TR-87-8

Computing Geometric Boolean Operations by Input Directed Decomposition

by

George Vanecek Jr.
Dana S. Nau

TR-1762 January 29, 1987

Computing Geometric Boolean Operations by Input Directed Decomposition

George Vaněček Jr.
Department of Computer Science

Dana S. Nau
Department of Computer Science and
Institute for Advanced Computer Studies

University of Maryland
College park, MD 20742

ABSTRACT

This paper presents an algorithm to perform regularized Boolean operations on collections of simple polygons. The algorithm accepts two arbitrarily complex collections of disjoint polygons and returns two collections of polygons corresponding to the union and intersection respectively. The algorithm is efficient and generalizes to higher dimensions.

Given two collections of polygons, the algorithm recursively decomposes them into fragments using splitting lines determined from the collections' edges. This approach, which is called *input directed decomposition*, maintains exact representations of objects, and easily classifies an edge into either the union or the intersection set. By the use of edge orientation information, ambiguities caused by objects that touch along an edge are avoided. After edge classification, edge connectivity of polygons is used to allow creation of the polygons belonging to the union and the intersection collections.

Keywords: Computational Geometry, divide-and-conquer, geometric intersections, regular sets, regular decomposition.

This work has been supported in part by the following sources: an NSF Presidential Young Investigator Award to Dana Nau, NSF NSFD CDR-85-00108 to the University of Maryland Systems Research Center, IBM Research, and General Motors Research Laboratories.

1. Introduction

This paper is concerned with the task of performing Boolean operations like union and intersection on geometric objects. The task is performed in such a way that the boundaries of the resultant object can be quickly determined. Many of the existing approaches to this problem are based on the idea of space decomposition, a direct application of the divide-and-conquer paradigm[8]. Most decomposition methods are based on either constructive solid geometry[9] (CSG) or quadtrees[2,11]. When CSG is used, an object is decomposed into Boolean combinations of primitives. In quadtree approaches, the space containing the object is partitioned into disjoint quadrants containing fragments of the object. Both methods have been used to solve the problem of performing binary Boolean operations.[1,5,10,13]

In quadtree methods, decomposition is normally done by recursively dividing a rectangular region into identically shaped rectangular subregions (this is called *regular* decomposition). Some regions can be recursively decomposed into "primitive" regions (which provide exact representations of regions of space) without any problem, but for certain regions (called *nasty* regions by [14]), regular decomposition no matter how deep, will always yield a non-primitive subregion. Thus, regular decomposition methods cannot provide a simple exact representation of the portion of the object that is within the nasty region.[1]

Another problem with quadtree methods based on regular decomposition arises when they are used for performing a Boolean operation such as intersection. When this is done, it is necessary to determine, for each edge of each primitive region of an object, whether it is inside or outside the intersection with the other object. This task is called set membership classification or *edge classification*, and the methods necessary for doing it are rather complex.

This paper describes an algorithm for binary Boolean operations on sets of mutually non-intersecting simple polygons, overcoming the various difficulties mentioned above. The

algorithm uses quadtree decomposition that is not regular. It uses spatial locality by decomposing space into non-overlapping convex regions. But unlike quadtree methods, that utilize regular decomposition, the approach used here decomposes space into non-regular convex regions. Regions are partitioned into two subregions by choosing splitting lines determined by the edges of the polygons to be decomposed.

This approach, called *input directed decomposition*, overcomes the above described problems occurring with quadtree approaches. First, input directed decomposition guarantees that the interactions between a primitive region of one object and the corresponding primitive region of the other object are quite simple—and thus the methods for edge classification can also be simple. Second, nasty regions do not exist—and thus the algorithm always creates an exact representation of the object.

One problem that occurs in the classification of edges is the "On/On" ambiguity, in which it is unclear whether or not a particular edge should be considered part of an object. This occurs when two objects share an edge. One method for resolving this ambiguity is by using neighborhood information[10]. In contrast, our algorithm avoids the "On/On" ambiguity entirely by keeping track of edge orientation. When edge orientation is maintained, the edge in question is represented two collinear edges, and whether or not each of these edges is in the object becomes obvious.

The organization of the rest of the paper is as follows: Sections 2.1-2.3 present definitions of various mathematical entities such as points, directed lines, simple polygons, regions, collections and fragments, and Section 2.4 describes the data structures used by the algorithm to represent these entities. Section 3 discusses the general description of the union/intersection algorithm. Section 3.1 through 3.4 give the details of the decomposition and edge classification. Section 4 outlines the creation of the polygons belonging to the union and the intersection. In section 4.1 the clean up operation, which eliminates unnecessary vertices, is given. Section 5 discusses implementation accuracy and the last section gives closing remarks.

2. Preliminaries

2.1. Points and Lines

A directed line in a two-dimensional Euclidean space is represented by a triple (A, B, C) denoting the equation $Ax + By + C = 0$. If (A, B, C) represents a line L, then the direction of L is the direction of the vector $(-B, A)$. $-L$ is the line represented by $(-A, -B, -C)$; i.e., the line occupying the same location as L, but having the direction $(B, -A)$.

Given two distinct points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, $\text{Line}(P_1, P_2)$ is a triple representing the directed line passing from P_1 towards P_2; i.e., the line passing through P_1 whose direction is $(x_2 - x_1, y_2 - y_1)$. It follows that $\text{Line}(P_1, P_2) = (a/f, b/f, c/f)$, where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_2 y_1 - x_1 y_2$, and $f = \sqrt{a^2 + b^2}$. Thus the directed line is normalized; i.e. $A^2 + B^2 = 1$.

Any point $P = (x, y)$ can be classified as being either to the *left of* L, $P \prec L$, to the *right of* L, $P \succ L$, or *on* L, $P \simeq L$. We can define

$$P \prec L \Leftrightarrow Ax + By + C < 0$$

$$P \simeq L \Leftrightarrow Ax + By + C = 0$$

$$P \succ L \Leftrightarrow Ax + By + C > 0$$

Since the line is normalized, the distance from P to L is

$$\text{Distance}(P, L) = \text{abs}(Ax + By + C).$$

Given a line $L = (A, B, C)$ and a point $P = (x, y)$ on the line, the directed line $\text{Perp}(L, P)$ perpendicular to L through P and pointing to the left of L is

$$\text{Perp}(L, P) = (-B, A, Bx - Ay).$$

Given two lines $L_1 = (A_1, B_1, C_1)$ and $L_2 = (A_2, B_2, C_2)$, their *intersection* is the point

$$\text{Inter}(L_1, L_2) = \begin{cases} (\alpha/\delta, \beta/\delta) & \text{if } \delta \neq 0 \\ \text{undefined} & \text{otherwise,} \end{cases}$$

where $\alpha = (B_1 C_2 - B_2 C_1)$, $\beta = (C_1 A_2 - C_2 A_1)$, and $\delta = (A_1 B_2 - A_2 B_1)$.

2.2. Polygons

We represent a simple (i.e., connected and non self-intersecting) planar polygon Π by a sequence of points $(P_0, P_1, \ldots, P_{n-1})$ of its *vertices*, given in the order in which they occur on the polygon's boundary. If $(P_0, P_1, \ldots, P_{n-1})$ are the vertices of a polygon Π and P_i is one of these vertices, then the *predecessor* and *successor* for P_i are $P_{i\ominus 1}$ and $P_{i\oplus 1}$ respectively, where $i \ominus 1 = (i-1) \bmod n$, and $i \oplus 1 = (i+1) \bmod n$. The *edges* of Π are $\left(\overline{(P_0, P_1)}, \overline{(P_1, P_2)}, \ldots, \overline{(P_{n-1}, P_0)}\right)$ and such that for each edge $\overline{(P_i, P_{i\oplus 1})}$, $P_i \neq P_{i\oplus 1}$. A vertex P_i is a *pseudo vertex* if it its two incident edges are collinear, i.e., if $P_{i\oplus 1} \simeq \text{Line}(P_{i\ominus 1}, P_i)$.

