Filling Gaps in the Boundary of a Polyhedron*

Gill Barequet† Micha Sharir‡

Abstract

In this paper we present an algorithm for detecting and repairing defects in the boundary of a polyhedron. These defects, usually caused by problems in CAD software, consist of small gaps bounded by edges that are incident to only one polyhedron face. The algorithm uses a partial curve matching technique for matching parts of the defects, and an optimal triangulation of 3-D polygons for resolving the unmatched parts. It is also shown that finding a consistent set of partial curve matches with maximum score, a subproblem which is related to our repairing process, is NP-Hard. Experimental results on several polyhedra are presented.

Keywords: CAD, polyhedra, gap filling, curve matching, geometric hashing, triangulation.

1 Introduction

The problem studied in this paper is the detection and repair of "gaps" in the boundary of a polyhedron. This problem usually appears in polyhedral approximations of CAD objects, whose boundaries are described using curved entities of higher levels (cf. [Sheng & Tucholke '91], [Dolenc & Mäkelä '91], [Bøhn & Wozny '92], and [Mäkelä & Dolenc '93]). In solid modeling the original boundary may be described as the unions and intersections of spheres, cones, etc., whereas in surface modeling it may be described by Bézier surfaces, Nurbs, etc. Some of the gaps are caused by missing surfaces, incorrect handling of adjacent patches within a surface, or (most commonly) incorrect handling of trimming curves, which are defined by the intersections of adjacent surfaces. The mesh points (that is, the computed vertices of the

*Work on this paper by the second author has been supported by Office of Naval Research Grant N00014-90-J-1284, by National Science Foundation Grants CCR-89-01484 and CCR-91-22103, and by grants from the U.S.-Israeli Binational Science Foundation, the Fund for Basic Research administered by the Israeli Academy of Sciences, and the G.I.F., the German-Israeli Foundation for Scientific Research and Development.

†School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel

‡School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel, and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA

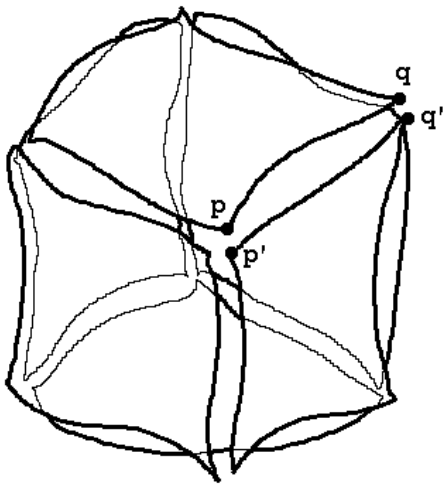


Figure 1: A cracked object

polyhedral approximation) along an intersection curve between two such surfaces are often computed separately along each of its two sides, thereby creating two different copies of the same curve, causing a gap to appear between the copies. In the simple case, different point sets might be produced but according to the same curve equation; in the more complicated case, different equations of the curve, one for each surface containing it, are used for the mesh point evaluations.

The effect of these approximation errors is invariably the same: the boundary of the resulting polyhedron contains edges which are incident to only one face (whereas in a valid representation each edge should be incident to exactly two faces), thereby creating gaps in the boundary and making the resulting representation invalid. Such gaps may make parts of the boundary of the approximating polyhedron disconnected from other parts, or may create small holes bounded by a cycle of invalid edges. Usually, each connected component is the result of the polyhedral approximation of a single surface or part of a solid in the original representation. Figure 1 shows a typical example, where thin cracks are curving between the surfaces of the cube-like model. This phenomenon does not usually disturb graphics applications, where the gaps between the surfaces are often too small to be seen, or are handled straightforwardly [Samet & Webber '88]. However, it may cause severe problems in applications which rely on the continuity of the boundary, such as finite element analysis ([Yerry & Shephard '83], [Ho-Le '88]), rasterization algorithms [Foley & Van Dam '84], etc.

This problem arises frequently in CAD applications ([Sheng & Tucholke '91], [Mäkelä & Dolenc '93]) and a significant number of currently available commercial CAD packages produce these gaps. According to our own practical experience, such gaps arise in almost every sufficiently large CAD file, so their detection and elimination is indeed a rather acute practical problem. [Sheng & Tucholke '91] refer to these errors as one of the most severe software problems in rapid prototyping. Many authors, such as [Dolenc & Mäkelä '91] and [Sheng & Hirsch '92], tried to avoid it already in the surface fitting triangulation process.

Traditional methods for closing gaps in edges and surfaces, mainly used in image processing, assume that the input is given as binary raster images in two or three dimensions. Errors in edge detector output were dealt with extensively (cf. [Pratt '78, §17] for a detailed discussion). Various morphological techniques were suggested in order to fix these errors, such as the *chamfer map* used in [Snyder, Groshong, Hsiao, Boone & Hudacko '92].

Previous attempts to solve this problem, based only on the polyhedral description of a model, used only local information, and did not check for any global consistency violations. [Bøhn & Wozny '92] treat only local gaps by iteratively triangulating them. They eliminate at each step the vertex which spans the smallest angle with its two neighboring vertices. Similarly, [Mäkelä & Dolenc '93] apply a minimum distance heuristic in order to locally fill cracks in the boundary of the polyhedron. We invoke a similar procedure (which uses a minimum area heuristic) at the second phase of our algorithm (see Section 6.2). To the best of our knowledge, no effort to solve this problem while considering the global consistency of the resulting polyhedron, based only on the polyhedral description of a model, was ever made. In order to do that, we give a precise definition of the problem, and develop a method for identifying instances of this problem and then resolving them.

Let us denote the collection of cycles of invalid edges on the polyhedron boundary by the term *borders*. The main problem that we face is to "stitch" these borders together, i.e. add new faces that close the gaps in the boundary such that the resulting polyhedron is valid. New faces are added by connecting points along the same or different borders. To achieve this we first have to identify matching portions of these borders (e.g. arcs pq and $p'q'$ in Figure 1), and then to choose the best set of matches, construct new facets (planar faces) connecting them, and fill (by triangulation) the remaining holes. Successful solutions to all these subproblems are described in detail in this paper.

For the purpose of identifying matching portions of the borders we use a *partial curve matching* technique, which was first suggested in [Kalvin, Schonberg, Schwartz & Sharir '86] and in [Schwartz & Sharir '87]. This technique, which uses the so-called *Geometric Hashing* method, originally solved the curve matching problem in the plane, under the restrictive assumption that one curve is a proper subcurve of the other one, namely:

> Given two curves in the plane, such that one is a (slight deformation of a) proper subcurve of the other, find the translation and rotation of the subcurve that yields the best least-squares fit to the appropriate portion of the longer curve.

This technique was extended and used in computer vision for automatic identification of partially obscured objects in two or three dimensions, an important problem in robotics applications of computer vision, which has attracted much attention. The geometric hashing technique was applied in [Hong & Wolfson '88], [Wolfson '90] and

in [Kishon, Hastie & Wolfson '91] in various ways for identifying partial curve matches between an input scene boundary and a preprocessed set of known object boundaries. This was used for the determination of the objects participating in the scene, and of the position and orientation of each such object.

We use a simplified variant of this technique, in which no motion of one curve relative to the other is allowed. However, our variant matches 3-dimensional curves. Since the scope of our problem is wider, we have to further process the information obtained by the matching step. We use the matching results for repairing most of the defects, and develop a 3-dimensional triangulation method for closing the remaining holes. This method is similar to the dynamic programming triangulation of simple polygons developed in [Klincsek '80].

2 Definition of the Problem

Consider the following description of the boundary of a polyhedron, where the boundary is represented by two lists: one contains all the vertices of the polyhedron, and the other contains all the facets. A facet is a collection of one or more polygons, all lying in the same plane in 3D. The first polygon is the *envelope* (outer boundary) of the facet, and the other polygons, if any, are *windows* in it (when the facet is not simply connected). Each polygon is specified as a circular sequence of indices in the vertex list. There is no restriction on the length of such an index sequence, hereafter referred to as the *size* of the polygon.

