Set Operations

Robust Set Operations on Polyhedral Solids

Christoph M. Hoffmann, Purdue University
John E. Hopcroft, Cornell University
and
Michael S. Karasick, IBM T.J. Watson Research Center

We describe an algorithm for performing regularized Boolean operations on polyhedral solids. Robustness is achieved by adding symbolic reasoning as a supplemental step to resolve possible numerical uncertainty. Additionally, numerical redundancy and numerical computation based on derived quantities are reduced as much as possible. We also discuss our experience with our implementation of the algorithm.

We present a robust algorithm for performing regularized set operations on polyhedral solids described using a boundary representation. That is, we give a reliable method for the regularized intersection, union, difference, and complement of polyhedral solids.

Algorithms for regularized set operations on polyhedral objects have been implemented before.[1-5] However, the robustness problem has not been addressed deeply, and certain input configurations of simple objects may lead to failure. An example illustrates the problem: Consider a unit cube. Take a second cube, obtained from the first by successive rotation about each principal axis by a small angle, and intersect it with the first cube. Many polyhedral modelers fail when the angle of rotation drops below 2 degrees, because the two cubes are sufficiently similar that errors are made when computing their intersection.[4]

The robustness problem is rooted in floating-point arithmetic. While floating-point calculation can distinguish object features that are sufficiently separated, it can never reliably determine their coincidence. Moreover, in a certain region of proximity, floating-point computation will give seemingly random results because of round-off errors. This region of criticality, in which many modelers fail, depends on the machine precision and on the nature of the computation. It cannot be addressed satisfactorily by declaring two features coincident whenever they are closer than some tolerance ε. Doing so leads to inconsistent decisions in certain situations. A more sophisticated approach is needed.

Because incidence testing is a fundamental operation in an intersection algorithm, we are ill-advised to base these tests on floating-point calculations alone. Conversely, using purely symbolic calculation or exact arithmetic is not the answer, because of ineffi-

0272-17-16/89/1100-0050$01.00 ©1989 IEEE

ciency and the fact that the original data is often inexact: For example, four planes meant to intersect at a common vertex probably intersect in sets of three at four distinct points near the vertex. An approach is needed that satisfies the following criteria: The method must be efficient, it must account for data imprecision, and it must make consistent incidence decisions.

In this article, we give a preliminary formulation of such a method; it relies on floating-point calculation when this is safe, and deduces relative position symbolically and reproducibly when floating-point calculation yields ambiguous results. It is perhaps not possible to formulate a complete and consistent general calculus for all geometric computations while maintaining the simplicity and efficiency of our approach, but it appears to be possible for many specific geometric operations. We have considered the problems of accuracy and robustness more generally elsewhere.[6,7]

We begin with an illustration of the robustness problem and how it is manifest in polyhedral intersection. Then, the chosen representation and a global description of the algorithm are given. Finally, we discuss how to structure and implement certain details to achieve robustness. A more detailed description of this material is available,[8] and a somewhat differently structured algorithm has been discussed.[7]

An example of intersection failure

The central difficulty in achieving robustness can be formulated as follows:

> A floating-point computation C is carried out. Depending on whether the result of C is zero, two geometric structures intersect or do not intersect (coincide or do not coincide). When the magnitude of the result of C is large, then nonintersection (nonincidence) can be determined with certainty. However, when the result has a small magnitude, then an uncertain decision must be made. When this decision conflicts with other, similar decisions made at other times during the computation, we could construct inconsistent data structures, which will cause the geometric algorithm to fail.

The difficulty of achieving consistency depends on the geometric operation and on the requirements of the data structures it accesses. For instance, we can prove fairly simply that two polygons can be robustly intersected, but we cannot yet prove a similar theorem about polyhedral intersection.[6]

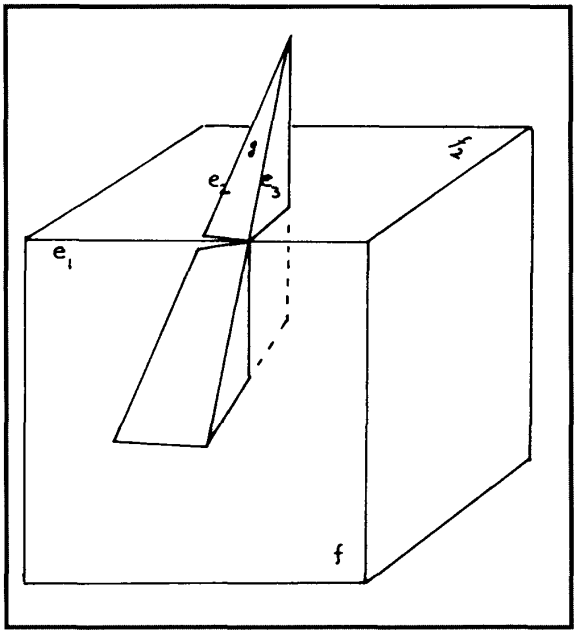


Figure 1. Cube and pyramid to be intersected.

