Repairing CAD Models*

Gill Barequet Subodh Kumar

Department of Computer Science
Center for Geometric Computing
Johns Hopkins University
Baltimore, MD 21218-2694
Email: {barequet,subodh}@cs.jhu.edu
URL: http://www.cs.jhu.edu/~{barequet,subodh}

Abstract

We describe an algorithm for repairing polyhedral CAD models that have errors in their B-REP. Errors like cracks, degeneracies, duplication, holes and overlaps are usually introduced in solid models due to imprecise arithmetic, model transformations, designer's fault, programming bugs, etc. Such errors often hamper further processing like finite element analysis, radiosity computation and rapid prototyping. Our fault-repair algorithm converts an unordered collection of polygons to a shared-vertex representation to help eliminate errors. This is done by choosing, for each polygon edge, the most appropriate edge to unify it with. The two edges are then geometrically merged into one, by moving vertices. At the end of this process, each polygon edge is either coincident with another or is a boundary edge for a polygonal hole or a dangling wall and may be appropriately repaired. Finally, in order to allow user-inspection of the automatic corrections, we produce a visualization of the repair and let the user mark the corrections that conflict with the original design intent. A second iteration of the correction algorithm then produces a repair that is commensurate with the intent. Thus, by involving the users in a feedback loop, we are able to refine the correction to their satisfaction.

1 Introduction

CAD (Computer-Aided Design) models are often represented as unordered lists of polygons – sometimes referred to as "soups of polygons." File formats like IGES [NCG91], DXF [Aut95] and STL [3DS89] (which is a *de facto* standard in the rapid-prototyping industry) allow users to represent models as such soups of polygons. Each polygon is listed independently as an ordered list of its vertex-coordinates, occasionally along with its normal vector. The collection of polygons is assumed to represent a complete model. Unfortunately this is often not the case. Typical problems include cracks (in the surface), degeneracies, duplication (of patches of the surface), holes and overlaps, as shown in Figure 1. Cracks

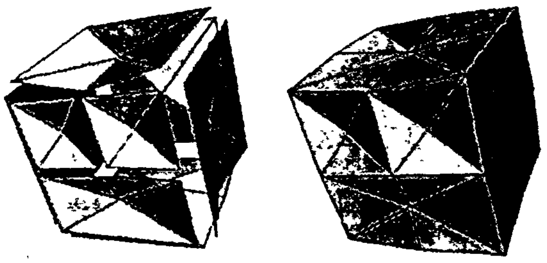

(a) Original model (b) Repaired model

Figure 1: A synthetic broken cube

occur due to inaccuracies in the data or in the process that creates the data. For example, a vertex shared by multiple polygons may be computed according to two surface equations, resulting in slightly different positions in space, causing a crack or an overlap. Such faults are especially common in models obtained by tessellating curved surfaces and are a result of different tessellations of the same boundary curve. These cracks often complicate further processing of data, specially if it requires inside-outside classification of points with respect to the model. For example, if a solid, P, is not closed, it is difficult to determine if a it interferes with another solid Q. The fabrication process in most rapid prototyping techniques often fails due to erroneous solid descriptions. Illumination algorithms compute images with artifacts in the presence of cracks in the surface due to leakage of light. [BMSW91] catalogs some common types of errors and proposes techniques to avoid them. Missing polygons are also a common source of holes in the model. Another possible error is the existence of dangling walls inside a solid or repetition of portions of the geometry; these are usually mistakes of the model designer and result from repeated operations. We present algorithms to eliminate dangling geometry, T-joints, holes and cracks in a polygonal solid model, and generate consistent polygon-orientations. We assume that the target model is a set of closed 2-manifolds [O'N66], but allow users to override this assumption.

1.1 Related Work

Several techniques for correcting some of the errors described above have been proposed in the past. Morvan and Fadel [MF96a, MF96b] describe a virtual environment that provides tools for model correction, controlled primarily by the user. Unfortunately, this can be a cumbersome and inef-

*Supported in part by the U.S. Army Research Office under Grant DAAH04-96-1-0013.

0-8186-8262-0/97 $10.00 Copyright 1997 IEEE.

ficient procedure for large models. Furthermore, it is easy to miss errors. Our approach makes the correction process automatic, at the same time including the user in the correction loop. The user can visualize the errors and guide the correction algorithm. Turk and Levoy [TL94], in order to generate polygonal models from range data, remove overlaps of polygons by clipping them. For our application, vertex-shifting seems to cause comparatively smaller perturbation to the input model. Rock and Wozny [RW92] present efficient data structures for computing exact matches between polygon edges for reconstructing the topology of the model. Bøhn and Wozny [BW92, BW93] present a technique based on Jordan curve construction for identifying holes bounded by edges at each of which only one facet occurs. They use local techniques for filling a hole by triangles. Mäkelä and Dolenc [MD93] also use local techniques for filling cracks in the model surface. Barequet and Sharir [BS95] describe a globally-consistent approach for identifying and filling holes. Unfortunately, when a large number of cracks is involved, simple-minded hole filling may result in an explosion of the number of polygons needed to describe the model. The approach of Murali and Funkhouser [MF97] is to determine regions of space that lie inside a solid using spatial partitioning and use the partition as the description of solids. Unfortunately, for degenerate cases, their output may be significantly different from the input even for small errors in the input. In our work we propose a heuristic which assumes that most of the cracks and overlaps occur due to numerical error in the computation of vertex coordinates, and hence shifts the positions of vertices by a small amount to eliminate the error. Our method ensures that no vertex is moved farther than a user-specified error-tolerance. Larger holes are filled using the triangulation technique of [BS95].