$\Pi = (P_0, P_1, \ldots, P_{n-1})$ is a *clockwise* polygon if $P_0, P_1, \ldots, P_{n-1}$ are given in clockwise order, and is a *counterclockwise* polygon (or a *hole*) otherwise. Π is a clockwise polygon if and only if

$$P_{k\oplus 1} \succ \text{Line}(P_{k\ominus 1}, P_k),$$

where P_k be the rightmost vertex of Π (or the uppermost of the rightmost vertices if there is more than one).

If Π is a polygon whose vertices are $(P_0, P_1, \ldots, P_{n-1})$, then $\neg\Pi$, the *reverse* of Π, is the polygon whose vertices are $(P_{n-1}, P_{n-2}, \ldots, P_0)$. Thus if Π is clockwise, then $\neg\Pi$ is counterclockwise, and vice versa.

2.3. Regions, Fragments and Collections

By a solid we mean a regular subset of two-dimensional Euclidean space (for a definition of a regular set, see [10]). If S is a solid and its boundary is a set of disjoint simple polygons $C = \{\Pi_1, \ldots, \Pi_2\}$, then C is called a *collection.*

A *region* is the convex intersection R of zero or more half-spaces, each of which may be either open or closed. Let R be a region, and let C' be the collection formed from C by

introducing pseudo vertices wherever edges in C cross boundaries of R. Then the *fragment* $f = \{e_1, \ldots, e_m\}$ of C in R is the set of all edges e_i of C' whose interiors are in R.

Let C_1 and C_2 be collections. Whenever we speak of the union or intersection of C_1 and C_2, we mean the regularized union and intersection (commonly denoted by $C_1 \cup^* C_2$ and $C_1 \cap^* C_2$) [10,13]. These operations are similar to ordinary union and intersection, except that they don't generate isolated vertices (i.e. vertices with no incident edge) and dangling edges (i.e. edges that do not form a boundary of a solid).

2.4. Data Structures

This section describes the basic data structures used in the union/intersection algorithm.

Each polygon Π is represented by a doubly linked circular list. Each *node* N of the polygon list corresponds to a vertex $p(N)$ of Π—or equivalently, it corresponds to the edge $e(N)$ with its initial vertex $p(N)$. Each node N contains a pointer to nodes $N_\ominus$ and $N_\oplus$ representing the predecessor and successor nodes of $p(N)$ in Π. The line equation triple, $l(N)$, for edge $e(N)$ is also stored in N.

Collection C is represented as a list of polygons. If Π is a polygon in a collection C, then the program creates fragments of C in such a way that no edge of Π is in more than one fragment of C. Thus, it is possible to represent fragments by placing additional links in the nodes used to represent polygons. A fragment f is represented by a singly linked circular list F called a *fragment list.* F contains one node for each edge in f, and each node N of F is also a node of one of the polygon lists mentioned above. Thus N has three links: one pointing to $N_\ominus$, one pointing to $N_\oplus$, and one pointing to another node in the same fragment. Figure 1 shows an collection and the corresponding data structure. In

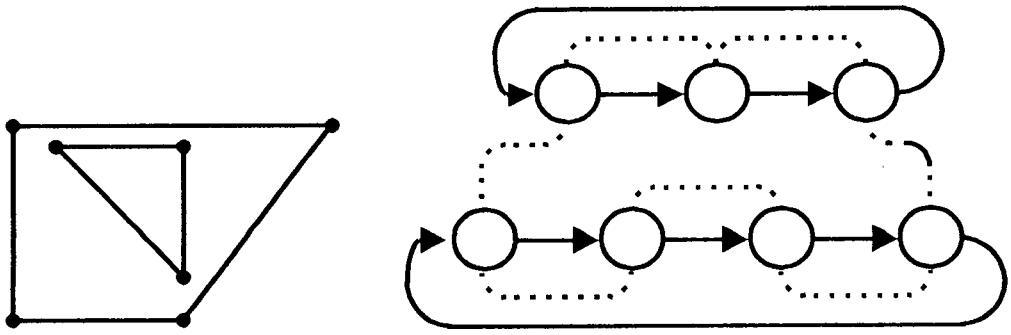


Fig. 1. A collection and its data structure.

the data structure, the polygon links are shown as solid arrows and the fragment links are shown as dotted links.

By convention in this paper, whenever f_i is a fragment, F_i is the corresponding fragment list, and vice versa. In our algorithms, if F_i is a fragment list and N is a node in the region R containing f_i, then the assignment $F_i := F_i \cup \{N\}$ assigns to F_i a representation of $f_i \cup \{e(N)\}$. This is implemented by inserting N into the list F_i. Similarly, the assignment statement $F_i := F_i - \{N\}$ assigns to F_i a representation of the difference $f_i - \{e(N)\}$. This is implemented by removing N from the list F_i. In practice, the node N to be removed from F_i will always be the first node in F_i; thus, no traversal of F_i will be necessary.

3. The Union/Intersection Algorithm

This section discusses an algorithm that computes both the union and the intersection of two collections C_1 and C_2. In order to compute the collections $C_1 \cup^* C_2$ and $C_1 \cap^* C_2$, the algorithm interprets C_1 and C_2 as fragments of some arbitrary enclosing region R. These fragments are then recursively decomposed, and their edges are classified (what this means is discussed below). The classified edges from both collections C_1 and C_2 are then relinked to form the collection $C_1 \cup^* C_2$ and $C_1 \cap^* C_2$. Finally, all remaining pseudo vertices are removed.

Let f_1 and f_2 be fragments of C_1 and C_2 respectively, sharing the same region. If e is an edge of f_1 or f_2, then edge classification of e consists of determining which of the fragments $f_\cup$, $f_\cap$, or $f_\emptyset$ contains e. These fragments, called classification fragments, contain the edges appearing in $C_1 \cup^* C_2$, $C_1 \cap^* C_2$, or neither of them.

Edge classification is done by transforming the fragment lists F_1 and F_2 into three fragment lists $F_\cup$, $F_\cap$, and $F_\emptyset$ called *classification lists*, which represent $f_\cup$, $f_\cap$, and $f_\emptyset$, respectively. Initially, the classification lists are all empty, but as each node of F_1 and F_2 is classified it is removed from F_1 or F_2 and put into one of the classification lists. During this process, connectivity of edges within a polygon is maintained at all times, and polygon boundaries are never broken.

Each edge must be classified. This is true regardless of whether the union alone or the intersection alone is desired. To create the connected polygons for the union, requires the relinking of the edges in $f_\cup$. As a biproduct of relinking the union edges, the connected polygons for the intersection remain. Therefore, the algorithm generates as output both the union and the intersection collections. Analogous to the computation of $C_1 \cup^* C_2$ and $C_1 \cap^* C_2$, a single operation suffices to compute both $C_1 -^* C_2$ and $C_2 -^* C_1$. Using the Union/Intersection algorithm, $C_1 -^* C_2$ could be computed as $\neg(\neg C_1 \cup^* C_2)$, where $\neg C = \{\neg \Pi \mid \Pi \in C\}$; $C_2 -^* C_1$ is simultaneously computed by $\neg C_1 \cap^* C_2$.

