RSVP: A Geometric Toolkit for Controlled Repair of Solid Models*

Gill Barequet Christian A. Duncan Subodh Kumar

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

Abstract

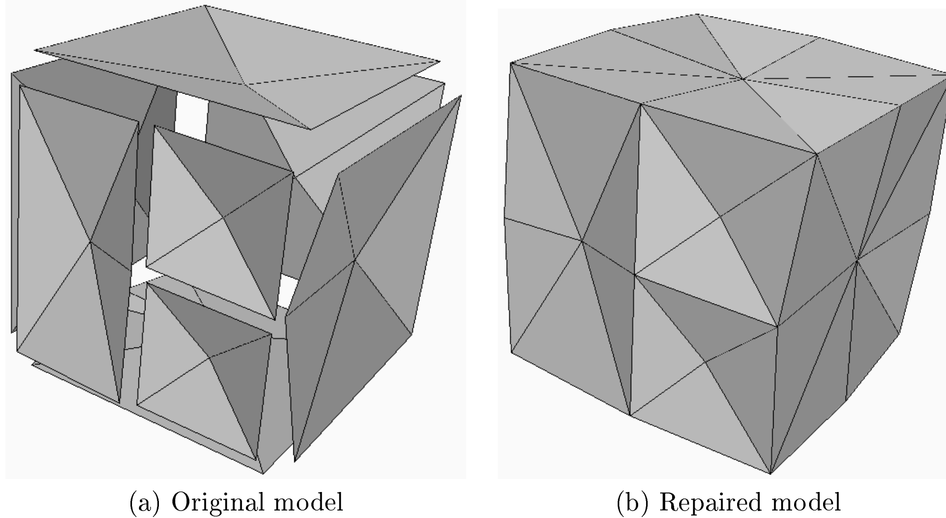
This paper presents a system and the associated algorithms for repairing the boundary representation of CAD models. Two types of errors are considered: *topological* errors, i.e., aggregate errors like zero-volume parts, duplicate or missing parts, inconsistent surface orientation, etc., and *geometric* errors, i.e., numerical imprecision errors like cracks or overlaps of geometry. The output of our system describes a set of clean and consistent 2-manifolds (possibly with boundaries) with derived adjacencies. Such solid representation enables the application of a variety of rendering and analysis algorithms, e.g., finite-element analysis, radiosity computation, model simplification, and solid free-form fabrication. The algorithms described here were originally designed to correct errors in polygonal B-Reps. We also present an extension for spline surfaces.

Central to our system is a procedure for inferring local adjacencies of edges. The geometric representation of topologically-adjacent edges are merged to evolve a set of 2-manifolds. Aggregate errors are discovered during the merging step. Unfortunately, there are many ambiguous situations, where errors admit more than one valid solution. Our system proposes an object-repairing process based on a set of user-tunable heuristics. The system also allows the user to override the algorithm's decisions in a repair-visualization step. In essence, this visualization step presents an organized and intuitive way for the user to explore the space of valid solutions and to select the correct one.

1 Introduction

CAD (Computer-Aided Design) models are often represented as unordered lists of polygons or surfaces — sometimes referred to as "soups" of polygons or surfaces. File formats like IGES [3], DXF [2] and STL [1] (the *de facto* standard in the rapid-prototyping industry) allow users to represent models as such soups. For example, each polygon may be listed independently as an ordered list of its vertex-coordinates, occasionally along with its normal vector. The collection of polygons is assumed to bound a complete solid model, i.e., a closed 2-manifold tamely embedded in $\mathbb{R}^3$ [18]. Unfortunately this is often not the case. Typical problems include tears or cracks in the surface, degenerate primitives, duplication (of surface patches or triangles), holes and overlaps, etc. (see Fig. 1). These errors often result from finite precision arithmetic, model transformations, designer's oversight, and programming bugs. Cracks may occur due

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



(a) Original model (b) Repaired model

Figure 1: A synthetic broken cube

to inaccuracies in the data or in the process that creates the data. For example, a vertex shared by several polygons may be computed according to two surface equations, resulting in slightly different positions in space, thereby 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.

The presence of these errors can complicate, and even preclude, application of algorithms that assume manifold properties. For example, cracks or degenerate geometry can result in leaking of light and create artifacts in a radiosity solution. Baum et al. [9] catalog some common types of errors and propose rules to avoid them. Other algorithms like visibility computation, constructive solid geometry, or collision detection, which require inside-outside classification of points with respect to a model, also fail due to errors in the model. For example, it is more difficult to determine whether two open 2-manifolds in 3-space intersect. Many surface simplification schemes [13, 20, 31] require consistently-oriented polygons with unambiguous adjacency information. The fabrication process in most rapid-prototyping techniques often fails due to erroneous solid descriptions. Accidental deletion of primitives or lack of complete information in 3d-scans also result in unintended holes in the model representation. Another possible error is the existence of dangling geometry 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, cracks, and overlaps in a solid model, and generate consistent orientations and adjacencies. We assume that the input model is an unordered collection of polygons (or splines) and that the target model is a set of closed 2-manifolds [27], but allow users to override this assumption.

1.1 Definitions

The problem for the polygonal (piecewise-linear) case may be stated as follows:

Input:

- $\{P_i\}$: A set of polygons, where each polygon is specified as an ordered sequence of vertices $\{\mathbf{v}_{i_j}\}$. Each vertex is specified by three real numbers $(x_{i_j}, y_{i_j}, z_{i_j})$ that represent its Euclidean coordinates.

Output:

- $\{\mathbf{v}_i\}$: A list of unique vertices, each specified by its three coordinates.
- $\{e_i\}$: A list of unique directed edges, each of the form (i_1, i_2). An edge (i_1, i_2) is geometrically realized by $(\mathbf{v}_{i_i}, \mathbf{v}_{i_2})$.

- $\{P_j\}$: A list of polygons, each one is specified by an ordered sequence of indices $\{j_k\}$. The geometric realization is obtained by replacing j_k by e_{j_k}.
- $\{S_i\}$: A set of oriented 2-manifolds (surfaces or solids). Each 2-manifold S_i consists of a set of polygons $\{P_j\}$.

Edges $e_l = (v_{l_1}, v_{l_2})$ and $e_m = (v_{m_1}, v_{m_2})$ in a 2-manifold are said to be topologically *adjacent* if either $l_1 = m_1$ and $l_2 = m_2$ or if $l_1 = m_2$ and $l_2 = m_1$. The edges are correctly adjacent if $l_1 = m_2$ and $l_2 = m_1$. This implies locally-consistent orientation. In a valid representation of a manifold every edge is correctly adjacent to at most one other edge. Every polygon should be oriented counterclockwise when it is viewed from outside the solid.

In case of splines, each surface patch is input as a mesh of control points. The output contains a common representation for boundary control points between two adjacent surfaces in a list of curves, instead of edges. Surfaces contain an index into this list.

Our goal is to generate a geometrically embedded set of oriented 2-manifolds. A geometric embedding maps the vertices of the complex to Euclidean vertices. Informally, we seek to construct geometric surfaces, each point of which is locally similar to a point on a plane. For further details on algebraic topology, the reader is referred to [18, 27]. We call two edges *coincident* if they geometrically overlap. Not all geometrically coincident edges may be topologically adjacent, however. Two edges e_l and e_m are said to be ε-*coincident* if either $v_{l_i}, i \in \{1, 2\}$, is no farther than ε from e_m or v_{m_i} is no farther than ε from e_l. Two ε-coincident edges e_l and e_m are *merged* by replacing both v_{l_1} (or v_{l_2}) and v_{m_1} by v'_1, and both v_{l_2} (or v_{l_1}) and v_{m_2} by v'_2.

A surface is said to be *globally consistent* if the normal of each polygon is consistently oriented outside the solid, i.e., each polygon is oriented counterclockwise.