1.2 Our Approach

We present a topologically-based geometric algorithm designed to rid polygonal boundary representations (B-REP) of solid models of some common errors. We define the problem as follows:

Given input:

A list of polygons, where each polygon is specified by an ordered sequence of vertices, and each vertex is specified by three real numbers that represent its x, y, and z coordinates.

Generate the following output:

• A list of unique vertices, each specified by its three coordinates.
• A list of unique directed edges, each specified by two vertex indices.
• A list of unique polygons, each specified by an ordered sequence of edge indices. Each polygon is oriented counter-clockwise when it is viewed from outside the model.

We have implemented a system called *RSVP* (Repair by Shifting Vertices of Polygons). RSVP takes as input a soup of polygons and outputs the adjacency structure of the corrected model. Each edge of the resulting solid appears exactly twice, in opposite orientations, on two adjacent polygons. A brief overview of RSVP follows:

- The repair algorithm generates manifolds with consistent normal vectors. It closes small cracks and fills larger gaps with polygons. Small overlaps are detected and separated. Holes, open geometry, zero-volume solids and T-joints are also handled.
- We first compute the connected components of the surface, which are typically open manifolds. An edge does not lie on the boundary of a component if it appears in exactly two polygons. All other edges are boundary edges. Duplicate polygons are also found at this stage.
- The next step is matching each boundary edge with another that may not necessarily coincide with it. We use an adjacency score for ranking all the edge-to-edge candidates for matching, We maintain a queue of candidates sorted by this score and process candidates in that order.
- We merge two boundary edges with the minimum score by moving their end points. We continue such merges until no more merges may be made without moving a vertex by more than ε, where ε is a user specified parameter. If the angle between two polygons in a component (either before or after merges) is 0°, the component is flagged as zero-volume.
- The holes and open components that remain after the merging process are identified by a second step of computing Jordan curves on the (modified) surface. The holes can be optionally triangulated, and the dangling or zero-volume parts can be discarded.
- We produce a visualization of the model repair. Often the vertex shift is too small and thus a naive model display is useless with respect to inspecting the erros. We describe a highlighting technique in conjunction with *zoom windows* for aiding the user inspect the faults on the surface and supervise the repair process.
- Once the user overrides a decision made by the system, further iterations are performed to obtain a better result.

The rest of the paper is organized as follows. Section 2 discusses the algorithm to generate connected components of the given model. Sections 3 and 4 describe our candidate ranking and processing algorithms. Section 5 presents the final triangulation step. Section 6 presents our repair visualization technique and section 7 describes our implementation and some experimental results. We end with some concluding remarks and possible future work in section 8.

2 Computing the Connected Components

To reduce the number of candidate matchings, we first compute components of the model that are correctly specified in the input model, i.e., we merge two polygon edges, $e_1 = v_1v_2$ and $e_2 = w_1w_2$, if $e_1 = e_2$, i.e., $v_1 = w_1$ and $v_2 = w_2$, and no other edge $e_3 = e_1$. The assumption here is that no accidental error can cause exactly two edges to match each other. If this assumption is invalid for an application, each polygon may be considered a single component and passed on to the merging phase.

We compute the connected components of the model using a process similar to that of [BS95]. We first construct G, the adjacency graph of the model. Each facet of the model is a vertex in G. G includes the graph-edge F_1-F_2, if faces F_1 and F_2 contain edges e_1 and e_2, respectively, such that $e_1 = e_2$. We use a depth-first traversal of G to construct the connected components of the model. During the traversal

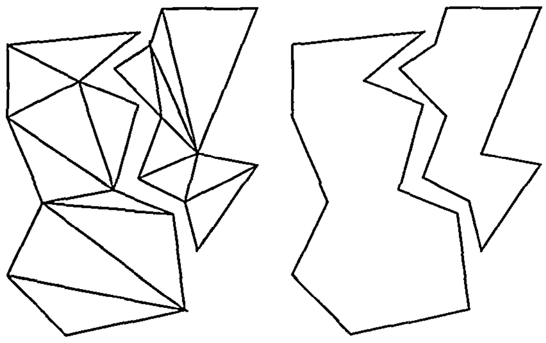


Figure 2: Boundaries of connected components

process we also orient all the facets of each component consistently. Each internal edge of the component appears exactly twice, in opposite orientations, in a pair of neighboring facets. The algorithm is also able to detect non-orientable surfaces while processing a "back-edge" in the depth-first order traversal. Back-edges are used to perform a consistency check between two facets whose respective orientations are already fixed. If the two orientation do not match, the component is non-orientable and the system reports the error as such.