As input to our algorithm, there is no restriction on the directions of the polygons. Eventually, in order to form an oriented 2-manifold, they will have to obey a consistency rule. For example, we require that all the facet envelopes appear in the clockwise direction when viewed from outside the polyhedron, and that all the window polygons appear in the counter-clockwise direction when viewed this way. Thus, the body of a polygon will always be on the right hand side of every directed edge which belongs to it, when viewed from the outside. Since the directions of all the input polygons are arbitrary, we have to orient them ourselves in these consistent directions. Note that each valid edge appears in exactly two facets, and that these two appearances are oppositely directed to each other.

The main problem addressed in this paper is the existence of gaps between and/or within parts of the polyhedron boundary. We want to identify correctly the matching portions of the borders, and fill them with additional triangles, such that no holes remain. The output should be an orientable manifold which describes a closed volume.

As the previously suggested recipes cited in the introduction, we may also allow the resulting boundary to intersect itself near its original borders. This often happens anyway in CAD approximations of curved surfaces. The borders of the approximating polyhedral surfaces are generated in very close locations (where they should really be coincident), potentially making the surfaces either totally disconnected or intersecting. We attempt to stitch together close borders, allowing the unified boundary to

intersect itself, as long as it remains oriented consistently. This means that, upon the termination of the algorithm, each edge should appear in exactly two facets and in opposite directions.

The fact that the resulting boundary may be self-intersecting can be regarded as a limitation of the proposed algorithm, in instances where the output should be free of this phenomenon. We allow this for two practical reasons. First, the input may already have this property, and our algorithm does not attempt to fix that. (However, in practice, our algorithm does not tend to create self-intersections when they do not exist in the input.) Second, our algorithm is mainly intended for repairing polyhedral boundary descriptions, to serve as input for other algorithms, which crucially rely on the continuity of the boundary. Usually, these algorithms are very robust regarding the existence of small self-intersections. One example is the class of scan-line rasterization algorithms, where the "inside" and the "outside" of the rasterized object must be well defined. Self-intersections are easily handled by simply counting the number of entrances to and exits from the object along a scanning line. Another example is the class of finite element analysis algorithms, which are based on processing the boundary of the object. Usually, these algorithms are not even aware that small self-intersections occur.

3 Overview of the Algorithm

Our proposed algorithm consists of the following steps:

1. Data acquisition:
   - Identify the connected components of the polyhedron boundary, and orient all the facets in each component with consistent directions.
   - Identify the border edges, each incident to only one face, and group them into a collection of border polygons, each being a cycle of border edges. Each open connected component of the polyhedron boundary is bounded by one or more border polygons.
2. Matching border portions:
   - Discretize each border polygon into a cyclic sequence of vertices, so that the arc length between each pair of consecutive vertices is equal to some given parameter.
   - Vote for border matches. Each pair of distinct vertices which belong to the discretized borders and whose mutual distance is below some threshold parameter, contributes one vote. The vote is for the match between these two borders with the appropriate shift, which maps one of the vertices to the other.

- Transform the resulting votes into a collection of suggestions of partial border matches, each given a score that measures the quality of the match.
- Choose a consistent subset of the above collection whose score is maximal. This step turns out to be NP-Hard, so we implement it using a standard approximation scheme.

3. Filling the gaps:

   - Stitch together each pair of border portions that have been matched in the above step, by adding triangles which connect between these portions. The new triangles should be oriented consistently with the facets along the borders.
   - Identify the remaining holes (usually appearing at the junctions of several matches).
   - Triangulate the holes, using a 3-D minimum area triangulation technique.

The following three sections describe the algorithm steps in detail.

4 Data Acquisition

The description of the boundary of the polyhedron is typically given in a file, output of a CAD system. Our system allows several input formats, without any restriction on the size of the polygons, and allowing facets to contain windows. Most of the file formats used by commercial CAD systems do not include adjacency information (between facets). When the input does not specify this information, our system generates it as a preprocessing step. For this purpose we sort the edges according to the id's of their endpoints, and transform every pair of two successive edges in this order, which have the same endpoints, into an adjacency relation between facets. The same information was computed in [Dolenc and Mäkelä '91] using an *Octree*-like data structure, and in [Rock and Wozny '92] using an *AVL* tree. The internal representation that our system actually uses in further steps is the *quad-edge* data structure described in [Guibas & Stolfi '85].

The connected components of the (broken) boundary of the polyhedron are computed in a simple depth-first search on the dual graph of the boundary of the polyhedron. This process also allows us either to orient all the facet polygons with consistent directions, or to detect that one or more components are not orientable. In the latter case we may either ignore the problematic components or halt the algorithm.

Locating the borders of the connected components of the boundary is straightforward. We consider each oriented facet polygon as a formal sum of its directed edges, and add up all these facets, with the convention that $\vec{e} + (-\vec{e}) = 0$. The resulting sum consists of all the border edges. Since each facet polygon is a directed cycle, the resulting sum is easily seen to represent a collection of pairwise edge disjoint directed

cycles. For convenience, we break non-simple border cycles into simple ones. Each connected component may be bounded by any number of border polygons.

Unlike previous related works on object recognition (see [Kalvin, Schonberg, Schwartz & Sharir '86], [Schwartz & Sharir '87], [Hong & Wolfson '88] and [Wolfson '90]), we do not smooth the borders. In our case, the data is not retrieved from a noisy raster image, and is assumed to be accurate enough, except for those computation errors which were introduced in the polyhedral approximation and which caused the gaps.

5 Matching Border Portions

5.1 Border Discretization

Each border polygon is discretized into a cyclic sequence of points. This is done by choosing some sufficiently small arc length parameter s, and generating equally-spaced points, at distance s apart from each other (along the polygon boundary). In analogy with the works on object recognition cited above, we may regard the resulting discretization as *signatures* (also called *footprints*) of the borders.

5.2 Voting for Border Matches

Naturally, two parts of the original object boundary, which should have shared a common polygonal curve but were split apart in the approximation, must have similar sequences of footprints along their common curve (unless the approximation was very bad). This follows from our definition of the footprints as the 3-D coordinates of the points. Thus, our next goal is to search for pairs of sufficiently long subsequences that closely match each other. In our approach, two subsequences $(p_i, \ldots, p_{i+\ell-1})$ and $(q_j, \ldots, q_{j+\ell-1})$ are said to closely match each other, if, for some chosen parameter $\varepsilon > 0$, the number of indices k for which $\|p_{i+k} - q_{j+k}\| \leq \varepsilon$ is sufficiently close to ℓ. We perform the following voting process, where votes are given to good point-to-point matches.

The borders are given as cyclic ordered sequences of vertices. We break each cycle at an arbitrary chosen vertex. Also, the direction of a border is implied by the chosen orientation of its connected component. Had it been chosen the other way, the border direction would have been reversed. A match between two border subsequences is called *direct* when the sequences of vertex indices of both borders are in the same (increasing or decreasing) order; a match is called *inverted* when one of the sequences is in an increasing order and the other is in a decreasing order.

Note the following:

- Adjacent components whose orientations are consistent should have an inverted

match. This match, if accepted, gives the combined component the same orientation as its two subparts (see Figure 2(b)), or the opposite of these orientations.

- A direct match implies that the orientations of the two components are not consistent. Hence, if the match is accepted, exactly one of the components (i.e. all its facets) should invert its orientation before gluing together the two components.

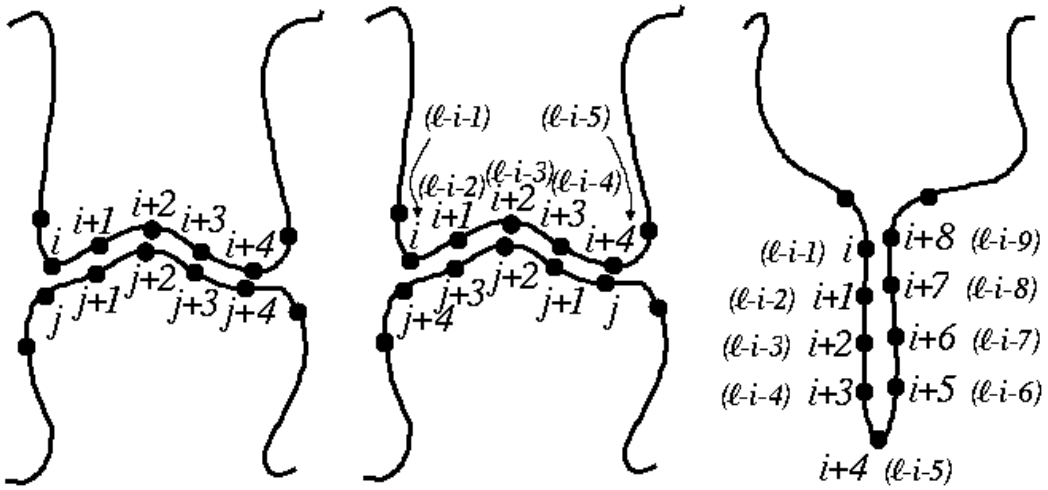
- If two border portions that bound the same component are matched, then only inverted matches are acceptable, or else the component will become non-orientable after gluing.