1.2 Algorithm Overview

We first present a topologically-based geometric algorithm designed to rid polygonal boundary representations (B-REP) of solid models of common errors. We later extend the algorithm to spline models in Section 6.

A fundamental premise of our system is that the intent of the designer is an important issue and must not be ignored by any comprehensive model-repairing system. Unfortunately, the original design intent is not always obvious. We have chosen to desist from implementing an expert system that guesses the intent and rather use only geometric algorithms driven by heuristics based on common properties of closed solid models. However, for such a system to be applicable to real models, we must produce *the* desired model, not just *a* "correct" model. We achieve this goal by involving the user in the repair loop. The system consists of two major components:

1. **Geometric component**: Using local neighborhood and global-consistency properties, this component derives topological adjacencies and unifies adjacent edges geometrically. A new position of the merged edge is determined so that the surface is locally a 2-manifold. This step requires no user interaction.
2. **Visualization component**: The system lets the user visualize and guide the fault repair. The challenge is to convey all geometric and topological structure in a simple and intuitive fashion without inundating the user with visual detail. A simple point-and-click interface allows the user to explore all correct solutions by altering any local decision made by the system and viewing the effect.

The system, called RSVP,[1] takes as input a soup of polygons and outputs the adjacency structure of the corrected model. RSVP may be used to eliminate common errors; it is not designed for full-fledged surface editing or modeling, but only for local and small changes in a geometric model. For example, it makes no attempt to perform any CSG operations to resolve intersecting solids. Also, if exact Euclidean vertex positions are important, our techniques are

[1] This name stood originally for Repairing by Shifting Vertices of Polyhedra; we then extended the system to handle spline models too.

not applicable. Indeed, one cannot even expect to correct errors in this context. The repair algorithm generates manifolds with normal vectors pointing outwards from the model. It closes small cracks and fills larger gaps with polygons. Small overlaps are detected and separated. Extraneous geometry, zero-volume parts and T-joints are also handled.

Here is a brief description of the algorithm:

- We first preprocess the model and construct a k-D tree [10] in which we store the vertices. Due to coherence, updates of the tree are inexpensive on the average. This k-D tree is used subsequently for efficient location of points, edges, and polygons.
- We then compute the connected components of the object, which are oriented 2-manifolds, often with boundaries. 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 matches each boundary edge with another ε-coincident boundary edge, where ε is a user-specified parameter bounding the maximum error to be repaired in the input model. These edges may be made adjacent by merging vertices if this operation does not violate manifold properties. We use an adjacency score for ranking all ε-coincident pairs of edges.
- We merge two boundary edges that have the lowest adjacency score. We continue such mergers until all subsequent mergers either result in intersecting components or require moving a vertex by more than ε from its original position. The repair process "converges" to the desired model faster if ε is a good estimate of the maximum numerical error in the input model (since in this case there are less user interventions).
- If the dihedral angle between two polygons in a component 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 that computes the boundaries of the modified surface. The holes are triangulated, and the dangling or zero-volume parts are 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 errors. We describe a highlighting technique in conjunction with zoom windows for helping the user inspect the faults on the surface and supervise the repair process.
- Once the user overrides a decision made by the system, the geometric sub-system takes over and recomputes new adjacencies, thus obtaining a new corrected model. This process is continued to user's satisfaction. In practice, only a few automatic decisions need to be overridden in most cases.

1.3 Related Work

A preliminary version of this work was recently published [6]. This paper reports several enhancements over [6], that include additional features in the geometric and visualization subsystems, maintenance of topological constraints at each stage, an improved vertex positioning algorithm, efficient geometric pruning to obtain significant speed-ups, and an extension to spline surfaces.

Several other techniques for correcting some of the errors described here have been proposed in the past. These techniques use variants of tolerancing schemes. In general, vertices closer than a user-specified parameter ε are merged. Such indiscriminate merging often fails since an appropriate value of ε is difficult to guess.

Morvan and Fadel [24, 25] describe a virtual environment that provides tools for model correction, controlled primarily by the user. Unfortunately, this can be a cumbersome and inefficient procedure for large models. Furthermore, it is easy to miss errors, and even to introduce new errors, if one just uses the two dimensional screen projection to infer three dimensional

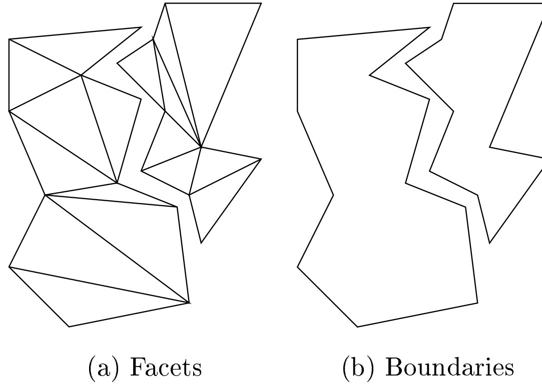
relationships. Our approach makes the correction process automatic, but includes the user in the correction loop at the same time. The user can visualize errors which are clearly highlighted in the rendering and can guide the correction algorithm. Geometric constraints are always maintained to avoid accidental introduction of errors. Turk and Levoy [30] remove overlaps of polygons by clipping them against each other in order to generate polygonal models from range data. Unlike their work, which handles a collection of scattered points, our system always has the recent version of a polyhedral model available, so that vertex-shifting causes smaller perturbation to the input. In addition, vertex-shifting closes gaps as well. Rock and Wozny [28] sort vertices using an AVL tree for efficient location of vertices in an ε neighborhood—these vertices are then merged. Bøhn and Wozny [11, 12] 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 [23] also use local techniques for filling cracks in the model surface. Barequet and Sharir [7] 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 [26] is to determine regions of space that lie inside a solid using spatial partitioning, and use the partition as the description of solids. This is a simple and promising technique which generates topologically-correct solids. Unfortunately, there is no control on the topology of the result, which can be significantly different from the input. This technique works well in the absence of degeneracies or narrow angles between adjacent polygons. The approach of Guéziec et al. [17] is similar to ours. They merge close vertices, but their work is primarily focussed on topological construction and does not consider geometric intersections. In addition, the candidate edge pairs for their stitching operation is restricted in order to avoid inconsistent topology without having to consider geometric properties. The "zipping" operation of Sheng and Meier [29] is also similar in flavor, but both [17] and [29] can violate our consistency constraints. In our work we propose a heuristic based on the observation that most of the cracks and overlaps occur due to numerical errors in the computation of vertex coordinates. RSVP corrects such errors by slightly shifting the vertex positions. Our method ensures that no vertex is moved farther than a user-specified error-tolerance. Larger holes are filled using the triangulation technique of [7].

The rest of this paper is organized as follows. Section 2 briefly describes our point and edge location data structure. Section 3 discusses the algorithm to generate connected components of the given model. Section 4 describes our candidate ranking and processing algorithms. Section 5 presents the final topological classification. We discuss the extension to spline surfaces in Section 6. Section 7 presents our repair visualization technique. In Section 8 we describes our implementation, analyze its performance, and give some experimental results. We conclude in Section 9 with future research directions.

2 k-D Trees

Recall that only edges that lie within the distance ε of each other are merged. Thus, for any given edge, the list of valid mergeable edges is expected to be small relative to the problem size. We therefore use a data structure which can efficiently answer range queries. Note that intersecting geometry are also proximate in Euclidean space. Thus range search can also be used to efficiently detect intersections. It is well-known that the k-D tree [10] is an efficient range searching data structure when the elements are expected to be spread uniformly in space, and its inherent simplicity allows us to process such range queries with minimal overhead.