We examine how polyhedral intersection might fail. Consider a cube and a triangular pyramid, shown in Figure 1, and consider computing their intersection. The two objects have been positioned such that edges e_1 and e_3 intersect. Edge e_2 approaches e_1 to within a small tolerance, but does not intersect.

The algorithm will determine independently, on the surface of each solid, a set of curves that are the intersection of the two surfaces of the solids. On the basis of this determination, the surfaces of the cube and of the pyramid will be subdivided into regions that are on the surface of the intersection and regions that are not on the surface of the intersection. However, since the intersection curves were determined independently on the solids, they happen to be incompatible. The cause of the incompatibility is as follows.

When determining a subdivision of the face f, to find a face of the intersection of the two objects, the algorithm intersects edges e_2 and e_3 with the face plane of f, and determines whether the two points lie on the edge e_1. Because of close proximity, it decides that both points are on e_1; that is, both e_2 and e_3 intersect e_1. In consequence, a short edge segment e_4 is created that lies on e_1. The computation for f and the analo-

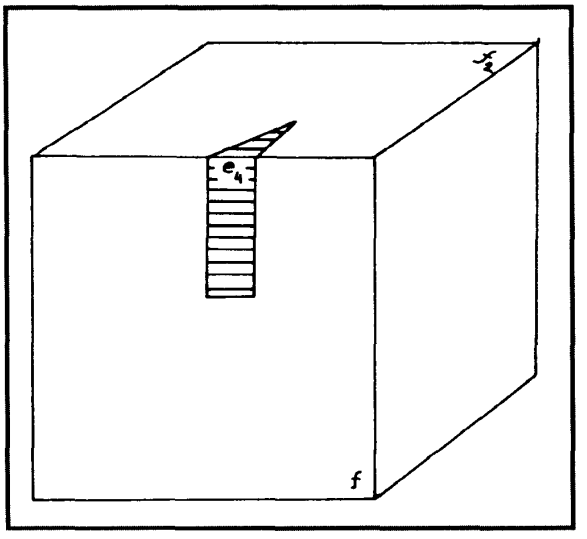


Figure 2. Subdivision of the cube's surface.

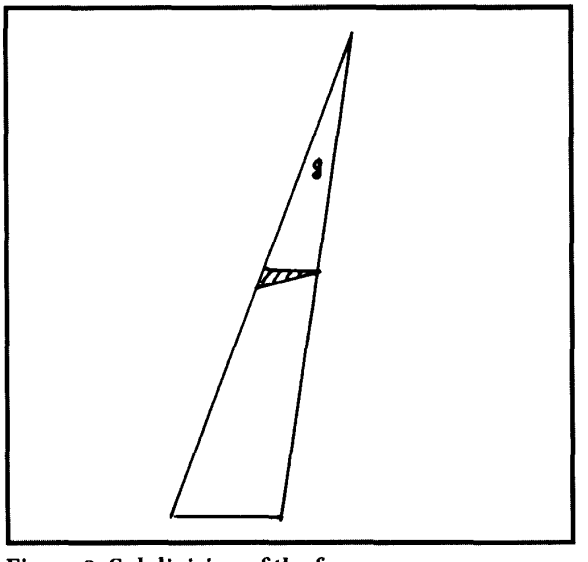


Figure 3. Subdivision of the face *g*.

gous computation for the top face f_2 of the cube results in a subdivision of the cube's surface shown in Figure 2.

Next the face g is analyzed, and the algorithm determines whether the vertices u and v of edge e_1 lie in the face plane of g. Because of the length of the edge, the vertices are at a considerable distance from the plane, and so it is decided that e_1 does not lie in the plane of g. Note that this decision contradicts the earlier finding that both e_2 and e_3 intersect e_1. The resulting subdivision of g is as shown in Figure 3. It is clearly incompatible with the subdivision of the cube, for the edges of the shaded triangle have no adjacent faces on the cube. Hence, the subsequent phase of assembling the surface of the resulting polyhedron cannot succeed, and the intersection algorithm fails because of the unexpected inconsistency in the data structures.

The problems of accuracy and robustness in geometric modeling are difficult, both on theoretical and practical grounds.[9] Clearly, each specific case such as the one just illustrated can be eliminated by restructuring the intersection algorithm. However, no general proof exists to date that this restructuring is possible without introducing other, different opportunities for inconsistency.

Representation

The difficulty of implementing a polyhedral solid modeler depends to a certain extent on the underlying representation. We choose to represent the surface as an orientable nonmanifold; that is, there are points whose neighborhoods are not homeomorphic to a disk, but the surface is bounded, enclosing a possibly infinite volume. This class of objects is closed under set operations. We have found that this representation delivers the simplest algorithms for intersecting objects in special positions.

We are especially concerned with limiting the redundancy of numerical data in our object representation, thereby reducing the opportunities to introduce inconsistencies when testing incidence. For this reason, the only numerical data used in our representation are the coefficients of the face plane equations. Vertex coordinates are specified implicitly as the intersection of three face planes. If a vertex is incident to more than three faces, its coordinates are defined as the intersection of three explicitly specified face planes. Auxiliary planes can be used to define edges and vertices when planes of the solid boundary are nearly tangent. All other model data are given in symbolic form—for example, which face planes intersect to define an edge, which edges are incident to a common vertex.