The procedure for union/intersection is shown below. The procedures called by it are defined in sections 3.1 through 3.4.

Function Union_Intersect(Collection C_1, C_2)
begin
Let fragment F_1 contain the edges of C_1
Let fragment F_2 contain the edges of C_2
$(F_\cup, F_\cap, F_\emptyset)$:= Decompose(F_1, F_2) (* *Classify all edges* *)
$C_\cup$:= MakePolygons($F_\cup$) (* *Relink the Union polygons* *)
$C_\cap$:= MakePolygons($F_\cap$) (* *Relink the Intersection polygons* *)
CleanUp($C_\cup$) (* *Remove pseudo vertices from* $C_\cap$ *)
CleanUp($C_\cap$) (* *Remove pseudo vertices from* $C_\cap$ *)

```
  Dispose(F_∅)                          (* Discard all edges in F_∅        *)
  return(C_∪, C_∩)                      (* return C1 ∪* C2 and C1 ∩* C2    *)
end(* Union_Intersect *)
```

3.1. Fragment Decomposition

The *Decompose* procedure, which controls the input directed decomposition, is based on divide-and-conquer and is an extension of the one-dimensional Quicksort[4]. *Decompose* accepts fragment lists representing two nonempty fragments f_1 and f_2. If each fragment contains exactly one edge and the two edges are collinear with common endpoints, then edge classification is performed on them immediately, as described in the next paragraph. Otherwise, the fragments must be decomposed further. In this case, *Decompose* calls the *FindSplitLine* procedure to find a splitting line L, and partitions both f_1 and f_2 into left subfragments f_{1l} and f_{2l} and right subfragments f_{1r} and f_{2r}, respectively (see Section 3.3). If one of the subfragments is empty, then all the edges of the other fragment are classified immediately by the *ClassifyFragment* procedure. Otherwise, *Decompose* calls itself recursively on the corresponding subfragments.

```
Function Decompose(Fragment F1, F2)
begin
  If F1 = {N} and F2 = {M} then          (* One edge in each fragment *)
    (* Check for collinear edges with common endpoints *)
    If p(N) = p(M) and p(N⊕) = p(M⊕) then
      return({N}, {M}, ∅)                (* Edges have same direction *)
    else If p(N) = p(M⊕) and p(N⊕) = p(M) then
      return(∅, ∅, {N, M})               (* Edges have opposite directions *)
  Line L := FindSplitLine(F1, F2)
  (F1l, F1r, G1l, G1r) := SplitFragment(F1, L)
  (F2l, F2r, G2l, G2r) := SplitFragment(F2, L)
  If F1l = ∅ then
    (F∪l, F∩l, F∅l) := ClassifyFragment(F2l, G1r)
  else If F2l = ∅ then
    (F∪l, F∩l, F∅l) := ClassifyFragment(F1l, G2r)
```

else
$(F_{\cup l},\ F_{\cap l},\ F_{\emptyset l}) :=$ Decompose$(F_{1l},\ F_{2l})$
If $F_{2r} = \emptyset$ **then**
$(F_{\cup r},\ F_{\cap r},\ F_{\emptyset r}) :=$ ClassifyFragment$(F_{1r},\ G_{2l})$
else If $F_{1r} = \emptyset$ **then**
$(F_{\cup r},\ F_{\cap r},\ F_{\emptyset r}) :=$ ClassifyFragment$(F_{2r},\ G_{1l})$
else
$(F_{\cup r},\ F_{\cap r},\ F_{\emptyset r}) :=$ Decompose$(F_{1r},\ F_{2r})$
return$(F_{\cup l} \cup F_{\cup r},\ F_{\cap l} \cup F_{\cap r},\ F_{\emptyset l} \cup F_{\emptyset r})$
end$(*$ *Decompose* $*)$

If each fragment contains exactly one edge and the two edges are collinear with common endpoints, then let the N and M be the nodes representing these edges. For classifying $e(N)$ and $e(M)$, there are two cases to consider. The first case is where the collinear edges have the same direction (i.e., $l(N) = l(M)$). This will occur when two solids are on the same side of a common boundary. In this case, one of the edges, say $e(N)$, must belong to the union and the other, $e(M)$, must belong to the intersection. The second case is where the collinear edges have opposite directions (i.e., $l(N) = -l(M)$). This will occur when two solids touch at an edge. In this case, both edges fall in the interior of the union and do not belong to any boundary in either $F_{\cup}$ nor $F_{\cap}$, so they are added to $F_{\emptyset}$.

3.2. Selecting A Splitting Line

If edge classification fails, then the fragments f_1 and f_2 are decomposed simultaneously by decomposing the region of space R that contains f_1 and f_2. This is normally done by intersecting R with the half-spaces H_l and H_r lying on either side of a line called a *splitting line*, thus producing a *left* subregion R_l and a *right* subregion R_r. We assume that H_l is an open half-space and H_r is a closed half-space. Thus, any edges of f_1 and f_2 lying on the splitting line will be in R_r rather than R_l.

There are two different ways in which the splitting line might be chosen. The approach used in previous geometric decomposition algorithms is regular decomposition, in which

regions are recursively divided into smaller regions of identical shape. In such cases, the splitting line is a vertical or horizontal bisector. For example, this is done in quadtrees, where a square region is divided into four equal quadrants, and in bintrees, where a rectangular region is bisected into two smaller rectangles either horizontally or vertically.

In contrast, our algorithm uses non-regular decomposition to divide a region R into convex subregions R_l and R_r that satisfy two desirable properties. These two properties are discussed below.

Property 1. R_l and R_r must both have nonzero area. This is because representational inconsistencies arise when trying to handle edges that cross regions of zero area. This property will be maintained if and only if a splitting line is not collinear with an edge of F_1 or F_2 that is on the boundary of R.

Property 2. R_l and R_r must each contain either a nonempty fragment of f_1 or a nonempty fragment of f_2. Without this property, a region could be split repeatedly without decomposing the fragments it contains. Indeed, by analogy to Quicksort, it is desirable in the union/intersection algorithm that R_l and R_r each contain substantial portions of both f_1 and f_2.

The choice of a splitting line is made by the function *FindSplitLine* shown below. *FindSplitLine* is given two fragment lists F_1 and F_2 occupying some region R. From these lists *FindSplitLine* selects two edges $e(N)$ and $e(M)$ and returns a splitting line L that does not lie on the boundary of R, and which satisfies Properties 1 and 2.

```
Function FindSplitLine(Fragment F1, F2) : Line
begin
 (* Select two edges for determining the separation line *)
 If |F1| = 1 and |F2| = 1 then
    Let N and M be nodes of F1 and F2 respectively
 else If |F1| > |F2| then
    Let N and M be two distinct nodes of F1
 else
    Let N and M be two distinct nodes of F2
```