The data structure is quite simple. Each node in the tree represents both a point from the data set and a splitting plane along one of the point's dimensions (for example, splitting according to the point's x-coordinate). Each node has two children, one pointing to all points on the left of the plane and another pointing to all elements on the right of the plane. Using this definition, the tree is easily recursively built and can be implemented to have a depth of no more than $O(\log n)$ size (where n is the total number of points).



(a) Facets (b) Boundaries

Figure 2: Boundaries of connected components

The query also proceeds in a similar fashion. Starting at the node, the current represented point is tested for inclusion in the query range (in our case, this is a ball). Next we compare the query range with the splitting plane. If any portion of the range falls on the left side of the plane, we recurse on the left child, and if any portion of the range falls on the right side, we recurse on the right child.

This simple, yet powerful, data structure, used on realistic CAD models and for small ranges answers each query efficiently; performances of around $O(\log n)$ per query can be expected. This is a vast improvement over the brute-force method (used in [6]).

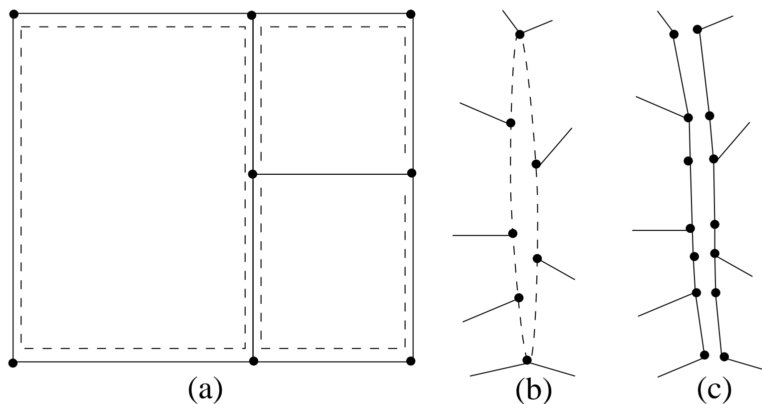
3 Component Construction

We start by considering each input polygon as an open manifold by itself. We evolve the complete model by merging edges of neighboring polygons. 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 unordered 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 there is 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 as a single component and passed on to the merging phase.

We compute the connected components of the model using a procedure similar to that of [7]. 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_i-F_j, if faces F_i and F_j contain edges e_l and e_m, respectively, such that e_l is adjacent to e_m. The graph-edge is marked with $\{l, m\}$. An edge $\{l, m\}$ is not included in the graph if a third edge e' is adjacent to e_l. The graph G is then constructed as follows:

FOR each unprocessed (input) edge e_i **DO**
    $\{e_j\}$ = `LocateEdge`(e_i); // Find edges adjacent to e_i
    IF $|\{e_j\}| = 1$ **THEN**
        Add the edge `Face`(e_i)-`Face`(e_j) to G.
    Mark e_i and $\{e_j\}$ as processed.
OD

We use a depth-first search of G to construct the connected components of the model. During the traversal 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 nonorientable surfaces while processing "back-edges" in the depth-first traversal. Back-edges are used to perform a consistency check between two facets whose respective orientations are already fixed. If the two orientations do not match, the component is nonorientable and the system reports the error.



(a) The boundary edges are shown dashed. (b) A zero-area polygon (dashed) may be created due to t-joints. The gap is magnified for clarity. (c) The zero-area polygon can be eliminated by introducing additional vertices on the adjacent polygons. Each vertex on the zero-polygon must be included on both sides. The components on each side are now topologically adjacent.

Figure 3: T-Joints

Once we partition a model into connected components, we have to find the boundary edges of each component, which may later be 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 (see [7]) of all the facets (as cycles of graph-edges). The result is the collection of the boundary polygonal contours (see Fig. 2).

Note that in case of a T-joint (Fig. 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 general, input edges must be subdivided by introducing additional vertices to eliminate T-joints (see Fig. 3(c)). This operation is simple when T-joints occur on coincident lines as shown in Fig. 3(a). However, cracks at T-joints require special handling. We introduce a new vertex v_i' on edge e_i if another vertex $v_i \notin e_i$, lies less than ε away from e_i (but not within ε-neighborhood of any endpoint of e_i). Unfortunately, naively introducing such vertices can result in undue increase in the number of vertices. We instead introduce vertices only if the corresponding match is promising, namely, only when splitting the edge lowers its minimum adjacency score (see section 4).

The component generated at this stage are considered “correct” parts of the model. Their topology remains unchanged for the rest of the repair procedure. Each component is oriented consistently. However, this orientation may be opposite to the final surface orientation in the model and may be reversed later. Also, boundary vertices may have their geometric location changed later. It is easy to see that these components may fail to be 2-manifolds and may self-intersect. However, one of the curves of intersection must lie in the interior of a polygon. Hence if the intersections between polygons present in the input are only along input edges, our algorithm is able to detect all intersections and generates a set of oriented 2-manifolds. Components without boundaries are marked as complete and are removed from further processing. Their orientation is made globally consistent (see section 4).

4 Adjacency Processing

Once connected components of a model are constructed and their boundary contours are computed, we generate a list of candidate boundary-edge pairs. Some of these candidates are later merged to generate new topological adjacencies. Note that even pairs of edges belonging to the same component are candidates for merging. In theory, there may exist a quadratic number of

topologically valid mergers. We use heuristics to eliminate most unpromising mergers.

4.1 Adjacency Score

We assign scores to candidates as follows: Let $e_1 = \overrightarrow{v_1v_2}$ and $e_2 = \overrightarrow{w_1w_2}$ be two boundary edges (see Fig. 4). We first assume that the boundary contours are oriented consistently with respect to each other, i.e., all the connected components are globally consistently oriented. We later (Section 4.3) describe how to remove this assumption to deal with more complex cases where we cannot rely on the original orientations of the facets.

The goal is to estimate the "desirability" of merging two directed edges e_1 and e_2: a lower score corresponds to a better matching candidate. For this purpose we estimate the magnitude of the geometric error: the area of the missing part of the two corresponding polygons in case of cracks, or the area of the extra parts in case of overlaps. The case of cracks is demonstrated in Fig. 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. Note that e_1 and e_2 need not be coplanar and hence the area between them is not well defined. We define the area using a linear parametrization $(0 \leq t \leq 1)$ of the edges as follows:

$$A(e_1, e_2) = \int_0^1 |\overrightarrow{e_1(t)e_2(t)}|^2 \mathrm{d}t = \frac{1}{3}\left(|\overrightarrow{v_2w_1}|^2 + |\overrightarrow{v_1w_2}|^2 + \overrightarrow{v_2w_1} \cdot \overrightarrow{v_1w_2}\right). \tag{1}$$

We define the normalized area, $\overline{A}(e_1, e_2) = \frac{A(e_1,e_2)}{|\overrightarrow{e_1}||\overrightarrow{e_2}|}$. Say, $l(t)$ is a linear parametrization of the line of intersection between the planes containing the two error polygons. To compute Δ, the estimate of error, we are actually interested in

$$\int_0^1 (|\overrightarrow{e_1(t)l(t)}|^2 + |\overrightarrow{l(t)e_2(t)}|^2)\mathrm{d}t$$

instead of $A(e_1, e_2)$. 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 dihedral angle between the planes supporting the error polygons. If we assume (as an approximation) that the error is symmetric, i.e., $|\overrightarrow{e_1(t)l(t)}| = |\overrightarrow{l(t)e_2(t)}|$, then

$$\Delta(e_1, e_2) \stackrel{\text{def}}{=} \int_0^1 \frac{|\overrightarrow{e_1(t)l(t)}|^2 + |\overrightarrow{l(t)e_2(t)}|^2}{|\overrightarrow{e_1}||\overrightarrow{e_2}|} = \frac{\overline{A}(e_1, e_2)}{(1-\cos(\theta))} = \frac{\overline{A}(e_1, e_2)}{(1 + N_1 \cdot N_2)},$$

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

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

$$\text{Score}(e_1, e_2) = U(e_1, e_2) + \Delta(e_1, e_2) + 0.5\Delta(e_1^-, e_2^-) + 0.5\Delta(e_1^+, e_2^+).$$

(See Fig. 4(a).) U is a user-specified score for certain candidate pairs. In order to prohibit merging of two edges e_1 and e_2, the user may set $U(e_1, e_2)$ to $+\infty$ ($-\infty$ to force a merger). Setting U is performed by the visualization sub-system described in section 7. By default, $U = 0$ for every candidate. While it is possible to use a continuous function with a maximum at $t = 0.5$ to weigh the Δ's, we found out that an equal-density distribution of weight between the matching pair and the adjacent pairs works well in practice. The key lesson is to use the adjacent edges—this results in each edge matching one pairing edge much more strongly than other edges. As a result minor variations of the weight do not usually affect the results.

