Using Tolerances to Guarantee Valid Polyhedral Modeling Results

Mark Segal

Silicon Graphics Computer Systems*

Abstract

A polyhedral solid modeler that operates on boundary representations of objects must infer topological information from numerical data. Unavoidable errors (due to limited precision) affect these calculations so that their use may produce ambiguous or contradictory results. These effects cause existing polyhedral modelers to fail when presented with objects that nearly align or barely intersect[10][7].

An object description associating a tolerance with each of its topological features (vertices, edges, and faces) is introduced. The use of tolerances leads to a definition of topological consistency that is readily applied to boundary representations. The implications of using tolerances to aid in making consistent topological determinations from imprecise geometric data are explored and applied to the calculations of a polyhedral solid modeler. The resulting modeler produces a consistent polyhedral boundary when given consistent boundaries as input.

CR Categories and Subject Descriptors: I.3.5 **[Computer Graphics]:** Computational Geometry and Object Modeling - *curve, surface, solid and object representations; geometric algorithms, languages, and systems;* G.4 **[Mathematical Software]:** Reliability and Robustness

Additional Key Words and Phrases: solid modeling, boundary representation, numerical uncertainty

1 Introduction

Constructive solid geometry (CSG) is a well-established technique for specifying solid objects. Finding an object's boundary given its CSG description entails computing the boundary of an intersection, union, or difference of solids given by their boundaries. Several methods have been used successfully to carry out these "boolean" operations on polyhedra[10][11][19]. All these methods compute intersections between the input boundaries and use this information to produce a resulting boundary.

*2011 N. Shoreline Blvd., Mountain View, CA 94043

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

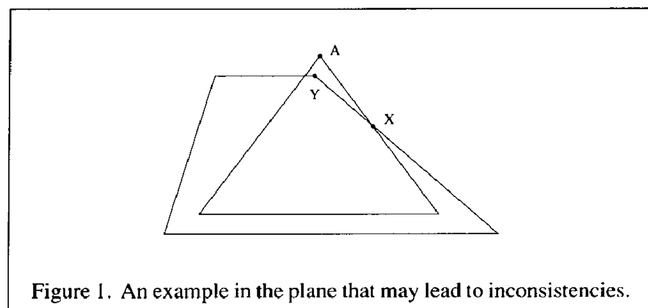


Figure 1. An example in the plane that may lead to inconsistencies.

A computed intersection may be inexact either because of the accumulated effects of round-off error in floating-point computations or because of uncertainty in the coordinates on which the calculation is based. Because computed intersections must be used to derive topology in the result, such topology may be ambiguous or inconsistent. These effects sometimes cause previously published algorithms to fail.

For instance, consider deciding whether a vertex of one polyhedral boundary lies in a face of the other[4]. If a vertex lies in a face, then the vertex's coordinates produce zero when substituted into the face's plane equation. However, if the face is not triangular, its plane equation may not be defined exactly, because the edges that define the face may not lie precisely in a single plane. The vertex's coordinates may similarly be uncertain.

The test may be modified by deeming the vertex to lie in the face's plane if it lies within some small specified distance of an approximate plane, but this decision is still arbitrary (based on the small value chosen) and care must be taken that it is not contradicted by some later decision. The problem is exacerbated by making other related decisions based on numerical data, such as whether or not two faces are coplanar, and by deriving the locations of new vertices and edges from inexact, previously computed coordinates. Any inconsistency may lead to a result which does not represent the boundary of a solid; it may even cause the modeler to fail without producing any output.

Figure 1 shows an example that may pose difficulties in two dimensions. An algorithm for finding polygon intersections might compute intersections among edges, break edges into segments at intersection points, and output those segments that lie inside of one polygon or the other (for two coincident segments, one of the two would be output). Suppose this algorithm were applied to the polygons in Figure 1, and suppose the intersection of an edge of the

©1990 ACM-0-89791-344-2/90/008/0105 $00.75

triangle and an edge of the trapezoid is found at X. Now suppose point Y is considered, and is found to lie close enough to A that it is deemed coincident with it (for clarity, the distance between A and Y is exagerated in the figure). But this is impossible since an intersection has already been found at X. Unless steps are taken to handle this situation, the algorithm's result may be missing an edge or contain an extra edge between X and A or X and Y.

A more concrete example is given in [10] and later cited by other authors[7][19]. Two congruent cubes are placed with their centers coincident but with one rotated relative to the other about some axis by a small angle. A polyhedral modeler that accomodates coplanar faces can handle either an angle of zero degrees or large angles, but most modelers fail for a range of angles near zero. The size of this critical interval can be made very small by careful attention to numerical issues in the calculation of intersections, but there is still a range of angles, however tiny, for which a modeler may fail.

Several methods have been proposed to address these issues. One method, exact rational arithmetic, attempts to represent all coordinates exactly as rational numbers[5], because intersections among features with rational coordinates have rational coordinates. More sophisticated techniques allow an arbitrary symbolic formula for each coordinate[2][1]. However, at some point the formula must be evaluated. If the coefficients and arguments (which are derived from the input objects' coordinates) are inexact, topological data derived from the formula may still be ambiguous.

If the input coordinates are exact[16], then an arbitrary precision calculation can achieve any accuracy. Unfortunately, the output is still of limited accuracy, so the results from one calculation cannot be used as input to the next. A grid may be placed over the computed object and features snapped to it[18], but this process may incorporate large portions of the grid into the object, creating a myriad of tiny "staircase" features.

Another method is to back up and increase the precision of an earlier calculation if it is later found that its accuracy is not great enough to resolve a topological determination[3]. Instead of recomputing previous calculations, their results may instead be perturbed as necessary so that they do not introduce ambiguity[8][21]. Other methods that yield robust algorithms in limited domains include ones for computing intersections in the plane[12][13] and finding intersections between convex polyhedra[9].

We concede that input coordinates may be inexact and that numerical errors may accumulate during computation. We give methods that have their roots in interval arithmetic[20][6][14], to measure that accumulation. Each vertex, edge, and face is assigned its own tolerance. These tolerances are updated during algorithm operation, allowing a modeler to locate regions in which numerical uncertainty may lead to topological ambiguity. Using maintained tolerances instead of arbitrarily selected "epsilons" makes it possible to decide these cases consistently, guaranteeing that a solid modeling result is always the boundary of a solid.

2 Overview

We begin by reviewing the essential means by which polyhedral boundaries in three dimensions are specified. Next we present a system of tolerances applied to these specifications and introduce a notion of consistency for inexact geometric objects. We describe modifications to the standard geometric notions of intersection, coincidence and containment that can be applied to objects incorporating tolerances. These concepts provide the framework necessary to build a polyhedral modeler that accomodates numerical uncertainty in its topological decisions. Then we describe the basic solid modeler and how it is modified to incorporate tolerance handling. We conclude with some examples and a discussion of the prospects of improving and extending the current implementation.

