition fail for many vertices. On the other hand, if we make this angle larger co-cones become thinner which may not allow some of the desired triangles to be captured as candidate triangles. We used $\frac{3\pi}{8}$ to define the co-cones both in theory and practice.

As mentioned earlier, numerical robustness is an important issue in our reconstruction software. Figure 6 shows the reconstruction for the dataset HALFSPHERE with floating point arithmetic and with filtered exact arithmetic. Numerical problem occurs in this dataset since it contains many co-planar points causing degeneracies in the Delaunay triangulation. As a result the Delaunay triangulation is computed incorrectly. CO-CONE takes 1.9 seconds computing time with floating point arithmetic compared to 3 seconds with filtered predicates. Figure 7 shows the effect of using filtered predicates instead of floating point arithmetic for the OILPUMP dataset. Floating point arithmetic takes 469 seconds for the reconstruction whereas filtered exact arithmetic takes 966 seconds. Considering the tradeoff between running time and robustness our experiments show that one should accept increased running time for reliable computations.

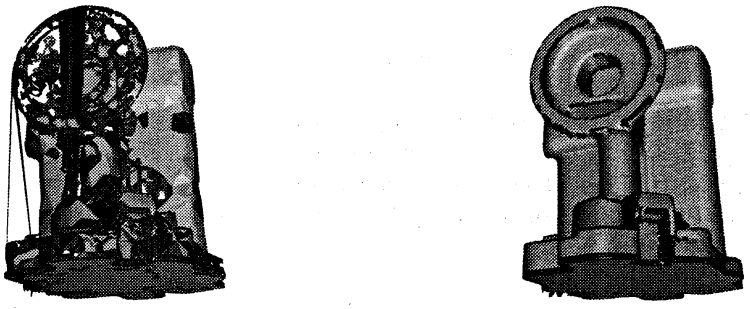
6.3 More datasets

We have experimented with many other datasets, some of which are listed below. In the figures the triangles incident with boundary vertices are shaded darker.



(a) (b)

Figure 6: Left and right figures show the reconstructed HALFSPHERE with floating point and exact arithmetic respectively.



(a) (b)

Figure 7: Reconstruction of the dataset OILPUMP (a) with floating point arithmetic and (b) with filtered exact arithmetic; sharp features are well detected.

The dataset ENGINE has three connected components. CO-CONE nicely separates the three connected components and identifies the boundaries of the two outer shells. It also identifies the sharp tip of the innermost component, see Figure 8.

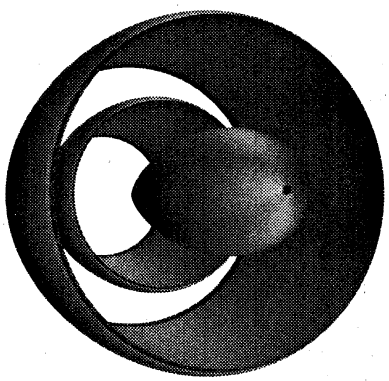
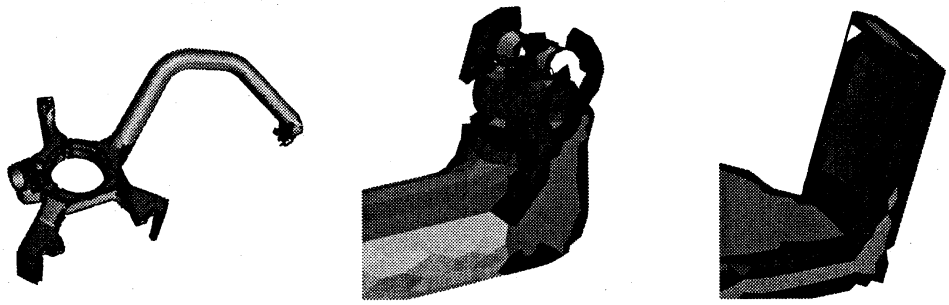


Figure 8: Boundaries and sharp tip in ENGINE are detected.

The dataset CAT: This dataset contains a single boundary at the bottom of the cat and some undersampled regions, especially at the ears and legs. Figure 9 shows the detected boundary and undersampled regions.

Figure 9: The dataset CAT.