A *vertex* is a point in Euclidean three-space, defined as the intersection of three planes, although we allow more than three planes to meet at a vertex. An *edge* is the line segment connecting two distinct vertices and is defined by the intersection of two planes, although we allow more than two faces to meet on an edge.

A *convex polyhedron* is the intersection of finitely many half-spaces of finite volume, each bounded by a plane. The *regularized* intersection (complement,

union, difference) of two polyhedra is the closure of the interior of their set-theoretic intersection (complement, union, difference).[10,11] These operations constitute the regularized set operations on polyhedra. Since we consider only regularized operations, we drop the adjective.

A *polyhedron* is either a convex polyhedron or it is the result of a finite sequence of set operations on convex polyhedra. Note that an edge of a solid can be adjacent to more than two faces, and each vertex can be incident to more than one corner. Such edges and vertices consist of surface points whose neighborhoods are not homeomorphic to a disk. Moreover, a polyhedron may have infinite volume, but it must have a finite, bounded surface area.

Each face of a polyhedron is bounded by one or more cycles of directed edges. Each edge in a cycle is directed so that the interior of the face is locally to the right. Associated with each directed edge is a *tangent vector* in the direction of the edge, and a *face-direction vector* that points orthogonally from the interior of the directed edge into the interior of the face belonging to the directed edge (see Figure 4).

The *shell* of a solid is a connected component of the surface of the solid. The representation for a solid is given as a list of representations, one for each shell. Each shell representation is given by lists of the faces, edges, and vertices of that shell. Our representation is called the *star-edge representation*.[8] It is equivalent to Weiler's nonmanifold representation for solids.[12] Related representations are described by Hanrahan,[13] and Dobkin and Laszlo.[14]

There may be more than two directed edges incident to a vertex on a face. Therefore, we store them in radially sorted order, about the vertex. Moreover, the edges incident to the same vertex on a face are paired so that two consecutive directed edges, in radial order, enclose face interior. In this case we speak of an *area-enclosing pair* of edges (see Figure 5). Similarly, the faces incident to a common edge are radially ordered about this edge, and these faces are paired so that consecutive faces enclose a wedge of solid interior. Here we speak of a *volume-enclosing pair* of faces.

It is well known that all Boolean set operations can be reduced to intersection and complement. For the star-edge representation, a solid is complemented by complementing each of its shells, and a shell is complemented by inverting the normal vector to each face, the orientation of each directed edge, and the radial order of the directed-edge lists for each vertex on a face. Thus such regularized set operations as union and difference can be efficiently implemented in terms of intersection.

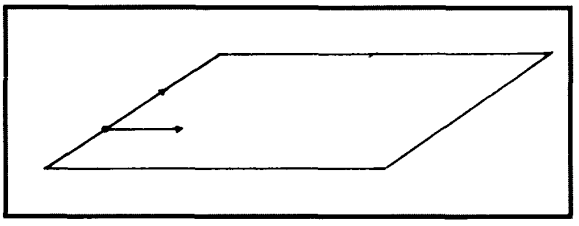


Figure 4. Edge-direction and face-direction vectors.

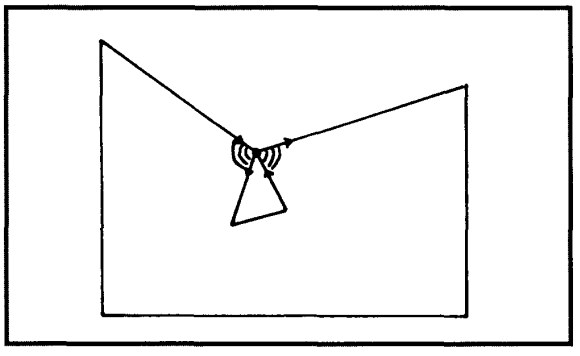


Figure 5. Area-enclosing pairs of edges.

The intersection algorithm

We describe the polyhedral intersection algorithm conceptually. The description does not touch on several robustness issues except in one respect: The asymmetric structure of the algorithm reflects the asymmetry of incidence tests that are described in a later section.

Conceptually, the algorithm to intersect solid A with solid B merges intersecting shells and retains or discards nonintersecting shells on the basis of a containment test:

1. Intersect every shell of A with every shell of B.
2. Merge intersecting shells into a set of shells that constitute a portion of the boundary of $A \cap B$.
3. Add all shells of A contained entirely within B, and add all shells of B contained entirely within A.

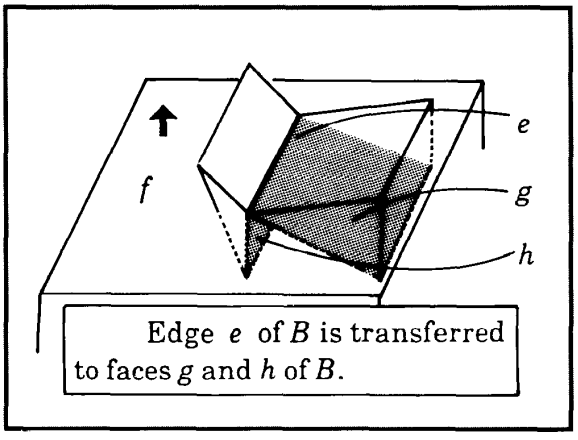


Figure 6. Transfer of edge segment of Type 4.