Once we partition a model into connected components, we must find the boundary edges of each component, which may be later merged with the boundary edges of other components. In addition, we need to determine the preceding and following boundary-edges for each edge in a component (which are used later for computing scores). To obtain the boundary contours of a component, we compute the binary sum [BS95] of all the facets (as cycles of graph-edges). The result is the collection of the boundary polygonal contours (see Figure 2). Note that in case of T-joint (Figure 3), there are two possible preceding or following edges; we maintain pointers to both. In the merging phase, we select the one with lower score. In our current implementation, such T-joints can result in zero area polygons. It is relatively simple to add a post-processing phase to eliminate zero-polygons, and even duplicate polygon vertices. Note that, in general, a simple deletion of a zero-polygon is not sufficient. Multiple polygons may be adjacent to it. This is highlighted in Figure 3(b). Additional vertices must be introduced to eliminate T-joints (Figure 3(c)).

3 Ranking the Matching Candidates

Once connected components of a model are constructed and their boundary contours computed, we generate a list of candidate boundary-edge pairs. The two (oriented) edges of a candidate pair belong to different components. We assign scores to candidates as follows: Let $e_1 = \overrightarrow{v_1 v_2}$ and $e_2 = \overrightarrow{w_1 w_2}$ be two boundary edges (see Figure 4).

We first assume that the boundary contours are oriented correctly, i.e., all the connected components appear in their correct orientations. We later (Section 4.2) describe how to remove this assumption to deal with more complex cases when we cannot rely on the original orientations of the facets.

Our goal is to evaluate the "desirability" of merging two directed edges e_1 and e_2. For this purpose we estimate, the magnitude of the geometric error: this could be the area of the missing part of the two corresponding polygons in case of cracks or that of the extra parts in case of overlaps. The case of cracks is demonstrated in Figure 4; an estimate for the missing geometry is shown lightly shaded. For a boundary-edge pair (e_1, e_2), we compute $\Delta(e_1, e_2)$, an estimate for the relative error; we normalize the area error by the lengths of e_1 and e_2 to prevent larger errors for longer edges. In order to compute $\Delta(e_1, e_2)$, we first compute the area of the region spanned by e_1 and e_2. This is done by linearly parametrizing (by t) e_1 and e_2, respectively, and computing:

$$A(e_1, e_2) = \int_0^1 |\overrightarrow{e_1(t)e_2(t)}|^2 \mathrm{d}t = \frac{1}{3}|\overrightarrow{v_2 w_2} - \overrightarrow{v_1 w_1}|^2 + \overrightarrow{v_2 w_2} \cdot \overrightarrow{v_1 w_1} \tag{1}$$

To compute Δ, instead of integrating $|\overrightarrow{e_1(t)e_2(t)}|^2$, as done in (1), we must integrate $|\overrightarrow{e_1(t)l(t)}|^2 + \overrightarrow{l(t)e_2(t)}|^2$, where l is the line of intersection of the error-polygons (Figure 4). Note that

$$|\overrightarrow{e_1(t)e_2(t)}|^2 = |\overrightarrow{e_1(t)l(t)}|^2 + |\overrightarrow{l(t)e_2(t)}|^2 + 2|\overrightarrow{e_1(t)l(t)}||\overrightarrow{l(t)e_2(t)}|\cos(\theta),$$

where θ is the angle between the error polygons. If we make the approximation that the error is symmetric, i.e., $|\overrightarrow{e_1(t)l(t)}| = |\overrightarrow{l(t)e_2(t)}|$,

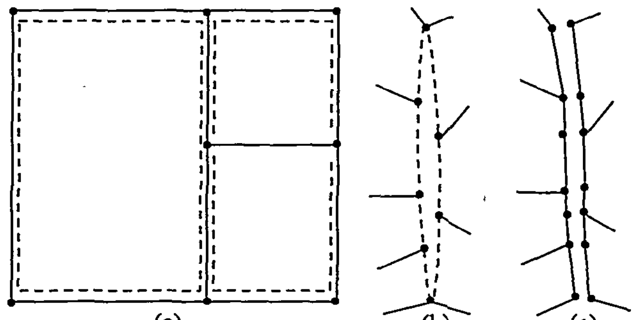
$$\Delta(e_1, e_2) = \frac{A(e_1, e_2)}{|\overrightarrow{e_1}||\overrightarrow{e_2}|\cos(\frac{\theta}{2})} = \frac{A(e_1, e_2)}{|\overrightarrow{e_1}||\overrightarrow{e_2}|N_1 \cdot N_2},$$

where N_1 and N_2, are the normals to the error polygons, respectively. Intuitively, the angle term induces our score to favor smoother matches over sharp bends.

