Environment and Planning B: Planning and Design, 1992, volume 19, pages 431 – 464

The arithmetic of maximal planes

R Krishnamurti
Department of Architecture, Carnegie-Mellon University, Pittsburgh, PA 15213, USA
Received 17 December 1991

Abstract. The geometry of shapes made up of finite planes is considered in detail. Algorithms on maximal planes for performing shape arithmetic are developed.

Introduction

This paper is a sequel to Krishnamurti (1992) in which the representation of shapes in terms of maximal spatial elements is considered. As pointed out in that paper, the maximal representation provides for the definition of shapes as definite geometrical objects with indefinitely many geometrical parts. In this paper, I examine the geometry of shapes made up solely of finite planes of nonzero area.

The following ideas are basic.

The algebra U_n, $n \geqslant 0$, is the least set of shapes made up of finite n-dimensional hyperplanes with nonzero measure, and is obtained by taking the closure under union and the Euclidean transformations (augmented with scale) of an appropriate set of n-dimensional hyperplanes (Stiny, 1991; see footnote (2) in Krishnamurti, 1992). Thus, U_2 is the algebra of shapes made up of finite planes of nonzero area.

A shape may consist of elements from different algebras. A shape may also consist of elements from the same algebra and belong to different aspects. For example, the shape consisting of the plan, elevation, and section of a building is made up of points, lines, and planes that belong to each drawing that describes the building. Thus, this shape would belong to the Cartesian product of the algebras $(U_0 \times U_1 \times U_2) \times (U_0 \times U_1 \times U_2) \times (U_0 \times U_1 \times U_2)$ where the parentheses are included merely for the purpose of illustration.

In general, a shape s is an ordered tuple of shapes $\langle s^1, s^2, \ldots, s^k, \ldots \rangle$, $k > 0$, where each s^k is a shape in the algebra U^k. Two shapes, s^i and s^j, $i \neq j$, may belong to the same algebra. Each U^k corresponds to an algebra U_n, for some $n \geqslant 0$, of shapes made up of n-dimensional planes. In this paper we concentrate on shapes in U_2.

A spatial element in a shape is *maximal* if it cannot be combined with other spatial elements (from the same algebra) in the shape to form a single larger spatial element. The *maximal representation* of a shape is its description in terms of its maximal spatial elements and is the smallest unique specification for the shape.

A spatial element is described by its *descriptor* and *boundary*, where the descriptor identifies the orientation of the element in an appropriate Euclidean space and the boundary identifies the position, size, and geometry of the spatial element. The descriptor partitions the elements of a shape into *co-equal* (that is, coincident, colinear, coplanar, or cohyperplanar, etc) equivalence classes. A shape may be organized as an ordered arrangement of classes of *co*-equal elements, that is, sets of spatial elements that share the same descriptor.

Spatial elements can be combined to form a larger element if they overlap, share boundaries, or if one element contains the other. Disjoint elements are relatively maximal and cannot be so combined. Spatial elements can be compared with other elements to form new elements. There are three basic operations to create

new elements and these correspond to the Boolean operations on shapes; union, difference, and intersection. Thus, spatial arithmetic can be captured by four basic geometric relations; *disjoint*, *overlap*, *share_boundary*, and *contain*, and by three basic operations; *combine*, *complement*, and *common*. In this paper, I develop the algorithms for these basic relations and operations on shapes defined by finite planes.

The algorithmic notations employed in this paper are based on a logic programming approach introduced and explained in Krishnamurti (1992) as are some of the basic predicates, other terms, and notations not explicitly defined here.

Last, as a visual prolegomenon, a cube with a square hole drilled through it is illustrated as a shape made up of maximal planes (figure 1). The maximal planes that make up the sides of the cube are shown. The reader may notice two of the maximal planes that define the cube have holes.

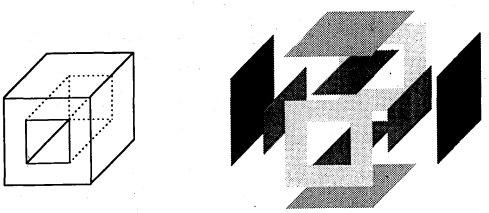


Figure 1. A cube with a square hole and the maximal planes that make up the sides of the cube.

Shapes in U_2

Shapes in U_2 consist of maximal planes. A maximal plane is specified by its boundary which is a set of polygons one of which is the *outer boundary* of the plane and all others are the *inner boundaries* of the plane. A maximal plane is *simple* if it has no inner boundaries. Nonsimple maximal planes contain holes. For a maximal plane, the lines of the inner boundaries lie inside the region defined by the outer boundary and at best touch the outer boundary at one point. Inner boundaries may touch another inner boundary at a point. Examples of maximal planes are shown in figure 2.

The simplicity of a maximal plane is easily established. Let M denote a maximal plane. M is represented by the set of simple polygons $\{b_0, b_1, \ldots, b_m\}$ where, by convention, b_0 is the outer boundary and all others are inner boundaries of M. Then,

$$simple(M) \leftarrow M = \{b_0\}\,. \tag{S1}$$

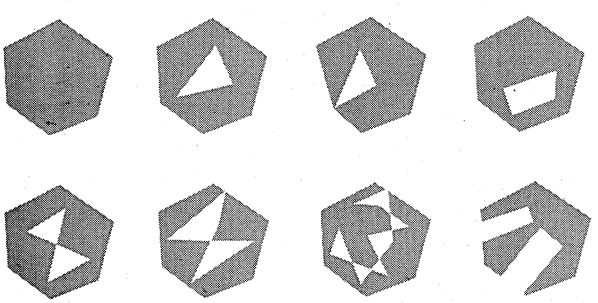


Figure 2. Examples of maximal planes.

Nonsimple maximal planes have an outer boundary and one or more inner boundaries. However, any boundary of a maximal plane, whether outer or inner, is a simple polygon as figure 2 illustrates. In some cases, the boundaries of a maximal plane are disconnected polygons as figure 3 demonstrates. In other cases, the combination of the outer and inner boundaries can be combined to form a single self-intersecting polygon as illustrated in figure 4. In still other cases, the boundaries of several maximal planes can be combined to form a single self-intersecting polygon. Figure 5 illustrates a self-intersecting polygon that is formed by four distinct maximal planes. Thus, a maximal plane can be considered to be formed by disconnected or disjoint planes each bounded by simple polygons some of which combine to form self-intersecting polygons.

The process of extracting maximal planes from a specification of polygons consists of decomposing self-intersecting polygons into simple planes that make up the region defined by the polygons, and of determining whether the boundaries of a collection of disjoint or disconnected simple planes are outer or inner boundaries of maximal planes.

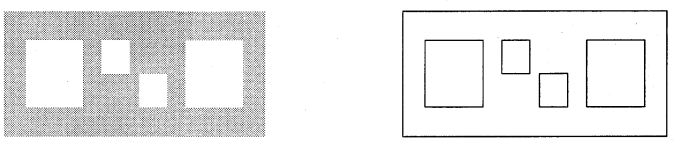


Figure 3. The boundaries of a maximal plane can be disconnected. In this case, the maximal plane consists of a single outer boundary and four inner boundaries.

Figure 4. A nonsimple maximal plane whose boundaries form a self-intersecting polygon.

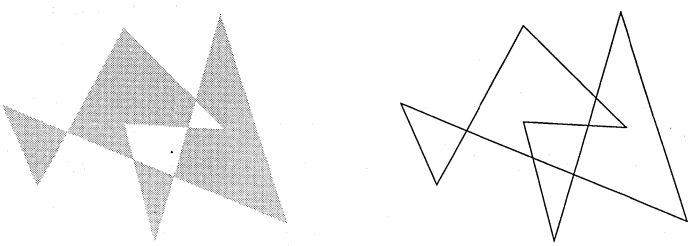


Figure 5. Four distinct simple maximal planes that form a single self-intersecting polygon.

Converting a self-intersecting polygon into maximal planes

The first step in decomposing a polygon into simple planes is to find the self-intersecting points of the polygon. Self-intersecting points act as 'articulation' points in the sense that these are the points at which the polygon splits into simpler polygons. There are two possible ways that a polygon can split at self-intersecting points. Either the split polygons share a common point which must be a self-intersecting point, or share a common side bounded by two self-intersecting points. These two cases are illustrated in figure 6.

By the Jordan curve theorem, every polygon divides the plane into two regions, an inside and an outside. The points bounded by the outer boundary of a

maximal plane are interior points and those bounded by its holes are exterior points. Points outside a maximal plane are also exterior points. The notion of interior and exterior points is useful in designating a simple plane as *interior* if it bounds interior points and *exterior* otherwise. Thus, the outer boundary of a maximal plane is interior and the inner boundaries are exterior. When a polygon is split at a self-intersecting point, the polygons formed are either all relatively interior or some are relatively interior and some are relatively exterior. When a polygon is split at two self-intersecting points, a region on one side of the line is relatively interior and the other region is relatively exterior. Moreover, the exterior region will not be a hole in any maximal plane. These observations are illustrated in figure 7.

It is convenient to treat a polygon P as a *necklace*[1] of points $\{p_1, p_2, \ldots, p_k, p_1\}$ where each consecutive pair of points $\{p_i, p_{(i \bmod k)+1}\}$ in the necklace is a side of the polygon. Moreover, the necklace can be rotated until p_1 is the left-most bottom corner of the polygon and $\{p_1, p_2\}$ is the side with the least gradient. This ensures that the necklace starts as a counterclockwise sequence of points around the polygon from the left-most corner.

Suppose we traverse the sequence along the sides comparing the current side with the other sides for self-intersecting points. Let $\{p, q\}$ be the current side and $\{r, s\}$ be another distinct side of the polygon that is being compared with. There are two possibilities. Either the sides intersect at, say, m coincident with both line segments, or they do not. Suppose the former is the case. We can replace the two sides by at most four line segments by inserting m *after* p and r in the sequence. That is, m splits the two sides into at most four segments; $\{p, m\}, \{m,q\}, \{r, m\}$, and $\{m, s\}$. Point m may coincide with one of the endpoints in which case we exclude the side $\{m, m\}$ from the sequence (see figure 8).