We describe only the method for merging intersecting shells, since the other steps are straightforward.

Merging intersecting shells

By far the most complex step of the algorithm is intersecting and merging shells. Conceptually, shells are intersected by intersecting their faces. If no intersections are found, the shells do not intersect. If there are intersections, then the shells are merged as required.

Assume that a shell of A and a shell of B intersect and must be merged. The standard approach proceeds as follows:

1. Intersect all faces of A with all faces of B, determining the correct subdivision of the appropriate faces.
2. Add to the subareas on the surface of $A \cap B$ the faces of A in the interior of B and the faces of B in the interior of A.

The numerical computations implementing these steps are asymmetric: As described later, a vertex of A may lie on every plane defining a vertex of B, but not vice versa. Therefore, the merging step has been restructured slightly by introducing a corresponding asymmetry, as follows:

1. For each face f of A, intersect its plane P with B, yielding a cross-face graph called G_P. Classify the areas into which G_P partitions P as being *inside*, *on* the surface, or *outside* of B, encoding this information suitably by orienting the edges of G_P.
2. Intersect G_P with the boundary of f, and determine which areas on P are faces of $A \cap B$.
3. Add all faces of A that are in the interior of B.
4. Transfer the relevant edges of subareas of f bounding $A \cap B$ to the corresponding faces of B, thereby subdividing the faces of B that intersect the surface of A.
5. Add all faces of B that are in the interior of A.

This approach should be interfaced with heuristics that quickly reject nonintersecting face-pairs, and combined with techniques from computational geometry to yield an asymptotically efficient algorithm.[15] Briefly, each face is boxed. The set of boxes is intersected to determine a subset of face-pairs that might intersect. The time required to find all intersecting boxes is $O(n \log^2(n) + k)$, where n is the number of boxes and k is the number of box-pair intersections.

The cross-sectional graph G_P

The graph G_P comprises the intersection of solid B with the plane P containing the face f of solid A. To construct it, edges of B are intersected with P, yielding *intersection points*. Certain intersection points are then linked by line segments representing the intersection of faces of B with P. The resulting graph G_P partitions P. The areas delimited by its edges and vertices consist of points that lie inside, outside, or on the surface of B. Such areas could then be labeled as **in**B, **out**B, or **on**B. If an area delimited by edges and vertices of G_P is on the surface of B, then we distinguish whether on B this surface area is oriented in the same way as P. If the orientation is opposite, then the area cannot be part of the surface of $A \cap B$. Accordingly, the label **on**B could be refined to $\mathbf{on}_{\mathbf{in}}B$ for areas oriented the same way, and $\mathbf{on}_{\mathbf{out}}B$ for areas oriented the opposite way. This area classification is effected by suitably orienting the directed edges of G_P, ignoring the distinction between **in**B and $\mathbf{on}_{\mathbf{in}}B$.

The construction of G_P requires many incidence tests. It is critical to robustness that the outcome of a test be the same if this test is repeated when constructing a graph for an adjacent face. This is achieved by annotating the data structures for the boundary elements of A and B. For example, an edge of A is annotated with the vertices, edges, and faces of B that the edge intersects, and the edges and vertices induced by the intersections are also annotated.

The edges of G_P are either edges of solid B or they are *cross-face edges* across faces of B. Each cross-face edge is the intersection of the interior of a face g of B with P. The edges of G_P are oriented so that the surface area of $A \cap B$ is locally to the right, as seen from the

exterior of solid A. After orienting edges, all face areas of $A \cap B$ are enclosed by oriented cycles of edges and are assembled from them.

Subdividing faces of solid A

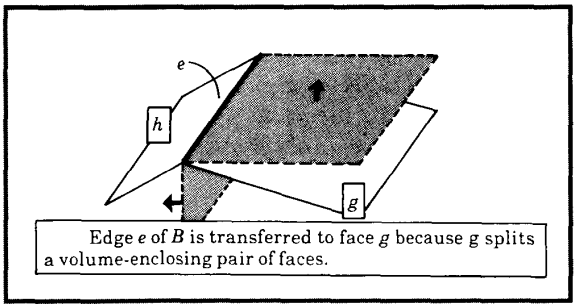
We subdivide a face of solid A to identify those subareas that are on the surface of $A \cap B$. The subdivision is effected by intersecting the boundary of that face with the associated cross-sectional graph. (Recall that G_P is the cross-section of solid B with plane P of face f.) The areas bounded by G_P are a set of faces or the 2D regularized complement of such a set. Intersecting G_P with f therefore has the flavor of an intersection algorithm of polygonal 2D objects, and many of the steps to be described here are analogous to the entire 3D intersection algorithm. The structural analogy with polyhedral intersection is seen by equating edge cycles with shells. The intersection of f with G_P is done as follows:

1. Intersect the edges of f with the edges of G_P.
2. At each intersection point, determine which incident edge segments bound faces of $A \cap B$.
3. Construct directed-edge cycles bounding the surface area of $A \cap B$, by traversing the directed edges of G_P and f between intersection points.
4. Add all additional directed edges and vertices that are contained, nonintersecting components of G_P and f.