(* *Find the splitting line using nodes* N *and* M *)*
If $p(M) \simeq l(N)$ **and** $p(M_\oplus) \simeq l(N)$ **then begin** (* *edges are collinear* *)
Let P be an endpoint of one of the edges so that the other edge is split
return(Perp($l(N)$, P)) (* *return a perpendicular to* $l(N)$ *)
end else If $e(N)$ is not on a region boundary **then**
return(**If** $p(M) \prec l(N)$ **or** $p(M_\oplus) \prec l(N)$ **then** $l(N)$ **else** $-l(N)$)
else If $e(M)$ is not on a region boundary **then**
return(**If** $p(N) \prec l(M)$ **or** $p(N_\oplus) \prec l(M)$ **then** $l(M)$ **else** $-l(M)$)
(* *Neither* $l(N)$ *nor* $l(M)$ *can be used. Compute a new splitting line* *)*
If ($l(N)$ and $l(M)$ are oppositely directed) **then**
return(Line(midpoint($p(N)$, $p(M_\oplus)$), midpoint($p(M)$, $p(N_\oplus)$)))
else
return(Line(midpoint($p(N)$, $p(M)$), midpoint($p(N_\oplus)$, $p(M_\oplus)$)))
end(* *Find Splitting Line* *)*

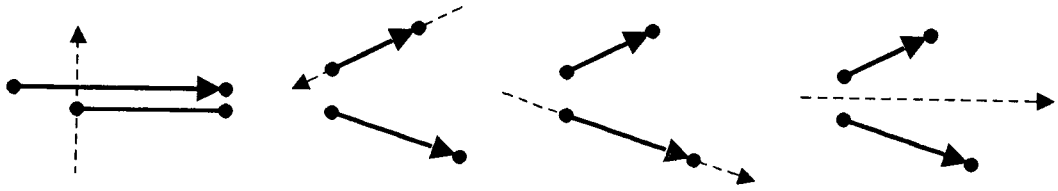


Fig. 2. Examples of Choosing a Splitting Line. L is the dotted line.

If f_1 and f_2 contain edges that are not on the boundary of R, $e(N)$ and $e(M)$ should be selected from among these edges. To eliminate searching for these edges, the fragment lists are ordered such that all edges that are not on the boundary precede all edges that are on the boundary.

When selecting $e(N)$ and $e(M)$, it is necessary to insure that they are not collinear with common endpoints. Since there is no way to separate such edges, the algorithm would not terminate. When possible, $e(N)$ and $e(M)$ are both taken from the same fragment, since a single fragment contains no collinear edges. The only case where this is not possible is where each fragment contains one edge each. In this case, the two edges cannot be both collinear and have common endpoints, since that condition was checked in function *Decompose*.

3.3. Splitting Fragments

Once *FindSplitLine* returns a splitting line L to *Decompose*, *Decompose* calls the *SplitFragment* procedure twice, to split the fragment lists F_1 and F_2.

Given a splitting line L and a fragment list for a fragment f, *SplitFragment* partitions f into the subfragments f_l and f_r defined as follows: f_l contains all edges of f which are to the left of L, and the left-hand portions of those edges which intersect L. f_r contains all edges of f which are to the right of L, all edges of f which lie on L, and the right-hand portions of those edges which intersect L. Mathematically, f_l and f_r are the following fragments:

$$\begin{aligned} f_r = \; & \left\{ \overline{(P,Q)} \in f \mid P \succeq L \wedge Q \succeq L \right\} \cup \\ & \left\{ \left(\overline{\mathrm{Inter}(\mathrm{Line}(P,Q),L),Q} \right) \mid \overline{(P,Q)} \in f \wedge P \prec L \wedge Q \succ L \right\} \cup \\ & \left\{ \left(\overline{P,\mathrm{Inter}(\mathrm{Line}(P,Q),L)} \right) \mid \overline{(P,Q)} \in f \wedge Q \prec L \wedge P \succ L \right\} \end{aligned}$$

$$\begin{aligned} f_l = \; & \left\{ \overline{(P,Q)} \in f \mid P \preceq L \wedge Q \prec L \right\} \cup \\ & \left\{ \overline{(P,Q)} \in f \mid Q \preceq L \wedge P \prec L \right\} \cup \\ & \left\{ \left(\overline{\mathrm{Inter}(\mathrm{Line}(P,Q),L),Q} \right) \mid \overline{(P,Q)} \in f \wedge P \succ L \wedge Q \prec L \right\} \cup \\ & \left\{ \left(\overline{P,\mathrm{Inter}(\mathrm{Line}(P,Q),L)} \right) \mid \overline{(P,Q)} \in f \wedge P \prec L \wedge Q \succ L \right\} \end{aligned}$$

Given a splitting line L and a fragment list for a fragment f, *SplitFragment* returns the new subfragments f_l and f_r created from partitioning f (see figure 3). For each of the subfragments f_l and f_r, if the subfragment is nonempty, then *SplitFragment* also selects for that subfragment a *classification vertex*, which is closest vertex to the splitting line. If several vertices having different coordinates are equally close to the splitting line, one is chosen arbitrarily. This vertex, which is determined by the function *GetClosestVertex*, is used by the *ClassifyFragment* function. For this vertex, a set of all nodes at that vertex is maintained (see section 3.5).

SplitFragment calls the function *SplitEdge* once for each edge of F that intersects L. *SplitEdge* takes a node N representing the edge that crosses L, and introduces a pseudo vertex (see Section 2.2) that splits $e(N)$ at L into two edges e_l and e_r (e_l and e_r are to the left and right of L, respectively). Thus, in the polygon list containing N, N is replaced by nodes N_l and N_r such that $e(N_l) = e_l$ and $e(N_r) = e_r$, and returns the pair (N_l, N_r).

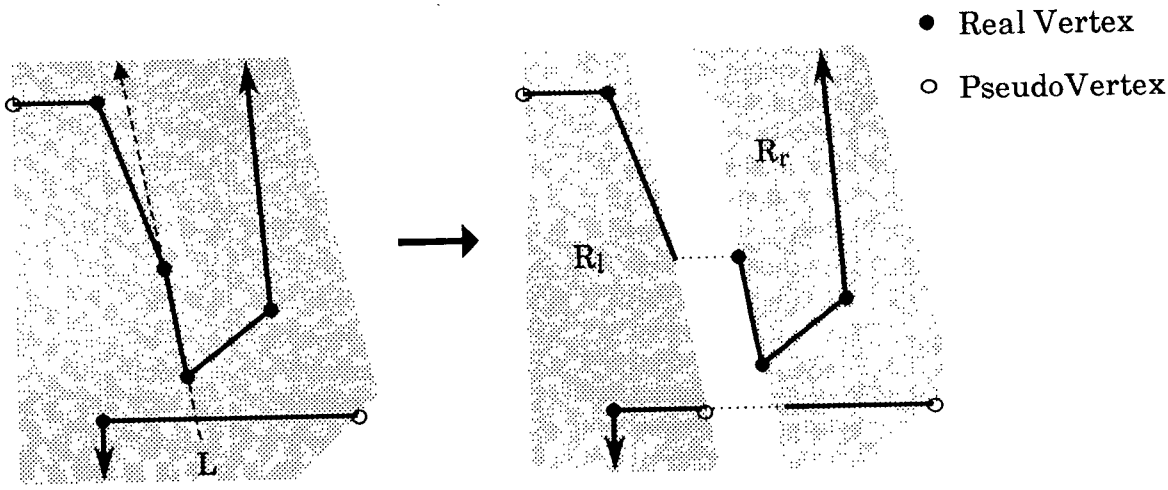


Fig. 3. Fragment Splitting.