The dataset KNUCKLE: This is a challenging dataset for reconstruction, because it contains many undersampled parts. This undersampling is mostly due to sharp features. Figure 10 shows how the CO-CONE algorithm deals with undersampling. Most of the sharp features are well detected. Two regions where undersampling is extreme are shown with a zoom.



(a) (b) (c)

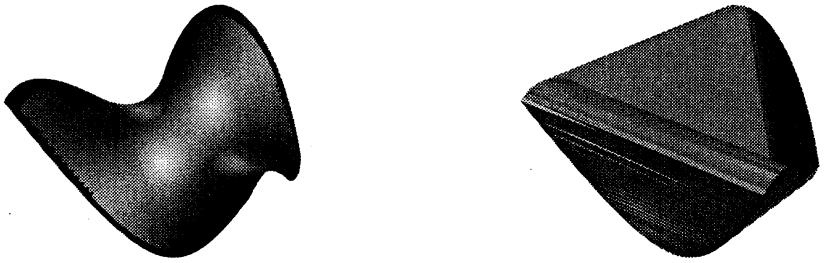
Figure 10: Reconstruction of the dataset KNUCKLE. In (a) the entire reconstruction is shown and in (b) and (c) two undersampled parts are zoomed.

The dataset MANNEQUIN: This dataset contains one boundary at the bottom and undersampled regions in eyes, lips and ears. Figure 11 shows the reconstruction of this dataset with and without boundary detection. Using boundary detection the CO-CONE detects the boundary at the bottom as expected.

The dataset MONKEY: This dataset is created using the function $(x, y) \mapsto x^3 - 3xy^2$ on the unit square. The graph of this function is called *monkey saddle*. The monkey saddle contains a single boundary which is perfectly detected. Figure 12 shows the monkey saddle and compares the output with the output of the CO-CONE algorithm without boundary detection.

(a) (b)

Figure 11: Reconstruction of the dataset MANNEQUIN (a) with boundary detection and (b) without boundary detection.



(a) (b)

Figure 12: Reconstruction of the dataset MONKEY (a) with boundary detection and (b) without boundary detection.

6.4 Running times

We tested the CO-CONE algorithm on many datasets. All the tests were done on a Silicon Octane computer with 300 Mhz MIPS processor and 512 MByte of main memory. For all datasets we used $\rho = 1.5$ and $\theta = \frac{\pi}{6}$ in the algorithm BOUNDARY.

For rendering we used *Geomview* [17], provided by the Geometry Center at the University of Minnesota.

The basic data of our experiments are summarized in Table 1. The running times are with respect to filtered exact arithmetic. Observe that in all cases the number of triangles is roughly twice the number of points, which is expected from Euler's formula.

Table 2 allows a closer look on the running times. We also included the timings when using floating arithmetic.

Since the CO-CONE algorithm is logically split into three steps: *Delaunay triangulation*, *Boundary detection* and *Surface reconstruction* we measure the CPU times for each of these three steps.

As expected the Delaunay triangulation computation is the most time consuming step. The running times using filtered arithmetic are almost twice as high as the running times using floating point arithmetic, but they are still in a reasonable range. The total running time shows a sub quadratic behavior with respect to the number of sample points.

object	number of points	number of triangles	running time(sec.)
CAT	10000	19826	311
ENGINE	11360	22356	1040
FOOT	20021	39997	592
KUNCKLE	6014	10791	208
MANNEQUIN	12772	25243	374
MONKEY	10000	19600	1620
OILPUMP	30933	61041	966

Table 1: Experimental data.

object	Delaunay	Boundary	Reconstruction
CAT	121-50	108-58	82-51
ENGINE	836-405	107-58	97-59
FOOT	233-99	224-119	138-103
MANNEQUIN	135-57	136-77	103-77
KNUCKLE	107-42	60-34	41-45
MONKEY	1328-	175-	117-
OILPUMP	414-154	313-172	239-143

Table 2: A closer look on the running time of the CO-CONE algorithm in seconds. The first times (before '-') are with respect to filtered arithmetic and the second times (after '-') are with respect to floating point arithmetic. (For the MONKEY dataset the CGAL code was not able to compute the Delaunay triangulation using floating point arithmetic).