3 Geometry, Topology, and Consistency

We distinguish three *topological features* in three dimensions: vertices, edges, and faces. A vertex is essentially a point. An edge is a connected subset of a line bounded by vertices. A face is a connected subset of a plane bounded by non-self-intersecting polygonal curves made up of edges lying in that plane. The affine subset of three-space corresponding to each feature (a point for a vertex, a line for an edge, and a plane for a face) is called a *flat* of dimension 0, 1 or 2. A particular flat is specified by giving *coordinates* that specify its position with respect to a fixed origin and basis.

A *geometric object* consists of an arbitrary set of particular topological features described in two parts. The first part, called *metric data*, are the coordinates that locate each feature in space. The second part is a list of *connectivities*. Each connectivity comprises a set of geometric features that intersect (called the connectivity *set*), together with a distinguishing feature that represents the intersection. For instance, several edges may meet at a vertex, or several faces may intersect in an edge.

An object's representation is *consistent* if every intersection among an arbitrary set of features is represented explicitly in the connectivities, and conversely, every connectivity represents an intersection computable from the metric data.

Consistency is desirable because connectivities encode most of an object's topology (a solid's genus, for example, can be found directly from its connectivities). Further, a solid modeler relies on connectivities to derive information about an object, such as determining which faces lie on either side of an edge or which edges emanate from a vertex. Inconsistency between connectivities and metric data may invalidate assumptions made in the modeler's algorithms, leading to the modeler's failure.

3.1 Numerical Uncertainty

In practice consistency may be difficult to achieve because a polyhedron's metric data are represented with limited precision quantities. Computing an intersection from these inexact quantities (even if carried out to arbitrary precision) may lead to ambiguous results.

To account for inaccuracy, and to quantify its effects during geometric computation, we introduce a *tolerance*[17] ϵ for each topological feature f describing the positional uncertainty of its corresponding flat a. A tolerance specifies a *tolerance region* about a flat. To define this region, let q be any point and a be a flat, and let

$$\text{dist}(q, a) = \min\{\|q - q_a\| \mid q_a \in a\}.$$

The region

$$r = \{q \mid \text{dist}(q, a) < \epsilon\}$$

is the tolerance region about the flat a (Figure 2).

A tolerance also specifies a tolerance region $tol(f)$ about a feature f. Because a feature's boundary may not lie exactly in its corresponding flat, the definition of a feature's tolerance region requires some notation. Consider the projection operator $P_a(y)$ that computes the perpendicular projection of the point or points of y

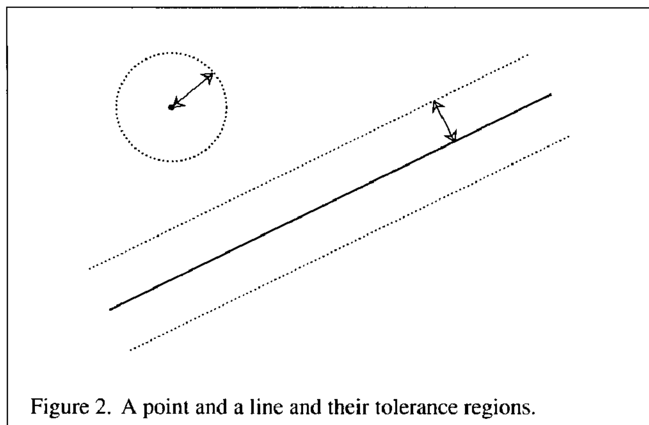
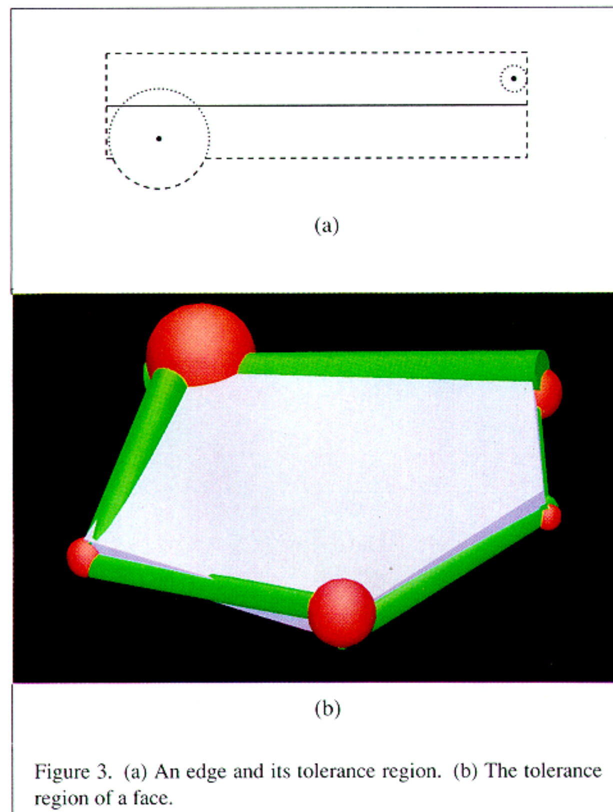


Figure 2. A point and a line and their tolerance regions.



(a)

(b)

Figure 3. (a) An edge and its tolerance region. (b) The tolerance region of a face.

onto the flat a. We also write $P_f(y)$ if f is a feature, in which case the projection is onto f's corresponding flat. Let $B(f)$ be the set of boundary features of f, and let a be f's corresponding flat. A point q lies in a feature f's tolerance region, denoted $tol(f)$, if either of two conditions hold:

1. q lies within some tolerance region of $B(f)$.
2. $\text{dist}(q, a) < \epsilon$ and $P_a(q)$ lies in the interior of the planar polygon or linear segment defined by $P_a(B(f))$.

The tolerance region of a vertex is the same as that of its point. Figure 3 shows the tolerance regions of an edge and a face.

3.2 Approximate Intersection

Consider the distance d between two flats a_1 and a_2

$$d = \min\{\|x_1 - x_2\| \mid x_1 \in a_1, x_2 \in a_2\}$$

where $x_i \in a_i$ means x_i is a point of flat a_i. Two flats *intersect within* ϵ if the distance between them does not exceed ϵ. Two flats a_i and a_j *approximately intersect* if they intersect within $\epsilon = \epsilon_i + \epsilon_j$. That is, if the tolerance regions of two flats intersect, then the flats approximately intersect. Similarly, approximate intersection of two features is defined as intersection of the corresponding tolerance regions. *Consistency* for inexact objects is the same as consistency for exact objects with intersection replaced by approximate intersection.

We require that polyhedral boundaries presented to the solid modeler be approximately consistent. The algorithm is designed so that given two approximately consistent polyhedral boundaries, the output will also be an approximately consistent polyhedral boundary. This ensures that the algorithm's output can be fed back into the algorithm as input.