Function SplitFragment(Fragment F; Line L)
begin
  Fragment $F_l := F_r := \emptyset$ (* *Clear the Left and Right subfragments* *)
  Set of Nodes $G_l := G_r := \emptyset$ (* *Initialize the Classification vertex set* *)
  for every node N in F **do begin**
    $F := F - \{N\}$ (* *Remove edge $e(N)$ from fragment F* *)
    If $p(N) \succeq L$ **and** $p(N_\oplus) \succeq L$ **then begin** (* *$e(N)$ is on or right of L* *)
      (* *N is on the boundary of the region* *)
      Mark N **If** $p(N) \simeq L$ **and** $p(N_\oplus) \simeq L$
      GetClosestVertex(L, G_r, N)
      $F_r := F_r \cup \{N\}$ (* *Place $e(N)$ in right subregion* *)
    end else If $p(N) \preceq L$ **and** $p(N_\oplus) \preceq L$ **then begin**
      (* *$e(N)$ is to the left of L* *)
      GetClosestVertex(L, G_l, N)
      $F_l := F_l \cup \{N\}$ (* *Place $e(N)$ in left subregion* *)
    end else begin (* *Edge(N) crosses L* *)

```
      (N_l, N_r) := SeparateEdges(N, L)          (* Split e(N) by L *)
      F_r := F_r ∪ {N_r}
      F_l := F_l ∪ {N_l}
    end
  end
  return(F_l, F_r, G_l, G_r)
end(* Split Fragment *)
```

In the *GetClosestVertex* procedure, G represents the current "best guess" for the vertex closest to the splitting line. Since a fragment list may contain several nodes for a single vertex—and since *GetClosestVertex* must keep track of all of these nodes—G is a set which may contain more than one node. *GetClosestVertex* chooses the vertex closest to L, and if necessary, updates G by comparing the vertex represented by G with the initial and terminal vertices $p(N)$, and $p(N_\oplus)$ of the edge $e(N)$.

```
Procedure GetClosestVertex(Line L; Set of Nodes G; Node N)
begin
  Point P := If G = ∅ then (∞, ∞) else p(G)
  (* Check initial vertex of edge e(N) *)
  If Distance(p(N), L) < Distance(P, L) then
    G := {N}                                  (* p(N) is closer *)
  else If P = p(N) then
    G := G ∪ {N}
  (* Check terminal vertex of edge e(N) *)
  If Distance(p(N_⊕), L) < Distance(P, L) then
    G := {N_⊕}                                (* p(N_⊕) is closer *)
  else If P = p(N_⊕) then
    G := G ∪ {N_⊕}
  If Distance(P, L) = Distance(p(N), L) = Distance(p(N_⊕), L) then
    G := G ∪ {N}                              (* Edge e(N) is parallel to L *)
end(* Get Classification Vertex *)
```

3.4. Empty Fragment Classification

Suppose that *Decompose* is decomposing two fragments f_1 and f_2. Then both are split into their corresponding left subfragments f_{1l} and f_{2l} and their right subfragments f_{1r} and f_{2r} about some splitting line L. Suppose that one of these subfragments is empty (without loss of generality, we may assume it is f_{1l}). That is, the nodes of F_1 fall entirely into F_{1r}, and some of these becomes the nodes at the classification vertex G_{1r}.

Let S be the solid having the collection C_1 as its boundary. Let f_1 be a fragment of C_1. If f_{1l} is empty, then the region R_l is either totally in the interior or totally in the exterior of S. If it is in the interior then all the edges of f_{2l} can be added to $F_\cap$. Otherwise, all the edges of f_{2l} can be added to $F_\cup$, without further decomposition of f_{2l}. This is done by the *ClassifyFragment* function shown below. To determine which of these cases hold, *ClassifyFragment* checks a point P in the left subregion against the edges incident to the classification vertex nodes in the set G_{1r} (see figure 4).

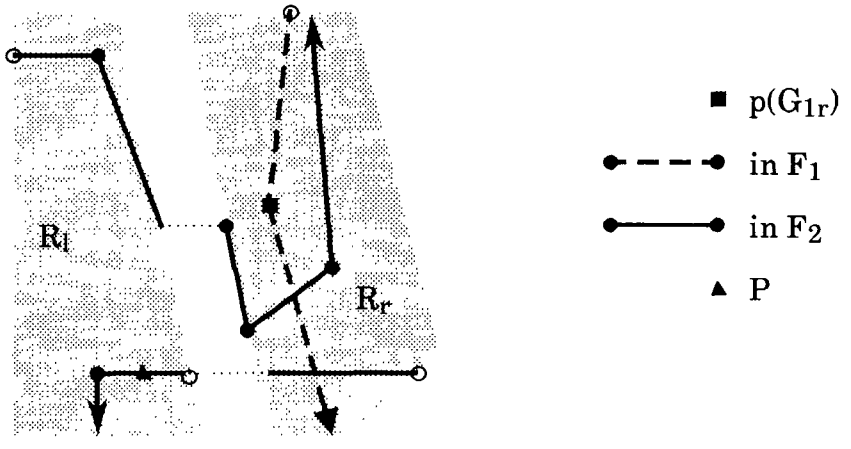


Fig. 4. Classification of the Fragment F_{2l}.

To determine whether R_l is in the interior or exterior, it suffices to examine the relationship between a single point of R_l and a single edge of F_{1r}. Let P be any point in R_l. Among the edges of F_{1r} that are incident on G_{1r}, let E be the one having the smallest angle to the line $Line(p(G_{1r}), P)$. Then P (and thus R_l) is inside S if and only if P is to the right of E.

Since the boundary of R_l is not explicitly stored, suitable candidates for P can only be found by examining the interiors of the edges contained in fragment f_{2l}. As any point will do, a point is selected that lies on an edge of f_{2l} and that does not correspond to $p(G_{1r})$, the classification vertex. *ClassifyFragment* selects this point, checks it agains $p(G_{1r})$, and adds the edges of f_{2l} to $F_\cup$ or $F_\cap$.

```
Function ClassifyFragment(Fragment F; Set of Nodes G)
begin
  Node M := any node of F
  Point P := any point on e(M) such that P ≠ p(G)
  Find edge e(N) incident to a vertex node in G such that
    the angle between e(N) and Line(p(G), P) is minimum
  If P ≻ l(N) then
    return(∅, F, ∅)                    (* Place all edges of F in F∩ *)
  else
    return(F, ∅, ∅)                    (* Place all edges of F in F∪ *)
end(* Classify Fragment *)
```

4. Creating Polygons by Resolving Transitions

After the original fragment lists f_1 and f_2 have been decomposed and all their edges have been classified, the classification lists $F_\cup$ and $F_\cap$ contain the proper edges of the $C_1 \cup^* C_2$ and $C_1 \cap^* C_2$ respectively, and $F_\emptyset$ contains the edges that do not belong to either the union or the intersection. Using the classification list $F_\cup$ and $F_\cap$, it is possible to create polygon lists representing the polygons bounding $C_1 \cup^* C_2$ and $C_1 \cap^* C_2$. This is done by relinking the nodes in the polygon lists representing C_1 and C_2. The relinking operation

is a destructive operation since all the edges of both C_1 and C_2 are used to construct the polygons in $C_1 \cup^* C_2$ and $C_1 \cap^* C_2$.

The way that the algorithm determines that a node N needs to be relinked is if that node is a *transition node*, i.e., a node whose incident edges are in different classification lists. This occurs if and only if $f(N) \neq f(N_\ominus)$, where $f(N)$ is defined to be the classification fragment containing $e(N)$.

If $N \in F_1$ is a transition node, then there must be another transition node $M \in F_2$ such that $p(N) = p(M)$ and $f(N) = f(M_\ominus)$. Thus, if the polygon list links between N and M are interchanged, then N will no longer be a transition node.