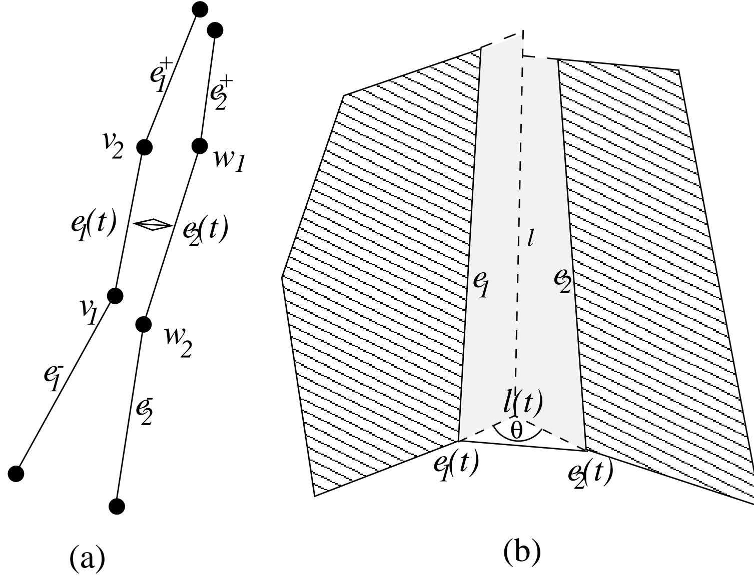


Figure 4: (a) A candidate matching pair of edges. Fig. (b) shows l, the line of intersection of the two planes containing e_1 and e_2.

For each candidate, we maintain its score, the respective edges, and their endpoints. In the implementation presented in [6] we sorted all the matching candidates in ascending order of score, and processed the candidates in this order. In the current implementation we insert all the candidates into a heap, and later remove one candidate at a time by using a standard `DeleteMin` operation on the heap. Asymptotically the two methods require $O(q \log q)$ time for all the queue operations, where q is the number of candidates. However, the heap allows us to evaluate and update the scores lazily, as only the minimum score needs to be uptodate at any instant.

Note that any of the vertices considered for the computation of the score could be introduced in an edge split (see the discussion of T-joints in Section 3). In particular, if the distance between vertices v_1 and w_2 is greater than 2ε, then one of the edges, say e_1, may be split into edges e_1^1 and e_1^2, provided that the score(e_1^1, e_2) or score(e_1^1, e_2) is lower than the best matching score of e_1.

4.2 Merging

For each boundary edge we maintain one bit indicating whether the edge has already participated in a merger. A pair of matching candidates is merged only if both its edges are still unmerged and such a merger does not introduce any self-intersections in the merged component. In order to match a pair of boundary edges $e_1 = \overrightarrow{v_1 v_2}$ and $e_2 = \overrightarrow{w_1 w_2}$, we merge v_1 with w_2 and v_2 with w_1.

Since it is possible that 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 position of the bucket.

Here is a formal description of the vertex-merging process:

Initialization:

FOR each boundary edge e
  Set e.handled := False;
FOR each boundary vertex v **DO**
  Create a bucket $b_v = \{v\}$;
  Set b_v.average := v;
OD

Iteration:

FOR each matching candidate (e_1, e_2) (where $e_1 = \overrightarrow{v_1 v_2}$ and $e_2 = \overrightarrow{w_1 w_2}$):
  IF e_1.handled = False **AND** maxShift$(e_1, e_2) \leq \varepsilon$ **AND** e_2.handled = False **THEN**
  DO
    Merge (v_1, w_2);
    Merge (v_2, w_1);
    Set e_1.handled := True;
    Set e_2.handled := True;
  OD

FUNCTION *Merge* (v, w):
IF $v = w$ **THEN**
  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 DO
    Set $b_v = b_v \cup b_w$;
    IF *Place* (b_v) = Success **THEN**
      Delete b_w;
    ELSE
      Discard the merger of b_v and b_w;
  OD
OD

maxShift(e_1, e_2) is a measure of the distance between the edges e_1 and e_2. The function *Place*(b) attempts to position a bucket b, i.e., relocate all the points in the bucket, such that all its points lie within distance ε of their original positions and no self-intersections are introduced in the manifold. This is a computationally expensive procedure to resolve exactly, hence we use a conservative heuristic. If *Place* returns a position, it is always guaranteed to be valid, but the function may also fail even if there exists a valid location for the bucket. The function *Place* proceeds as follows:

FUNCTION *Place* (b):
Relocate the vertices of b at b.average;
FOR all vertices $v \in b$ **DO**
  IF some polygon P, incident to v, intersects polygon P' **THEN**
    Move b towards v (but not beyond v) until P does not intersects P';
  IF no intersections remain **THEN**
    Report Success;
OD
Report Failure;

4.3 Orientation Checking

If the original orientations of facets are not consistent, we need the following modifications (using some methods of [7]) to our algorithm:

- Orient all the facets of each connected component consistently with respect to other facets of the component.
- For each pair of boundary edges, compute *two* matching candidates. The additional candidate consists of the same edges with the additional 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, the candidate can be processed only if it passes the consistency test. Otherwise, while processing this candidate we unite the two respective connected components, and the type of the candidate and the current orientations of the components determine whether we need to reorient one of the components in order to maintain global consistency. A components is reoriented inverting the orientations of all its facets.

5 Component Classification

After all components have been grown as much as possible, some of them may be closed solids without boundaries, while others may still be open. Only outward normal orientation remains to be verified for the closed solids. However, open solids do not have a well-defined inside and outside. These components may be open due to a number of reasons:

1. Open components were part of the original design.
2. These are extraneous parts, either due to duplicate geometry or erroneously retained geometry.
3. The parameter ε was not an accurate upper bound on the error.
4. A hole was accidentally left during the design in the model.

In case 1, we leave the open components as they are. In case 2, we discard the extraneous geometry. In cases 3 and 4, we fill the hole using the algorithm outlined in the section 5.1. Alternatively, the user may increase ε and apply the repair process on the new version of the model. Unfortunately, it is not always clear whether an open component should remain unchanged, discarded, or closed. We use the following heuristic for this classification:

- Fill all holes.
- If the total area of the triangles filling the hole exceeds the surface area of the open component, mark the component as open and discard the hole triangulation;
- Otherwise, close the hole by retaining the triangulation.

We normally do not discard any components, unless all its polygons are duplicates of other polygons. In practice, we have found a few cases of hanging geometry to justify a heuristic for this case. The visualization stage allows the user to click on any open, filled or closed component and delete it from the model.