7 Conclusions

In this paper we present an algorithm to detect the regions of undersampling in data that are sampled from some surface. This provides a unified approach to reconstruct surfaces with boundaries and to identify the regions of non-smoothness or high curvature where undersampling does occur. The existing algorithms with theoretical guarantees fail miserably on such data sets since they assume that the data is derived from a smooth surface without any boundary. As exhibited by our empirical results, our boundary detection algorithm correctly identifies the vertices that are visibly lying on the boundary. Also, it identifies the regions of non-smoothness and high curvature effectively in practice.

A probable follow up of this work is to investigate how this algorithm can be used to reconstruct non-smooth surfaces. After detecting the regions of non-smoothness, can we repair the surface to fill up the "holes"? Currently research on this question is under progress.

References

[1] N. Amenta and M. Bern. Surface reconstruction by Voronoi filtering. *Discr. Comput. Geom.*, **22**, (1999), 481–504.

[2] N. Amenta, M. Bern and D. Eppstein. The crust and the β-skeleton: combinatorial curve reconstruction. *Graphical Models and Image Processing*, **60** (1998), 125-135.

[3] N. Amenta, M. Bern and M. Kamvysselis. A new Voronoi-based surface reconstruction algorithm. *SIGGRAPH 98*, (1998), 415-421.

[4] N. Amenta, S. Choi, T. K. Dey and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. *Proc. 16th. ACM Sympos. Comput. Geom.*, (2000), 213–222.

[5] D. Attali. r-regular shape reconstruction from unorganized points. *Proc. 13th ACM Sympos. Comput. Geom.*, (1997), 248–253.

[6] C. Bajaj, F. Bernardini and G. Xu. Automatic reconstruction of surfaces and scalar fields from 3D scans. *SIGGRAPH 95*, (1995), 109–118.

[7] J. D. Boissonnat. Shape reconstruction from planar cross-sections. *Computer Vision, Graphics, and Image Processing* **44** (1988), 1–29.

[8] J. D. Boissonnat and F. Cazals. Smooth surface reconstruction via natural neighbor interpolation of distance functions. *Proc. 16th. ACM Sympos. Comput. Geom.*, (2000), 223–232.

[9] B. Curless and M. Levoy. A volumetric method for building complex models from range images. *SIGGRAPH 96*, (1996), 303-312.

[10] T. K. Dey and J. Giesen. Detecting undersampling in surface reconstruction. *manuscript*, 2000.

[11] T. K. Dey and P. Kumar. A simple provable algorithm for curve reconstruction. *Proc. ACM-SIAM Sympos. Discr. Algorithms*, (1999), 893–894.

[12] T. K. Dey, K. Mehlhorn and E. A. Ramos. Curve reconstruction: connecting dots with good reason. *Comput. Geom. Theory Appl.*, **15**, 229–244.

[13] T. K. Dey and R. Wenger. Reconstructing curves with sharp corners. To appear in *16th ACM Sympos. Comput. Geom.*, 2000.

[14] H. Edelsbrunner. Shape reconstruction with Delaunay complex. *LNCS 1380, LATIN'98: Theoretical Informatics*, (1998), 119–132.

[15] H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graphics*, **13**, (1994), 43–72.

[16] H. Edelsbrunner and N. Shah. Triangulating topological spaces. *Proc. 10th ACM Sympos. Comput. Geom.*, (1994), 285–292.

[17] `http://www.geomview.org`

[18] J. Giesen. Curve reconstruction, the TSP, and Menger's theorem on length. *Proc. 15th ACM Sympos. Comput. Geom.*, (1999), 207–216.

[19] C. Gold. Crust and anti-crust: a one-step boundary and skeleton extraction algorithm. *15th. ACM Sympos. Comput. Geom.*, (1999), 189–196.

[20] H. Hiyoshi and K. Sugihara. Voronoi-based interpolation with higher continuity. *Proc. 16th. ACM Sympos. Comput. Geom.*, (2000), 242–250.

[21] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald and W. Stuetzle. Surface reconstruction from unorganized points. *SIGGRAPH 92*, (1992), 71-78.

[22] M. Melkemi. $\mathcal{A}$-shapes and their derivatives. *13th ACM Sympos. Comput. Geom.* , (1997), 367–369.

[23] D. Zorin and P. Schröder. Subdivision for modeling and animation. *SIGGRAPH 99 Course Notes.*.

[24] `http://www.cs.ruu.nl/CGAL`