In the simplest case, M will also no longer be a transition node after this interchange operation. There may be more complicated cases in which some number $n > 2$ of transition nodes represent identical vertices. In such cases, all n nodes can be made non-transition nodes in $n - 1$ relinkings or less. At each point where a vertex occurs, the algorithm maintains a set of all nodes with that vertex. To find the corresponding transition node M such that $p(M) = p(N)$, it is not necessary to examine all the nodes in all the classification lists, but only the nodes in this set. When a transition node is relinked it is removed from this set. In figure 4, nodes belonging to a single set are shown circled.

The *MakePolygons* procedure traces around each polygon in F, finds all transition nodes M such that $p(M) = p(N)$, and relinks them. After each polygon has been traced, the polygon is added to the collection being constructed. *MakePolygons* is first invoked on $F_\cup$. Once its nodes have been relink, *MakePolygons* is invoked on $F_\cap$. The nodes in $F_\emptyset$ do not have to be checked explicitly, since if $F_\cup$ and $F_\cap$ contain no transition nodes, $F_\emptyset$ also cannot contain transition nodes. Since all edges in $f_\emptyset$ fall in the interior of $C_1 \cup^* C_2$, $F_\emptyset$ may be discarded.

```
Function MakePolygons(Fragment F) : Collection
begin
  Collection C := ∅
  While F ≠ ∅ do begin              (* For each polygon Π in F do... *)
    Let Node N ∈ F and let Π be its polygon    (* Pick any edge e(N) *)
    While N ∈ F do begin              (* Trace around the polygon *)
      F := F − {N}                    (* Remove edge e(N) from F *)
      If f(N⊕) ≠ f(N) then begin      (* N⊕ is a transition node *)
        Find another transition Node M such that
          p(M) = p(N⊕) and f(M) = f(N)
        Relink N⊕ and M    (* Make edge e(M) be the next edge on Π *)
      end
      N := N⊕                         (* Advance to next edge on boundary *)
    end
    C := C ∪ Π                        (* Add polygon to the collection *)
  end
  return(C)                           (* Return the collection of polygons *)
end(* MakePolygons *)
```

Figure 5 shows the two fragments $F_{\cup}$ and $F_{\cap}$ before and after invoking the procedure *MakePolygons*. Edges of $F_{\cup}$ are shown as solid lines and the edges of $F_{\cap}$ are shown as dotted lines. Collinear edges are offset slightly to show their connectivity.

4.1. Pseudo Vertices Cleanup

In the process of fragment splitting, edges are split and pseudo vertices are introduced. Many of these vertices are transition vertices and will be relinked. However, some pseudo vertices remain and must be removed in order to prevent proliferation of redundant pseudo vertices in following operations.

Removing pseudo vertices from a collection is a simple task of removing each node N for which $e(N)$ is collinear with $e(N_{\ominus})$. The *CleanUp* procedure performs this operation.

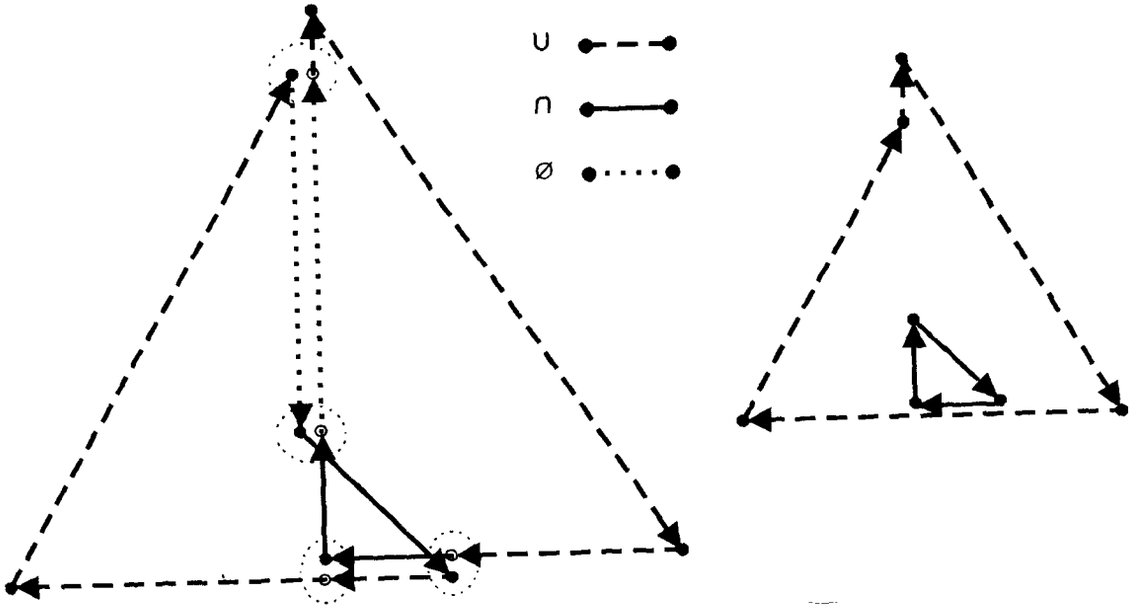


Fig. 5. Before and after MakePolygons.

```
Procedure CleanUp(Collection C)
begin
  for every polygon Π ∈ C do
    for every vertex p(N) in Π such that p(N_⊖) ≃ l(N) do begin
      (* N is a pseudo vertex *)
      Unlink p(N) from Π                    (* Link up N_⊖ with N_⊕ *)
      Dispose(N)                            (* Discard Pseudo Vertex *)
    end
end(* CleanUp *)
```

A splitting line is usually collinear with some edge e. Any edge that crosses the splitting line without crossing e will cause a pseudo vertex to be created—but the number of cases where this occurs is usually small. Any edge that crosses the splitting line by crossing e will cause two pseudo vertices to be created—but these pseudo vertices are transition vertices. Thus, during relinking, these vertices will become actual vertices rather than pseudo vertices.

The case in which the maximum number of transition vertices is created before relinking consists of two collections C_1 and C_2 of one polygon and n edges each, in which each

edge of C_1 crosses all n of the edges of C_2. In this case, there will be approximately $2n^2$ pseudo vertices created. However, there will be $n^2/4$ polygons of four vertices each in the intersection and $n^2/4$ polygons in the union. During relinking, all of the pseudo vertices except one becomes an actual vertex.

Experimentally, the number of pseudo vertices remaining after relinking has been found to be generally small, averaging around one-third the total number of vertices in the original collections.

5. Implementation Accuracy

A pathological manifestation that occurs when implementing any solid modeling algorithm using floating point is the *epsilon* problem. We store all numbers in double precision that yields 15 digits of accuracy, but set the equality check at only 9 digits of accuracy; (i.e. $\epsilon = 10^{-9}$). Computations that are sensitive to floating point errors must be normalized using the ϵ.

The point/line predicates ($\prec$, $\simeq$, $\succ$, ...) are coded using Eval^ϵ, which is Eval^ϵ, which is

$$\text{Eval}^\epsilon(P, L) = \begin{cases} 0 & \text{if abs}(Ax + By + C) \leq \epsilon \\ Ax + By + C & \text{otherwise.} \end{cases}$$

Point equality $P_1 = P_2$ is

$$P_1 =_\epsilon P_2 \Leftrightarrow |x_1 - x_2| + |y_1 - y_2| \leq \epsilon.$$

One important consideration in the implementation is to insure that the predicates agree. That is, since each vertex P is on at least two lines L_1 and L_2, then for any other point Q the implementation must guarantee that $P = Q$ iff $Q \simeq L_1 \wedge Q \simeq L_2$. A disagreement between predicates like $P = Q$ and $Q \simeq L$ will result in erroneous edge classification. The use of normalized line equations and careful computation of the point of intersection of two lines prevents the predicates from disagreeing.