All the border vertices are preprocessed for *range-searching*, so that, for each vertex v, we can efficiently locate all the other vertices that lie in some ε-neighborhood of v. We have used a simple heuristic projection method, which projects the vertices onto each of the X-, Y- and Z-axes, and sorts them along each axis. Given a query ε-neighborhood, we also project it onto each axis, retrieve the three subsets of vertices, each being the set of vertices whose projections fall inside the projected neighborhood on one of the axes, then choose the subset of smallest size, and finally test each of its members for actual containment in the query neighborhood. While this method may be inefficient in the worst case, it works very well in practice. Orthogonal range queries can be answered more efficiently by using the *range tree* data structure [Mehlhorn '84, p. 69], or by *fractional cascading*, like in [Chazelle '88].

The positions along a border sequence b, whose length is ℓ_b, are numbered from 0 to $\ell_b - 1$. Assume that the querying vertex v is in position i of border sequence b_1. Then, each vertex retrieved by the query, which is in position j in border sequence b_2, contributes a vote for the direct match between borders b_1 and b_2 with a shift equals to $(j - i) \pmod{\ell_{b_2}}$, and a vote for the match between the borders $\overline{b_1}$ (the *inverted* b_1) and b_2, with a shift equals to $(j - (\ell_{b_1} - 1 - i)) \pmod{\ell_{b_2}}$. (The latter is the inverted match between the borders b_1 and b_2, as defined above.) As noted above, we allow only inverted matches between two portions of the same border or between borders which bound the same component; otherwise we would introduce a topological error by creating a non-orientable surface.

All these cases are illustrated in Figure 2. Match (a) is direct, hence the orientation of one of the components should be inverted. The corresponding shift is $(j - i)$. Match (b) is inverted, hence the two involved components are consistent. The corresponding shift is $(j - \ell + i + 5)$, where the small indices are those of the inverted top border. Finally, match (c) is between a border to itself, where the shift, when inverting the right portion, is $(2i - \ell + 9)$.

Obviously, matches between long portions of borders are reflected by a large number of votes for the appropriate shift between the matching borders. Since there might be small mismatches between the two portions of the matching borders, or the arc length along one portion may not exactly coincide with the arc length along the other portion, it is most likely that a real match will be manifested by a significant



(a) (b) (c)

Figure 2: Matches between borders

peak of few successive shifts in the graph that plots the number of votes between two borders (possibly the same one) as a function of the mutual shift. Note that there might be several peaks in the same graph. This implies that there are several good matches with different alignments between the same pair of borders.

Keeping track (for each peak) of the portions of the borders which voted for this alignment, we can infer the endpoints of the corresponding match (or matches) between these two borders. We extend the matches as much as possible, based on the neighborhood of the peak, allowing sporadic mismatches, insertions or deletions up to specified limits.

Each match is given a *score*. The score may reflect not only the number of votes for the appropriate shift, but also the Euclidean length of the match and its quality (measured by the closeness of the vertices on its two sides). Our settings of these parameters are described in Section 8.

Note that the ε parameter for the range-searching queries is not a function of the input. It is rather our *a priori* estimation of the physical size of the gaps created between the original surfaces. Setting ε to a value which is too small (or too large) may cause a degradation in the performance of the algorithm. In the first case, close points will not be matched, thus border matches will not be found. In the second case, too many "false" votes will result in losing the correct border matches among too many incorrect matches. Note also that, due to the implementation, each point-to-point match actually contributes two votes, but this does not spoil the voting results.

In almost all the cases small portions of the borders are included in more than one candidate match. This happens when several borders occur in close locations, usually at the junction of three or more cracks. We simply eliminate those portions common to more than one significant candidate match.

5.3 Pruning the Suggestions

The result of the voting step is a set of suggestions for matches between portions of borders. Our next goal is selecting a consistent subset of these suggestions with maximal score. Accepting a direct match implies the inversion of exactly one of the two borders, whereas accepting an inverted match implies the inversion of both borders or of none of them. Each border has to be oriented in one of the two possible directions, and, given an assignment of border orientations, we may accept only inverted matches relative to these orientations. Each assignment of border orientations is scored by the sum of the scores of the accepted matches. Naturally, for each orientation assignment there exists the inverse assignment, where all the borders are oriented in the opposite directions. In the absence of prior preferences, the score of the inverse assignment is equal to the score of the original one. We look for the assignment of border orientations whose score is maximal.

Unfortunately, this problem turns out to be NP-Hard. In order to prove that, let us rephrase the problem using graph terminology:

> *Choose-1.* Consider a weighted graph $G = (V, E)$. The vertices of V appear in pairs $(v_i, \overline{v}_i)$. The edges of E also appear in pairs: each pair either connects v_i to v_j and $\overline{v}_i$ to $\overline{v}_j$, or connects v_i to $\overline{v}_j$ and $\overline{v}_i$ to v_j. Each pair of edges has the same weight. The problem is to choose one vertex out of each pair, such that the total weight of the edges connecting the selected vertices is maximal.

The graph problem is equivalent to the border matching problem, at least in an abstract non-geometric setting. Each pair of vertices corresponds to the two possible orientations of a border. Each pair of edges represents a match: a straight link for an inverted match, and a cross link for a direct match. Selecting one vertex out of each pair stands for the choice of the orientation of the corresponding border, and the edges connecting the selected vertices correspond to the accepted border matches. The weights of the edges stand for the scores of the matches, and in both problems we search for an optimal (maximum) choice. (It is not clear whether each abstract instance of the graph problem has a geometric interpretation. Hence, our analysis will only imply that the abstract part of the border matching problem, whose input candidate matches may be arbitrary, is NP-Hard.)

The following problem is known to be NP-Hard[1] [Garey & Johnson '79, p. 210]:

> *Max Cut.* Given a graph $G = (V, E)$, and a weight $w(e) \in Z^+$ for each $e \in E$, find a partition of V into disjoint sets V_1 and V_2 such that the sum of the weights of the edges from E that have one endpoint in V_1 and one endpoint in V_2 is maximal.

[1] Originally, it was stated as a decision problem which turned out to be NP-Complete. Similar reasoning to that in the proof given here shows that the decision version of *Choose-1* is NP-Complete too.

Theorem 1 *The* Max Cut *problem is NP-Hard.*

Proof: By a reduction from *Maximum 2-Satisfiability* [Karp '72]. □

Theorem 2 *The* Choose-1 *problem is NP-Hard.*

Proof: By a reduction from *Max Cut*. Given an instance graph $G = (V, E)$ of the *Max Cut* problem, we build a graph G^* for the *Choose-1* problem. For each vertex $v_i \in V$, we construct a pair of vertices v_i^1 and v_i^2 in G^*. For each edge e between two vertices v_i and v_j, we construct a pair of edges in G^*, which connect v_i^1 to v_j^2 and v_i^2 to v_j^1, and assign them the weight $w(e)$. The construction is linear in the size of the input. It is trivial to verify that a selection of vertices in the *Choose-1* problem which yields maximal weight of the corresponding selected edges implies optimal partition of the vertices in the *Max Cut* problem (where V_1 is the set of all vertices v_i for which v_i^1 was selected, and V_2 is the complementary set), and vice versa. □

It is important to note that relaxing the requirement that the weights of the two edges of the same pair are equal does not make the problem any easier. Theorem 2 implies that the relaxed problem is NP-Hard too, or else the original *Choose-1* problem would not have been NP-Hard. But even without Theorem 2 we could prove that the relaxed problem is NP-Hard, by a reduction from the problem of finding an *Independent Set* with a maximal size in a graph.