If $\{p, q\}$ intersects with two or more sides, we process the intersection points in order of their distance from p. Suppose the current sequence, P, is $\{..., p, q, \ldots, r, s, ...\}$.

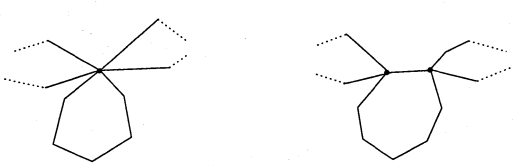


Figure 6. Splitting a polygon at self-intersecting points.

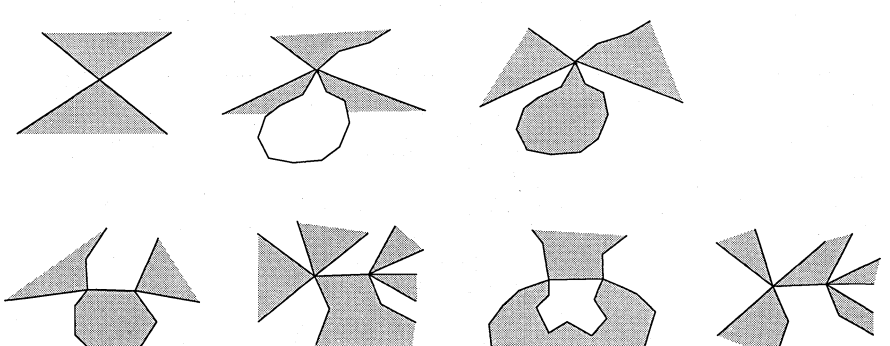


Figure 7. Relative interior and exterior polygons formed at self-intersecting points. The shaded parts indicate interior regions.

[1] A necklace is a sequence that wraps around onto itself. That is, each consecutive pair of elements, including the last and first, are related to each other in the same way.

Then, after inserting m into the sequence, we get a new sequence, P',

$P' = \{..., p, m, q, ..., r, m, s, ...\}$ if $m \neq p, q, r, s$,
$P' = \{..., p, q, ..., r, m, s, ...\}$ if $m = p$ or q and $m \neq r, s$,
$P' = \{..., p, m, q, ..., r, s, ...\}$ if $m \neq p, q$ and $m = r$ or s,
$P' = \{..., p, q, ..., r, s, ...\}$ if $m = p$ or q and $m = r$ or s.

This process is repeated until all the self-intersecting points have been determined. Figure 9 illustrates the procedure for finding the five self-intersecting points in the polygon shown in figure 5.

The routine *resequence* generates all self-intersecting points and places them in the proper position in a necklace of points specifying a polygon. The routine returns a sequence that contains the corners of the polygon and all self-intersecting points. A polygon will have at least three points, so the input to *resequence* will have at least four points, the first point being included at least twice. *Resequence* has the following definition.

resequence($\{p\}, \varnothing$). (R1)

resequence($\{p, q\} + Necklace$, *Sequence*) ←
¶ find all points of intersection of $\{p, q\}$ and the other lines of the polygon
¶ such that the points are between p and q, and arrange them in increasing
¶ distance from p
self_intersections($\{p, q\}$, *Necklace*, I_{pq}, N) ∧
¶ recursively traverse from q
resequence($\{q\} + N$, S) ∧
resequence_side($\{p, q\}$, I_{pq}, S_{pq}) ∧
merge_sequence(S_{pq}, S, *Sequence*). (R2)

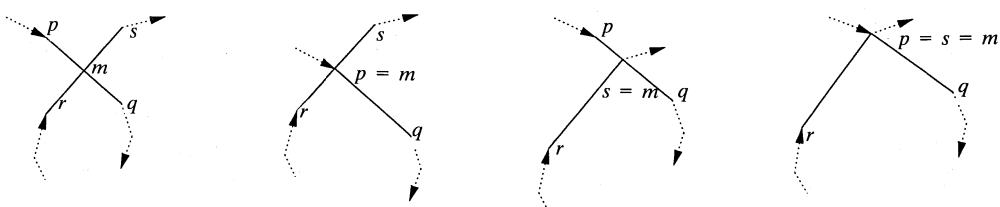
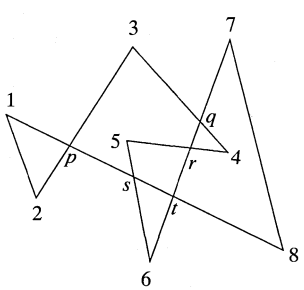


Figure 8. Possible self-intersecting points when two sides intersect.



$P = \{1, 2, 3, 4, 5, 6, 7, 8, 1\}$

$\{2, 3\}$ and $\{8, 1\}$ intersect at p
$P = \{1, 2, \check{p}, 3, 4, 5, 6, 7, 8, \check{p}, 1\}$

$\{3, 4\}$ and $\{6, 7\}$ intersect at q
$P = \{1, 2, p, 3, \check{q}, 4, 5, 6, \check{q}, 7, 8, p, 1\}$

$\{4, 5\}$ and $\{6, q\}$ intersect at r
$P = \{1, 2, p, 3, q, 4, \check{r}, 5, 6, \check{r}, q, 7, 8, p, 1\}$

$\{5, 6\}$ and $\{8, p\}$ intersect at s
$P = \{1, 2, p, 3, q, 4, r, 5, \check{s}, 6, r, q, 7, 8, \check{s}, p, 1\}$

$\{6, r\}$ and $\{8, s\}$ intersect at t
$P = \{1, 2, p, 3, q, 4, r, 5, s, 6, \check{t}, r, q, 7, 8, \check{t}, s, p, 1\}$

Figure 9. Finding the self-intersecting points of a polygon identifies the self-intersection point currently determined by the algorithm.

The first rule R1 is a terminating rule when the processed sequence contains just one point, namely the start point. The second rule R2 needs further explanation. We take the first two points, $\{p, q\}$, in the sequence and find all its points of intersection with the remaining lines in the sequence. We continue resequencing starting at q.

In the course of resequencing a polygon, in order to include the self-intersection points in their correct position in the sequence, we traverse the polygon a side at a time. Suppose $\{p, q\}$ is the current side. At this stage, we determine all self-intersection points that are coincident on the segment $\{p, q\}$. The predicate *self_intersections* returns two lists. One is the list, I_{pq}, of self-intersection points coincident with line $\{p, q\}$ in increasing order of their distance from p. The points in I_{pq}, if any, are inserted between p and q in the required sequence. The other is the updated list of points that have yet to be traversed.

self_intersections$(\{p, q\}, \{p_1\}, \varnothing, \varnothing)$. (SeI1)

self_intersections$(\{p, q\}, \{r, s\} + Necklace, I_{pq}, \{r\} + N_{pq}) \leftarrow$
¶ $\{p, q\}$ and $\{r, s\}$ intersect at m
$MEET(\{p, q\}, \{r, s\}, m) \wedge$
self_intersections$(\{p, q\}, \{s\} + Necklace, I, N) \wedge$
¶ if m is not identical to either r or s, insert it between r and s in the sequence
¶ if m is not identical to either p or q, insert it between p and q in the sequence
ifelse$(m = r \vee m = s, equate(N_{pq}, N), equate(N_{pq}, \{m\} + N)) \wedge$
ifelse$(m = p \vee m = q, equate(I_{pq}, I), INSERT\_POINT(m, p, I, I_{pq}))$. (SeI2)

self_intersections$(\{p, q\}, \{r, s\} + Necklace, I_{pq}, \{r\} + N_{pq}) \leftarrow$
¶ $\{p, q\}$ and $\{r, s\}$ do not intersect at a point coincident with both segments
not $MEET(\{p, q\}, \{r, s\}, m) \wedge$
self_intersections$(\{p, q\}, \{s\} + Necklace, I_{pq}, N_{pq})$. (SeI3)

There are three cases that *self_intersections* must consider: (1) when it reaches the end of the necklace; (2) when the current line meets another line at a self-intersecting point which is inserted into the sequence in at most two places, depending on whether the intersection point coincides with an endpoint of one of the two lines; (3) when the current line does not meet another line at a point coincident with either line. The routine *MEET* determines the intersection point that is considered between the endpoints of two given lines. *MEET* fails if there is no intersection point coincident with either line. *MEET* uses standard geometric properties and its definition is assumed. The routine *INSERT_POINT* inserts the self-intersection point m in a list containing all such points on the line $\{p, q\}$ arranged in increasing order of their distance from p. Its definition is straightforward and also left to the reader.

The insertion mechanism in rule SeI2 is expressed by the two *ifelse* statements. The two *ifelse* rules subsume the four-case insertion rule given at the top of page 435. *Ifelse* has the following standard definition.

ifelse$(Cond, Then, Else) \leftarrow Cond \wedge Then$. (If1)

ifelse$(Cond, Then, Else) \leftarrow$ *not* $Cond \wedge Else$. (If2)

The routine *equate* unifies the first argument which must be a variable to the second argument which has been instantiated. *Equate* has the following trivial definition.

equate(X, X).

The sequence of points produced by procedure *resequence* includes the self-intersecting points and each of these occurs at least twice. Hence, if each point in the sequence excluding the last point occurs just once, then the polygon and its corresponding maximal plane must be simple.

Suppose the polygon is not simple and suppose we traverse the resequenced necklace of points in order. Then, three possibilities can occur and these are illustrated in figure 10.

In the first case, the self-intersecting point, m, acts as an articulation point. That is, there is a subsequence of the form $\{m, p_i, p_{i+1}, \ldots, p_j, m\}$ where m denotes a self-intersecting point and p_k, $i \leqslant k \leqslant j$ are not self-intersecting. The sequence $\{m, p_i, p_{i+1}, \ldots, p_j\}$ is a polygon which forms the outer boundary of a maximal plane or the inner boundary of a maximal plane. In either case, we can repeatedly remove these subsequences until it is no longer possible to do so, by applying the following rule:

$$P_1, m, p_i, p_{i+1}, \ldots, p_j, m, P_2 \rightarrow P_1, m, P_2 \text{ and } \{m, p_i, p_{i+1}, \ldots, p_j\} \text{ is a polygon.} \quad (1)$$

That is, we replace the sequence $P_1, \{m, p_i, p_{i+1}, \ldots, p_j\}, m, P_2$ by the sequence P_1, m, P_2 and output $\{m, p_i, p_{i+1}, \ldots, p_j\}$ as a polygon.