6. Conclusion

Efficient algorithms for solving general geometric intersection problems are becoming an important part of many areas such as computational geometry[7], and computer graphics. The development of this algorithm stemmed from the goal of improving on the complexity of existing methods by taking advantage of the well understood divide-and-conquer paradigm. We conjecture that the complexity is $O(\max(n \log n, m))$, where n is the number of edges in both C_1 and C_2 and m is the number of edges in both $C_1 \cup^* C_2$ and $C_1 \cap^* C_2$.[12] This would be more efficient than existing intersection algorithms that operate in time in excess of $O(n^2)$. Current work is being conducted to show the algorithm's correctness and establish its average time complexity. The complexity will be verified statistically by automatically generating collections of polygons and collecting parameters such as the number of times edges are examined.

This algorithm has been successfully implemented in C and runs under Unix on a Sun-3 workstation. The algorithm is implemented as a part of a two dimensional modeling package including a user and a graphic interface that uses the SunCore graphics package. The entire system runs under the Sun window environment.

Our AI group at the University of Maryland is developing a system called SIPS to perform an automated manufacturing task known as generative process planning[6]. The future success of this project will necessitate the use of a solid modeler which can efficiently solve intersection problems on three dimensional solids. The two dimensional algorithm described in this paper can be directly extended to three dimensional polyhedra by using splitting planes instead of splitting lines.[3] We intend to use this approach as the basis for a solid modeler based on boundary representation and input directed decomposition.

Acknowledgements

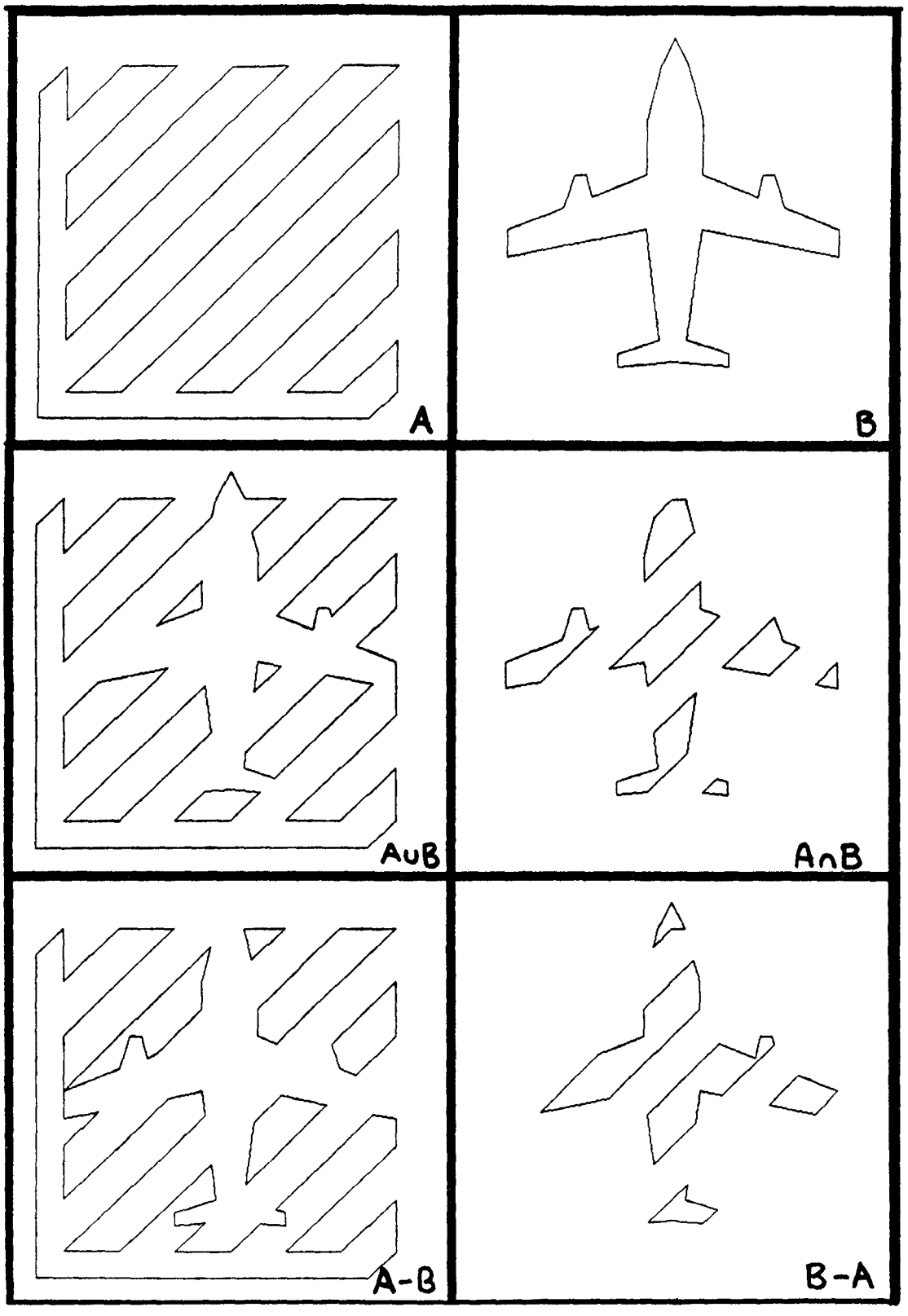
The authors wish to thank Dave Mount, Gary Knott, and Hanan Samet for their help, suggestions, and incouragements in completing this paper.