To compute the final Score(e_1, e_2), we also consider the edge-pairs immediately preceding and following (e_1, e_2): Score(e_1, e_2) =

$$U_s(e_1, e_2) + \Delta(e_1, e_2) + 0.5\Delta(e_1^-, e_2^-) + 0.5\Delta(e_1^+, e_2^+)$$

U_s is a user-specified score for certain candidates. In order to prohibit merging of two edges e_1 and e_2, we set $U_s(e_1, e_2)$ to $+\infty$ ($-\infty$ to cause a merge). Setting of U_s is controlled by the visualization algorithm described in section 6. By default $U_s = 0$ for each candidate. While it is possible to use a continuous function (that achieves a maximum at



(a) The boundary edges are shown dashed. (b) A zero-area polygon (dashed) may be created due to t-joints. The distances have been enhanced for clarity in these diagrams. (c) The zero-area polygon may be eliminated by introducing *additional vertices on the adjacent polygons. Each unique* vertex on the zero-polygon must be included on both sides. The components on each side are now topologically adjacent.

Figure 3: T-Joints

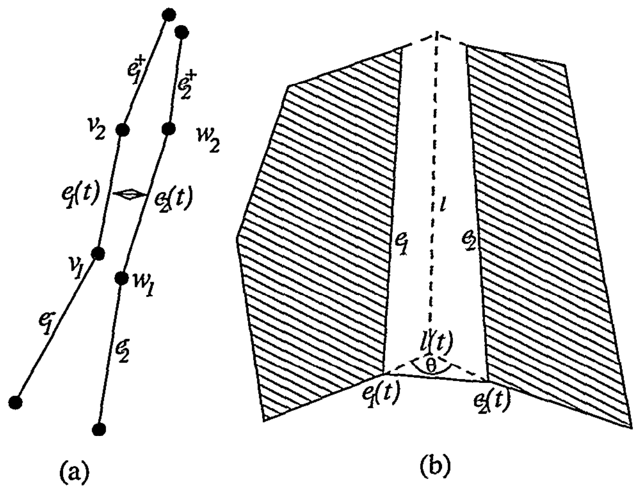


Figure 4: A candidate matching pair of edges and its score computation. Figure (b) shows l, the line of intersection of the two nominal planes containing e_1 and e_2, respectively. The nominal plane for an edge e_1 is constructed using the farthest point from it on its polygon.

$t = 0.5$ between e_1 and e_2) to weight the Δs, empirical evidence shows that an equal distribution of weights between the matching pair and the adjacent pairs works well.

We sort all the matching candidates in ascending order of score (a lower score corresponds to a better matching candidate), and maintain for each score the respective edges and their endpoints.

4 Processing the Matching Candidates

We process all the matching candidates in ascending order of score. For each boundary edge we maintain one bit which indicates whether the edge was already matched or not. Processing a matching candidate is performed only if both its edges were still un-merged and such a merge does not introduce any self-intersections. (Self intersections are ignored if the score is $-\infty$.) In order to match a pair of boundary edges $e_1 = \overrightarrow{v_1v_2}$ and $e_2 = \overrightarrow{w_1w_2}$, we merge v_1 with w_2 and v_2 with w_1. The merging is performed by replacing a pair of vertices by their average, the mid-point on the line-segment which connects between the two points. Since it is possible that, for example, v_1 and w_2 were already merged, or that v_1 (or w_2) was already merged with another vertex, this process maintains *buckets* of merged vertices. Initially, every vertex is set to a singleton bucket. Merging two vertices then actually amounts to merging the two buckets that contain them. Each bucket maintains the original coordinates of all the vertices that it contains, so that when we merge two buckets we are able to recompute the average of all involved vertices.

4.1 Vertex Merging

A formal description of the vertex-merging process follows:

```
1. Initialization:
   FOR each boundary edge $e$
       set handled($e$) := False;
   FOR each boundary vertex $v$
       create a bucket $b_v = \{v_i\}$ and
       set $b_v$.average:= $v_i$;
2. Iteration:
   FOR each matching candidate $(e_1, e_2)$ (where
           $e_1 = \overrightarrow{v_1v_2}$ and $e_2 = \overrightarrow{w_1w_2}$):
       IF handled($e_1$) = False AND
          maxShift $< \varepsilon$ AND
          handled($e_2$) = False DO
           Merge $(v_1, w_2)$;
           Merge $(v_2, w_1)$;
           Set handled($e_1$) := True;
           Set handled($e_2$) := True;
       OD

FUNCTION Merge $(v, w)$:
IF $v = w$ RETURN;
ELSE DO
    Locate the buckets $b_v$ and $b_w$ that
        contain $v$ and $w$, respectively;
    IF $b_v = b_w$ THEN
        RETURN;
    ELSE
        Set $b_v = b_v \cup b_w$;
        Recompute $b_v$.average;
        Delete $b_w$;
    FI
```

4.2 Orientation Checking

If the original orientation of facets is not consistent, we need to make the following modifications to our algorithm:

- Orient all the facets of each connected component consistently with respect to other facets in the component.
- Compute, for each pair of boundary edges, *two* matching candidates. The additional candidate consists of the same edges but with the assumption that the orientation of one of them should be inverted. The order of the considered four vertices (along one boundary contour) is also inverted for computing the score of this candidate.
- Before the vertex-merging process, initialize each boundary edge with a reference to its connected component.
- During the vertex-merging process, when two edges are matched (either in their regular orientations, or when one is inverted), the orientation *consistency* between the two respective connected components is checked. If the two edges belong to the same connected component, then the candidate can be processed only if it passes the consistency test. Otherwise, processing this candidate causes the uniting of the two respective connected components, in which case the type of the candidate, and previous possible reorientations of the components, determine whether we have to reorient one of the components (that is, invert the orientations of all its facets) in order to maintain global consistency.