In the second and third cases, two distinct self-intersecting points m and n form a common side to two polygons. That is, there is subsequence $\{m, p_i, p_{i+i}, \ldots, p_j, n\}$ where the p_k, $i \leqslant k \leqslant j$ are no longer self-intersecting points. It may be that m and n are not connected. We are only interested in the case when $\{m, n\}$ is a side of the polygon in which case there must be a subsequence $\{n, m\}$ that occurs later in the sequence. When $\{n, m\}$ is a side of the self-intersecting polygon, two possibilities arise. Either the polygon formed from the sequence $\{m, p_i, p_{i+1}, \ldots, p_j, n\}$ is an

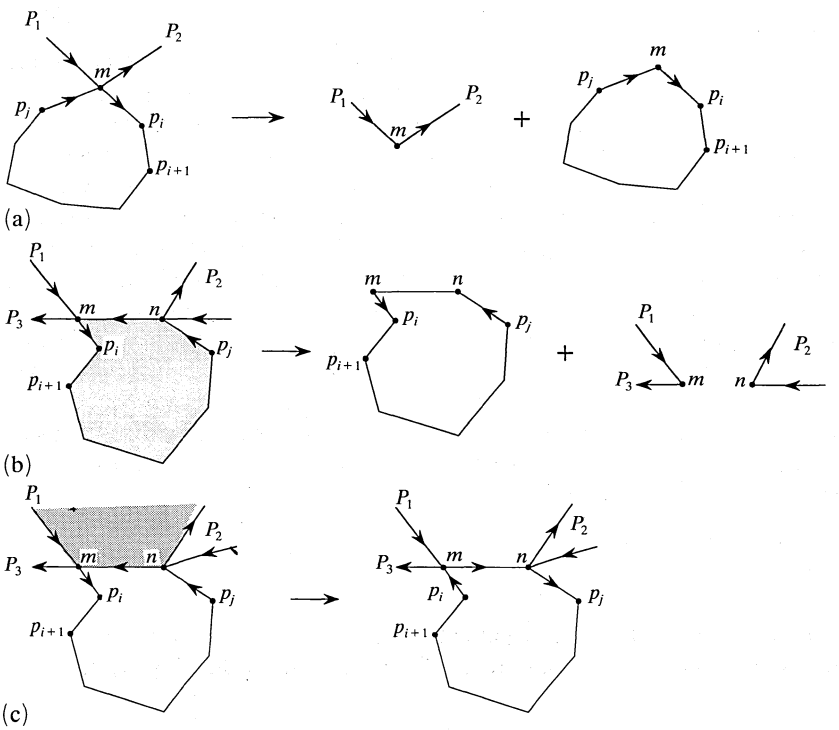


Figure 10. Three possibilities that arise at self-intersecting points when traversing a sequence of points that describes a polygon.

interior polygon or the polygon defines a hole that shares sides with two or more interior polygons. In the first of these subcases, we apply the rule

$$P_1, m, p_i, p_{i+1}, \ldots, p_j, n, P_2, n, m, P_3 \rightarrow P_1, n, P_2, n, m, P_3 \text{ and } \{m, p_i, p_{i+1}, \ldots, p_j, n\} \text{ is a polygon .} \quad (2)$$

In the second of these subcases, we apply the rule

$$P_1, m, p_i, p_{i+1}, \ldots, p_j, n, P_2, n, m, P_3 \rightarrow P_1, m, n, P_2, n, p_j, p_{j-1}, \ldots, p_i, m, P_3 . \quad (3)$$

As an illustration, we apply these rules to the polygon in figure 9 to obtain the constituent simple polygons. We traverse the sequence in the given order, pushing points onto a stack until we visit a point that has already been pushed onto the stack at which stage we pop the stack till we hit the point again. The following table outlines the procedure.

Current point	Stack	Rule and polygon
r	1, 2, p, 3, q, 4, r	(3)
t	1, 2, p, 3, q, r, 5, s, 6, t	(2) and $\{s, 6, t\}$
t	1, 2, p, 3, q, r, 5, t, r, 4, q, 7, 8, t	(1) and $\{r, 4, q, 7, 8, t\}$
p	1, 2, p, 3, q, r, 5, s, p	(1) and $\{3, q, r, 5, s, p\}$
1	1, 2, p, 1	(1) and $\{2, p, 1\}$

This procedure will work if the following definitions for *resequence_side* and *merge_sequence* invoked in rule R2 are adopted.

resequence_side$(\{p, q\}, I_{pq}, \{p\}+I_{pq})$.
merge_sequence$(S_{pq}, S, Sequence) \leftarrow APPEND(S_{pq}, S, Sequence)$[2] .

Of course, we still need to split the sequence into simple polygons along the lines indicated above.

Once the simple polygons have been determined, we can check whether these simple polygons correspond to inner or outer boundaries of maximal planes. Essentially we compare the simple polygons pairwise to see if one is inside the other.[3] If so, then one of the polygons is potentially an inner boundary of, and the other is potentially an outer boundary of, some maximal plane. If not, they belong to different maximal planes. In this way, we can build a list of polygons, some inside other polygons some outside.

Before I present an algorithm to compare simple polygons to extract the maximal planes, we should take another look at the resequencing algorithm.[4] The procedure described above for extracting simple polygons from a sequence of points is needlessly cumbersome, has many special cases, and involves extensive search. So, instead of simply producing a new sequence of points to describe the polygon, it is worth considering the lines that make up the sides of the polygon.

Each side of a polygon can be considered to be made up of a list consisting of pairs or triples of points of the form $\{p, m\}$ and $\{m, p, q\}$. In the case of triples, m denotes a self-intersection point. We add the convention that each side, $\{p, q\}$, of

[2] *APPEND* concatenates two lists and it is defined in Krishnamurti (1992).

[3] We only need to check one point from each polygon to test for insideness. An algorithm for polygon checking is given in Preparata and Shamos (1985).

[4] The versions of *resequence*, *isolate_polygon* and *compare_planes* described in the paper are based on improvements due to Rudi Stouffs (personal communication).

the polygon satisfies $p < q$. That is, each side is processed from its left-most (bottom-most) point. Thus,

resequence_side$(\{p, q\}, I_{pq}, S_{pq}) \leftarrow$
  $p < q \wedge$
  APPEND$(I_{pq}, \{q\}, I) \wedge$
  sequence_side$(\{p\}+I, S) \wedge$
    equate$(S_{pq}, \{\{p, car(I)\}\}+S)$.

resequence_side$(\{p, q\}, I_{pq}, S_{pq}) \leftarrow$
  $p > q \wedge$
  REVERSE(I_{pq}, I_{qp})[5] $\wedge$
  APPEND$(I_{qp}, \{p\}, I) \wedge$
  sequence_side$(\{q\}+I, S) \wedge$
    equate$(S_{pq}, \{\{q, car(I)\}\}+S)$.

Resequence_side returns a sorted list of pairs and triples, each consisting of a point and all points connected to it by a line. That is, if $\{p, m_1, \ldots, m_k, q\}$ is the sequence of endpoints and self-intersecting points on the side $\{p, q\}$, $p < q$, of the polygon, then *resequence_side* produces the list $\{\{p, m_1\}, \{m_1, p, m_2\}, \ldots, \{q, m_k\}\}$. Moreover, $p < m_1 < \ldots < m_k < q$. If the side has no self-intersecting points, the list $\{\{p, q\}, \{q, p\}\}$ is returned.

The predicate *car* invoked in the *ifelse* clause returns the first element of a list.

Sequence_side has the following simple definition.

sequence_side$(\{p, q\}, \{\{q, p\}\})$.

sequence_side$(\{p, q, r\}+I, \{\{q, p, r\}\}+S) \leftarrow$
  sequence_side$(\{q, r\}+I, S)$.

Once a side of the polygon has been processed, the sorted list that is produced has to be merged with the sorted lists produced for the remaining sides of the polygon. The reason for this is given by the fact that a self-intersecting point is produced whenever at least two sides of the polygon cross each other. Thus, if $\{p, q\}$ and $\{r, s\}$ intersect at m, then the triple $\{m, p, q\}$ produced when processing side $\{p, q\}$ and the triple $\{m, r, s\}$ produced when processing side $\{r, s\}$ have to be merged to form the quintuple $\{m, p, q, r, s\}$ which indicates that m is linked to the points p, q, r, and s by lines. This is accomplished by *merge_sequence* which has the following definition.

merge_sequence$(S, \varnothing, S)$.

merge_sequence$(\varnothing, S, S) \leftarrow S \neq \varnothing$.

merge_sequence$(\{P\}+Seq\_1, \{Q\}+Seq\_2, \{P\}+Merged) \leftarrow$
  $car(P) < car(Q) \wedge$
  merge_sequence$(Seq\_1, \{Q\}+Seq\_2, Merged)$.

merge_sequence$(\{P\}+Seq\_1, \{Q\}+Seq\_2, \{M\}+Merged) \leftarrow$
  $car(P) = car(Q) \wedge$
  APPEND$(P, cdr(Q), M) \wedge$
  merge_sequence$(Seq\_1, Seq\_2, Merged)$.

merge_sequence$(\{P\}+Seq\_1, \{Q\}+Seq\_2, \{Q\}+Merged) \leftarrow$
  $car(P) > car(Q) \wedge$
  merge_sequence$(\{P\}+Seq\_1, Seq\_2, Merged)$.

[5] *REVERSE* reverses the order of the elements in a list. Its definition is straightforward and is omitted.

In other words, *merge_sequence* produces an adjacency list of points for each corner and self-intersecting point of the polygon. The predicates *car* and *cdr* have the same definition as in functional programming; that is, *car* returns the first element of a list and *cdr* returns the remaining list.

The preceding discussion for determining the maximal planes from a self-intersecting polygon can be summarized as follows.

get_maximal_planes(*Polygon*, *Planes*) ←
  resequence(*Polygon*, *PointAdjacencyLists*) ∧
  isolate_planes(*PointAdjacencyLists*, ∅, ∅, ∅, *Planes*).

Isolate_planes is described in the next section.