Adding faces induced by B

When the processing described above is complete, all faces of $A \cap B$ that lie on the surface of A have been generated. The final step in the shell-merging algorithm is the subdivision of faces of B. That is, we add to $A \cap B$ all missing faces that lie on the surface of B. The simplest way to obtain those faces is to run the algorithm with the roles of A and B interchanged, but a more direct approach results in greater robustness.

First we examine the known edges and vertices of $A \cap B$. We identify those that are adjacent to faces g of B extending inside A. Note that they are found from the vertex, edge, and face annotations described in the section on the cross-sectional graph. That is, every intersection point or line between g and a face of A is transferred to g.[8] As an intersection line is transferred to g, the area it bounds on g is classified as **in**A or **out**A. Again, this information is expressed by orienting edge segments on the faces of B. Finally, all faces of B are examined, and the relevant face areas on those faces are assembled into faces of $A \cap B$. Note that areas that lie on the surface of both A and B can be ignored, since they were classified as $\mathbf{on}_{\mathbf{in}}B$ and have already been accounted for.



Edge e of B is transferred to face g because g splits a volume-enclosing pair of faces.

Figure 7. Transfer of edge segment of Type 5.

Let P be the plane containing the face f of A. In f, a directed edge of $A \cap B$ is one of the following:

1. An edge segment of f bounding an **in**B area of G_P.
2. An edge segment of f bounding an $\mathbf{on}_{\mathbf{in}}B$ area of G_P.
3. A cross-face edge segment of G_P not coincident with an edge of f.
4. An edge segment of B not coincident with any edge of f.
5. An edge segment of G_P coincident with an edge of f.

When considering the segment for transfer to the appropriate faces of B, we proceed as follows. All edges of Type 1 can be ignored, for no face of B intersects in that edge. An edge segment e of Type 2 is also located on a face g of B; e is transferred to g by creating a directed edge of e on g with orientation opposite to the directed edge of e already created. A cross-face edge-segment $\bar{e}$ of Type 3 is a portion of the intersection of g with P. It is added to g by creating a directed edge of $\bar{e}$ on g with opposite orientation to that directed edge of $\bar{e}$ already created. An edge segment e of Type 4 is transferred to each face of B adjacent to e and below P (see Figure 6). Finally, an edge segment of Type 5 is transferred to all those faces of B that lie between volume-enclosing face pairs of the coincident edge of A. In Figure 7, face g is inside A, whereas face h is not. This transfer is implemented by merging the radially ordered directed edges of the two coincident edges of A and B.

Robustness in the operations

Robustness is achieved primarily by designing reliable basic operations with which the algorithm is implemented. Our approach is based on three concepts:

- By understanding the inherent error of the floating-point calculations involved, we can distinguish between trustworthy and inconclusive results. A decision based on inconclusive, numerical results is checked for consistency with earlier decisions where possible.
- There is no redundancy in the numeric input data. This eliminates the possibility of contradictory input data.
- To contain the propagation of errors, all numerical computations are based on input data whenever possible.

The basic operations and tests on which the consistency and correctness of the algorithm depend are demonstrated by the following examples:

- *Incidence tests.* Does a vertex lie on a plane, do two vertices coincide, or do two edges intersect?
- *Ordering operations.* What is the relative order of points along a line, and what is the radial order of directed lines originating in a common point?
- *Pairing operations.* Which pair of faces enclose volume at an edge, and which pair of edges enclose face area at a vertex?

What is fundamentally different about incidence tests, as opposed to the ordering and pairing operations, is that we are comparing features from distinct objects and thus these features can be arbitrarily close: They can be closer than the accuracy of the model data. Thus, there is no guarantee that all ambiguities can be resolved, and we will have to make an arbitrary choice at some point. This choice must be consistent with related choices.

Whenever a decision is needed as to whether a feature of one object coincides with a feature of another object, we perform the necessary numerical computation. If the features are separated by some predetermined tolerance, we can safely assume that the features do not coincide. However, we can never numerically determine that features do coincide, because of the uncertainty caused by numerical round off. Thus, we need some way to make a positive decision when features are within this tolerance. Whenever features are sufficiently close, we are free to conclude that they do or do not coincide, provided that we do not make a decision that is inconsistent with some known fact or previous decision.