These considerations are very similar to those of [BS95].

5 Filling Holes

If the user sets a small value for ε, the maximum allowed vertex-shift, there may be holes left in the models at the end of the vertex merging phase. Some of these holes may be due to large position errors, others may be due to missing polygons. A boundary edge could remain unmatched either because it did not appear in any matching candidate, or its potential matching edges were merged with other boundary edges.

Identifying the remaining holes is done in the same way the original boundary contours were located (see Section 2). These are indeed the boundary polygons of the new object after the vertex-merging step.

Our system automatically triangulates these holes (unless over-ridden by the user). A triangulation of a three-dimensional polygonal contour C is a collection of triangles which define a simply-connected 2-manifold whose boundary is C.

Barequet et al. show in [BDE96] that the decision whether a three-dimensional polygon is triangulable (that is, whether it has a triangulation which does not intersect itself) is $\mathcal{NP}$-complete. For practical reasons, we do not check whether a hole is triangulable or not. As in [BS95], we use a simple dynamic-programming technique which computes the triangulation of a hole with n edges in $O(n^3)$ time. Following [BST96], we minimize a measure which is a linear combination of the area of the triangles and the ratio (for each triangle) between the lengths of the longest and the shortest edges. Let $A(t)$ be the area of a triangle t, and let $L(t)$ (resp. $S(t)$) be the lengths of the longest (resp. shortest) edges of t. The weight of a triangle t is of the form $\omega_{\text{area}}A(t) + \omega_{\text{ratio}}L(t)/S(t)$, where ω_{area} and ω_{ratio} are user-defined weights. The rationale of this measure is the aim both to minimize the surface area of the triangulation and to avoid, as much as possible, long skinny triangles. Minimizing the total area of the triangulation is guided by the intuition that the unknown surface only whose boundary (the hole) is known is the one that minimizes the "tension" as in a soap bubble. Avoiding long skinny triangles is done because many external applications (such as finite-element analysis) rely not only on the continuity of the surface but also on its regularity.

Our experimentation shows that in practice such a triangulation indeed produces "intuitively-correct" filling of holes. Note, however, that for larger holes, it may not be clear if these are legitimate holes, or just an artifact of dangling geometry. If the sum of the area of the filled triangles is smaller than the sum of the area of the rest of the component polygons, we leave the holes marked as holes, otherwise we mark the component as dangling. The choice of the shift-bound, ε, is important for some cases. For example, two well separated hemispheres may each end up separately closed by triangulation. The correct solution in this case might be to close the two hemispheres together, i.e., to either extend the two components towards each other, or triangulate the region between the two components. Automatic control of ε and inter-component triangulation is left as future work. In our current scheme we let the user set a large enough ε, possibly after repair-visualization.

6 The Visualization Loop

While automated error correction speeds up the detection and elimination of errors, often there are multiple correct solutions. The original intent of the designer may not be followed by the heuristic for edge matching. For example, see Figure 5(a). Without considering the context, we cannot unequivocally choose between Figures 5(b) and (c), either of which could be the intended shape.

We, therefore, involve the designer in the correction loop to override incorrect decisions made by the automated process. A subsequent iteration of RSVP avoids matchings marked incorrect by the user.

A simple rendering of the faulty or corrected model is not usually effective: it is not easy to recognize the errors. We augment the simple rendering (of the input model) by highlighting the computed holes. Inspection is done in two phases, one for each type of errors – Erroneous geometry: cracks and overlaps and Extraneous geometry: inside walls, holes, zero-volume components, etc.

Dangling geometry is identified in RSVP by incomplete components, i.e., components that contain edges with a single polygon adjacent to them after the merging process.

For visualization, the cracks correspond to the merged edges. We display the cracks in red color and the rest of the model in blue to clearly highlight the repaired crackes. In addition, we also assign a different color to each solid. These are helpful in identifying cases like that in Figure 5.

Typical errors are so small that it may not be easy to find them in a simple rendering. To help visualize the errors we employ two techniques:

Fault line : Each boundary edge is drawn using a single pixel thick line (called fault line). Thus even small errors are represented in the image and can be easily located.

Zoom window : Sometimes it becomes necessary to see the actual geometry in the neighborhood of a given error. Since the dimensions of errors are usually quite small (with resect to the surrounding features of the model), the user needs to significantly zoom-in to the error. Unfortunately, this results in a loss of context and can be confusing. We allow the user to maintain context, but pop up a second (and recursively more) zoom window which shows the zoomed-in view around the selected point of the model (see Figure 10).

To highlight cracks we use two-sided flat shading of the model. All vertices that belong to the same bucket (see Section 4) are shown in the same color. In addition, any window showing an augmented input model can also be replicated with the second window showing the corrected model. A typical use of this tool is to focus in on a fault line, pop up a zoom window which shows the error in detail, and then replicate the zoom window showing the corresponding part of the corrected model. The user can select a fault line that is marked for separation. On the next iteration of the automatic correction, the score of the selected pair is artificially increased by setting U_s to $+\infty$ (see Section 3).