Determining a set of maximal planes from a set of lines

The analogous problem to splitting a self-intersecting polygon into planes is the determination of maximal planes from a given set of lines. The algorithm is quite simple. Although it is preferable that each line in the set corresponds to a side of a polygon, it is not necessary. Any line that is not the side of a polygon is ignored. We convert the lines into a sorted list of adjacency lists of points. We then isolate the maximal planes from the point adjacency lists.

The routine *maximal_planes* takes a set of lines, constructs the point adjacency list and then isolates maximal planes.

maximal_planes(*Lines*, *Planes*) ←
  make_adjacency_lists(*Lines*, *PointAdjacencyLists*) ∧
  isolate_planes(*PointAdjacencyLists*, ∅, ∅, ∅, *Planes*).

To isolate the planes we do a depth-first search on the lines by starting each traversal in a counterclockwise direction starting from the left-most bottom point. We maintain a stack of points and search along a clockwise direction from the current top of stack. If the line has an endpoint already on the stack, we have found a polygon. There are two possibilities. Either the stack is empty or it is not. If the stack is not empty, then the polygon that is found must be interior to another polygon that has yet to be found (because the sides are traversed in a clockwise manner). If the stack is empty, the polygon that is found is exterior to all polygons found from the current path (because we always start a traversal in a counterclockwise manner from the left-most bottom available point). This polygon together with all polygons inner to it is then tested against the currently determined maximal planes to determine whether it can be the inner or outer boundaries of a maximal plane. The algorithm maintains two additional auxiliary lists, one for the polygons that are constructed from the current path and one for the list of maximal planes that have been previously found. The procedure terminates when all lines have been examined.

isolate_planes(∅, *Stack*, *Polygons*, *Aux*, *Planes*) ←
  ¶ if the stack is not empty, the current path does not complete a polygon
  compare_each_plane(*Polygons*, *Aux*, *Planes*). (IP1)

isolate_planes(*AdjList*, ∅, ∅, *Aux*, *Planes*) ←
  ¶ empty stack – start a counterclockwise traversal {*p*, *q*} from the left-most
  ¶ bottom available point
  ¶ remove side from the adjacency lists
  AdjList ≠ ∅ ∧
  start_path(*AdjList*, {*p*, *q*}, *NewList*) ∧
  isolate_planes(*NewList*, {*q*, *p*}, ∅, *Aux*, *Planes*). (IP2)

isolate_planes(*AdjList*, {*q*, *p*} + *Stack*, *Polygons*, *Aux*, *Planes*) ←
¶ find the most clockwise side {*q*, *r*} to {*p*, *q*}
¶ if *r* is on the the stack, we have found a polygon—process it
MEMBER({*q*} + *Q*, *AdjList*) ∧
continue_path(*AdjList*, {*q*, *p*}, {*r*, *q*}, *NewList*) ∧
MEMBER(*r*, *Stack*) ∧
end_path(*NewList*, {*r*, *q*, *p*} + *Stack*, *Polygons*, *Aux*, *Planes*). (IP3)

isolate_planes(*AdjList*, {*q*, *p*} + *Stack*, *Polygons*, *Aux*, *Planes*) ←
¶ find the most clockwise side {*q*, *r*} to {*p*, *q*}
¶ if *r* is not on the stack, continue traversal
MEMBER({*q*} + *Q*, *AdjList*) ∧
continue_path(*AdjList*, {*q*, *p*}, {*r*, *q*}, *NewList*) ∧
not MEMBER(*r*, *Stack*) ∧
isolate_planes(*NewList*, {*r*, *q*, *p*} + *Stack*, *Polygons*, *Aux*, *Planes*). (IP4)

isolate_planes(*AdjList*, {*q*, *p*} + *Stack*, *Polygons*, *Aux*, *Planes*) ←
¶ if we cannot find a side from *q*, {*q*, *p*} is a dangling line; backtrack to *p*
¶ and continue
not MEMBER({*q*} + *Q*, *AdjList*) ∧
isolate_planes(*AdjList*, {*p*} + *Stack*, *Polygons*, *Aux*, *Planes*). (IP5)

In the course of traversing a polygon there are three kinds of decisions that have to be made. The first is to decide where to start a new path which is always along the counterclockwise edge at the left-most bottom available point. The edge will correspond to the one with the least gradient of all lines from this point.

start_path({{*p*} + *P*} + *AdjList*, {*p*, *q*}, {*P'*} + *NewList*) ←
find_ccw_edge(*p*, *P*, *q*) ∧
¶ remove side {*p*, *q*} from the adjacency lists of *p* and *q*
DELETE(*q*, *P*, *P'*)[6] ∧
DELETE({*q*} + *Q*, *AdjList*, *A*) ∧
DELETE(*p*, *Q*, *Q'*) ∧
INSERT({*q*} + *Q'*, *AdjList*, *NewList*)[6].

Second, when we have traversed a sequence of lines, decide which edge to follow next which is at the least clockwise angle from the most recent edge traversed.[7]

continue_path(*AdjList*, {*q*, *p*}, {*r*, *q*}, *NewList*) ←
¶ find the nearest clockwise edge {*q*, *r*} to {*p*, *q*}
¶ remove side {*q*, *r*} from the adjacency lists of *q* and *r*
DELETE({*q*} + *Q*, *AdjList*, *A*) ∧
find_cw_edge({*q*, *p*}, *Q*, *r*) ∧
DELETE(*r*, *Q*, *Q'*) ∧
DELETE({*r*} + *R*, *A*, *A'*) ∧
DELETE(*q*, *R*, *R'*) ∧
APPEND({*r*}, *R'*, *R''*) ∧
ifelse(*Q'* = ∅, *INSERT*({*R''*}, *A'*, *NewList*),
INSERT({{*q*} + *Q'*, *R''*}, *A'*, *NewList*))[8].

[6] *DELETE* removes a member of a list and returns the remaining elements in the list. *INSERT* adds an element into a list in the correct position (determined by the imposed order relation).

[7] If the adjacency list for a point is arranged as a doubly linked circular list of edges arranged in counterclockwise order around the point, the next edge in clockwise or counterclockwise order requires $O(1)$ pointer operations.

[8] *INSERT*({*x*, *y*}, *L*, *N*) is shorthand for the conjunction *INSERT*({*x*}, *L*, *M*) ∧ *INSERT*({*y*}, *M*, *N*).

The third and last decision to be made occurs when we have found a polygon. There are two possibilities. If the polygon returns to the start point of the current path, then it is an outer polygon to all polygons that have been thus far found from the current path. We take this polygon and all its inner polygons and compare them against the maximal planes that have been determined so far. If the polygon returns to a point which is not the start point of the current path, we have found a polygon which is inner to the polygon that returns to the start point. We save it and resume the traversal from the current point. The two possibilities are captured by the following rule.

end_path(*AdjList*, {*r*,*q*} + *Stack*, *Polygons*, *Aux*, *Planes*) ←
  ¶ extract polygon from current path
  ¶ if path is not empty, save polygon and continue traversal from current point
  POP_STACK(*r*, *Stack*, *P*, {*r*} + *NewStack*) ∧
  NewStack ≠ ∅ ∧
    isolate_planes(*AdjList*, {*r*} + *NewStack*, {*P*} + *Polygons*, *Aux*, *Planes*).

end_path(*AdjList*, {*r*,*q*} + *Stack*, *Polygons*, *Aux*, *Planes*) ←
  ¶ extract polygon from current path
  ¶ if path is empty, compare polygon and its inner polygons with known planes
  ¶ and start a new traversal from the leftmost available point
  POP_STACK(*r*, *Stack*, *P*, {*r*} + *NewStack*) ∧
  NewStack = ∅ ∧
    compare_planes({*P*} + *Polygons*, *Aux*, *NewPlanes*),
    isolate_planes(*AdjList*, ∅, ∅, *NewPlanes*, *Planes*).

Rule IP1 is the terminating rule. Under normal conditions, when each line is a side of a polygon, the stack will eventually become empty and all polygons found off the most recently traversed path would have been tested against the known maximal planes. Moreover, under these conditions rule IP5 will never be invoked. However, should the stack not be empty, then there are lines which are not sides of any polygon. Further, there may be polygons that have been found off the path corresponding to the points on the stack. Two possibilities arise. Either these polygons are all interior to some maximal plane or they are all simple planes. In all cases, the following rule for *compare_each_plane* will work.

compare_each_plane(∅, *Planes*, *Planes*).

compare_each_plane({*P*} + *Polygons*, *Aux*, *Planes*) ←
  compare_planes({*P*}, *Aux*, *NewAux*) ∧
  compare_each_plane(*Polygons*, *NewAux*, *Planes*).

Comparing a plane with a set of maximal planes

We are now ready to test a plane for maximality against a known set of maximal planes. We make the assumption that the plane to be tested is known to be completely inside one of the maximal planes or is outside all the maximal planes. The condition is satisfied by the algorithms considered in the previous two sections.

Essentially the algorithm compares the outer boundary of the given plane with the outer boundary of each of the maximal planes. Two possibilities arise. Either it lies inside the outer boundary of one maximal plane or it lies outside all. In the latter case, the given plane is maximal and is appended to the list of maximal planes. In the first case, two further possibilities arise. Either the outer boundary of the given plane lies inside one of the holes of the maximal plane in which case the given plane is maximal or it lies outside all the holes, in which case the outer

boundary of the tested plane becomes an inner boundary of the maximal plane and its inner boundaries become simple maximal planes. See figure 11.