3.3 Coincidence, Containment, and Alignment

In the exact case, a feature of lower dimension may be *contained* in a feature or flat of higher dimension. Two features or flats of the same dimension may *coincide* if the two sets of points that they define are equal. We say that a feature f_1 *aligns* with feature f_2 if f_1 is contained in f_2's flat. Any feature contained in or coincident with another must also be aligned with it.

A feature f_1 *approximately aligns* with f_2 (with corresponding flats a_1 and a_2) if $\text{dist}(q, a_2) < \epsilon_1 + \epsilon_2$ for every $q \in P_{a_1}[tol(f_1)]$. f_1 is *approximately contained* in f_2 if it is approximately aligned with f_2 and if $P_{a_2}[tol(f_1)] \subseteq P_{a_2}[tol(f_2)]$. f_1 and f_2 are *approximately coincident* if $B(f_1)$ approximately coincides with $B(f_2)$. Two vertices are approximately coincident if they approximately intersect. See Figure 4.

3.4 Restrictions on Feature Alignments

The use of inexact metric data requires some restrictions on the allowable positions of object features if an object is to make sense under the familiar rules of exact geometry. These restrictions exclude certain feature constellations whose topology would be unclear. We assume that individual objects presented to the modeler as input follow the restrictions. One of the main concerns in the design of the modeler is ensuring that it detects and eliminates any violations of these restrictions in its result.

Consider alignment of features of equal dimension in the exact case, denoted by $f_1 \sim f_2$ if f_1 aligns with f_2. Such alignment is symmetric and transitive: if f_1, f_2, and f_3 are all features of the same dimension, and if $f_1 \sim f_2$, then $f_2 \sim f_1$; further, if $f_2 \sim f_3$, then $f_1 \sim f_3$. This is not automatically true for approximate alignment as Figure 4 shows.

However, for correspondence with the exact case, we also demand symmetry and transitivity of alignment of equal-dimensional features in the approximate case. Furthermore, to simplify data structures, we require that equal-dimensional features never approximately coincide; such coincidence is represented with a single feature.

Similarly, we declare illegal any intersecting feature tolerance regions if the associated boundary tolerance regions intersect in

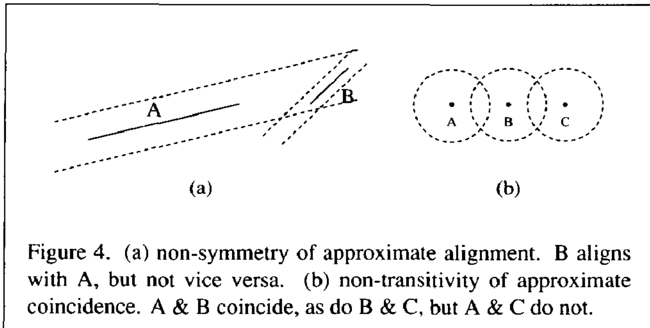


Figure 4. (a) non-symmetry of approximate alignment. B aligns with A, but not vice versa. (b) non-transitivity of approximate coincidence. A & B coincide, as do B & C, but A & C do not.

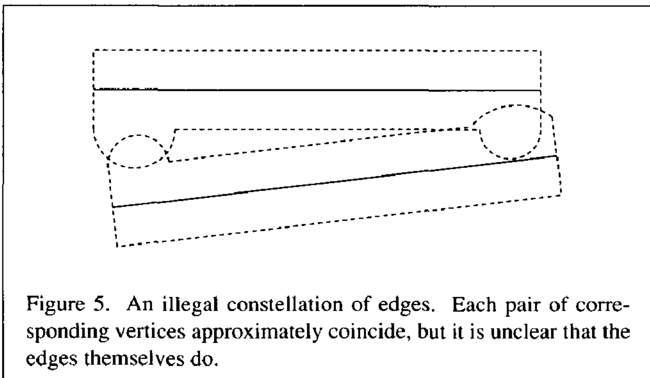


Figure 5. An illegal constellation of edges. Each pair of corresponding vertices approximately coincide, but it is unclear that the edges themselves do.

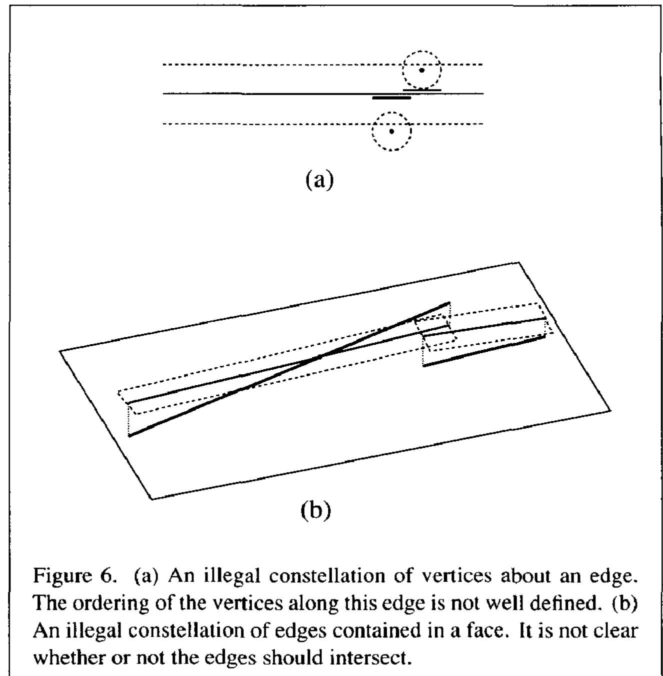


Figure 6. (a) An illegal constellation of vertices about an edge. The ordering of the vertices along this edge is not well defined. (b) An illegal constellation of edges contained in a face. It is not clear whether or not the edges should intersect.

more than one connected component. Figure 5 shows an example of the situation that this restriction is designed to exclude.

Finally, if two distinct features are approximately aligned with and intersect some higher dimensional feature, and if their projections onto the higher dimensional feature's ideal flat also approximately align and intersect (Figure 6), then we require the two features to be approximately aligned.

4 A Reliable Solid Modeler

The techniques for achieving consistent object descriptions in the presence of numerical uncertainty form the basis for a reliable polyhedral modeler. By reliable we mean that, given the descriptions of two consistent boundary representations, the modeler always runs to completion and that its result is always the consistent description of a boundary representation. In most cases, the connectivities of this result precisely correspond to the connectivities that would be obtained using exact inputs with infinite precision arithmetic. In some cases, when numerical uncertainties lead to arbitrary decisions, the exact result may have topologically complex regions of small size that are collapsed into simpler regions in the modeler's output.

4.1 Algorithm Overview

The basic algorithm[19] operates by detecting and removing intersections among pairs of candidate faces. It does this by tracing around the edges of one face and finding their intersections with the other face's plane; the procedure is repeated with the faces roles reversed. The intersection points are collected and sorted into order along the intersection line. Each face's contours are then sliced apart between these points to partition the faces into parts that intersect only along shared edges, if at all. Information gathered during cutting operations is used after all faces have been partitioned to select the contours appropriate to the desired CSG result.