In practice, though, getting an inconsistent collection of candidate matches is very rare (provided that we do not accept matches that are too short). This is because the boundary of real models is always intended to be orientable. In the rare cases where our algorithm erroneously produces a match that is wrongly directed, this usually happens when the match is very short, and then the two match orientations are suggested, thus overlap, and are therefore eliminated.

It is trivial to decide whether the set of suggested matches is consistent or not. We do that in a DFS-like process on another graph, whose vertices are the connected components of the polyhedron boundary, and whose edges are the suggested matches between them. We arbitrarily choose a vertex (a component) as the start of the search, and assign to it an arbitrary orientation; each traversed edge (match) implies a consistent orientation of the newly visited component. All we have to do is to check consistency of all pairs of vertices connected by back-edges of the DFS. This step is applied for each connected component of the new graph.

If the set of suggested matches is not consistent, we use the following simple heuristic. We maintain a collection of pairwise-disjoint sets of connected components of the polyhedron boundary, where the components in each set have been stitched together by already accepted matches. Initially, each connected component of the polyhedron is put in a separate singleton set. In each step we examine one match, in decreasing score order. If the match connects between components of different such sets, then we merge the two sets, (and if necessary, invert the orientation of all the components of one set), and accept the match. In case the match connects between

components of the same set, we accept the match only if it is consistent with the current component orientations; otherwise it is rejected.

We implemented this heuristic using a disjoint-set data structure, originally proposed in [Galler & Fischer '64]. The only operation we had to add is the inversion of orientation of all the members of one set, if necessary, in merging two sets. We do that with no extra charge as part of the regular *makeset*, *find* and *link* operations (as is the terminology in [Tarjan '83]). Instead of maintaining the orientation of a component, we indicate (using one bit) whether it is consistent with the immediate ancestor component in the rooted tree, which implements the set of components. At the beginning, we perform all the *makeset* operations, creating sets which all contain only one component consistent with itself. Processing a match suggestion requires two *find* operations. During the traversal of the path from a component to the root of its set, we also compute the consistency state between the component and the root. Thus, if the match connects components of the same set, we can directly conclude whether the match is consistent with the previous ones or not, and accept only consistent matches. If the match connects components of different sets, we *link* the two sets, and reset the consistency bit of the component which ceases to be a root. It now points to the other root, which has just become the root of the merged set.

Finally, in case one connected component of the polyhedron boundary has several borders, we must orient them with consistent directions. This can be done simply by adding artificial "matches" between these borders, with scores which dominate those of real matches. Thus, their acceptance is guaranteed.

We summarize below the complete procedure for pruning match suggestions:

1. Add new match suggestions for each connected component of the polyhedron boundary which has several borders. These matches should define consistent directions to all the borders of a component, and should be scored higher than every original match suggestion (the scores are needed only if the set of match suggestions is not consistent).

2. Check whether all the match suggestions are consistent. For this purpose build a graph G^m, where each connected component of the polyhedron boundary is a vertex in G^m, and each match suggestion is an edge in G^m. Choose arbitrarily a vertex of G^m and assign an arbitrary orientation to it. Perform a depth-first search in G^m. For every edge of the search, assign an orientation to the newly visited vertex according to the match corresponding to the edge. For every back-edge of the search, check whether its corresponding match is consistent with the already assigned orientations of its two incident vertices. If all the back-edges are consistent, then accept all the match suggestions and go to step 5. Otherwise proceed to step 3.

3. Sort all the match suggestions according to their decreasing score order. Construct (*makeset*) a singleton set for each connected component C_i of the boundary, where i goes from 1 to the number of connected components. Maintain

for each connected component its *inconsistency* with the root of its set, represented as an external flag, and initialize it to be false. (We use *inconsistency* flags rather than *consistency* flags in order to simplify the computation of the relative consistency between nodes in the structures.)

4. Process sequentially all the match suggestions in their decreasing score order. For each match suggestion between components C_i and C_j do the following:

   - *Find* the set S_{k_i} (S_{k_j}) which contains the connected component C_i (C_j). Denote its root by C_i^* (C_j^*). Recompute the *inconsistency* of C_i (C_j) with respect to C_i^* (C_j^*), as the exclusive or of the *inconsistency* flags encountered along the path from C_i (C_j) to C_i^* (C_j^*).
   - If $k_i \neq k_j$ then accept the match suggestion and *link* S_{k_i} and S_{k_j}. Assume without loss of generality that C_i^* now points to C_j^* as its parent. Reset the *inconsistency* of C_i^* according to the accepted match.
   - If $k_i = k_j$ and C_i and C_j are *consistent* (namely, their *consistencies* are either both true or both false), then accept the match suggestion.
   - Otherwise, if $k_i = k_j$ and C_i and C_j are not *consistent*, then reject the match suggestion.

5. Orient all the connected components in a way consistent with the accepted matches. Each connected component is now labeled to indicate whether it should be inverted or not. If we reach this step from step 2, then each component is classified according to whether it is consistent with the root vertex of the DFS on G^m (or with its local root, if G^m is a forest). Otherwise, if we reach this step from step 4, then each component is classified according to whether it is consistent with the root of its set (again, there might be more than one set, if the accepted matches did not connect between all the components of the polyhedron boundary). Inverting a component is performed by inverting the directions of all the facets of the component.

6 Filling the Gaps

6.1 Stitching the Matching Borders

Each match consists of two directed polygonal chains, which are very close to each other in the three-dimensional space. We arbitrarily choose one end of the match, and "merge" the two chains as if they were sorted lists. In each step we have pointers to the current vertices, u and v, in the two chains, and make a decision as to which chain should be advanced, say v advances to a new vertex w. Then we add the new triangle $\triangle uvw$ to the boundary of the polyhedron (oriented consistently with the borders), and advance the current vertex (from v to w) along the appropriate chain. When we reach the last vertex of one chain, we may further advance only the other one. This process terminates when we reach the last vertices of both chains.

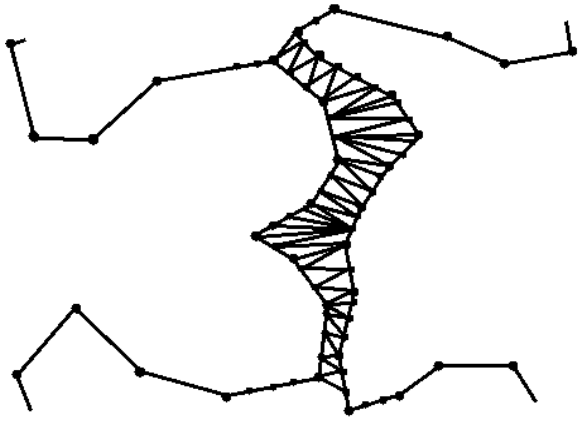


Figure 3: Stitching a match

Several advancing rules were examined, and the following simplest one proved itself the best. Assume that the current vertices of the two chains are v_i^1 and v_j^2, which are followed by v_{i+1}^1 and v_{j+1}^2, respectively. Then, if $|\overline{v_i^1 v_{i+1}^1}| + |\overline{v_j^2 v_{i+1}^1}| < |\overline{v_i^1 v_{j+1}^2}| + |\overline{v_j^2 v_{j+1}^2}|$, we advance the first chain; otherwise we advance the second chain[2]. This bears close resemblance to the merging of two sorted lists, and turns out to produce reasonably looking match triangulations. Figure 3 shows such a triangulation. We may further examine the sequence of newly added triangles, and unify adjacent coplanar (or nearly coplanar) triangles into polygons with larger sizes. Alternative advancing rules were described in [Christiansen & Sederberg '78] and in [Ganapathy & Dennehy '82].

As an alternative to this merge simulation, we also examined match triangulation by the procedure described in Section 6.2. This method turned out to produce rather unaesthetic results, although it yielded a minimum-area triangulation.

6.2 Filling the Holes

After stitching the borders, we are likely to remain with small holes at the junctions of cracks, as illustrated in Figure 5(g). These are 3-dimensional polygons, which are composed of portions of the borders which were not matched. This happens either because they do not meet the matching threshold, or because they belong to two or more overlapping matches, and are thus removed.

Identifying the holes is done in exactly the same way the original borders were located (see Section 4). In fact, these holes are the borders of the new boundary after the stitching phase. We found out that the best way to fix these holes was by a triangulation which minimizes the total area of the triangles. We, therefore, need to solve the following problem:

Given a 3-dimensional closed polygonal curve P, and an objective function

[2]That is, we advance so that the newly added triangle has smaller perimeter. Actually, we used the squares of the distances, with equally good results.