The routine *compare_planes* together with auxiliary routines *inside_innerplanes* and *inner_to_simple_planes* compares a plane against a list of maximal planes by using the procedure described above. Initially the list of maximal planes is empty. Each plane is represented by the list of polygons $\{b_0, b_1, ...\}$, where b_0 is the outer boundary. If the plane is simple, it is represented by the singleton list $\{b_0\}$.

compare_planes$(P, \varnothing, \{P\})$. (CP1)

compare_planes$(P, \{M\} + Planes, \{M, P\} + Planes) \leftarrow$
¶ if the outer boundary of P is inside the outer boundary of M,
¶ compare it against the inner boundaries of M
inside$(car(P), car(M)) \wedge$
inside_innerplanes$(car(P), cdr(M))$. (CP2)

compare_planes$(P, \{M\} + Planes, \{M'\} + NewPlanes) \leftarrow$
¶ if the outer boundary of P is inside the outer boundary of M
¶ and outside the inner boundaries of M, it becomes an inner boundary of M
¶ and the inner boundaries of P become simple maximal planes.
inside$(car(P), car(M)) \wedge$
not inside_innerplanes$(car(P), cdr(M)) \wedge$
APPEND$(\{car(M), car(P)\}, cdr(M), M') \wedge$
inner_to_simple_planes$(cdr(P), IP) \wedge$
APPEND$(IP, Planes, NewPlanes)$. (CP3)

compare_planes$(P, \{M\} + Planes, \{M\} + NewPlanes) \leftarrow$
¶ if the outer boundary of P is not inside the outer boundary of M,
¶ repeat the procedure selecting another maximal plane.
not inside$(car(P), car(M)) \wedge$
compare_planes$(P, Planes, New\_Planes)$. (CP4)

¶ *inside_innerplanes* will not succeed if the maximal plane is originally simple
inside_innerplanes$(P, \{M\} + Inner) \leftarrow$
inside(P, M).

inside_innerplanes$(P, \{M\} + Inner)$
not inside$(P, M) \wedge$
inside_innerplanes$(P, Inner)$.

¶ *inner_to_simple_planes* converts a set of polygons to a set of simple planes
inner_to_simple_planes$(\varnothing, \varnothing)$.

inner_to_simple_planes$(\{P\} + Inner, \{\{P\}\} + Planes) \leftarrow$
inner_to_simple_planes$(Inner, Planes)$.

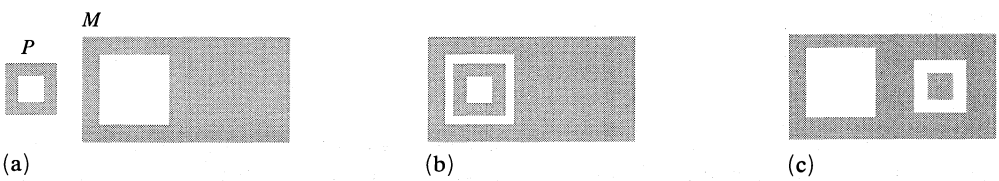


Figure 11. Comparing a plane P against a maximal plane M. (a) Outer boundary of P is outside outer boundary of M. (b) Outer boundary of P is inside an inner boundary of M. (c) Outer boundary of P is inside outer boundary of M but outside inner boundaries of M.

Combining simple maximal planes into a maximal plane

A basic operation on simple planes is to take two maximal planes and combine them into a single maximal plane. Because each plane is specified by its outer boundary, this is equivalent to comparing two polygons to produce a single polygon that encloses the area occupied by both polygons. We can make the following observation: if p is a corner of one of the polygons that is outside the other polygon, then p is a corner of the required polygon (see figure 12).

Let us make some further observations. First, we note that the boundary of the combined maximal plane is made up of lines from the boundaries of both planes that lie outside the other. Consequently, we can take the shape union defined on sets of lines to combine these outer lines (with respect to the other plane) to form the boundaries of the combined maximal plane. The combination of two simple maximal planes can introduce holes as illustrated by figure 13.

Second, we note that as a consequence of the Jordan curve theorem, any line joining two points outside (inside) a given polygon intersects the polygon no or an even number of times. Equivalently, a line jointing a point outside and a point inside a polygon must intersect the polygon an odd number of times. The exceptional case is when one or both endpoints of a line is coincident with a bounding line and these cases have to be treated separately. That is, we can preprocess each polygon with respect to the other by determining the intersection points, distinct from endpoints, coincident with each line by using a variation of the polygon resequencing algorithm described earlier. That is, each maximal line $\{p_i, p_j\}$ is replaced by the sequence of points $\{p_i, m_{ij}^l, \ldots, m_{ij}^n, p_j\}$ where $m_{ij}^k \neq p_i, p_j, 0 \leqslant k \leqslant n$, is an intersection point on the line $\{p_i, p_j\}$; $n = 0$ or an even number if both endpoints are outside or if both endpoints are inside (see figure 14).

Third, we may assume that the two planes overlap or share boundaries. The case when one plane contains the other is taken care of by the rules for shape union (see rules CU3 and CU4 in Krishnamurti, 1992). In the disjoint case, maximal planes do not combine.

Last, we may assume that the boundaries are given by their maximal line representation.

We consider the combination of two simple planes when the planes share bounding lines and when they overlap as separate cases.

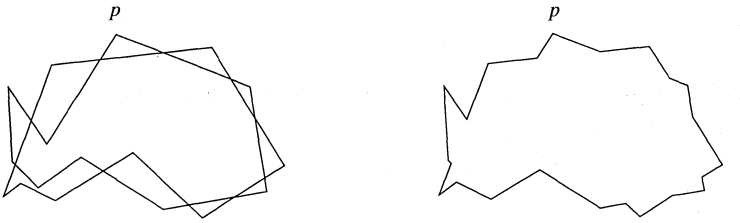


Figure 12. Two simple polygons and their combined polygon.

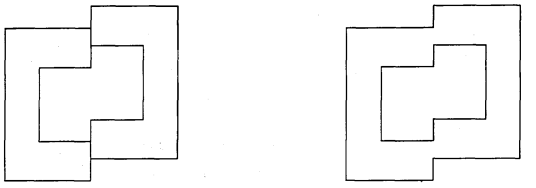


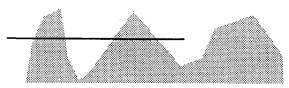
Figure 13. The combination of two simple planes can introduce holes.

Case 1: *Suppose the two planes share bounding lines but do not overlap.*

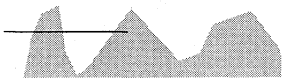
In this case, the common lines are inside the combined region as can be observed from figure 23 (below). Hence, if we remove the common bounding lines from both boundaries we are left with lines whose union gives rise to polygons, one of which forms the outer boundary and others of which, if any, correspond to the inner boundaries. Thus, we may describe the algorithm for the combination of two simple planes P and Q as follows:

simple_share_combine(P, Q, *Planes*) ←
  shape_difference_in_U$_1$(P, Q, P') ∧
  shape_difference_in_U$_1$(Q, P, Q') ∧
  shape_union_in_U$_1$(P', Q', *Lines*) ∧
  maximal_planes(*Lines*, *Planes*).

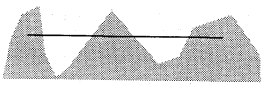
This procedure is illustrated in figure 15.



both endpoints of line outside polygon



one endpoint of line outside polygon and the other inside



both endpoints of line inside polygon

both endpoints of line coincident with polygon

one endpoint of line coincident with polygon

Figure 14. The intersection of a line and a polygon.

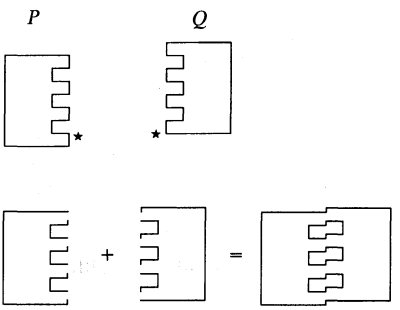


Figure 15. The combination of two maximal planes that share boundary lines. ⋆ is a reference point.

Case 2: *Suppose the planes overlap.*

There are two subcases to consider. The first case is when neither shape shares any boundary lines; the second case is when some boundary lines are shared.

Case 2.1: *The overlapping planes do not share boundary lines.*

In this case, the fact that the two shapes do not share any boundary lines can be established by an empty shape intersection of the boundary lines of the two polygons. Here, we separate the boundary lines of a polygon into two categories: those that lie inside the other polygon and those that lie outside. That is, we can define a routine that splits the lines of the polygon P with respect to another polygon Q (and vice versa) into two sets: I_P, comprising the lines of P inside Q; and O_P, comprising the lines of P outside Q. Figure 16 illustrates the inside and outside lines relative to two polygons.

Thus, if the two polygons, P and Q, overlap, are unequal, and do not share boundary lines, we can define a rule *split_polygons* that splits the boundaries of both P and Q into outer and inner fragments with respect to the other.

split_polygons$(P, Q, I_P, O_P, I_Q, O_Q, \varnothing) \leftarrow$
  ¶ an empty shape intersection establishes that two planes do not share
  ¶ boundary lines
  [*shape_intersection_in_U*$_1(P, Q, S) \wedge S = \varnothing] \wedge$
  ¶ split the boundary of a plane with respect to the other
  split_boundary$(P, Q, I_P, O_P) \wedge$
  split_boundary(Q, P, I_Q, O_Q).

The version of *split_boundary* described below splits unshared bounding lines of any polygon into two classes: outer fragments and inner fragments. A modified version that deals with shared and unshared bounding lines is presented in a subsequent section. Here we assume that the lines are expressed as classes of *co*-equal lines.

split_boundary$(\varnothing, Q, \varnothing, \varnothing)$. (SpB1)

split_boundary$([l_P]+P, Q, I, O) \leftarrow$
  ¶ split the class of *co*-equal lines into inner and outer fragments with respects
  ¶ to Q
  split_lines$([l_P], Q, [i], [o]) \wedge$
  ¶ repeat the process for the other lines in P
  split_boundary$(P, Q, I', O') \wedge$
  ¶ the fragment classes may be empty—so use append
  [*APPEND*$([i], I', I) \wedge$ *APPEND*$([o], O', O)]$. (SpB2)

Split_lines splits each line into fragments of the line that lie inside or outside Q. The procedure terminates when we have processed each class of lines in P.