5.1 Filling Holes

When the user sets a small value for ε, the maximum allowed vertex-shift, there may be holes left in the model 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 are located (see Section 3). These are indeed the boundary polygons of the new object after the vertex-merging step. Our system automatically triangulates these holes (unless this feature is turned off 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 [5] that the decision whether a three-dimensional polygon is triangulable, i.e, 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 [7], we use a simple dynamic-programming technique which computes the triangulation of a hole with n edges in $O(n^3)$ time. Following [8], 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 $L(t)$ (resp., $S(t)$) be the lengths of the longest (resp., shortest) edge of t. The weight of a triangle t is of the form $\omega_{\mathrm{area}}A(t) + \omega_{\mathrm{ratio}}L(t)/S(t)$, where ω_{area} and ω_{ratio} are user-defined weights. The rationale for this measure is the aim to minimize the surface area of the triangulation while avoiding, 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 is not clear if these are legitimate holes or just artifacts of dangling geometry. If the total area of the filling triangles is greater than that of the component itself, we mark the component as dangling. The choice of the shift-bound, ε, is important in some cases. For example, two well-separated hemispheres may each end up closed separately by triangulating their boundaries. But 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 to triangulate ("stitch") 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.

5.2 Global Consistency

We reorient all closed components such that each normal of a component locally points outside the component. Since open components do not have well defined inside or outside, they are ignored during this step. Orienting a solid correctly requires one ray-shooting operation if no nesting of solids is allowed or checked. Our system handles multi-shell solids as well. If a component is enclosed within another component, it is considered to be a hollow shell (3D void) cut out from the outer solid. Thus "outside the solid" actually corresponds to inside the shell. We identify the solid nesting-hierarchy by shooting rays at each solid, and infer the hierarchy from the obtained parentheses-sequences of hitting points. An invalid sequence implies intersection between two solids without a proper nesting relation between them.

As mentioned above, the input may contain intersecting solids which still intersect after the repair operation. In such cases there is no clear inner or outer shell, and the algorithm leaves normals of intersecting shells pointing outwards. Since intersecting shells are not parts of the target model, we do not perform any expensive operation to detect all intersections. Some intersections may be missed and thus some shells may be oriented incorrectly. Such cases are

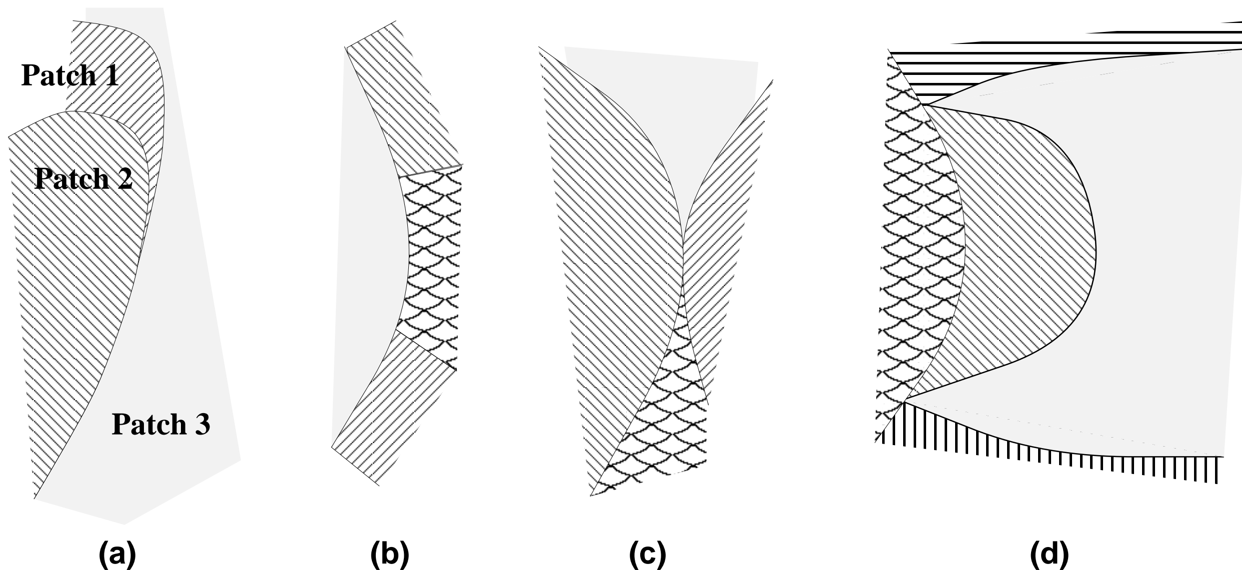


Figure 5: Tangency cases for smooth surfaces

passed on to the visualization step and it is the responsibility of the user to invert the orientation of such shells. The user may, of course, also invert the orientation of open components.

6 Extensions and Ongoing Work

Since many CAD models are designed by using NURBs (Non Uniform B-Splines), Bézier surfaces, or other spline forms, a fault-repair system would not be complete without repairing spline models. For our purpose the spline $S(u,v)$ is described by a mesh of control points $p_{ij}, i \in \{0..m\}$, $j \in \{0..n\}$, for some integers m and n. The points $p_{0,j}$, $p_{m,j}$, $p_{i,0}$, and $p_{i,n}$ are the four corners of the boundary of the surface. We denote the four sequences of boundary control points that connect the four corners as border *chains*. The simplest algorithm would tessellate each surface patch into a set of triangles and then apply a polygon-repair algorithm. However, this scheme would destroy the smooth surface representation, which is often undesirable. A more viable approach considers the boundary control points of the spline surface. In this scheme, two boundary curves are unified by merging their control chains. This algorithm proceeds as follows:

1. Insert the chain control points into a k-D tree.
2. Compute the connected components. As in the polygonal case, create an adjacency graph: two patches are connected by an edge in the graph, if they uniquely share a boundary chain.
3. Compute adjacency scores as a sum of scores of corresponding pairs of consecutive edges. ε-coincidence now defines the proximity of a boundary chain to another boundary chain.
4. Merge points of boundary chains with low adjacency score by sequentially merging corresponding points of the two chains.

An advantage of this approach is that some of the functions used for polygonal repair may be used also for repairing spline surfaces. However, there are a few problems with this simple extension:

1. Two matching curves must be of the same degree, i.e., they must have the same number of control points for evaluation of score and subsequent merger.

Figure 6: To test a preliminary version of our spline-correction code, we introduced random changes (between 0 and 1) to the control points of the Utah teapot.

2. A simple addition of edge scores artificially raises the score of a pair of higher degree curves, compared to lower-degree curve pairs.
3. Boundary chains must match in their entirety, and hence special handling of T-joints is needed.
4. Entire curve mergers imply that rather long control chains must be merged. As a result cases as shown in Fig. 5(a) and (b) cannot be handled.
5. Merging boundary chains, one point at a time may alter the shape of the boundary curve. Indeed, arbitrary mergers could alter the shape of the surfaces and violate continuity or other constraints. This is a compromise we chose to make in the current implementation.

Fortunately, most of these problems can be addressed rather easily. In order to gain finer control over mergers, it is useful to have relatively small surface patches. We use knot insertion [14] to first subdivide larger NURBs surfaces into Bézier patches. Individual Bézier boundary curves are more easily merged. This conversion also eliminates in practice most cases of the type shown in Fig. 5. However, tangency cases like that shown in Figs. 5(c,d) can still occur. These can be resolved by curve subdivision. Once all mergers have been completed, the inverse process of knot deletion [4, 22] (and curve joining [14]) may be used to restore the original form of representation, if required. If the degrees of two ε-coincident boundary curves are not equal, we degree-elevate the lower-degree curve. The degree may later be reduced back, but that requires a slight modification of the adjacency data structures as a single index into a list of curves can no more be stored with the corresponding surface pairs.