There is nothing that makes this algorithm especially amenable to using tolerances to assure reliable behavior in the presence of numerical uncertainty. Other algorithms could be similarly equipped, although the exact measures taken depend on the particular form of the algorithm and the stages that it goes through to achieve a modeling result. The essence of achieving reliability is to ensure that every stage of the algorithm leaves the geometric object on which it is working consistent and free from any violations of the restrictions presented in the last section.

4.2 Data Structures

The algorithm encodes a geometric object in a data structure consisting of four items. These data structure items completely encode both the metric data and connectivities of a geometric object. The first three items correspond to features of dimension 0, 1, and 2, respectively.

Vertices

A vertex's position is described by three coordinates giving its location relative to a fixed origin and basis. In addition, each vertex has a list that indicates which edges emanate from it.

Edges

An edge's flat is described by a parametric line equation given by a point and a unit direction. Its boundary is described by pointers to its two bounding vertices. Each edge also has a list that describes which faces are incident on it (more precisely, each item in the list is a pointer to the use of the edge within a face's contour). Edges

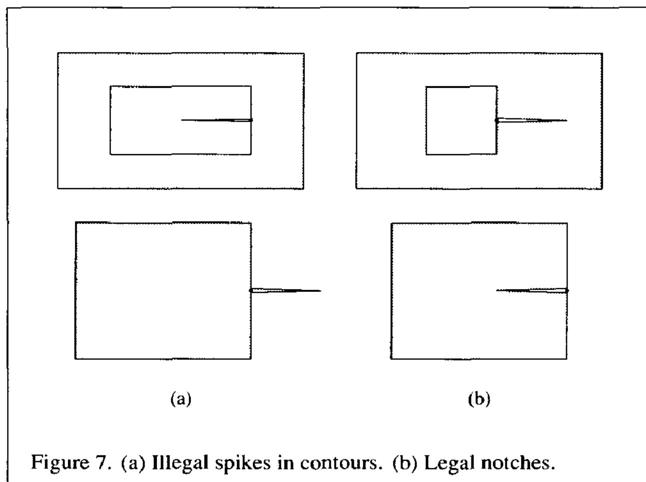


Figure 7. (a) Illegal spikes in contours. (b) Legal notches.

are unique; there can be no more than one edge between any pair of vertices.

Faces

A face's flat is described by an implicit plane equation given by a point on and a unit normal to the plane. Its boundary is described by one or more pairwise disjoint contours, some of which may be holes. Each contour is a list of edges that form a non-self-intersecting loop.

A contour must bound an open planar set so that the local orientation of the contour is defined at each vertex (Figure 7). This restriction ensures that a contour does not contain zero-width "spikes," although zero-width notches are allowed (the algorithm may insert such notches into faces as partitioning proceeds).

Solid Boundaries

The boundary of a solid is simply a list of faces that make it up.

Tolerances and Related Quantities

Three quantities are kept with each feature f in addition to the coordinates that specify its flat's position. The first of these is a tolerance describing the accuracy of these coordinates. The second, kept for edges and faces only, is an *extent*, denoted $ext(f)$. This value gives the maximum deviation of the feature's tolerance region from its specified flat. The third quantity is called the *near* value, denoted $near(f)$. This value is initialized to a large positive number when the modeler starts. Whenever the distance between a feature and some other feature is computed, and the pair of features are deemed not to intersect, the near value is updated so that it is the minimum distance to all features found to not intersect the feature so far. The use of the extent and near values will become apparent as the algorithm is described.

If objects are presented to the algorithm without edge or plane equations or feature tolerances, the equations and tolerances are computed. Each vertex in the input is assigned an arbitrary small tolerance, typically 10^{-10} for an object centered about the origin whose longest edges have approximate length 1. This makes the ratio of the tolerance to the vertex coordinates a few times the machine precision when using IEEE double-precision floating-point, guaranteeing that round-off error will generally not affect tolerances. An edge's equation is found by finding the line that passes through its two endpoints. The tolerance assigned to such a computed edge equation is zero, because the line must intersect the vertices' tolerance regions. The extent for the edge is the larger of the two endpoints' tolerances. A face's equation is computed by applying Newell's algorithm[15] to each of the faces contours and averaging the results. The tolerance is found by substituting the coordinates of each vertex in the face's boundary into the computed equation and recording the maximum deviation from zero. The extent is this deviation added to the maximum of the vertices tolerances.

4.3 Topological Modifications

As the algorithm proceeds, it modifies the topology of the input to incorporate computed intersections. These modifications are:

1. Breaking an edge into two edges at a new vertex where the edge crosses a face.
2. Inserting new edges into a face along an intersection line.
3. Merging coplanar faces, coincident edges, and coincident vertices.

Of these operations, (1) and (2) are purely topological in that they do not require alteration of metric data or tolerances. The operations in (3), however, require the selection of new coordinates and a new tolerance for the merged feature. Further, merging a pair of features may require topological changes beyond the merge itself.

Vertex Merging

If there is an edge between two merged vertices, it must be eliminated. This means excising the corresponding pointer from every contour that uses the edge. Further, there may be an edge from each of the merged vertices to a third vertex; such edges must be merged into a single edge.

Edge Merging and Edge Breaking

If two distinct edges are bounded by the same two vertices (as may happen after a vertex merge), then the edges must be merged. Every reference to the either of the original edges must be replaced with a reference to a single new edge. If the two merged edges belong to a contour with three edges that is not a hole (Figure 8a), then merging the edges would create an illegal two-edged contour. In this case the vertices at the ends of this illegal contour are merged, eliminating the contour. Further, merging edges may create a spike in a contour that has more than three edges (Figure 8b). This possibility is detected by checking if the angle made by the unmerged edges in the contour is less than 180°. If so, the spike is removed from the contour.

Two edges that have no boundary vertices in common *overlap* if one approximately aligns with the other and they approximately intersect. They also overlap if they violate the restriction illustrated in Figure 6b. Two overlapping edges e_1 and e_2 must be broken into a series of two or three non-overlapping ones. This breaking creates two edges with identical bounding vertices that must be merged. Figure 8c shows how a merged edge resulting from an edge break can create a spike. This spike is eliminated when the edges are merged.