$\mathcal{F}$ defined on all triangles (called *weight* in the sequel), find the triangulation of P (i.e., a collection of triangles spanned by the vertices of P, so that each edge of P is incident to only one triangle, and all other triangle edges are incident to two triangles each), which minimizes the total sum of $\mathcal{F}$ over its triangles.

For this purpose, we closely follow the dynamic programming technique of [Klincsek '80] for finding a polygon triangulation in the plane, which minimizes the total sum of edge lengths. Let $P = (v_0, v_1, \ldots, v_{n-1}, v_n = v_0)$ be the given polygon. Let $W_{i,j}$ $(0 \leq i < j \leq n-1)$ denote the weight of the best triangulation of the polygonal chain $(v_i, \ldots, v_j, v_{j+1} = v_i)$. We apply the following procedure:

1. For $i = 0, 1, \ldots, n-2$, let $W_{i,i+1} := 0$, and for $i = 0, 1, \ldots, n-3$, let $W_{i,i+2} := \mathcal{F}(v_i, v_{i+1}, v_{i+2})$. Put $j := 2$.

2. Put $j := j + 1$. For $i = 0, 1, \ldots, n-j-1$ and $k = i + j$ let

$$W_{i,k} := \min_{i<m<k} [W_{i,m} + W_{m,k} + \mathcal{F}(v_i, v_m, v_k)].$$

   Let $O_{i,k}$ be the index m where the minimum is achieved.

3. If $j < n-1$ then go to step 2; otherwise the weight of the minimal triangulation is $W_{0,n-1}$.

4. Let $\mathcal{S} := \emptyset$. Invoke the recursive function *Trace* with the parameters $(0, n-1)$.

```
Function Trace (i, k):
if i + 2 = k then
    S := S ∪ △v_i v_{i+1} v_k;
else do:
    a. let o := O_{i,k};
    b. if o ≠ i + 1 then Trace (i, o);
    c. S := S ∪ △v_i v_o v_k;
    d. if o ≠ k − 1 then Trace (o, k);
od
```

At the termination of the algorithm, $\mathcal{S}$ contains the required triangulation of P. For our purposes, $\mathcal{F}(u, v, w)$ is taken to be the area of the triangle $\triangle uvw$. In practice, in order not to totally ignore the aesthetics of the triangulation, we actually added a measure of "beauty" of a triangle to the weight function $\mathcal{F}$, by making it slightly depend on the lengths of the triangle edges and on the spatial relations between them. This avoided in most of the cases the creation of long skinny triangles. Figure 4 shows an example of a triangulation of a hole.

As in the stitching process, we may also test edges shared by newly added triangles, and unify groups of adjacent coplanar (or nearly coplanar) triangles into polygons with larger sizes.

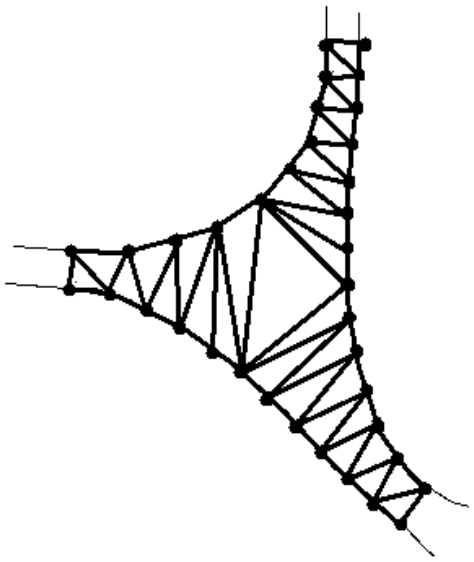


Figure 4: Minimum area triangulation of a hole

7 Complexity Analysis

We measure the complexity of the algorithm as a function of two variables: k, the size of the input (say, the number of vertices of the original object), and n, the total number of vertices along the border edges after the discretization. We denote the number of components by c, and the number of match suggestions by m. Naturally, c and m are very much less than n. We also denote the number of triangulated holes by h, and the complexity of the ith hole ($i = 1, \ldots, h$) by ℓ_i.

We do not regard the computation of the connectivity (facets adjacency) information as part of our algorithm, since this should be part of the input. However, our preprocessing step generates this information, if needed, in $O(k \log k)$ time, which is dictated by the sorting of the input vertices. The connected components of the boundary of the polyhedron can be found in time linear in k. The time needed for finding their borders is also $O(k)$. The discretization of the borders takes $O(n)$ time.

As in [Hong & Wolfson '88], the voting step, if it uses a hash table, can be executed with expected $O(n)$ running time. This expected running time is due to the nature of hashing, and does not assume anything about the geometry of the input polyhedron to the algorithm. Nevertheless, it assumes a reasonable choice of the proximity parameter ε, which should yield in average a constant number of output vertices for each range-searching query. Improper choice of ε, say equal to the size of the whole object, will result in $\Theta(n^2)$ access operations to the hash table, but no matches will be identified in this case. We infer the matches from the voting results also in $O(n)$ time. (We could also achieve an $O(n \log^2 n)$ deterministic running time by using fractional cascading.)

Since choosing the maximal consistent set of matches is NP-Hard (at least in its

abstract non-geometric setting), we instead just check for consistency. As a DFS in a graph, it requires only $O(m)$ time. If the collection of match suggestions is not consistent, we invoke the heuristic described in Section 5.3, that takes $O(m \log m + m\alpha(m,c))$ time[3], which is negligible. Stitching the borders, as merging sorted lists, is linear in their sizes. Thus, the required time for all these operations is, again, $O(n)$.

Finding the remaining holes is now done in $O(k+n)$ time, since this is the size of the new version of the object. The triangulation of each hole is done in time cubic in its size. So the triangulation of all the holes requires $O(\sum_{i=1}^{h} \ell_i^3)$ time. Since $\sum_{i=1}^{h} \ell_i \leq n$, it follows from the averages inequality that $\sum_{i=1}^{h} \ell_i^3 \leq n^3/h^2$. In the worst case, where there is a hole whose complexity is proportional to that of the whole input, this term is $O(n^3)$. Nevertheless, these cases are very unlikely in practice. Usually, the number of holes is linear in the number of surfaces (and hence borders), and their size is very small. The size of a hole is usually bounded by some constant (especially with a proper choice of the proximity parameters that control the voting process), so this step, although it is asymptotically inefficient, does not require more than $O(n)$ time in practice.

To conclude, the whole algorithm runs on practical instances in average $O(k+n)$ time, which is optimal. If we also count the connectivity computation in the preprocessing, the algorithm runs in expected $O(k \log k + n)$ time. In unrealistic cases, where the hole(s) are as complex as the polyhedral boundary itself, the running time may climb to as high as $O(k+n^3)$ (or to $O(k \log k + n^3)$, when we also compute the connectivity).

The following section describes our rather comprehensive experimentation with the algorithm. In all cases that we tried, the running time was indeed small, and the cost of the (theoretically expensive) hole-stitching step was invariably negligible.

8 Experimental Results

We have implemented the whole algorithm on a Digital DECstation 5000/240 and on a Sun SparcStation II in C. We have experimented with dozens of CAD files whose boundaries contained gaps, and obtained excellent results in most of the cases. These CAD files were generated by various CAD systems, such as Euclid-IS (vendor: Matra Datavision), Unigraphics (McDonnell Douglas), Catia (IBM/Dassot), CADDS-4X (Computer Vision), ME (Hewlett Packard), Pro-Engineer (Parametric Technologies), and many others. We note that most of the problems occurred when stand-alone computer programs translated curved objects from neutral file format, e.g. VDAFS [VDA '87] or IGES [NIST '91], into their polyhedral approximations. Fewer problems appeared when the CAD systems performed the same task, using a built-in function which converts the internal data into an output file which contains a polyhedron description. The files described industrial models, primarily parts and subunits of the

[3] $\alpha(m,n)$ is an extremely slowly growing functional inverse of Ackermann's function. For all practical values of m and n, $\alpha(m,n)$ does not exceed a very small constant; cf. [Tarjan '83].

automotive industry, which were extracted from CAD systems for the fabrication of three dimensional prototypes. Most of the models were specified in the STL [3DS '88] file format, which is the de-facto standard in the rapid prototyping industry.