We also modify the adjacency score to reflect the merging of chains rather than edges. The score for a pair of control chains $\mathcal{P}_i = \{p_{i_k}\}$ and $\mathcal{P}_j = \{p_{j_k}\}, k = 0, \ldots, n$ is given by:

$$U(\mathcal{P}_i, \mathcal{P}_j) + \frac{1}{n+2}\left[\left(\sum_{k=0}^{n-1} \Delta(e_{i_k}, e_{j_k})\right) + \Delta(e_i^-, e_j^-) + \Delta(e_i^+, e_j^+)\right]$$

where e_i is the edge (p_i, p_{i+1}), e_i^- is the last edge of the chain preceding $\mathcal{P}_i$ (recall that there are four control chains on the boundary of the surface), and e_i^+ is the first edge of the following chain. Notice that the Δ's are normalized by the number of edges. This ensures that longer chains (having more edges) are not penalized.

After the merging phase is complete, the rest of the procedure is identical to that of the polygonal case. We currently fill holes with triangles as described in Section 5.1.

7 The Visualization Loop

While a fully automatic process for error correction of solid models is desirable for both time and cost effectiveness, it is not practical. Multiple options resulting in different "correct" models

Figure 7: Demonstration of multiple options. (a) and (d) show four components corresponding to two cubes with one wall of each, separated from the rest of the body. The shaded solids (b) and (c) result by making the respective choices shown in (e,f) in wire-frame. The candidate matches are highlighted using the white marker. Note that (f) results in a component contained within the cube, which may optionally be discarded.

are often available. Typically, only one of these is acceptable based on the designer's intent. The original intent of the designer may not be followed by the heuristic for edge matching. For example, see Fig. 7(a). Without considering the context, we cannot unequivocally choose between Figs. 7(b) and 7(c), either of which could be the intended shape. Any theoretical guarantees, e.g., obtaining the topologically and geometrically closest correct model to the input not only would be inefficient, but could also fail to produce the "intended" model. Instead of trying to develop a language to let users specify this design intent, we chose to involve users in the correction process, and let them override any decisions made by the autonomous part. The premise of this approach is that the decision of the automatic part is almost always correct, thus necessitating only a small number of user overrides. The challenge is to present the information to the user in a fashion that the cases requiring override become immediately apparent. Our goal is to provide enough information on-screen to draw attention to faults without deluging the screen with detailed rendering of geometry. At the same time we allow the user the freedom to customize the visualization.

7.1 Visualization Setup

Our system provides simultaneous visualization of two forms of the model. The erroneous form and the corrected form. While it is possible to switch between two forms in the same window, or even generate an animation from one to the other, it is often necessary to compare the result of an operation. A side-by-side rendering of two forms provides a more intuitive way to compare options. Furthermore, one-to-one correspondences are maintained between the geometry in windows. For instance, a primitive selected in the *error window* is automatically selected in the *repair window* too.

Sometimes it is necessary to observe the actual geometric detail in the neighborhood of a given error. Note that the size of an error is usually quite small (with respect to the surrounding features of the model), and that the user needs to significantly scale up the image in order to

visualize the error. Unfortunately, this results in a loss of context and orientation, and may confuse the viewer. We alleviate these problems by using separate *zoom windows.* The user can maintain context in the primary windows and pop up zoom windows which show the zoomed-in view around the selected region of the screen. The context is maintained by synchronized rendering of the same parts of the model in both the primary window and its zoom windows.

A simple, even if highly zoomed up, 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 various types of errors.

7.2 Visualization Operations

The user can display any of the following aspects of fault repair: computed components, merged edges, resolved T-joints, surface orientation, zero-volume solids, open components, and filled holes. To delineate different components we provide a unique, well separated, color to each component. This is helpful in identifying cases like that in Fig. 7. The colors black and white are reserved: the background is black and highlights are white. We also color all back faces in white to make cracks apparent. In addition to separately colored solids, we provide an option to color all front faces in a color different from the back-face color to make the cracks even more obvious. However, even with such rendering, most cracks are not immediately recognizable. We use *fault-lines* to help visualize the mergers better. Fault lines are boundaries of components and are rendered as single pixel wide lines. With fault lines (see Color Plate (b)), cracks are easily visible. Again, all front faces may be colored using a single color different from the color of the fault lines to make the distinction clear. In addition to the fault lines, the error window also optionally displays *bucket-lines*, edges connecting vertices of a bucket (see Section 4.2) to the final position of the bucket. The user can select a pair of fault lines. This pair is marked for separation. On the next iteration of the automatic correction, the score of the selected pair is artificially increased by setting the user-controlled score factor U to $+\infty$ (see Section 4).

Editing of topological errors is typically performed using a different rendering scheme. This scheme, when selected, renders most polygons in wire-frame. Currently we use a checkered stipple pattern to obtain this rendering. Only the polygons of special solids are rendered shaded. The special solids, depending on user's selection, may be these of one of the following types:

1. Dangling geometry: retain on click, otherwise discard.
2. Holes (the holes are shown triangulated in the repair window): retain hole on click, otherwise leave triangulated.
3. open components: fill hole on click, otherwise retain as is.
4. Zero-volume solids: retain on click, otherwise discard.
5. User selected component: discard on click.

Our experimentation shows that turning off the wire-frame rendering of components, either based on a timer to provide a blinking effect, or directly controlled by the user, provides a cleaner visual screen space and makes faults more apparent. The user may actively click on any shaded component and have its type marked differently, based on the context. Note that clicks are transitive, i.e., a dangling geometry is re-classified as an open component on the first click, as a hole on the second click, and back to dangling on the third click. Components re-classified as holes are completed by hole-filling. Surface orientations are inverted by clicking on components in a different phase, after the topology-editing phase.

The Color Plate demonstrates some common visualization and override operations. Plate (a) shows a model of an F15e fighter consisting of 2,000 polygons. The fault lines are shown in Plate (b). The user can select a component for close inspection, as shown in Plate (c). The selected component is rendered shaded, while other components are hatched. Overlapping geometry is visualized in Plates (d-f). The user can again select the particular dangling component (e). If the user allows the system to discard this component, the retained copy becomes visible and may be highlighted for verification. Plates (g-i) concentrate on holes. Plate (g) shows a

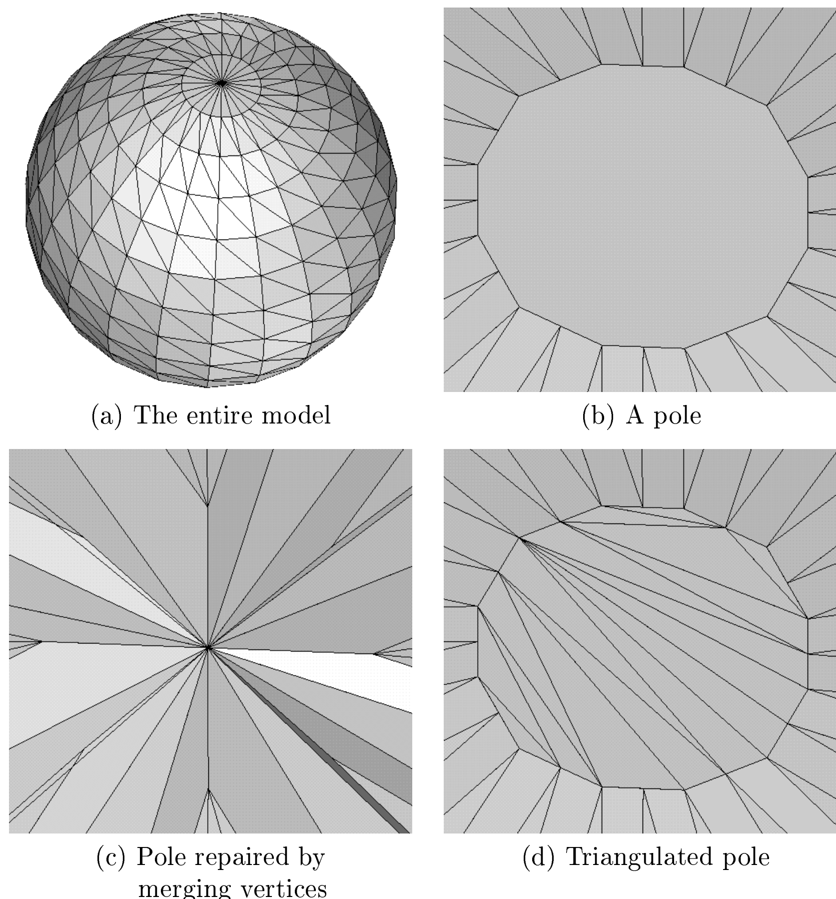


Figure 8: A ball with holes in the poles

hole which was erroneously filled in the first pass (Plate (h)). Upon user intervention, the hole is left open (Plate (i)). The F15e model required a single user iteration to fix all errors. Plates (j) and (k) show examples of other models.