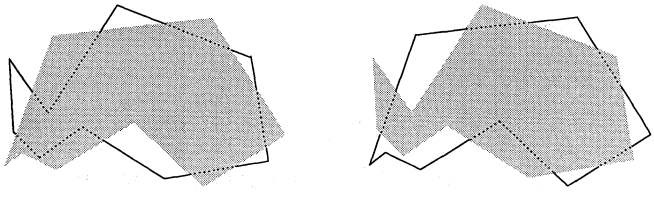


Figure 16. The inside and outside lines of a simple polygon with respect to another simple polygon.

split_lines($\varnothing$, Q, $\varnothing$, $\varnothing$). (SpL1)

split_lines($l+s_l$, Q, I, O) ←
¶ l is a line—find all points of intersection of l with the boundary of Q
¶ *cuts* is the number of cuts that the scan line corresponding to l makes with
¶ Q to one side of the first endpoint of l
intersection(l, Q, M, *cuts*) ∧
¶ insert only those points coincident with l
INSERT_POINTS(M, l, l') ∧
¶ if *cuts* is even or zero, first point of l is outside Q
¶ otherwise, the first point of l is inside Q
ifelse(*cuts mod* 2 = 0,
¶ l represents an alternating sequence of outer and inner fragments
outer(l', [i_l], [o_l]),
¶ else l represents an alternating sequence of inner and outer fragments
inner(l', [i_l], [o_l]) ∧
¶ repeat the process for the remaining lines
split_lines(s_l, Q, I', O') ∧
¶ the fragment classes may be empty—so use append
[*APPEND*([i_l], I',I) ∧ *APPEND*([o_l], O', O)]. (SpL2)

Split_lines applies a scan-line technique where the current line defines a scan-line. All points of intersection of the line with the boundary of the other polygon and coincident with the line are found. These points together with endpoints of the line are arranged as an ordered 'left-to-right' sequence of points. In addition, we compute the number of cuts of the scan-line with the boundary of the polygon which is to the 'left' side of or equal to the 'left' most endpoint of the line. As a consequence of the Jordan curve theorem, this endpoint is inner if the number of cuts is odd and outer, otherwise.

The coincident points of intersection between the current line and the lines in Q are determined by *intersection* as are the number of cuts of the scan-line to the left of the current line. *INSERT_POINTS* inserts the intersection points into the specification of a line. That is, if $\{p, q\}$ is a line which is coincident with a point of intersection $m \neq p, q$, then $\{p, q\}$ is replaced by the set $\{p, m, q\}$ to give rise to fragments $\{p, m\}$ and $\{m, q\}$. If there are two or more intersection points coincident with a line, then the points are arranged in order of their distance from the tail of the line. If $\{p, q\}$ are a pair of points that specify a line such that $p < q$, then tail p is the *left-most* point of the line. The definitions for *intersection* and *INSERT_POINTS* are left to the reader.

The predicates *inner* and *outer* alternatively mark successive fragments of the current line as inner or outer with respect to the other polygon determined from a sequence of intersection points between the endpoints of a side of a polygon.

inner($\{p\}$, $\varnothing$, $\varnothing$).
inner($\{p, q\}+S$, $\{p, q\}+I$, O) ← *outer*($\{q\}+S$, I, O).

outer($\{p\}$, $\varnothing$, $\varnothing$).
outer($\{p, q\}+S$, I, $\{p, q\}+O$) ← *inner*($\{q\}+S$, I, O).

We can now define the procedure to combine the boundaries of two simple maximal planes into the boundary of a single maximal plane when they overlap but do not contain one another and do not share boundary. The combination rule when one plane contains the other is trivial to define and is omitted.

simple_combine(P, Q, *Planes*) ←
  split_polygons(P, Q, I_P, O_P, I_Q, O_Q, $\varnothing$) ∧
  *shape_union_in_*U_1(O_P, O_Q, *Lines*) ∧
  maximal_planes(*Lines*, *Planes*). (SC1)

The second line of the rule states that the bounding lines of the combined shape is given by the union of the outer bounding lines of the two shapes.

Case 2.2: *Planes overlap and share boundary lines.*

Since the shared fragments correspond to the shape intersection of the two boundary lines and other lines belong to the shape differences of the two shapes, we can combine these algorithms together to split the boundaries of the two polygons.

split_polygons(P, Q, I_P, O_P, I_Q, O_Q, S) ←
  ¶ a nonempty shape intersection establishes that two planes share boundary
  ¶ lines
  [*shape_intersection_in_*U_1(P, Q, S) ∧ $S \neq \varnothing$] ∧
  ¶ obtain the nonshared fragments
  *shape_difference_in_*U_1(P, S, P') ∧
  *shape_difference_in_*U_1(Q, S, Q') ∧
  ¶ split the nonshared boundaries of the two planes with respect to the other
  split_boundary(P', Q, O_P, I_P, $\varnothing$) ∧
  split_boundary(Q', P, O_Q, I_Q, $\varnothing$).

In this case, the fragments of lines that coincide with a bounding line can be considered as outer or inner lines depending on the lines adjacent to the fragments. As the common fragment lies on simple polygons, each endpoint of the fragment is connected to exactly two lines, one from each polygon. These lines may themselves be outer, inner, or shared. So, a shared fragment may be regarded as outer to one of the polygons and inner to the other.

If the two shapes are equal, all lines will be shared by the two polygons; otherwise there is at least one endpoint of a boundary line that lies outside the other polygon and there is at least one endpoint of a boundary line that lies inside the other polygon. However, the equality case is handled by the shape_union algorithm (see rule CU3 and in particular, by rule CU3a, in Krishnamurti, 1992).

The possible neighborhood situations surrounding a shared fragment are illustrated in figure 17. Shared fragments are depicted by the thicker lines and inner fragments are depicted by dashed lines. In general, the lines in the upper half of each diagram belong to polygon P and the lines in the lower half belong to polygon Q. The central thick line indicates the shared fragment under consideration.

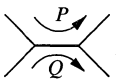
The classification for shared lines for combining two simple planes can now be described. A shared fragment is included in the union of the two planes if and only if its endpoints are not adjacent to two outer fragments one from each plane; otherwise, we must include the shared fragment in order to complete a traversal of the boundary of the combined plane. In figure 17, this means that the shared fragments shown in cases (a), (b), and (f) are omitted in the union of the two polygons whereas those shown in cases (c) and (e) are included. The shared fragment in case (d) is included if and only if one of the adjacent shared fragment is also included. As all fragments will be shared if and only if the two polygons are equal, for unequal polygons there will be a nonshared fragment adjacent to a shared fragment.

classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, $\varnothing$, $\varnothing$). (CSLU1)

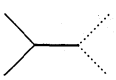
classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, $\{l\}+S$, $\{l\}+SU$) ←
¶ if one endpoint is adjacent to a pair of inner-outer fragments, it is included
¶ for union
[*at_one_end*(l, O_P, I_Q) ∨ *at_one_end*(l, I_P, O_Q)] ∧
classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, S, SU). (CSLU2)

classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, $\{l\}+S$, SU) ←
¶ if both endpoints p, q of a shared fragment are adjacent to shared fragments,
¶ postpone decision until later
connected_to(l, S, S) ∧
APPEND(S, $\{l\}$, S') ∧
classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, S', SU). (CSLU3)

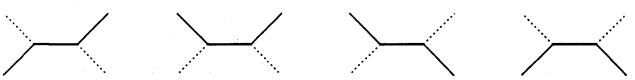
classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, $\{l\}+S$, $\{l\}+SU$) ←
classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, S, SU) ∧
¶ if one of the endpoints of a shared fragment is adjacent to a shared
¶ fragment,
¶ and the other to an included line or both are adjacent to included lines,
¶ include this line
[*connected_to*(l, S, SU) ∨ *connected_to*(l, SU, SU)]. (CSLU4)



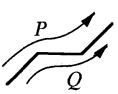
(a) shared line incident to outer lines



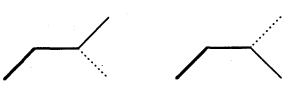
(b) shared line incident to outer lines at one endpoint and inner lines at the other



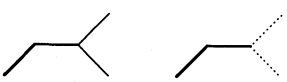
(c) shared line incident to a pair of inner-outer lines at each endpoint



(d) shared line incident to shared lines



(e) shared line incident to a shared line at one endpoint and a pair of inner-outer lines at the other



(f) shared line incident to a shared line at one endpoint and inner or outer lines at the other

Figure 17. Possible neighborhood spatial situations at shared boundary lines. The central thick line denotes the shared fragment under consideration. In general, the lines on the upper half belong to polygon P and lines in the lower half belong to polygon Q.

classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, $\{l\}+S$, SU) ←
¶ otherwise, shared fragment is not included
not[*at_one_end*(l, O_P, I_Q) ∨ *at_one_end*(l, I_P, O_Q)] ∧
not connected_to(l, S, S) ∧
classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, S, SU). (CSLU5)

At_one_end is a predicate that determines if one of the endpoints of the shared line is adjacent to two lines each taken from distinct sets of lines. *Connected_to* determines if both endpoints of the shared line are each adjacent to a line from two sets. Both *at_one_end* and *connected_to* rely on *adjacent_to* which determines if a point is an endpoint of a line in a list.

at_one_end($\{p, q\}$, L_1, L_2) ←
¶ a line is given by its endpoints $\{p, q\}$
[*adjacent_to*(p, L_1) ∧ *adjacent_to*(p, L_2)] ∨
[*adjacent_to*(q, L_1) ∧ *adjacent_to*(q, L_2)].

connected_to($\{p, q\}$, L_1, L_2) ←
¶ a line is given by its endpoints $\{p, q\}$
[*adjacent_to*(p, L_1) ∧ *adjacent_to*(q, L_2)] ∨
[*adjacent_to*(q, L_1) ∧ *adjacent_to*(p, L_2)].

adjacent_to(p, L) ←
MEMBER($\{p, r\}$, L) ∨ *MEMBER*($\{r, p\}$, L).

The result of classifying shared boundary lines for the union of two simple planes is illustrated by the example in figure 18, where some shared fragments are included in the union and others are not, depending on the lines adjacent to them.

We can now define the procedure to combine the boundaries of two simple maximal planes into the boundary of a single maximal plane when they overlap and share boundary lines.

simple_combine(P, Q, *Planes*) ←
split_polygons(P, Q, I_P, O_P, I_Q, O_Q, S) ∧
classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, S, SU) ∧
shape_union_in_U$_1$(O'_P, O'_Q, O_{PQ}) ∧
shape_union_in_U$_1$(SU, O_{PQ}, *Lines*) ∧
maximal_planes(*Lines*, *Planes*). (SC2)

Observe that the single rule SC2 subsumes rule SC1 (when $S = \varnothing$) and the definition for *simple_share_combine* (when $I_P = I_Q = \varnothing$).