The tuning of the parameters was very robust, and large parameter ranges produced nearly identical results. We usually used 0.1 mm as the discretization parameter, and 0.5 mm for the voting threshold ε (for models whose global size was between 2 to 20 cm in all dimensions). We allowed up to two successive point mismatches along a match. A point-to-point match contributed the amount of $1/(d+0.1)$ to the match score, where d was the distance between the two points. We considered only match suggestions which received more than 10 votes and whose scores were above 25.0. $\mathcal{F}$ was taken to be $0.85A+0.05P+0.10R$, where A was the area of the triangle, P was its perimeter, and R was the ratio between the largest and the smallest of its three edges.

All these parameters were user defined, but modifying them did not achieve any better results. Therefore, the program set these defaults for the parameters, except for the voting threshold ε. As noted above, this should be the user estimation of the physical size of the gaps between the original surfaces. Although there was a large enough range of valid ε settings, the algorithm failed more often if this parameter was chosen improperly. A too small ε resulted in the loss of matches due to the lack of votes. On the other hand, a too large ε resulted in a noisy voting table. This usually caused "intuitive" matches not to have scores sufficiently larger than incorrect match suggestions. Even when such intuitive matches were identified, they overlapped with erroneous match suggestions and were therefore eliminated. In all cases, too many unmatched border portions were passed to the triangulation step. When the remaining holes were not local (with respect to the geometric features of the original model), their triangulations constructed new surfaces far beyond the intention of the model designer.

Figure 5(a) shows a synthetic example of an object similar to a cube, whose boundary was broken into eight components, none of which is planar. Figure 5(b) shows the same object in a wire-frame representation. Figures 5(c,d) show the object after the whole repairing process. Figures 5(e,f,g,h) show the different steps of the algorithm. The borders are shown in (e), the matches (after stitching the borders) are shown in (f), the remaining holes are shown (with the same perspective view) in (g), and their minimal area triangulations are shown in (h).

Figure 6(a) shows another synthetic example, where a sphere contains very small holes in the poles. Figures 6(b,c) show one of the poles, before and after triangulating the hole, respectively.

Real examples turned out to be simpler, since the matching borders were much longer and closer than in the synthetic examples. Many polyhedral approximations of surface-modeling objects were repaired using the algorithm described in this paper.

A typical example (a hollowed trimmed box) is shown in Figure 7. The front side of the object is shown in Figure 7(a), whereas its bottom side is shown in Figure 7(b).

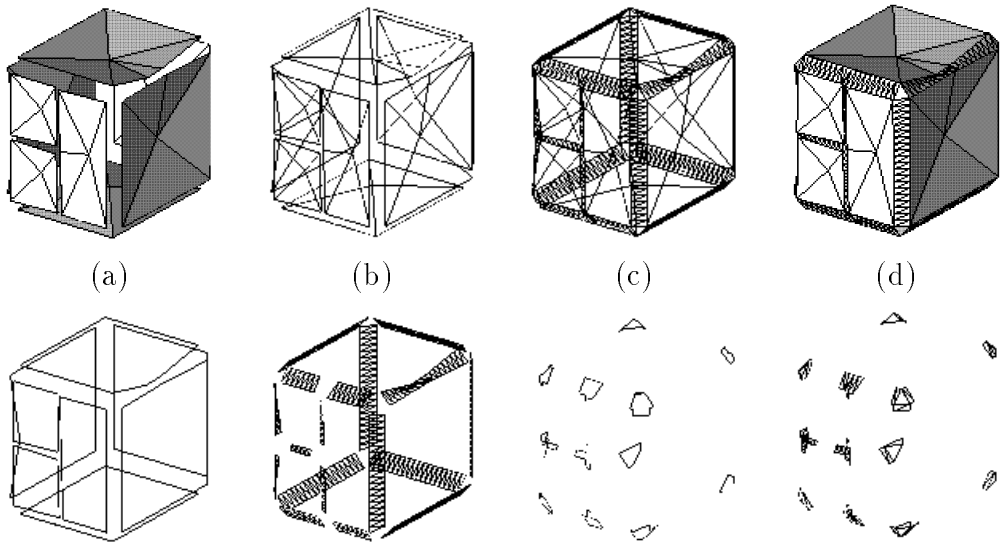


Figure 5: A synthetic example

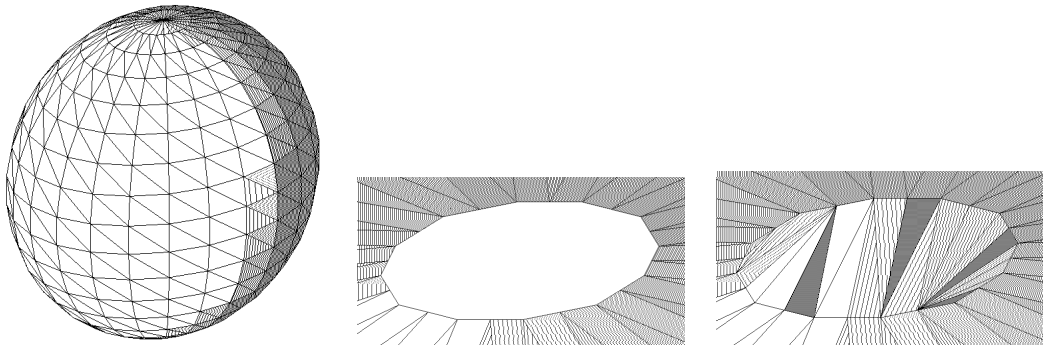


Figure 6: A sphere with perforated poles

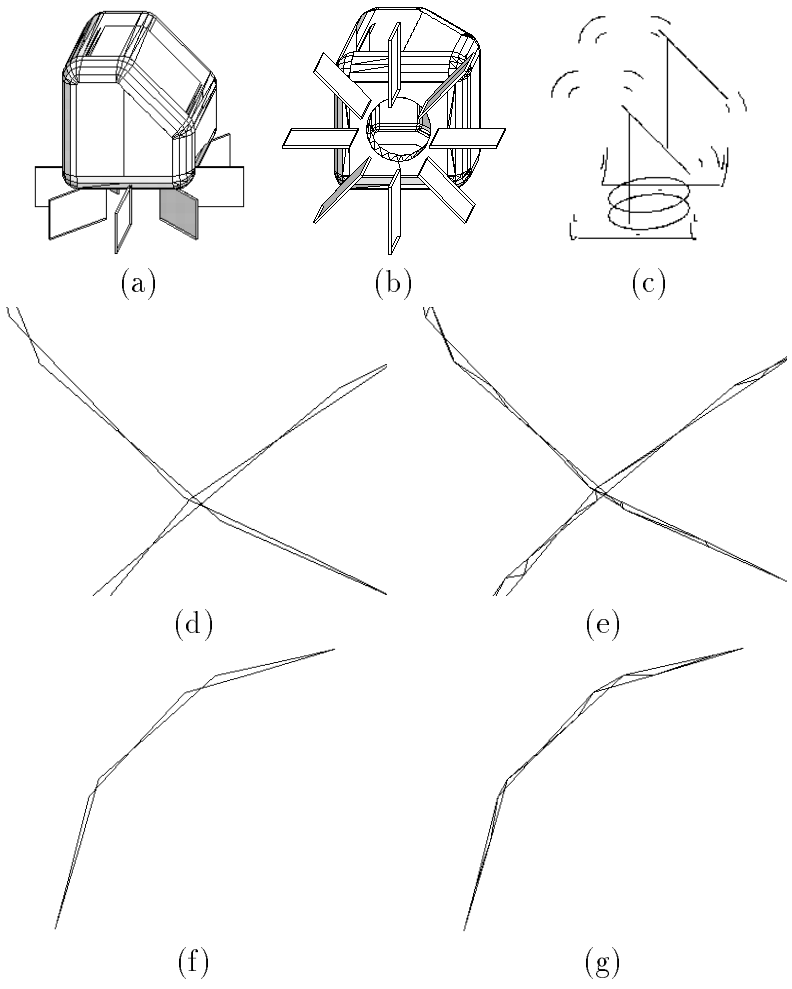
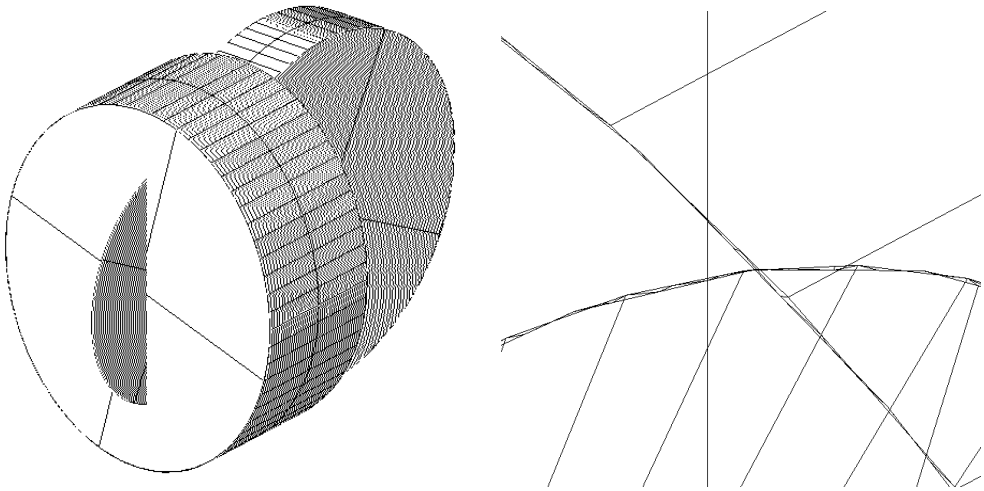