8 Implementation and Experimental Results

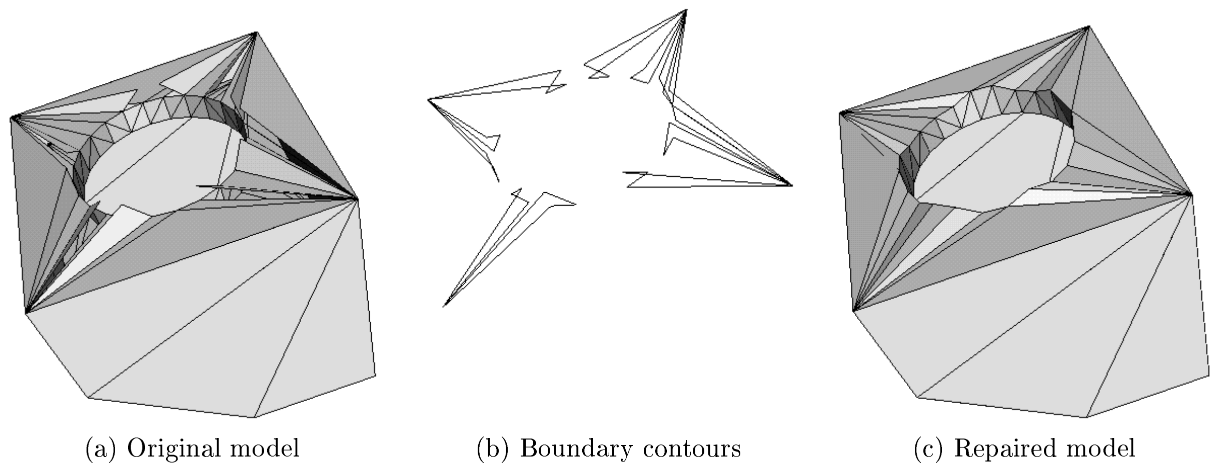
We have implemented RSVP in C on an IRIX platform. The visualization component is based on OpenGL. We have experimented with the algorithm on several data files (with thousands of polygons) obtained from CAD systems.

Fig. 1(a) shows a synthetic cube with a "broken" surface, while Fig. 1(b) shows the repaired model after vertex merging. (In this example the boundary polygons of the repaired object were also triangulated.)

Fig. 8(a) shows a ball with tiny holes in its poles. One such hole is displayed in Fig. 8(b). Vertex merging results in the construction shown in Fig. 8(c). Alternatively, if we avoid any vertex shifting (and thus, merging) by setting a small-enough maximum-allowed shift, ε, the holes are triangulated as shown in Fig. 8(d). (These two experiments are reflected by two entries in Table 1.)

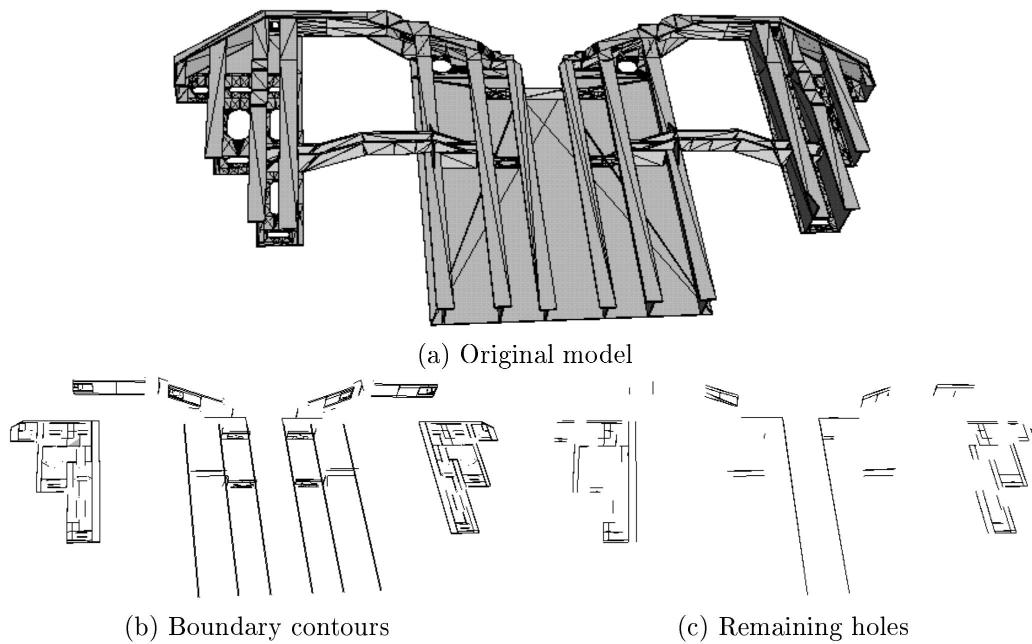
Fig. 9(a) shows a cracked mechanical T-box. The boundary contours of this model are displayed in Fig. 9(b). The repaired T-box is shown in Fig. 9(c).

Fig. 10(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 Fig. 10(b). We avoided merging of vertices whose mutual distance was above 10



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

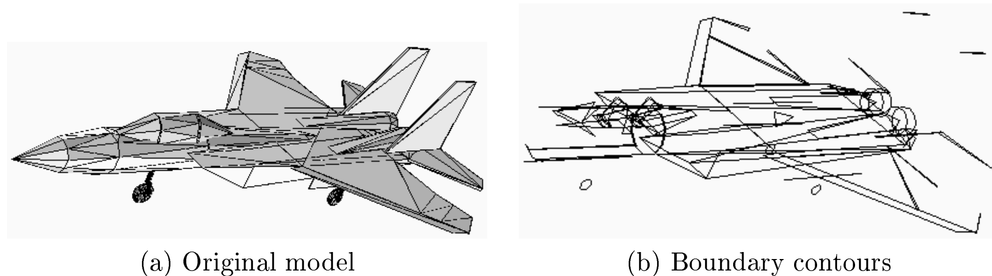
Figure 9: A mechanical T-box



(a) Original model

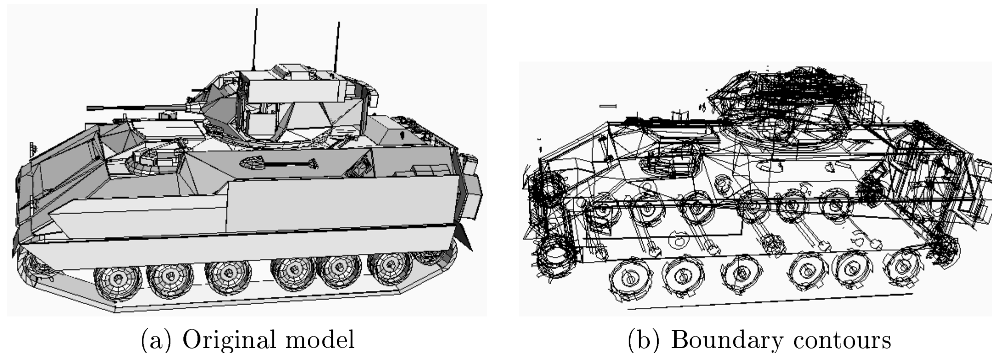
(b) Boundary contours (c) Remaining holes

Figure 10: A portion of a submarine storage and handling room



(a) Original model (b) Boundary contours

Figure 11: A model of an F15e fighter



(a) Original model (b) Boundary contours

Figure 12: A model of an M2 Bradley tank

units. Consequently, some holes remained after the vertex merging step (see Fig. 10(c)). These holes were then triangulated so as to obtain a fully repaired model. Similarly, Figs. 11(a,b) show a model of an F15e fighter and the gaps in this model, respectively. Fig. 12(a) shows a model of an M2 Bradley tank which underwent the same repairing process by using several values of ε (see Table 1). Fig. 12(b) shows the gaps in this model. In this example the remaining holes need not be triangulated in most cases.