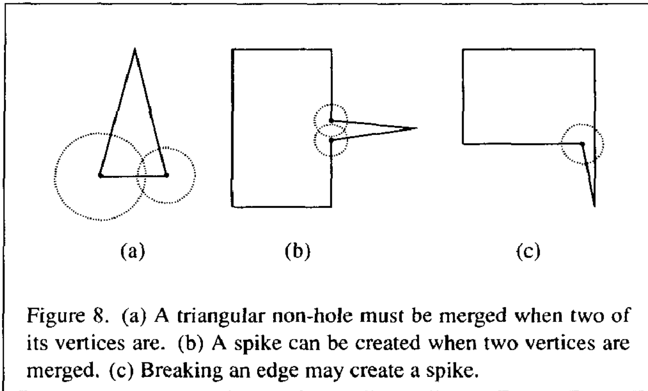


Figure 8. (a) A triangular non-hole must be merged when two of its vertices are. (b) A spike can be created when two vertices are merged. (c) Breaking an edge may create a spike.

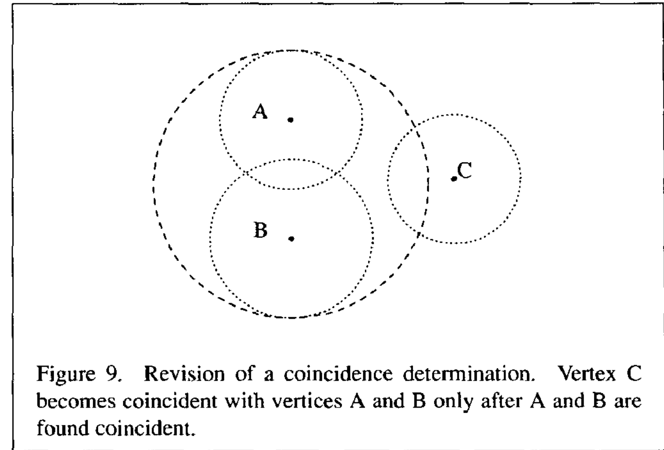


Figure 9. Revision of a coincidence determination. Vertex C becomes coincident with vertices A and B only after A and B are found coincident.

Coplanar Faces

Two faces that approximately align and intersect must be merged into a set of non-overlapping contours just as edges are. In general, this would require using an algorithm to partition intersecting polygons in the plane, but no such action is taken. Instead, we rely on there being two contours on either side of an edge. For an edge that lies in the plane of the coplanar faces, one of these contours belongs to one of the coplanar faces. The other contour belongs to a face that does not lie in the coplanar faces' plane. This third face will induce partitions in each of the coplanar faces. Eventaully (after all such edge-sharing faces have been considered), the coplanar contours will be partitioned into non-overlapping parts.

Perturbation of Merged Features

Merging two features f_1 and f_2 into a single new feature f requires finding coordinates and a tolerance for that feature. To maintain consistency, we must have $tol(f_1) \cup tol(f_2) \subseteq tol(f)$. As much as possible, the coordinates of f's flat are selected so as to minimize f's tolerance subject to this inclusion condition. For a pair of vertices, this amounts to finding the sphere of minimum radius that encloses two intersecting spheres. For a pair of edges, an attempt is made to find the bounded cylinder that encloses two intersecting bounded cylinders. For a pair of faces, a less than optimal approach is taken for the sake of simplicity. Newell's algorithm is applied to each of the contours in the coplanar pair and the results are averaged to find a new plane equation. The coordinates of each of the vertices of the edges in $B(f)$ are substituted into the new plane equation and the maximum deviation recorded. Then for each $e \in B(f)$ with vertices v_1 and v_2, the coordinates of $P_e(v_1)$ and $P_e(v_2)$ are substituted into the plane equation and the maximum deviation found. The largest of these two maximum deviations becomes f's tolerance. The procedure ensures that f approximately contains its boundary.

Both $ext(f)$ and $near(f)$ must also be assigned a value. $ext(f)$ must be recomputed from f's boundary features (this is done during the computation of its new coordinates and tolerance), while $near(f) = \min\{near(f_1), near(f_2)\}$. If f_1 and f_2 are vertices or edges, they may be contained in a feature of higher dimension. The extent of this feature may be affected when f_1 and f_2 are merged, so every feature that includes f_1 or f_2 must be located and a test made to see if its extent needs updating.

Backtracking

A merged feature f may be assigned a tolerance that is larger than either of the tolerances of the original features f_1 and f_2. This may create an inconsistency, because a third feature f_3 may have previously been determined not to intersect f_1 or f_2 individually, but f, with its larger tolerance region, may now approximately intersect f_3. Figure 9 shows the situation with vertices. If vertex C is considered against vertices A and B individually, vertex C will be found not coincident with either one. However, if vertex A is then considered against vertex B, these will be deemed coincident. A new tolerance region will be constructed that encloses those of A and B, contradicting the previous conclusion that C did not coincide with either A or B.

This situation is accomodated by maintaining the near value for each feature. If a feature f's tolerance is increased (i.e. by merging two features) and the new tolerance exceeds $near(f)$, an inconsistency may have been created. If this occurs, $near(f)$ is reset to a large positive value, and the algorithm is rerun from the beginning. While this *backtracking* is certainly expensive, it is essential to ensure that no inconsistencies are introduced. The cost of backtracking could be reduced by employing a spatial subdivision to limit those features that are reconsidered, but this has not been implemented.

Backtracking may lead to further backtracking, but any recursion must eventually terminate. The reason is that when backtracking occurs, all intersections between features considered so far will have been accounted for. Thus, any modifications made during a backtracking phase can only reduce the number of features (by merging them). At worst the process ends when all features that had been considered when backtracking started are merged into a single feature with a large tolerance.

4.4 Algorithm Operation

The first step in partitioning a pair of faces f_1 and f_2 is to compute the line of intersection between their planes a_1 and a_2. The direction of this line is given by $\mathbf{u} = \mathbf{n}_1 \times \mathbf{n}_2$ where $\mathbf{n}_i$ is the normal to a_i. A third plane is introduced with normal $\mathbf{u}$ and passing through a point in one of the faces boundaries. The intersection point of the three planes gives a point $\mathbf{v}$ on the intersection line. The intersection line tolerance is made large enough so that the tolerance region about the intersection line encloses $tol(a_1) \cap tol(a_2)$. If the face's

tolerances are ϵ_1 and ϵ_2, respectively, the tolerance turns out to be

$$\epsilon = \left(\frac{\epsilon_1^2 + \epsilon_2^2 + 2\epsilon_1\epsilon_2 \cos\alpha}{\sin^2\alpha}\right)^{1/2}$$

where $\sin\alpha = \|\mathbf{n}_1 \times \mathbf{n}_2\|$.

4.5 Coplanarity

If $\frac{\epsilon}{\max\{\epsilon_1, \epsilon_2\}} > n$, where n is a user-specifiable *coplanarity factor*, the faces are tested for coplanarity. n is typically between 100 and 1000. Small values force the algorithm to deem pairs of faces that intersect in a relatively large dihedral angle to be coplanar.

If the ratio exceeds n, the faces are deemed coplanar if two vertices in one face's boundary lie on opposite sides of the other's plane, or if any vertex of one face lies within a distance equal to the extent of the other. This second condition ensures that two faces whose tolerance regions intersect in more than one connected component are found to be coplanar (see Figure 5).