Figure 7: A real example



(a) (b)

Figure 8: Three intersecting "drums"

The borders are shown in Figure 7(c), and all of them are matched, leaving only six holes. Since the holes are so small, they appear as dots in this scale. Figures 7(d,e) are closeups of the area near the left intersection of the two pairs of circular borders (as seen in (c)), before and after stitching, respectively. Figures 7(f,g) are closeups of a hole before and after its minimum-area triangulation.

Figures 8 and 9 show two more real cases. Figure 8(a) shows two intersecting "drums", where a third part protrudes out of the near drum. Figure 8(b) is a closeup of two tiled matches (from a different point of view), in a wire-frame representation. Figure 9(a) shows a complex of five open hollowed cylinders. One long gap separates between the inner and outer surfaces of the complex. Two portions of the single border, that completely bounds this gap, were fully matched and tiled. Figure 9(b,c) are closeups of the tiled match.

Figure 10 shows three more real examples which our algorithm fully cured. We summarize the performance of our implementation on all the examples described above in Table 1. All the time measurements were taken on a Digital DECstation 5000/240.

9 Conclusion

We have proposed in this paper an algorithm for solving a practical problem, namely the detection and repair of gaps in the boundary of a polyhedron. This problem arises quite frequently in practice, during the computation of polyhedral approximations of CAD models, whose boundaries are described using curved entities of higher levels.

We applied several techniques in our solution. First, we used a partial curve matching technique, adapted from computer vision, for identifying matching border

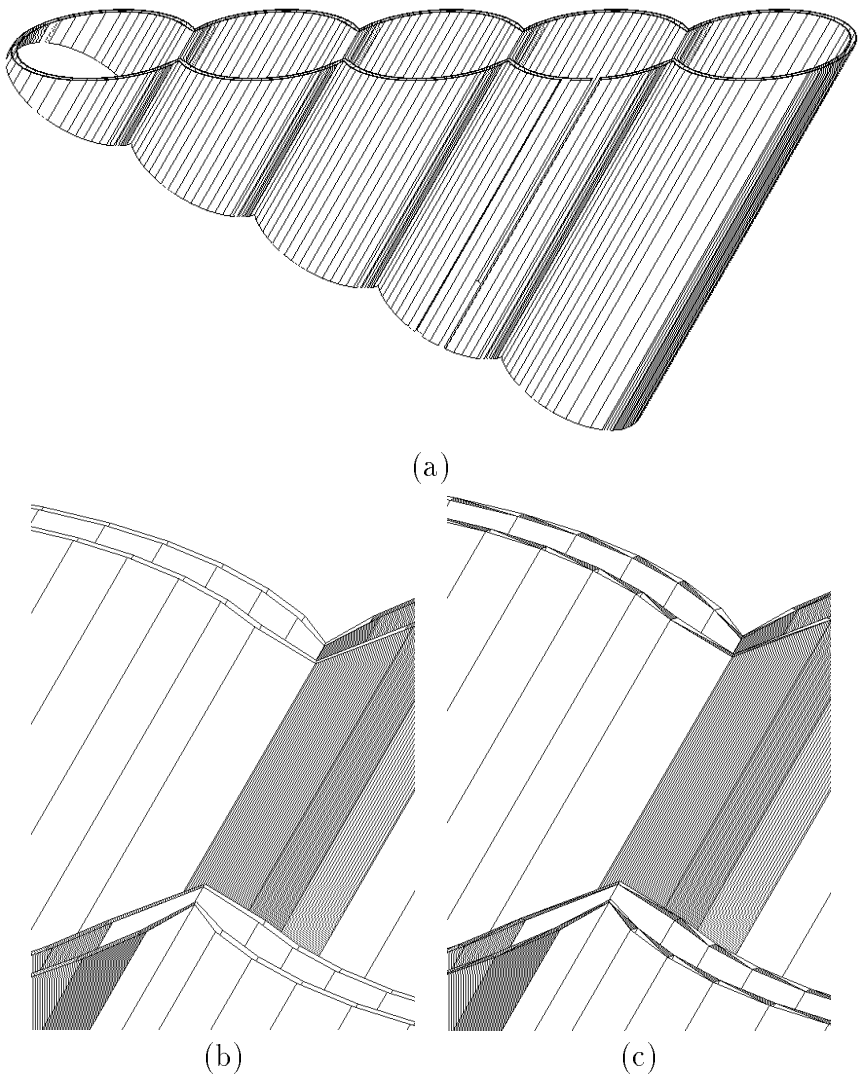
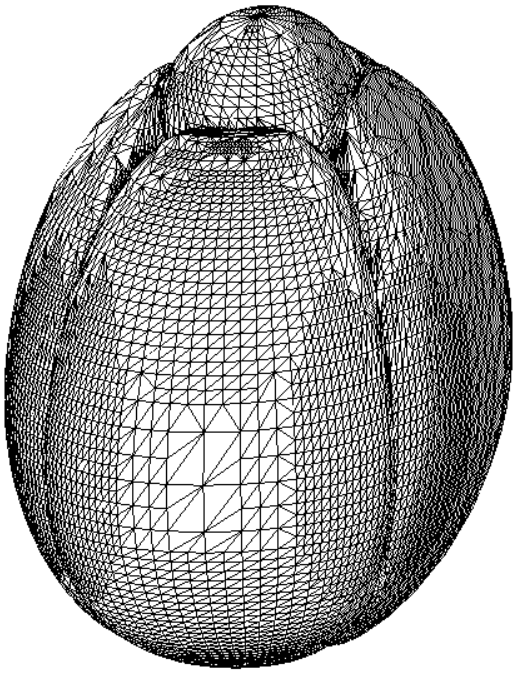
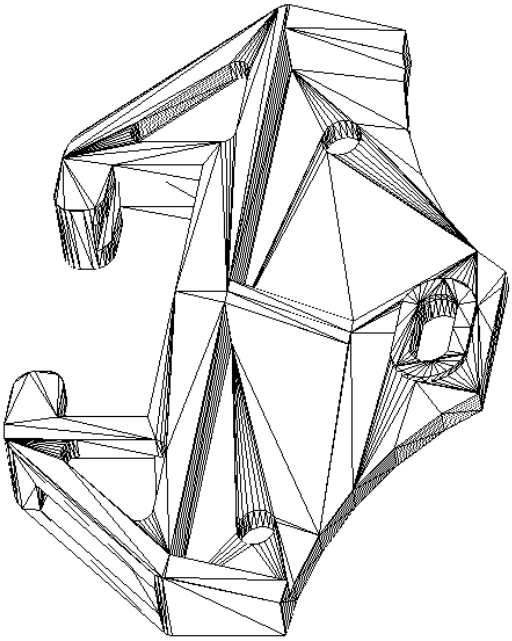