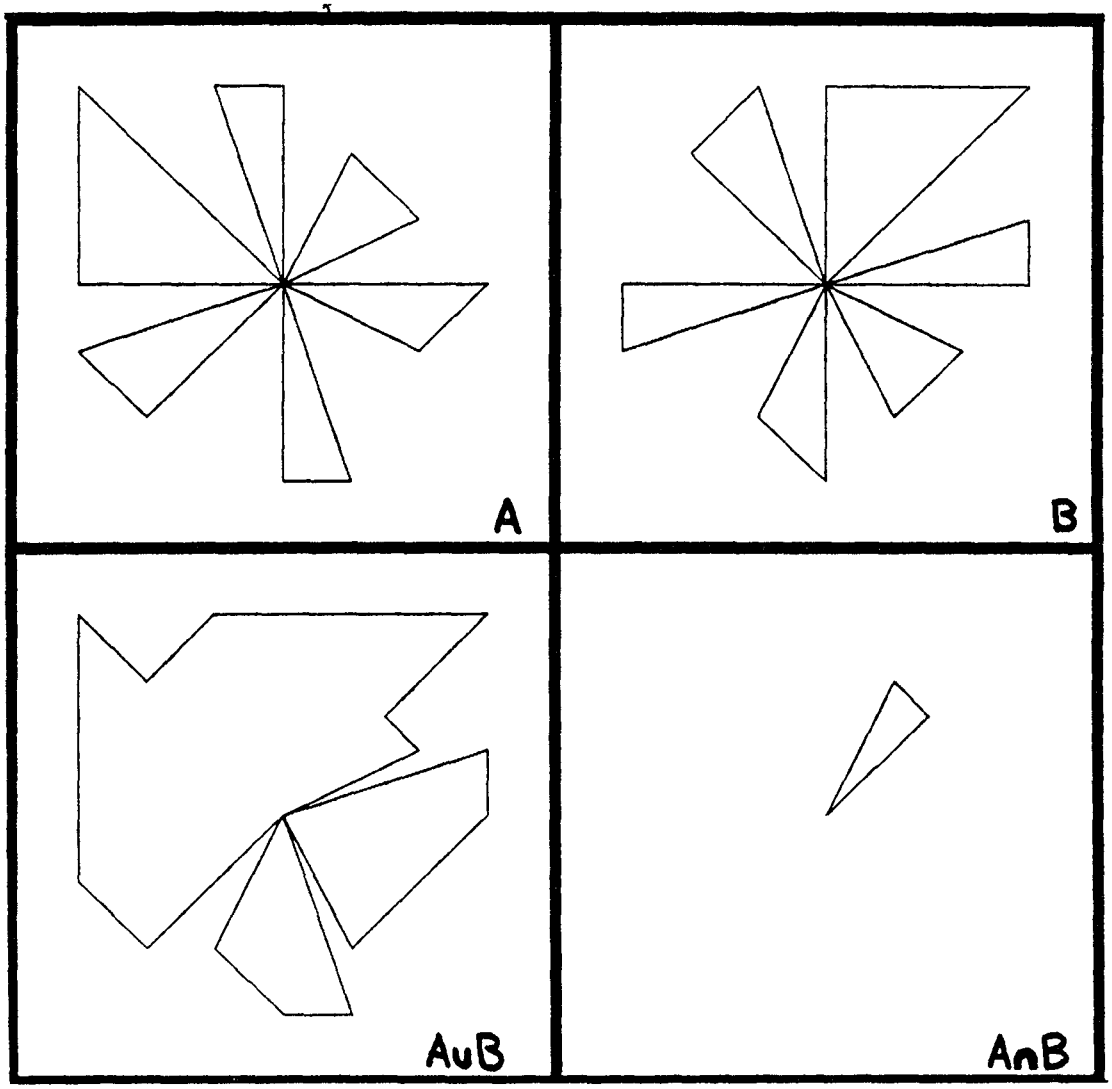
7. Bibliography

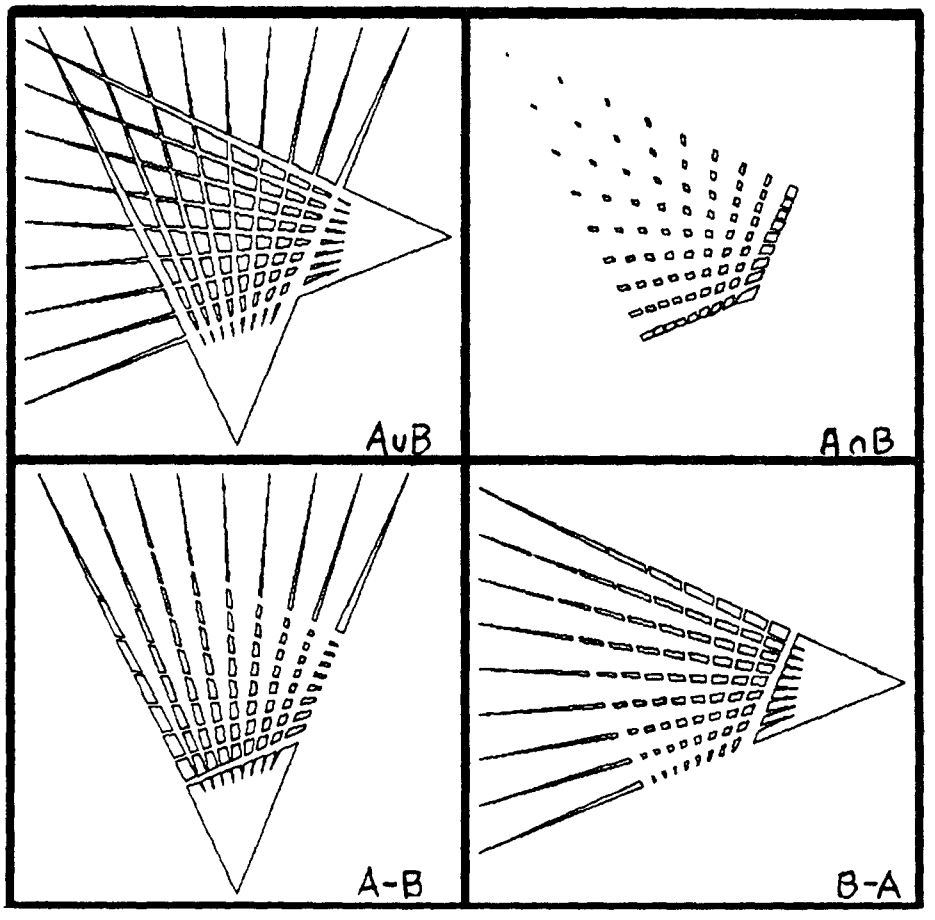
1. Ayala D., Brunet P., Juan R., and Navazo I., Object Representation by Means of Nonominimal Division Quadtrees and Octrees, *ACM Trans. on Graphics* **4**, 1 (Jan 1985), 41-59.
2. Bentley J. L., Multidimensional divide and conquor, *Comm. ACM* **23**, 4, (April 1980) 214-229.
3. Fuchs, H., Kedem, M. Z., and Naylor, F. B., On visible surface generation by a priori tree structres, *Conf. Proc. of Siggraph '80* **14**, 3, (1980), 124-133.
4. Hoare, C. A. R., Quicksort, *The Computer Journal* **5**, (1962), 10-15.
5. Hunter, G. M., Operations on images using quad trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **1**, 2 (April 1979), 145-153.
6. Nau, D. S., and Gray, M., Hierarchical Knowledge Clustering: A New Representation for Problem-Solving Knowledge, *in* J. Hendler, *Expert Systems: The User Interface*, Ablex, 1987, to appear.
7. Preparata, Franco P., and Shamos, Michael Ian, Computational Geometry, An Introduction, Springer-Verlag, 1985.
8. Purdom P. W. Jr., and Brown C. A., *The Analysis of Algorithms*, Holt, Rinehart and Winston, 1985.
9. Requicha, A. A. G., and Voelcker, H. B., Constructive solid geometry, *Production Automated Project* Tech. Memo 25 University of Rochester, Rochester, NY, (Nov. 1977).
10. Requicha, A. A. G., and Voelcker, H. B., Boolean Operations in Solid Modeling Boundary Evaluation and Merging Algorithms, *Proc. IEEE* **73**, 1 (Jan 1985), 30-44.
11. Samet, H., The quadtree and related hierarchical data structures, *ACM Computing Surveys* **16**, 2 (June 1984), 187-260.
12. Shamos, I. M., and Hoey Dan, Geometric Intersection Problems *7th Annual IEEE Symp. on Foundations of Comp. Sci.* (Oct. 1976), 208-215.
13. Tilove, R. B., Set Membership Classification: A Unified Approach to Geometric Intersection Problems, *IEEE Trans. Comp.* C-29, 10 (Oct 1980), 874-883.
14. Wyvill, G., and Kunii, T. L., A functional model for constructive solid geometry, *The Visual Computer*, **1** (1985), 3-14.

Appendix:

Attached are several hardcopies of the Sun's display printed on an Imagen. The display shows selected examples of two collections each and their corresponding union and intersection, or their subtraction.


A
B
A∪B
A∩B
A-B
B-A


A
B
AuB
AnB


AuB
AnB
A-B
B-A