We can achieve the combination of simple planes including the classification of shared fragments via boundary traversal rather than by the scan-line techniques described above. An outline of the boundary-traversal approach for combining simple convex maximal planes is described in the next section.

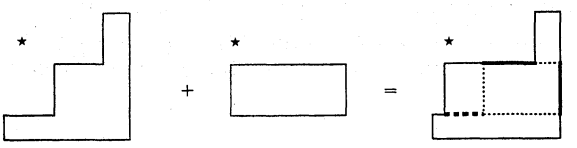


Figure 18. Classifying shared boundary lines when combining simple planes. ⋆ is a reference point. Shared fragments are shown as emboldened lines.

Combining overlapping simple convex maximal planes via boundary traversal

A *convex* plane is one whose boundary is a convex polygon. In this section we describe a boundary (polygon) traversal algorithm for combining convex maximal planes. The convexity of the planes will guarantee that the combined plane is simple but not necessarily convex. The crux of the algorithm is to construct the boundary of the combined maximal plane by extending a partial path made up of fragments of bounding lines. At each stage in the traversal, we determine which part of a bounding line the traversal must advance along. Obviously, the fragment of the line must lie outside the other polygon. By successively following a connected sequence of lines that lie outside the other polygon we can determine the boundary of the combined maximal line.

Let p be the left-most bottom corner of a simple bounding polygon; p must be the tail of two lines. Let q be the head of one of the lines. If we follow line $\{p, q\}$ and then follow the other line connected to q and then follow the lines likewise connected, we obtain a traversal of the polygon. That is, we can determine a sequence of points for any polygon starting at the left-most point. If q is the head of the line with the smaller gradient, the traversal is counterclockwise; otherwise, the traversal is clockwise (see figure 19).

For any two simple polygons, either one of their left-most bottom corners lies outside the other polygon or both corners may coincide. If they do not coincide, then both may lie outside the other polygon in which case we choose the left-most of the two. If they do coincide, we choose the polygon which has a line with the smaller of the two gradients. (Of course, if the endpoints points of this line coincide, we repeat the process with the endpoint of this line until we find a line with a smaller gradient.) Starting at this corner, say p_0 in polygon P, we traverse the boundary along the line $\{p_0, p_1\}$ keeping the inside of the plane to the left or right of the line, depending on whether the traversal is counterclockwise or not. Three cases arise: (1) the line does not intersect or overlap any other line; (2) the line intersects a bounding line of the other plane; (3) the line overlaps a bounding line of the other plane.

Suppose we traverse the boundary counterclockwise. The sequence of points that specify the bounding polygon determines a counterclockwise traversal. Suppose $\{p_{i-1}, p_i\}$ is the current edge and P the current polygon.

For case (1), both p_{i-1} and p_i must lie outside Q. We add $\{p_{i-1}, p_i\}$ onto the current path, which can be maintained as a stack, and advance along $\{p_i, p_{i+1}\}$.

For case (2), suppose $\{q_{j-1}, q_j\}$ is the line that intersects with $\{p_{i-1}, p_i\}$ at m, then we have reached an alternating point and we have to traverse along the boundary of the other plane. Because we are traversing the boundary counterclockwise, the inside of P is to the left of $\{p_{i-1}, p_i\}$. Moreover, q_{j-1} must be inside P or be coincident with $\{p_{i-1}, p_i\}$. Otherwise, we would have intersected the current path earlier and the traversal would be advancing along $\{q_{j-1}, q_j\}$ instead. There are four spatial situations that can arise, depending on whether p_{i-1} is the tail or head of line $\{p_{i-1}, p_i\}$. These are illustrated in figure 20. Though not illustrated in figure 20, m may equal q_{j-1} without altering the spatial situations. In this case, we add $\{p_{i-1}, m\}$ to the path and advance along $\{m, q_j\}$, making Q the current polygon.

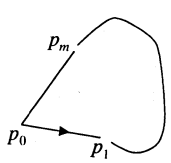


Figure 19. Starting a traversal of a polygon at its left-most bottom corner.

For case (3), there are three subcases to consider. If $q_j > p_i$, we add $\{p_{i-1}, q_j\}$ to the path and advance along $\{q_j, q_{j+1}\}$, making Q the current polygon. If $p_i > q_j$, we add side $\{p_{i-1}, p_i\}$ to the path and advance along $\{p_i, p_{i+1}\}$. If $p_i = q_j$, we have to determine whether p_{i+1} lies outside Q, or q_{j+1} lies outside P. Only one of these points will do so. We add $\{p_{i-1}, p_i\}$ to the path and advance along the appropriate bounding line, switching the current polygon if necessary.

The algorithm terminates when we return to the start point of the traversal.

If the planes are not convex, the algorithm will find the outer boundary of the combined plane. There may remain some processing to determine the inner boundaries.

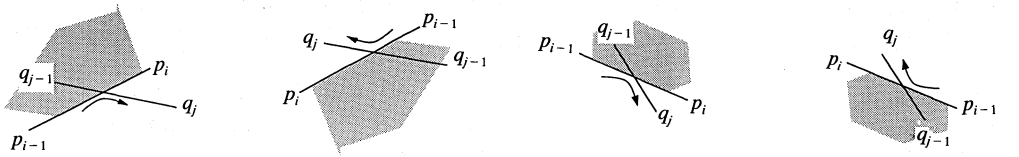


Figure 20. The four intersection situations that can arise. The shaded part indicates the inside region.

Intersection of two simple maximal planes

The intersections of two simple maximal planes are the planes common to both planes. As before we can approach the problem as a boundary traversal algorithm or as the shape union of specific bounding lines. For convex simple planes we can adapt a boundary traversal algorithm described in Preparata and Shamos (1985). Here, we define an algorithm based on shape operations on sets of lines. Figure 21 illustrates the intersection of simple maximal planes. The reader will notice that the shape common to the two simple planes is a shape made up of three maximal planes.

As before, we split the boundary of each plane into three sets, one containing lines lying outside and the other containing lines lying inside the other plane and shared fragments. Now the intersection of the two planes consists of the inside lines of both planes together with certain shared lines. Thus,

simple_common(P, Q, *Planes*) ←
  split_polygons(P, Q, I_P, O_P, I_Q, O_Q, S) ∧
  classify_shared_lines_for_intersection(I_P, O_P, I_Q, O_Q, S, SI) ∧
  shape_union_in_U$_1$(I_P, I_Q, L) ∧
    shape_union_in_U$_1$(L, SI, *Lines*) ∧
  maximal_planes(*Lines*, *Planes*).

Note that, as figure 21 illustrates, the intersection may yield more than one maximal plane. The routine *maximal_planes* will take a set of lines and return a set of disjoint maximal planes. For plane intersection, the set of lines will form a self-intersecting polygon and the routine *isolate_planes*, suitably modified, can be used instead.

Because the planes common to the two original planes are bounded by lines of the planes inner with respect to the other, we include just those shared fragments

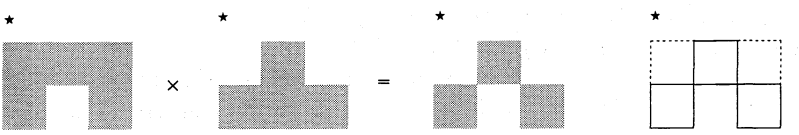


Figure 21. The intersection of two simple maximal planes. ⋆ denotes a reference point.

whose endpoints are each adjacent to exactly one inner fragment or whose endpoints are adjacent to shared fragments one of which has been deemed to be included. In other words, the shared fragments that are included for intersection are precisely those that are included for union [see figure 17(c), (d) and (e)].

classify_shared_lines_for_intersection(I_P, O_P, I_Q, O_Q, S, SI) ←
  classify_shared_lines_for_union(I_P, O_P, I_Q, O_Q, S, SI). (CSLI1)

The rule for plane intersection returns a nonempty plane only when the two planes overlap or when one contains the other.

Relative complement of a simple maximal plane with respect to another

Like plane intersection, the relative complement of one maximal plane with respect to the other may produce more than one maximal plane. But unlike plane intersection, the planes in the relative difference may not be simple. That is, they may contain holes. Thus, each plane in the difference is specified by an outer boundary and zero or more inner boundaries.

The relative complement algorithm is approached in a similar fashion to union and intersection. The boundary of each plane is split into three sets, one containing lines lying outside, one containing lines lying inside the other plane, and one containing shared bounding lines. Now the difference of the two planes consists of the outer lines of the first plane, the inner lines of the second and some shared lines.

simple_complement(P, Q, *Planes*) ←
  split_polygons(P, Q, I_P, O_P, I_Q, O_Q, S) ∧
  classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, S, SD) ∧
  shape_union_in_U$_1$(O_P, I_Q, L) ∧
    shape_union_in_U$_1$(L, SD, *Lines*) ∧
  maximal_planes(*Lines*, *Planes*).

Shared fragments are bounding lines in the shape difference only if either an endpoint is adjacent to two outer or to two inner fragments [see figure 17(a), (b) and (f)], or both endpoints are adjacent to shared fragments one of which has been deemed to be included [see figure 17(d)].

classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, ∅, ∅). (CSLD1)

classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, $\{l\}+S$, $\{l\}+SD$) ←
  ¶ if l is adjacent to outer(inner) fragments at one endpoint, it is included for
  ¶ difference
  [*at_one_end*(l, O_P, O_Q) ∨ *at_one_end*(l, I_P, I_Q)] ∧
  classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, S, SD). (CSLD2)

classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, $\{l\}+S$, SD) ←
  ¶ if both endpoints p, q of a shared fragment are adjacent to shared fragments,
  ¶ postpone decision until later
  connected_to(l, S, S) ∧
  APPEND(S, $\{l\}$, S') ∧
  classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, S', SD). (CSLD3)

classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, $\{l\}+S$, $\{l\}+SD$) ←
  classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, S, SD) ∧
  ¶ if one of the endpoints of a shared fragment is adjacent to a shared fragment,
  ¶ and the other to an included line or both are adjacent to included lines,
  ¶ include this line
  [*connected_to*(l, S, SU) ∨ *connected_to*(l, SU, SU)]. (CSLD4)

classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, $\{l\}+S$, SD) ←
¶ otherwise, it is not included for difference
not[*at_one_end*(l, O_P, O_Q) ∨ *at_one_end*(l, I_P, I_Q)] ∧
not connected(l, S, S) ∧
classify_shared_lines_for_difference(I_P, O_P, I_Q, O_Q, S, SD). (CSLD5)

Simple_complement subsumes the case when P contains Q. Figure 22 shows three examples of the difference of two planes and the outline of the bounding lines that make up the boundaries of resultant shape difference. The examples illustrate the possibilities that may arise from comparing two simple planes. In the first example, three maximal planes result; in the second, the result remains a simple plane; and in the third, the shape difference is a single maximal plane with an inner boundary.