At first it appears that a powerful theorem prover is necessary to determine whether a given decision is independent of previously made decisions. However, by understanding the types of inconsistencies that arise in a limited domain, such as set operations on solids, we can develop a small set of tests that maintain consistency for situations that arise in a specific algorithm. Whenever we must make a logical decision about the relative position of two possibly coincident features, we apply these tests to see if they separate these features. If not, then we say that these features coincide. We say this because our experience in such situations indicates that features coincide, and we are simply observing the result of numerical round off. At some small separation we must declare that two features do coincide, or we will get many features topologically distinct, but infinitesimally separated. We will still get a certain number of small structures, but they will be necessary to keep the topology consistent during construction of the resulting object. Afterward a post-processor might be used to perturb the face equations to eliminate structures of a size smaller than some tolerance. In this way a new object is constructed with some minimum feature separation.

This philosophy of using symbolic reasoning to ensure consistency of logical decisions when using numeric calculations should have widespread application in areas outside geometric modeling.

Accuracy

The accuracy and dependability of a floating-point calculation depends on both the machine precision and on the computation at hand. With each numerical computation is associated an uncertainty estimate ε. A logical decision based on two numerical values is reliable, provided the values differ by at least the sum of the two uncertainty estimates. If the numerical computation does not support a reliable logical decision, then additional symbolic computation is done to resolve the ambiguity. Two basic floating-point calculations are involved when testing incidence:

1. Given a vertex defined by the intersection of three planes, compute its coordinates.
2. Given point a and plane equation P, evaluate $P(a)$ to determine whether a is on P.

The first computation can be implemented by Gaussian elimination, and its associated uncertainty can be estimated from the condition number of the linear system and the machine precision.[16] The second computation can be done by substituting and evaluating the point coordinates into the plane equations. Some authors evaluate instead a 4×4 determinant.[17]

While it is not possible to exceed machine precision efficiently, we can easily assume less precision than is

actually delivered. We are able to do this by supplying an input parameter to the algorithm specifying a nominal machine precision. This is very useful when studying experimentally the effects of changing the tests for deciding incidence.

Let ε_a denote the error for determining the coordinates of point a. Points a and b are *distant* if they are separated by at least $\varepsilon_a + \varepsilon_b$. Otherwise, these two points are *near*. Note that near points may, but need not, coincide.

Now assume that we test whether a lies on plane P. The accuracy of this answer, $\varepsilon_{a,P}$, depends on ε_a and the accuracy of evaluating the equation of P. Certainly, a is distant from P if the magnitude of $P(a)$ exceeds $\varepsilon_{a,P}$. Otherwise, a is near P and a may lie on P. In the following discussion we drop all subscripts and collectively refer to the uncertainty estimates as ε.

Vertex incidence testing

One test performed when computing the intersection of two solids determines whether vertex v coincides with vertex w. This test can be implemented by testing whether v is on each of the three planes whose intersection defines w. This is not, however, equivalent to testing whether w lies on the planes intersecting in v. In fact, the test is neither symmetric nor transitive. For this reason, the incidence rules developed below must be limited to carefully chosen situations.

All incidence tests first make the necessary floating-point calculations. If the features in question are distant, no further action is required. Otherwise the features are near, and we examine adjacent features to obtain information on which to base a decision.

Vertex on plane. We test whether vertex v lies on plane P as follows:

1. If the magnitude of $P(v)$ is greater than ε, then v is distant from P, and the sign of $P(v)$ determines whether v is above or below P.
2. Otherwise, we examine the intersection of P with each edge incident to v. Consider an edge e incident to v and a vertex w, where w is shown to be off P by a recursive invocation of the vertex-on-plane test. If e intersects P at point a far from v, then v is not on P; v and w are on the same side of P if a is not in the interior of e; otherwise, v and w are on opposite sides of P.
3. Finally, if Step 2 fails to classify v, then v is on P.

Vertex on edge. First, we determine if vertex v is on the defining planes, P and Q, of edge e. If v is on P but not on Q, we know whether v is to the left or right of $P \cap Q$ from testing whether v is on Q. If v is on both P and Q, then we determine if v lies on or between the planes defining the endpoints of e.

Vertex on vertex. Vertex v is coincident with vertex w if v lies on each of the three planes that define w.

Note that asking whether v is coincident with w might result in a different answer than asking whether w is coincident with v. To avoid such an inconsistency we always perform this test asymmetrically, by asking if a vertex of solid A is coincident with a vertex of solid B. Another approach would have been to make the test symmetric by also requiring w to be on each plane defining v. However, if the first test succeeded and the latter test failed, we would be in the situation where v is on the defining planes of a vertex, but not coincident with their intersection.

Our tests do not guarantee transitivity of vertex coincidence. However, this cannot introduce an inconsistency, since the minimum separation between features on objects guarantees that at most one vertex from each object can be in the same neighborhood.

Edge and face-incidence tests

Edge and face-incidence tests arise in our intersection algorithm as follows. A plane P of a face f of solid A can intersect solid B in a collection of faces, edges, vertices, points in the interior of edges, and line segments in the interior of faces. Together, these elements form a cross section of solid B. We must identify intersections of f with this cross section. If f is adjacent to another face g, then the cross sections induced by the planes of f and g must agree, and it is critical to enforce this agreement. For example, if we determine that an edge common to f and g intersects an edge of B in the plane of f, then this common edge must intersect that same edge of B in the plane of g. Structured in this way, three questions arise when deciding edge intersection. Given face f contained in plane P of solid A, and edge e of f defined by the intersection of P with plane Q, we ask if e intersects one of the following:

1. An edge e' of B where e' is in P.
2. An edge e' of B where e' is not in P.
3. A cross-face edge $\bar{e}$ defined by the intersection of P with a face of B.