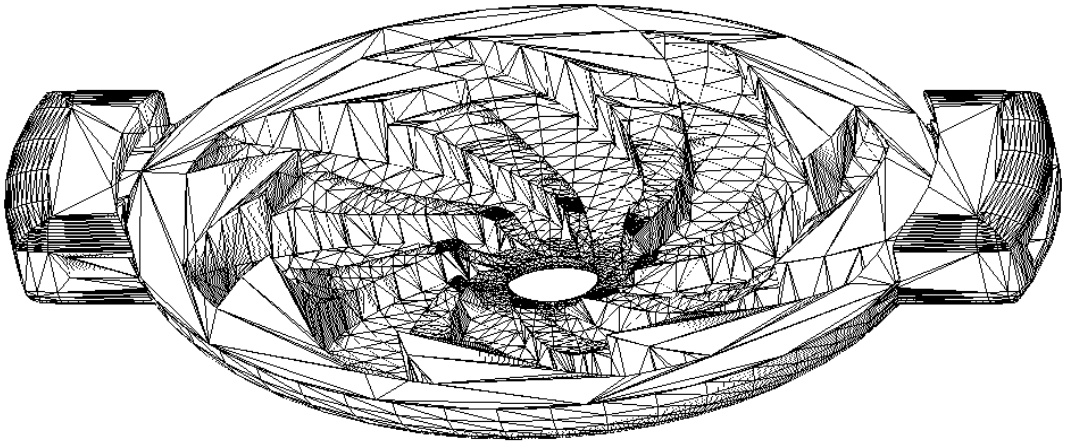
Figure 9: A complex of five open hollowed cylinders



(a) Impeller



(b) Drawer



(c) Adapter

Figure 10: Three more real examples

Model	Complexity			Borders	Border Points	Matches	Holes	Hole Points
	Vertices	Edges	Facets					
Synthetic	45	111	37	8	278	18	12	61
Sphere	432	2,448	816	2	48	0	2	48
Box	1,545	3,244	609	32	5,305	30	6	47
Drums	826	1,712	372	19	3,106	35	22	233
Cylinders	1,571	3,132	783	1	1,592	1	0	0
Impeller	7,284	43,635	14,545	11	47	0	11	47
Drawer	458	2,817	939	1	203	1	0	0
Adapter	6,539	44,682	14,894	27	84	0	27	84

Model	Time (Seconds)		
	Stitching	Triangulation	Total
Synthetic	0.48	0.20	0.68
Sphere	0.13	0.06	0.19
Box	10.14	0.60	10.74
Drums	5.95	0.20	6.15
Cylinders	3.60	0.00	3.60
Impeller	1.12	0.35	1.47
Drawer	0.24	0.00	0.24
Adapter	2.48	0.06	2.54

Table 1: Performance of the algorithm

portions. Then, we chose a (nearly) maximal consistent set of match candidates, and stitched them together. Finally, we identified the remaining holes and employed a minimum area triangulation of 3-D polygons in order to fill them.

This method can be applied to other geometric problems, where portions of 3-dimensional polygonal curves should be matched. In a companion paper [Barequet & Sharir '93], we are presently investigating the extension of our research to the problem of piecewise linear interpolation between parallel polygonal slices.

Acknowledgment

The authors wish to thank Haim Wolfson for useful discussions concerning the geometric hashing technique and its applications. The first author would also like to thank Cubital Ltd. for contributing data files.

References

Barequet, G. and Sharir, M. (1993), Piecewise-linear interpolation between polygonal slices, Technical Report 275/93, Department of Computer Science, Tel Aviv University.

Bøhn, J.H. and Wozny, M.J. (1992), Automatic CAD-model repair: Shell-closure, *Proc. Symp. on Solid Freeform Fabrication*, Dept. of Mech. Eng., Univ. of Texas at Austin, 86–94.

Chazelle, B. (1988), A functional approach to data structures and its use in multidimensional searching, SIAM J. of Computing 17 (3), 427–462.

Christiansen, H.N. and Sederberg, T.W. (1978), Conversion of complex contour line definitions into polygonal element mosaics, Computer Graphics 13, 187–192.

Dolenc, A. and Mäkelä, I. (1991), Optimized triangulation of parametric surfaces, Technical Report TKO-B74, Helsinki University of Technology, to appear in Mathematics of Surfaces IV.

Foley, J.D. and Van Dam, A. (1984), *Fundamentals of Interactive Computer Graphics*, Addison Wesley.

Galler, B.A. and Fischer, M.J. (1964), An improved equivalence algorithm, Comm. of the ACM 7, 301–303.

Ganapathy, S. and Dennehy, T.G. (1982), A new general triangulation method for planar contours, ACM Transactions on Computer Graphics 16 (3), 69–75.

Garey, M.R. and Johnson, D.S. (1979), *Computers and Intractability, A Guide to the Theory of NP-Completeness*, Freeman and Co., San Francisco.

Guibas, L. and Stolfi, J. (1985), Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, ACM Transactions on Graphics 4 (2), 74–123.

Ho-Le, K. (1988), Finite element mesh generation methods: A review and classification, Computer-Aided Design 20 (1), 27–38.

Hong, J. and Wolfson, H.J. (1988), An improved model-based matching method using footprints, *Proc. 9th Int. Conf. on Pattern Recognition*, 72–78.

Kalvin, A., Schonberg, E., Schwartz, J.T. and Sharir, M. (1986), Two-dimensional, model-based, boundary matching using footprints, Int. J. of Robotics Research 5 (4), 38–55.

Karp, R.M. (1972), Reducibility among combinatorial problems, in: Miller, R.E. and Thatcher, J.W., eds., *Complexity of Computer Computations*, Plenum Press, New York, 85–103.

Kishon, E., Hastie, T. and Wolfson, H. (1991), 3-D curve matching using splines, J. of Robotic systems 8, 723–743.

Klincsek, G.T. (1980), Minimal triangulations of polygonal domains, Annals of Discrete Mathematics 9, 121–123.

Mäkelä, I. and Dolenc A. (1993), Some efficient procedures for correcting triangulated models, *Proc. Symp. on Solid Freeform Fabrication*, Dept. of Mech. Eng., Univ. of Texas at Austin.

Mehlhorn, K. (1984), *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, Brauer, W., Rozenberg G. and Salomaa A., eds., Springer-Verlag.

Pratt, W.K. (1978), *Digital Image Processing*, Wiley, NY.

Rock, S.J. and Wozny, M.J. (1992), Generating topological information from a "bucket of facets", *Proc. Symp. on Solid Freeform Fabrication*, Dept. of Mech. Eng., Univ. of Texas at Austin, 86–94.

Samet, H. and Webber, R.E. (1988), Hierarchical data structures and algorithms for computer graphics; Part II: Applications, IEEE Computer Graphics & Application 8 (4), 59–75.

Schwartz, J.T. and Sharir, M. (1987), Identification of partially obscured objects in two and three dimensions by matching noisy characteristic curves, Int. J. of Robotics Research 6 (2), 29–44.

Sheng, X. and Hirsch, B.E. (1992), Triangulation of trimmed surfaces in parametric space, Computer-Aided Design 24 (8), 437–444.

Sheng, X. and Tucholke, U. (1991), On triangulating surface model for SLA, *Proc. 2nd Int. Conf. on Rapid Prototyping*, Dayton, OH, 236–239.

Snyder, W.E., Groshong, R., Hsiao, M., Boone, K.L. and Hudacko, T. (1992), Closing gaps in edges and surfaces, Image and Vision Computing 10 (8), 523–531.

Tarjan, R.E. (1983), *Data Structures and Network Algorithms*, SIAM, Philadelphia.
National Institute of Standards and Technology (NIST), MD (1991), *The Initial Graphics Exchange Specification (IGES), Version 5.1.*

Verband der Automobilindustrie e.V. (VDA), Germany (1987), *VDA Surface Interface, Version 2.0.*

Yerry, M.A. and Shephard, M.S. (1983), A modified quadtree approach to finite element mesh generation, IEEE Computer Graphics & Applications 3 (1), 39–46.

Wolfson, H.J. (1990), On curve matching, IEEE Transactions on Pattern Analysis and Machine Intelligence 12 (5), 483–489.

3D Systems, Inc. (1988), *Stereolithography Interface Specification.*