4.6 Finding Edge-Face Intersections

If the ratio of the edge tolerance to the maximum of the two face tolerances is not too large, then the intersections of one of the face's edges with the other face's plane are found. For each edge, there are four possibilities: (1) the edge does not intersect the plane, (2) the edge intersects the plane in one boundary vertex, (3) the edge lies entirely in the plane, or (4) the edge crosses the plane.

These possibilities are distinguished by computing the signed distance of each of the edge's endpoints to the plane. The signed distance is deemed to be zero if the vertex lies within tolerance of the computed intersection line. If the line's equation is $\mathbf{v} + t\mathbf{u}$ and the vertex's coordinates are $\mathbf{p}$, then the vertex lies on the intersection line if

$$\|\mathbf{p} - \mathbf{v}\|^2 - [\mathbf{u} \cdot (\mathbf{p} - \mathbf{v})]^2 \leq (\epsilon + \epsilon_p)^2$$

where ϵ is the intersection line tolerance and ϵ_p is the vertex's tolerance.

If the vertex v does not lie on the intersection line its signed distance d_v to the plane is obtained by substituting its coordinates into the plane equation $d = \mathbf{n} \cdot (\mathbf{p} - \mathbf{o})$ where $\mathbf{n}$ is the plane unit normal and $\mathbf{o}$ is the point in the plane. $|d_v|$ is compared with both the vertex's and other face's near value; each value, if less than $|d_v|$, is replaced with $|d_v|$.

Even if the vertex does not lie on the intersection line, it may still intersect one of the other face f's boundary features. If $|d_v| < ext(f)$, then each of the other face's boundary features is checked to see if it intersects the v. If so, the v's distance from the plane is zero.

If $\text{sign}(d_{v_1}) = \text{sign}(d_{v_2})$ for the edge e's vertices v_1 and v_2, then e does not cross the other face's plane. If one sign is positive and the other is negative, e crosses the plane and the intersection point x between e and the plane is computed. The tolerance for x is ϵ. This guarantees that x approximately intersects both faces' planes as well as the computed intersection line. $near(x)$ is $\min\{near(e), dist(x, v_1), dist(x, v_2)\}$. The computed intersection point, along with auxilliary information indicating the edge and contour from which it came, are added to the list of intersection points.

If one of an edges' vertices lies in the other face's plane, the vertex is entered into the list of intersection points. Information about edge adjacency and local orientation of the contour at the vertex is also recorded.

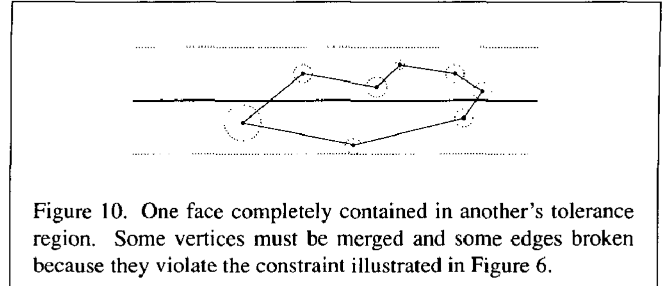


Figure 10. One face completely contained in another's tolerance region. Some vertices must be merged and some edges broken because they violate the constraint illustrated in Figure 6.

If it turns out that all the edges of one face lie in the other's plane, the faces are deemed coplanar. This may happen in spite of the intersection line having a relatively small tolerance if one face's edges are all about the same length as the other face's tolerance. Even if this is the case, the vertices are still recorded in the intersection point list. Some of the vertices may have to be merged (Figure 10); such merging is done after sorting the points.

Finally, it may happen that one or both of an edges' endpoints lie off of the other face's plane, but the edge itself is approximately contained in the plane. This can occur if an edge e's tolerance is large compared to those of its endpoints. This situation is detected by determining if $P_e(v_1)$ and $P_e(v_2)$ (v_1 and v_2 are e's vertices) are both approximately contained in the other face's plane. If so, then the plane's tolerance is made large enough so that both of the original vertices approximately intersect the plane. If the new tolerance exceeds the face's near value, backtracking occurs. Otherwise, the current face pair is rerun with the increased face tolerance.

4.7 Feature Merging

After finding the intersection points of each face's edges with the other's plane, the points are sorted into order along the intersection line. This is accomplished by computing the value $t = (\mathbf{x} - \mathbf{v}) \cdot \mathbf{u}$ for each intersection point $\mathbf{x}$ (this effectively projects the point onto the calculated intersection line). and sorting in order of increasing t. Points that appear more than once are sorted secondarily using orientation information.

With the points ordered, a search is made for points that lie close enough together that they must be merged. Each point in the sorted list is checked against the next point. If they approximately intersect, the vertices defining the points are merged into a single vertex. This process may be repeated, eventually coalescing several vertices. In addition to vertex merging, edges that lie on the intersection line must be broken at intersection points that fall between their endpoints. Performing this operation ensures that pairs of edges that overlap are broken into a sequence of non-overlapping segments.

Vertex and edge merging may invalidate some of the data about orientation and incident edges stored with the intersection points. Sometimes it is difficult to determine how to update these data. In these cases the current face pair is simply rerun.

4.8 Cutting

The list of sorted and merged intersection points is used to determine where to slice contours to partition the pair of faces. Figure 11 shows some examples of cuts that are inserted into pairs of simple faces.

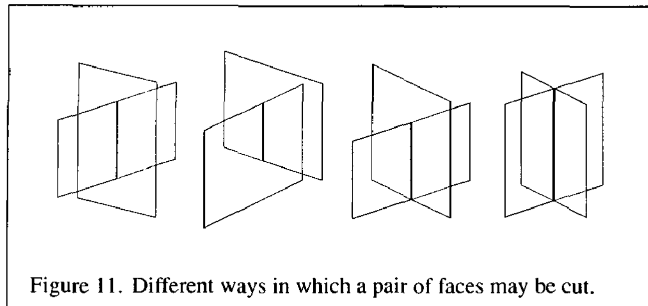


Figure 11. Different ways in which a pair of faces may be cut.

Either or both endpoints of a cut may correspond to newly calculated intersection points. Therefore an edge giving rise to such a point must be broken at that point before the cut is made. The edge is not broken when the intersection is computed because it is not known at that time whether or not the point lies in the interior of the face that the edge intersects. The point at which the edge is to be broken may approximately intersect one of the edges endpoints. If this is the case, the cut is made, after which the vertices are merged.