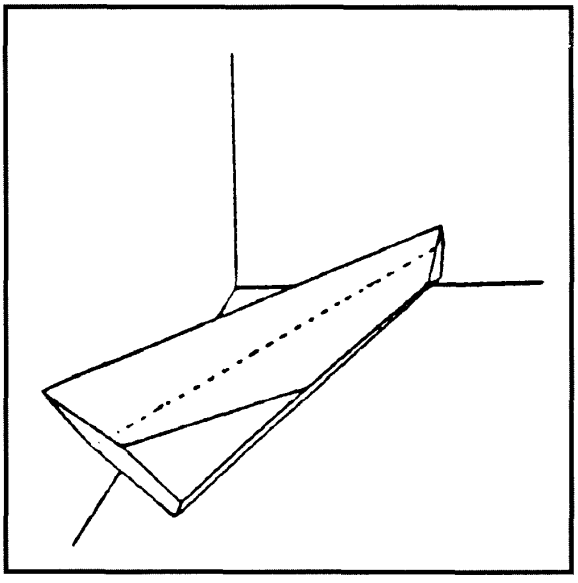


Figure 8. Sample heptahedron.

Each edge may be intersected in the interior or near a vertex. To distinguish among these cases, we test whether e' or $\bar{e}$ intersects the line $P \cap Q$ that contains e. Then we check whether e' or $\bar{e}$ contains, in addition, a vertex of e.

If e' intersects both P and Q transversally, then e' intersects the line $P \cap Q$ if the two intersection points are near $P \cap Q$.[8] Assume then that e' is in P; that is, both vertices lie in P. If the vertices of e' are on opposite sides of Q, then e' intersects the line $P \cap Q$ in an interior point. If only one vertex of e' is in Q, then e' intersects $P \cap Q$ in that vertex. Now if the interior of e' intersects $P \cap Q$, then we test whether e' intersects in the interior of e or at a vertex of e. To do so, we test whether e' also intersects the lines $P \cap R$ or $P \cap S$, where R and S are planes defining a vertex of e. If both vertices of e' lie on Q, then e and e' are collinear. Here further testing is required to determine how the edges overlap.[8]

A cross-face edge $\bar{e}$ is by definition in P. Its interior intersects $P \cap Q$ if the endpoints of $\bar{e}$ are on opposite sides of Q. Note that these endpoints are either vertices of B or points in the interior of edges of B. We examine the planes defining the vertices of e to determine whether $\bar{e}$ intersects e in its interior or at a vertex. If one or both endpoints of $\bar{e}$ lie on the plane R that defines a vertex v of e, or if both endpoints of $\bar{e}$ lie on the same side of R, then we can determine whether we have an intersection. Otherwise, we test the vertices of e against the plane of the face of B that induces $\bar{e}$.

Experience with the algorithm

As a simple test object for robustness we used the unit-cube example outlined in the beginning of the article. After rotating a unit cube about each principal axis by a small amount, we obtained a second cube with which the first one was intersected. As Laidlaw, Trumbore, and Hughes have pointed out, most modelers break when the angle of rotation is less than 2 degrees.[4] More specifically, above a certain angle β an intersection is correctly computed, and below a certain angle α the objects are so close that the modeler cannot distinguish them. The critical region of failure is thus $\delta = \beta - \alpha$. With the heuristics that they advocate, Laidlaw, Trumbore, and Hughes achieve a critical range $\delta = 0.4$ degree. In our algorithm, the critical range is $\delta = 10^{-10}$ degree, with an ε of 10^{-6}.

A nonregular heptahedron, shown in Figure 8, was rotated through various angles and intersected with itself. As the angle of rotation was decreased, the intersection gracefully converged to the original heptahedron as various vertex-pairs were deemed coincident. With $\varepsilon = 10^{-4}$, final convergence occurred at 1/10,000th of a degree. Other robustness experiments have been reported.[8]

Recall that all objects to be intersected have no features smaller than a given tolerance ε. After a set operation, however, the resulting object may well have such features, and in a postprocessing step we might wish to adjust its surfaces to eliminate them. Such an adjustment must include, among other things, the elimination of short edges, called ε-edges.

Suppose that an ε-edge e exists. If one of the vertices adjacent to e is of degree 3, then e can be removed by tilting the third face incident to that vertex. If the face has at most two vertices of degree 4 or higher, then the face can be rotated about the line through these two high-degree vertices without creating new ε-edges. Otherwise, if the face has at most three high-degree vertices, one of which is of degree 4, then tilting the face about the other two high-degree vertices transfers the ε-edge to a new set of faces, from which it might be removed. ■

Acknowledgments

This work was supported in part by NSF grants CCR 86-19817, DMC 88-07550, and DMC 86-17335; and ONR contracts N00014-86-K-0465, N00014-86-K-0281, and N00014-86-K-0591.