Extraneous geometry editing is normally performed after the user is satisfied with the results of the erroneous geometry editing. At the end of merge phase, extraneous geometry is marked as triangulated hole (shown in red), dangling piece (shown in green) or zero volume (shown in blue). If a component is zero-volume in addition to being dangling, it is shown in cyan. The complete components are either shown in gray wire-frame or not shown at all, based on user's preference. The users can select (using the mouse) any extraneous component they wish to change. The display of such selected components is changed to wire-frame in appropriate color (to facilitate "undo" operation). A selected dangling component may be marked to be retained as-is, or re-classified as one with hole. A triangulated hole may be selected to be removed. Subsequently, the rest of the component, now dangling, may either be retained or discarded. All unselected zero-volume solids and dangling components are saved into an error-file and removed from the model. Any dangling components re-classified as holes are completed by hole-filling.

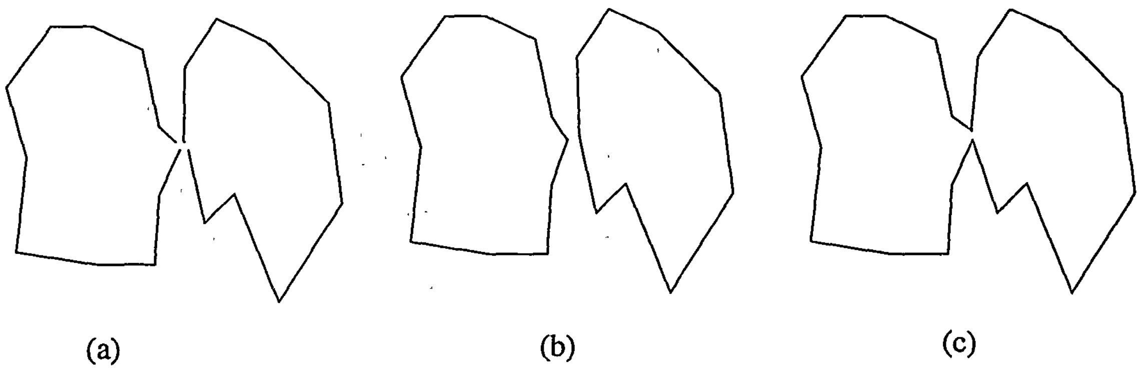


Figure 5: Design intent: Different possible correct solutions: (a) and (b). RSVP chooses one based on scores, say (b). If (c) was the original intent, the visualization stage lets the user mark the boundary which ensures that a second iteration of RSVP generates (c). The third possibility of self-intersecting parts is eliminated since that would generate an inconsistently oriented solid.

7 Implementation and Experimental Results

We have implemented the algorithm in C on a Silicon Graphics workstation. The visualization component is based on OpenGL. We have experimented with the algorithm on several data files (with many hundreds of thousand polygons) obtained from CAD systems.

Figure 1(a) shows a synthetic cube with a "broken" surface, while Figure 1(b) shows the repaired model after vertex merging.

Figure 6(a) shows a ball with a tiny holes in its poles. One such hole is displayed in Figure 6(b). Vertex merging results in the construction shown in Figure 6(c). Alternatively, if we avoid any vertex shifting by setting small enough maximum allowed shift, the holes are triangulated (by minimizing the total area of the triangles) as shown in Figure 6(d).

Figure 7(a) shows a mechanical T-box which was cracked manually. The boundary contours of this model are displayed in Figure 7(b). The T-box repaired by vertex merging is shown in Figure 7(c).

Figure 8(a) shows a portion of a model of a submarine storage and handling room with dimensions $45,433 \times 10,952 \times 8,407$ coordinate units. The boundary contours of this model are displayed in Figure 8(b). We avoided merging of vertices whose mutual distance was above 10 units. Consequently, some holes remained after the vertex merging step. These holes were then triangulated so as to obtain a fully repaired model.

Table 1 summarizes the performance of our implementation on all the examples described above.

8 Conclusion

A number of applications in computer graphics, e.g, virtual prototyping, global illuminations, surface simplification, etc., rely on complete adjacency graph of the model. By moving vertices by a small distance, we are able to construct such an adjacency graph. In practice, we found that minor variations of score weights and ε did not affect the results of our repair, since most boundary edges match one edge much more strongly than other potential edges. By automating majority of the repair, while allowing the user to visualize the errors and override any corrections, we believe we have achieved a good balance between quickly correcting the model and maintaining the original intent of the design.