Table 1 summarizes the performance of our implementation on the examples described above. Some of the running times reported here differ from those of [6] since the introduction of a k-D tree (to locate match candidates) and of a heap (to store the candidates) expedited the algorithm significantly (except for toy models, where the overhead dominates the running time).

9 Conclusion

A number of applications in computer graphics, e.g., virtual prototyping, global illumination, 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 the system to be quite stable. Small variations of score weights and ε did not affect the results of the repairing process, since in most cases a boundary edge matches one edge more strongly than all the other candidate edges. We believe that by automating most of the repairing process, while allowing the user to visualize the errors and override automatically-performed corrections, we achieve is a good balance between fast repairing the model and maintaining the original intent of the design. We consider such user involvement essential for successful model-repair system.

However, our system is only a step towards a robust and comprehensive repairing tool for correcting 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 are currently not detected by our system. A small box lying on top of a large box always results in two separate solids. A more general

Model	Initial Vert.	Initial Facets	Boundary Poly.	Boundary Edges	Matches Cand.	Matches Processed	Remaining Holes	Final No. of Vertices
Cube	55	37	8	55	68	26	0	29
Ball	432	816	2	48	1,035♣	23	0	403
			(step omitted)				2	432
T-Box	80	132	12	48	1,128♣	24	0	66
Sea Boat	5,863	10,625	705	2,503	7,369	979	116	4,892
F15e	1,198	2,237	58	333	69	38	39	1,151
Tank ($\varepsilon = 0.1$)	19,020	12,890	1,635	13,229	1,024	802	1,298	18,113
($\varepsilon = 0.3$)					2,536	1,202	1,182	17,576
($\varepsilon = 1.0$)					16,031	2,506	816	15,868
Torpedo Room	12,898	20,504	2,501	10,612	421	379	2,226	12,520

(a)

♣Allowing a very large tolerance.

	Time (Seconds) Loading Data	Pre-processing	Preparing Cand. List	Processing Cand. List	Post-processing	Filling Holes	Saving Data	Total
Cube	0.00	0.00	0.02	0.01	0.00	0.00	0.00	0.03
Ball	0.06	0.00	0.06	0.04	0.05	0.01	0.01	0.23
	0.06	0.00		(step omitted)		0.02	0.02	0.10
T-Box	0.01	0.01	0.01	0.00	0.01	0.00	0.00	0.04
Sea Boat	1.13	0.10	3.13	0.79	0.70	0.45	0.19	6.49
F15e	0.19	0.01	0.13	0.04	0.07	0.10	0.05	0.59
Tank	2.38	0.20	3.99	0.30	0.58	(1.46)♠	0.51	7.96
	2.38	0.20	4.32	0.75	0.62	(1.73)♠	0.50	8.77
	2.38	0.20	5.93	3.77	0.72		0.44	13.44
Torpedo Room	2.36	0.22	8.86	0.10	0.69		0.44	12.67

(b)

♠Not counted for the total running time.

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.

approach that includes full-scale CSG operations could generate a single manifold surface in such cases. We need to better integrate the geometric and visualization sub-systems. Currently, they run in separate stages. Ideally, visualization should accompany geometric repair and any user override should be instantly reflected in the evolving model. While efficiency is not as crucial as for off-line model correction, if RSVP were to be generalized and included in a modeling package that detects errors at modeling time, further improvements in efficiency are required. Furthermore, the 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. The additional feature of user-defined constraints would be another important enhancement of the system. For example, one might specify that two given polygons must always be maintained at right angle to each other, while their vertices move during mergers. Similarly, one must impose C^1 or C^2 continuity constraints on the *corrected* spline surfaces for the result to be really useful. Furthermore, any holes must be filled with spline patches, not with triangles. Correcting trimmed spline models is also an important step towards error-free models.

10 Acknowledgements

We thank Ken Fast, Greg Angelini, Jim Boudroux, and Electric Boat corporation for making the model of torpedo storage and handling system available for our testing. The Bradley model is courtesy of Army research lab and Viewpoint Data Labs.

References

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

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

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

[4] E. Arge, M. Dæhlen, T. Lyche, and K. Mørken. Constrained spline approximation of functions and data based on constrained knot removal. In J. Mason and M. Cox, eds., *Algorithms for Approximation II*, Lecture Notes in Mathematics, pages 4–20. Chapman and Hall, London, 1990.

[5] 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. To appear in *Computational Geometry: Theory and Applications.*

[6] G. Barequet and S. Kumar. Repairing CAD models. In *Proc. IEEE Visualization*, pages 363–370, Phoenix, AZ, 1997.

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

[8] G. Barequet, D. Shapiro, and A. Tal. History-driven reconstruction of polyhedral surfaces from parallel slices. In *Proc. IEEE Visualization*, pages 149–156, San Fransisco, CA, 1996.

[9] 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. (SIGGRAPH Proceedings).

[10] J.L. Bentley, Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

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

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

[13] J. Cohen, A. Varshney, D. Manocha, and G. Turk et al. Simplification envelopes. In *Proc. ACM SIGGRAPH*, pages 119–128, 1996.

[14] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide.* Academic Press Inc., 1993.

[15] D. Filip. Blending parametric surfaces. *ACM Transactions on Graphics*, 8(3):3–16, 1989.

[16] R. Goldman and T. Lyche, eds. *Knot Insertion and Deletion Algorithms for B-spline Curves and Surfaces.* SIAM, Philadelphia, 1993.

[17] A. Guéziec, G. Taubin, F. Lazarus, and W. Horn. Cutting and stitching: Efficient conversion of a non-manifold polygonal surface to a manifold. Computer science research report rc20935(92693), IBM Research Division, 1997.

[18] C.M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, San Mateo, CA, 1989.

[19] C.M. Hoffmann. Algebraic and numeric techniques for offsets and blends. In W. Dahmen, M. Gasca, and C. Micchelli, eds., *Computations of Curves and Surfaces*, pages 499–528. Kluwer Academic Publishers, 1990.

[20] H. Hoppe. Progressive meshes. In *Proc. ACM SIGGRAPH*, pages 99–108, 1996.

[21] K. Hsu and D. Tsay. Corner blending of free-form n-sided holes. *IEEE Computer Graphics and Applications*, 18(1):72–78, 1998.

[22] T. Lyche and K. Mørken. A discrete approach to knot removal and degree reduction algorithms for splines. In J. Mason and M. Cox, eds., *Algorithms for Approximation*, pages 67–82. Oxford University Press, 1987.

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

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

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

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

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

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

[29] X. Sheng and I.R. Meier. Generating topological structures for surface models. *IEEE Computer Graphics & Applications*, 15(6):35–41, November 1995.

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

[31] J. Xia and A. Varshney. A dynamic view-dependent simplification for polygonal models. In *Proc. IEEE Visualization*, pages 327–334, San Fransisco, CA, 1996.