References

1. I.C. Braid, "The Synthesis of Solids Bounded by Many Faces," *CACM*, Vol. 86, No. 4, April 1975, pp. 209-216.
2. M.A. Wesley et al., "A Geometric Modeling System for Automated Mechanical Assembly," *IBM J. Research and Development*, Vol. 24, No. 1, Jan. 1980, pp. 64-74.
3. A.A.G. Requicha and H.B. Voelcker, "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms," *Proc. IEEE*, Vol. 73, No. 1, Jan. 1985, pp. 30-44.
4. D.H. Laidlaw, W.B. Trumbore, and J.F. Hughes, "Constructive Solid Geometry for Polyhedral Objects," *Computer Graphics* (Proc. SIGGRAPH), Vol. 20, No. 4, Aug. 1986, pp. 161-170.
5. M. Mantyla, "Boolean Operations of 2-Manifolds through Vertex Neighbourhood Classification," *ACM Trans. Graphics*, Vol. 5, No. 1, Jan. 1986, pp. 1-29.
6. C.M. Hoffmann, J.E. Hopcroft, and M.S. Karasick, "Towards Implementing Robust Geometric Computations," *4th ACM Symp. Computational Geometry*, ACM, New York, 1988, pp. 106-117.
7. C.M. Hoffmann, *Geometric and Solid Modeling*, Morgan Kaufmann, San Mateo, Calif., 1989, Chaps. 3, 4.
8. M. Karasick, *On the Representation and Manipulation of Rigid Solids*, doctoral dissertation, McGill Univ., Montreal, 1988.
9. J. Bokowski and B. Sturmfels, "On the Coordinatization of Oriented Matroids," *Discrete Computational Geometry*, Vol. 1, 1986, pp. 293-306.
10. K. Kuratowski and A. Mostowski, *Set Theory*, North-Holland, Amsterdam, 1968.
11. A.A.G. Requicha, "Mathematical Models of Rigid Solid Objects," Tech. Memo 28, Production Automation Project, Univ. of Rochester, Rochester, N.Y., 1977.
12. K. Weiler, *Topological Structures for Solid Modeling*, doctoral dissertation, Rensselaer Polytechnic Inst., Troy, N.Y., 1986.
13. P.M. Hanrahan, *Topological Shape Models*, doctoral dissertation, Univ. of Wisconsin, Madison, Wis., 1985.
14. D.P. Dobkin and M.J. Laszlo, "Primitives for the Manipulation of Three-Dimensional Subdivisions," *Proc. 3rd ACM Symp. Computational Geometry*, ACM, New York, 1987, pp. 86-99.
15. K. Mehlhorn, *Data Structures and Algorithms*, Vol. 3, Springer-Verlag, New York, 1984, pp. 185-244.
16. G. Forsythe and C.B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1967, Chap. 8.
17. K. Sugihara and M. Iri, "Geometric Algorithms in Finite-Precision Arithmetic," Research Memo 89-01, Mathematical Eng. and Information Physics Dept., Univ. of Tokyo, 1988.

Christoph M. Hoffmann is a professor in the Computer Science Department at Purdue University. He has also held a faculty position at the University of Waterloo, Canada, and visiting faculty positions at the University of Kiel, West Germany, and Cornell University. His research interests are programming languages and compiler design, algorithms for computational algebra and graph theory, and solid modeling and robotics.

Hoffmann received his BS in mathematics from the University of Hamburg in West Germany, his MS in mathematics from Indiana University, and his PhD in computer science in 1974 from the University of Wisconsin. He is a member of ACM and SIAM.

Hoffmann can be reached at the Department of Computer Sciences, Purdue University, West Lafayette, IN 47907.

John E. Hopcroft is the Joseph C. Ford Professor of computer science and chairman of the Department of Computer Science at Cornell University. He has also served for three years on the faculty of Princeton University. Hopcroft's research interests include the analysis of algorithms, formal languages, automata theory, and graph algorithms. His most recent work has been in the theoretical foundations of solid modeling and robotics.

Hopcroft received his PhD in electrical engineering from Stanford University in 1964. He received the A.M. Turing award in 1986, and is a fellow of the American Academy of Arts and Sciences, the American Association for the Advancement of Science, IEEE, and the National Academy of Engineering.

Hopcroft's address is Department of Computer Sciences, 4130B Upson Hall, Cornell University, Ithaca, NY 14853.

Michael S. Karasick is a research staff member with the modeling systems project at the IBM T.J. Watson Research Center. His research interests include geometric modeling, computational geometry, and parallel algorithms.

Karasick received his BS in computer science from the University of Manitoba in 1981, and MS and PhD degrees in computer science from McGill University in 1983 and 1989. He was a member of the robotics research group at Cornell University's Department of Computer Science from 1985 to 1988. Karasick is a member of ACM and IEEE.

Karasick can be contacted at the IBM T.J. Watson Research Center, Box 218, Yorktown Heights, NY 10598.