However, our system is but a first step towards a robust and an ultimate repair tool that can be used to correct or detect any error in solid models. While our technique generates globally consistent models, it is targeted primarily at removing bulk errors (extraneous geometry) and small positional errors (erroneous geometry). For example, large intersecting polygons may not be detected. A small box lying on top of a large box always results in two separate solids. A more general approach that includes full-scale CSG operations could generate a single manifold surface in such cases. Efficiency was mostly ignored in our prototype implementation; a number of steps were implemented using brute force algorithms. While efficiency is not crucial for off-line model correction, if RSVP were to be generalized and included in a modeling package that detects errors at modeling time, better algorithms and data structures must be used. For example, a lazy evaluation of scores may significantly speed-up processing time. Further, ability to make incremental changes to the model after user-intervention, to display part-names and maintain file pointers and to add call-back data-paths to modeling software could make RSVP a viable plug-in. Generalization of our algorithm to parametric surface models is another important ingredient for a complete modeling tool.

9 Acknowledgements

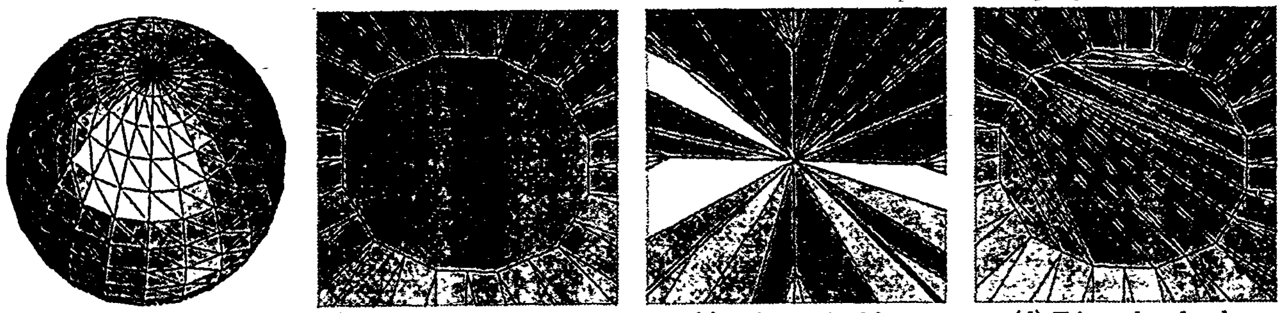
We thank Ken Fast, Greg Angelini and Jim Boudroux and Electric Boat corporation for making the model of torpedo storage and handling system available for our testing.

References

[3DS89] Stereolithography interface specification. SLA CAD, 3D Systems Inc. (Valencia, CA), 1989. p/n 50065-S01-00.

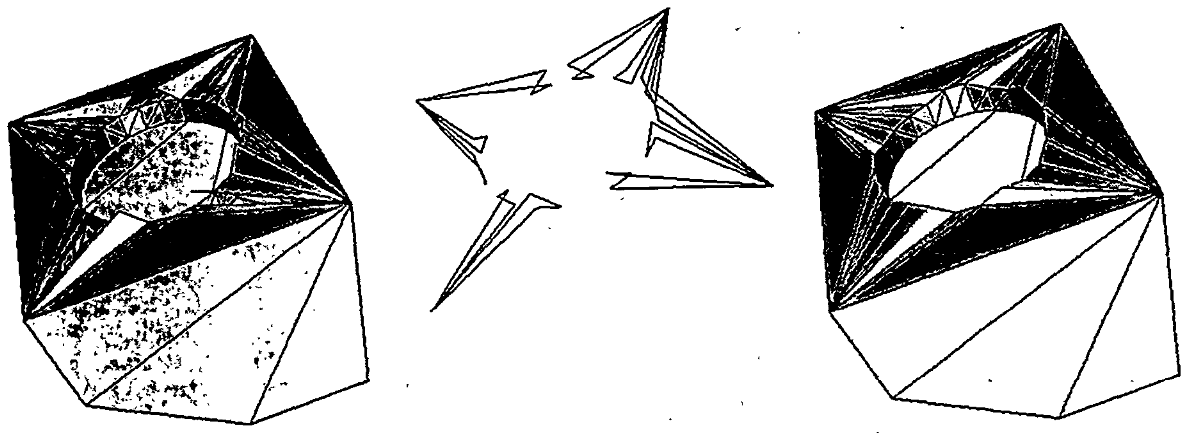
[Aut95] *Data Interchange Format.* http://www.autodesk.com, 1995. AutoCAD Release 13.

[BDE96] G. Barequet, M. Dickerson, and D. Eppstein. On triangulating three-dimensional polygons. In *Proc. 12th Ann. ACM Symp. on Computational Geometry*, pages 38–47, Philadelphia, PA, May 1996.