5 Results

Figure 12 shows the results of a union of two cubes centered at the origin where one cube is rotated through a small angle about a line through the origin with direction $(1,2,3)^T$ and the other is aligned with the coordinate axes. All computations were carried out in double-precision IEEE floating-point on a Silicon Graphics IRIS 4D/210VGX workstation. To make the results of this test easier to see, the face coplanarity factor was set to the rather low value of 20. The figure on the left was obtained with an angle of 9.59863838340879299° while that on the right was produced with an angle of 9.59863838340879300°. The entries in the corresponding rotation matrices differed by no more than one significant bit in their mantissas. The algorithm was also tested for many values of the rotation angle slightly above and below these two values. For all higher values, the object produced was similar to the one at right; for all lower values, the object was similar to the one at left. Further transitions occur as the angle is decreased even more until the object finally becomes a cube (but with non-planar faces) at an angle of about 1.5°. Figure 13 shows the object in Figure 12 in terms of its feature's tolerance regions.

If the coplanarity factor is increased to a more reasonable factor of 500, the angle at which the transition occurs is approximately 0.2°. In this example, even high coplanarity factors (10^5) with tiny angles (about 0.001°) produce a boundary of something other than a cube (but with some very short edges). This is because starting face and edge tolerances are zero, so computed vertex tolerances are set to the minimum value (10^{-10}). If face tolerances are initialized to about 10^{-6}, however, such high coplanarity factors and small angles cause computed vertex tolerances to become so large as to engulf the original vertices, eventually collapsing the cubes into a single vertex. Higher coplanarity factors can be used to achieve reasonable results if the input tolerances are reduced, but a vertex's tolerance may not be reduced so far that its ratio to the coordinates whose inaccuracy it quantifies approaches the machine precision.

Figure 14 shows the union of a pair of more complex objects, and Figure 15 shows the same union computed with a small coplanarity factor and with the final tolerances displayed.

Figure 16 shows that the modeler can be used to generate complex objects without the tolerances causing any feature merging. This object was generated by starting with two congruent cubes in different orientations and computing their intersection. Then the resulting objects were rotated with respect to one another and the union taken and so on until, after eight iterations, the final result was obtained. In this object tolerances remain imperceptibly small because features are not merged (with the default tolerances and a coplanarity factor of 1000). If the process is carried out for another iteration, however, some features are near enough to one another that they are merged, and tolerances increase. Continued iteration causes tolerances to grow; features are rapidly collapsed so that after 11 iterations, the object is reduced to a single vertex.

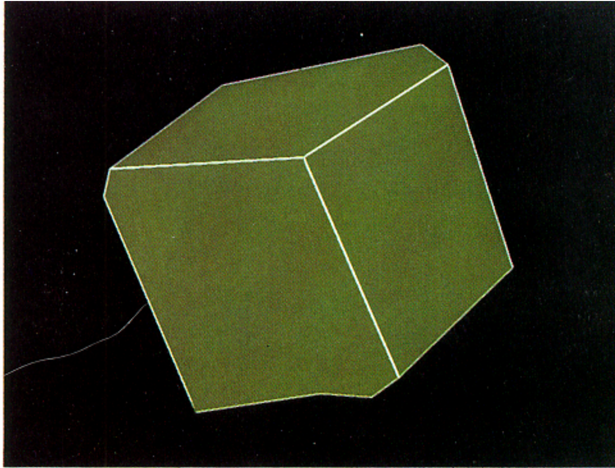
6 Conclusions

We have presented a tolerance scheme that accounts for the consistent definition of boundary representations whose specification may be numerically uncertain. This scheme has been applied to a polyhedral modeler computing CSG operations on boundary representations. The modeler always succeeds in producing a result that is a consistent polyhedral boundary. Doing so may entail the localized collapse of several features into a single feature with a large tolerance.

Achieving this behavior has its costs; the algorithm runs 5-10 times slower, even when backtracking is not required, than an earlier version that performed only cursory tolerance checking. If n is the number of faces, the algorithm runs in time $O(n^2)$ without backtracking, although the average time can be made $O(n)$ by employing a spatial subdivision. Careful attention to making certain checks only when absolutely necessary, and replacing often called functions (such as the distance of a vertex to a plane) with in-line code would certainly reduce the required time. Even so, the resulting algorithm would likely be at least a factor of two slower than a similar algorithm that is not guaranteed to work in all cases.

With backtracking, the worst case behavior is $O(e^f)$ where f is the number of features. In practice, it is difficult to construct a case that achieves this behavior, although if backtracking occurs once it typically occurs several times. For most object constellations no backtracking occurs; it is only necessary when features approximately coincide and then only if there are other features near the coincident ones.

If the input to the algorithm has numerical uncertainties, there may be no other solution than to occasionally collapse features. In normal situations, such as computing the union of two imperfectly abutting objects, this collapsing leads to modest tolerance increases that have only local effect. Only in unusual situations, such as those designed to stress the algorithm shown in Figures 12-16, are the tolerance increases compounded enough to be objectionable. In any case, it makes sense to report large tolerances so that a user or higher-level program equipped with heuristics can be alerted to regions where unexpected behavior may occur. Perhaps tolerances can be decreased or features slightly perturbed to remove such behavior. In some cases the algorithm is too conservative in assigning tolerances to computed features because it has no access to global relationships among these features. It may be possible to design a post-processor that decreases tolerances while maintaining consistency. One possibility is to use a relaxation algorithm that perturbs feature coordinates in an attempt to reduce an objective function made up of a weighted sum of all the features' tolerances.



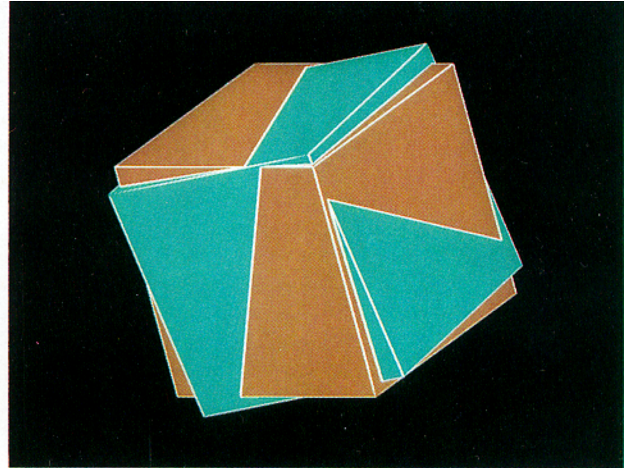


Figure 12. Union of two cubes.

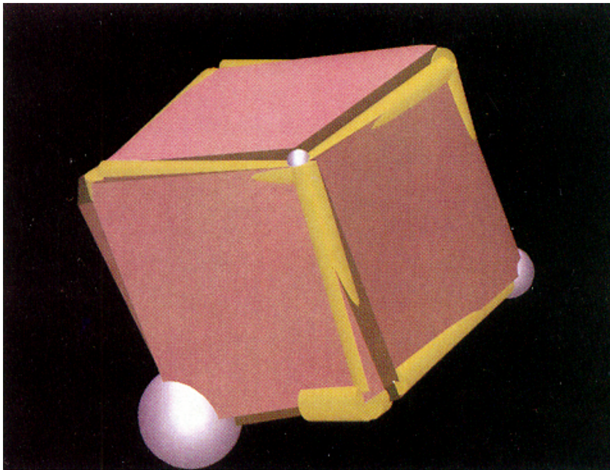


Figure 13. Union of two cubes with tolerance regions.

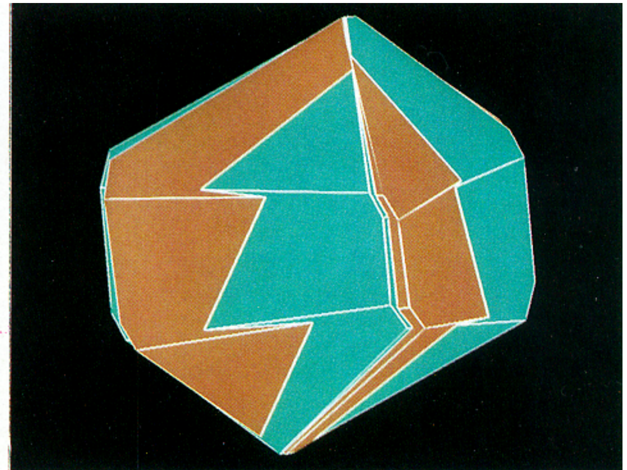


Figure 14. Union of two objects

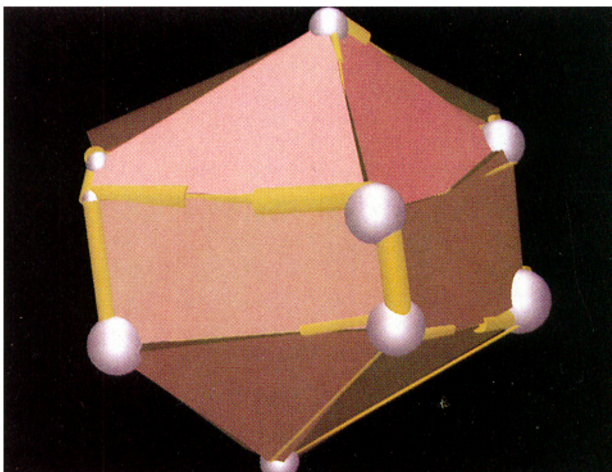


Figure 15. Union of two objects with tolerance regions.

Figure 16. A complex object formed by a series of CSG operations

The tolerance scheme can, in principle, be applied to other algorithms in geometric modeling. As already noted, polyhedral modelers other than the one described here can be modified to incorporate tolerances. Further, there is nothing inherently three-dimensional about a system of tolerances, so that the same techniques can be applied to geometric algorithms that operate in higher dimensions. Finally, it should be possible to extend the tolerance scheme to accomodate curved objects. The topological features of such objects are the same: vertices, edges, and faces. In this case the tolerances would not only reflect how accurately a feature is specified but also the accuracy of the computation used to evaluate points that lie on it. The challenge for this application is to extend the notions of approximate containment and coincidence to features that may intersect in several connected components or in isolated regions of tangency.

Acknowledgements

Many thanks to Carlo Séquin for his continued encouragement and support. Thanks also to Dan Baum, Derrick Burns, and Bob Drebin for helpful comments and criticisms.

References

[1] Dennis S. Arnon. Topologically reliable display of algebraic curves. Proceedings of SIGGRAPH'83 (Detroit, Michigan, July 25–29, 1983). In *Computer Graphics 17*,3 (July 1983), 219–227.

[2] John Francis Canny. *The Complexity of Robot Motion Planning*. PhD thesis, MIT, 1987.

[3] David Dobkin and Deborah Silver. Recipes for geometry & numerical analysis—part I: An empirical study. In *Proceedings of the Fourth ACM Symposium on Computational Geometry*, pages 93–105, Urbana, Illinois, 1988.

[4] A. Robin Forrest. Computational geometry and software engineering: Towards a geometric computing environment. In David F. Rogers and Rae A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 23–37. Springer-Verlag, New York, 1987.

[5] W. Randolph Franklin, Peter Y.F. Wu, and Sumitro Samaddar. Prolog and geometry projects. *IEEE CG & A 6*,11 (November 1986), 46–55.

[6] Leonidas Guibas, David Salesin, and Jorge Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proceedings of the Fifth ACM Symposium on Computational Geometry*, pages 208–217, 1989.

[7] Cristoph M. Hoffmann, John E. Hopcroft, and Michael E. Karasick. Robust set operations on polyhedral solids. *IEEE CG & A 9*,6 (November 1989), 50–59.

[8] Cristoph M. Hoffmann, John E. Hopcroft, and Michael S. Karasick. Towards implementing robust geometric computations. In *Proceedings of the Fourth ACM Symposium on Computational Geometry*, pages 106–117, Urbana, Illinois, 1988.

[9] John E. Hopcroft and Peter J. Kahn. A paradigm for robust geometric algorithms. Technical Report TR 89-1044, Department of Computer Science, Cornell University, October 1989.

[10] David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes. Constructive solid geometry for polyhedral objects. Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18–22, 1986). In *Computer Graphics 20*,4 (August 1986), 161–170.

[11] Martti Mäntylä. Boolean operations of 2-manifolds through vertex neighborhood classification. *ACM Transactions on Graphics 5*,1 (January 1986), 1–29.

[12] Victor Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence 37* (1988), 377–401.

[13] Victor Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *Proceedings of the Fifth ACM Symposium on Computational Geometry*, pages 197–207, 1989.

[14] S.P. Mudur and P.A. Koparkar. Interval methods for processing geometric objects. *IEEE CG & A 4*,2 (February 1984), 7–17.

[15] I.E. Sutherland, R.F. Sproull, and R.A. Schumacker. A Characterization of Ten Hidden-Surface Removal Algorithms. *Computing Surveys 6*,1 (March 1974), 1–55.

[16] Thomas Ottmann, Gerald Thiemt, and Christian Ullrich. Numerical stability of geometric algorithms. In *Proceedings of the Third ACM Symposium on Computational Geometry*, pages 119–125, Waterloo, Ontario, 1987.

[17] Aristides A. G. Requicha. Toward a theory of geometric tolerancing. *International Journal of Robotics Research 2*,4 (Winter 1983), 45–60.

[18] Mark Segal and Carlo H. Séquin. Consistent calculations for solid modeling. In *Proceedings of the First ACM Symposium on Computational Geometry*, pages 29–38, 1985.

[19] Mark Segal and Carlo H. Séquin. Partitioning polyhedral objects into non-intersecting parts. *IEEE CG & A 8*,1 (January 1988), 53–67.

[20] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1980.

[21] Chee-Keng Yap. A geometric consistency theorem for a symbolic perturbation scheme. In *Proceedings of the Fourth ACM Symposium on Computational Geometry*, pages 134–142, Urbana, Illinois, 1988.