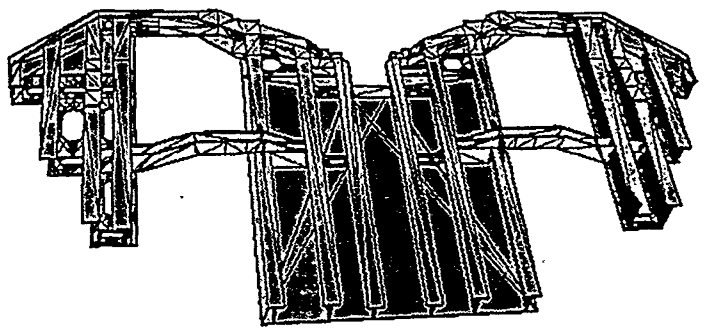
(a) The entire model (b) A pole (c) Pole repaired by (d) Triangulated pole
merging vertices

Figure 6: A ball with holes in the poles

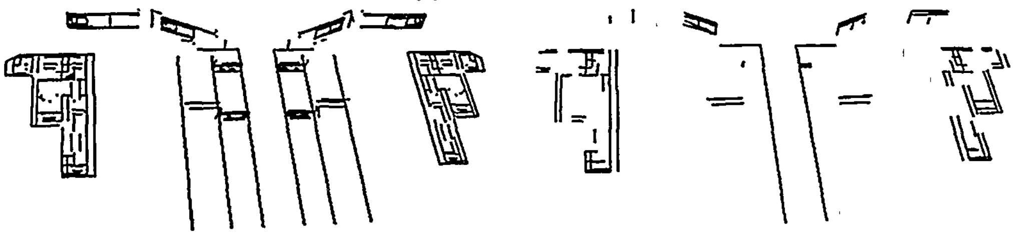


(a) Original model (b) Boundary contours (c) Repaired model

Figure 7: A mechanical T-box



(a) Original model



(b) Boundary contours (c) Remaining holes

Figure 8: A portion of a submarine storage and handling room

Model	Initial		Boundary		Matches		Holes Remaining	Final Vertex
	Vert.	Facets	Poly.	Edges	Cand.	Processed	after merges	Count
Cube	55	37	8	55	1,485	27	0	29
Ball	432	816	2	48	1,128	24	0	402
T-Box	80	132	12	48	1,128	24	0	66
Sea Boat	5,863	10,625	705	2,503	4,724	1,002	110	4,888

(a)

	Time (Seconds)							
	Loading Data	Pre-processing	Preparing Cand. List	Processing Cand. List	Post-processing	Filling Holes	Saving Data	Total
Cube	0.01	0.00	0.02	0.00	0.00	0.00	0.00	0.03
Ball	0.07	0.00	0.02	0.00	0.05	0.02	0.01	0.17
	0.07	(step omitted)				0.26	0.01	0.34
T-Box	0.01	0.01	0.01	0.00	0.01	0.00	0.00	0.04
Sea Boat	1.13	0.10	5.37	0.26	0.74	0.45	0.22	8.27

(b)

Table 1: Performance of the algorithm: (a) shows the geometric properties at various stages of our algorithms. (b) shows the time spent in different stages

[BMSW91] D. Baum, S. Mann, K. Smith, and J. Winget. Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. *ACM Computer Graphics*, 25(4):51–60, 1991.

[BS95] G. Barequet and M. Sharir. Filling gaps in the boundary of a polyhedron. *Computer Aided Geometric Design*, 12(2):207–229, 1995.

[BST96] G. Barequet, D. Shapiro, and A. Tal. History-driven reconstruction of polyhedral surfaces from parallel slices. In *Proceedings of IEEE Visualization'96*, pages 149–156, 1996.

[BW92] J.H. Bøhn and M.J. Wozny. Automatic cad-model repair: Shell-closure. In H.L. Marcus et al., editor, *Proc. Solid Freeform Fabrication Symposium*, pages 86–94, U. Texas, Austin, Texas USA, August 1992.

[BW93] J.H. Bøhn and M.J. Wozny. A topology-based approach for shell-closure. In P.R. Wilson et al., editor, *Geometric Modeling for Product Realization*, pages 297–319. North-Holland, 1993.

[KM95] S. Kumar and D. Manocha. Efficient rendering of trimmed NURBS surfaces. *Computer-Aided Design*, 27(7):509–521, July 1995.

[MD93] I. Mäkelä and A. Dolenc. Some efficient procedures for correcting triangulated models. In H.L. Marcus et al., editor, *Proc. Solid Freeform Fabrication Symposium*, pages 126–134, U. Texas, Austin, Texas USA, August 1993.

[MF96a] S.M. Morvan and G.M. Fadel. IVECS: An interactive virtual environment for the correction of .STL files. In *Conference on Virtual Design*, U. California, Irvine, August 1996.

[MF96b] S.M. Morvan and G.M. Fadel. IVECS, interactive correction of .STL files in a virtual environment. In *Proc. Solid Freeform Fabrication Symposium*, U. Texas, Austin, Texas USA, August 1996.

[MF97] T. Murali and T. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In *Symposium on Interactive 3D Graphics*, pages 155–162, Providence, RI, 1997.

[NCG91] Initial graphics exchange specification (iges) version 5.1. National Computer Graphics Association, 1991.

[O'N66] B. O'Neill. *Elementary Differential Geometry*. Academic Press, 1966.

[RW92] S. J. Rock and M. J. Wozny. Generating topological information from a 'bucket of facets'. In H.L. Marcus et al., editor, *Proc. Solid Freeform Fabrication Symposium*, pages 251–259, U. Texas, Austin, Texas USA, August 1992.

[TL94] G. Turk and M. Levoy. Zippered polygon meshes from range images. In *Proceedings of ACM SIGGRAPH*, pages 311–318, 1994.