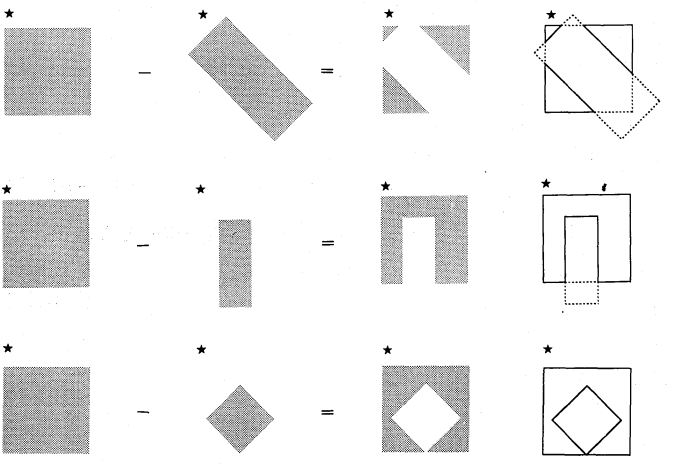


Figure 22. Maximal planes formed by the difference of two simple maximal planes. ⋆ denotes a reference point.

Unified treatment of shared fragments for shape operations on simple maximal planes
We have just seen that shared fragments are classified for inclusion differently, depending on the type of operation and the neighborhood of the shared fragments. Although this classification is fine for operations on simple planes, it adds a measure of difficulty when considering operations on nonsimple maximal planes. A better approach is to classify the shared fragments in a uniform way independent of the underlying operation and to select just those fragments that have to be included for the specific operation. In this section, a uniform classification of shared fragments is considered.

Consider any maximal line of the boundary of a polygon and compare it against the boundary of another polygon. The line can be fragmented by points of intersection into three classes: inner fragments, outer fragments, and shared fragments. Assume that the endpoints of any line, $\{p, q\}$, are arranged in lexicographical, termed *left-to-right*, order such that $p < q$. Consider the left-most endpoint of the first shared fragment. It is adjacent to an inner fragment, or an outer fragment, or it is the first fragment in the set of fragments that make up the line. Consider the line itself and treat it as a scan-line. Count the number of time it cuts the boundary of the other polygon at points less than, termed to the *left* of, or equal to the least endpoint of a fragment of the line. If the number of cuts is even or zero, the first fragment must be outer or is shared inside on the *right*.

If it is odd, the first fragment must be inner or shared inside on the *left*. That is, we can split each line of the boundary into inner and outer fragments, or *left-shared* or *right-shared* fragments. Observe that a left-shared fragment is a fragment that would have been deemed inner by considering an alternating sequence of fragments from left to right. Likewise, a right-shared fragment is a fragment that would have been deemed outer by an alternating sequence of fragments from left to right.

Thus, when comparing two simple planes we split the boundary of each plane with respect to the other into inner, outer, left-shared, and right-shared fragments. The following modifications to *split_boundary* and *split_lines* accomplish this.

¶ *split_boundary* splits the bounding lines of a plane into inner, outer, left-shared,
¶ and right-shared fragments with respect to another plane
split_boundary$(\varnothing, Q, \varnothing, \varnothing, \varnothing, \varnothing)$. (SpB1*)

split_boundary$([l_P]+P, Q, I, O, L, R) \leftarrow$
  split_boundary$(P, Q, I', O', L', R') \wedge$
  split_lines$([l_P], Q, [i], [o], [l], [r]) \wedge$
  APPEND$([i], I', I) \wedge$
  APPEND$([o], O', O) \wedge$
  APPEND$([l], L', L) \wedge$
  APPEND$([r], R', R)$. (SpB2*)

¶ *split_lines* splits a class of *co*-equal lines of a plane into inner, outer, left-shared,
¶ and right-shared fragments with respect to the other plane
split_lines$(\varnothing, Q, \varnothing, \varnothing, \varnothing, \varnothing)$. (SpL1*)

split_lines$(l+s_l, Q, I, O, L, R) \leftarrow$
  ¶ this part is basically unchanged
  split_lines$(s_l, Q, I', O', L', R') \wedge$
  intersection$(l, Q, M, cuts) \wedge$
    INSERT_POINTS$(l, M, l') \wedge$
  ifelse(*left mod* 2 = 0, *outer*$(l', [i_l'], [o_l'])$,
                inner$(l', [i_l'], [o_l'])) \wedge$
  ¶ this part includes the changes
  ¶ compute the shared fragments on l, if any
  shape_intersection_in_U$_1(\{l\}, Q, S) \wedge$
  ¶ isolate the left-shared and right-shared fragments of l
  shape_intersection_in_U$_1([i_l'], S, [l_l]) \wedge$
    shape_intersection_in_U$_1([o_l'], S, [r_l]) \wedge$
  ¶ isolate the inner and outer fragments of l
  shape_diffference_in_U$_1([i_l'], [l_l], [i_l]) \wedge$
    shape_difference_in_U$_1([o_l'], [r_l], [o_l]) \wedge$
  ¶ the fragment classes may be empty—so use append
  $[$*APPEND*$([i_l], I', I) \wedge$ *APPEND*$([o_l], O', O)] \wedge$
  $[$*APPEND*$([l_l], L', L) \wedge$ *APPEND*$([r_l], R', R)]$. (SpL2*)

The predicates *inner* and *outer* have been previously defined.

We can now consider the descriptions for the shape operations on two simple planes. If we examine figure 17, we notice that the shared fragments that are included for union and intersection correspond to those that are identically classified with respect to the boundary of the other plane, and for difference the included

shared fragments correspond to those that are oppositely classified. Thus,

simple_combine$(P, Q, Planes) \leftarrow$
split_boundary$(P, Q, I_P, O_P, L_P, R_P) \wedge$
split_boundary$(Q, P, I_Q, O_Q, L_Q, R_Q) \wedge$
shape_intersection_in_U$_1(L_P, L_Q, L_{PQ}) \wedge$
shape_intersection_in_U$_1(R_P, R_Q, R_{PQ}) \wedge$
shape_union_in_U$_1(L_{PQ}, R_{PQ}, S) \wedge$
shape_union_in_U$_1(O_P, O_Q, L) \wedge$
shape_union_in_U$_1(L, S, Lines) \wedge$
maximal_planes$(Lines, Planes)$.

simple_common$(P, Q, Planes) \leftarrow$
split_boundary$(P, Q, I_P, O_P, L_P, R_P) \wedge$
split_boundary$(Q, P, I_Q, O_Q, L_Q, R_Q) \wedge$
shape_intersection_in_U$_1(L_P, L_Q, L_{PQ}) \wedge$
shape_intersection_in_U$_1(R_P, R_Q, R_{PQ}) \wedge$
shape_union_in_U$_1(L_{PQ}, R_{PQ}, S) \wedge$
shape_union_in_U$_1(I_P, I_Q, L) \wedge$
shape_union_in_U$_1(L, S, Lines) \wedge$
maximal_planes$(Lines, Planes)$.

simple_complement$(P, Q, Planes) \leftarrow$
split_boundary$(P, Q, I_P, O_P, L_P, R_P) \wedge$
split_boundary$(Q, P, I_Q, O_Q, L_Q, R_Q) \wedge$
shape_intersection_in_U$_1(L_P, R_Q, L_{PQ}) \wedge$
shape_intersection_in_U$_1(R_P, L_Q, R_{PQ}) \wedge$
shape_union_in_U$_1(L_{PQ}, R_{PQ}, S) \wedge$
shape_union_in_U$_1(O_P, I_Q, L) \wedge$
shape_union_in_U$_1(L, S, Lines) \wedge$
maximal_planes$(Lines, Planes)$.

Shape operations on maximal planes

The preceding discussion pertains to simple maximal planes. However, it is not unreasonable to expect objects of interest to contain indentations or holes. That is, the planes of interest are nonsimple. The inclusion of inner boundaries adds a measure of difficulty in defining shape operations. The situation when the maximal planes are nonsimple is dealt with in the next three sections.

Shape union

Figure 23 illustrates examples of shape union of two arbitrary planes.

Consider the two planes P and Q, one of which is nonsimple. We may assume that they can combine; that is, if P and Q overlap, share bounding lines, or one contains the other. In addition, the planes combine in different ways, depending on how Q overlaps the holes in P.

Suppose $P = \{o_P, i_P\}$, and $Q = \{o_Q\}$. Let $\dot{+}$, $\dot{-}$ and $\dot{\times}$ respectively denote general union, difference, and intersection operators on planes or holes defined by simple polygons. We add the proviso that, if the polygons just share boundaries or are otherwise disjoint, the intersection operation produces an empty plane, and the difference operation returns the first plane. When the planes are disjoint, the union operation returns the two planes.

The shape union of P and Q is given by,

$$P+Q = \begin{cases} \{\{o_P, i_P\}, \{o_Q\}\}, & \text{if } P \text{ and } Q \text{ are disjoint,} \\ \{\{(o_P \dot{+} o_Q), (i_P \dot{-} o_Q)\}\}, & \text{if } P \text{ and } Q \text{ overlap or if } P \text{ contains } Q, \\ \{\{(o_P \dot{+} o_Q), i_P\}\}, & \text{if P and } Q \text{ share boundary.} \end{cases}$$

The union of two overlapping planes is a simple plane if Q covers the hole in P completely and is nonsimple otherwise. In the latter case, the boundary of the hole in the shape union is given by the relative complement of the boundary of the hole (considered as a simple plane) and the outer boundary of Q. The overlapping case subsumes the containment case.

If P contains Q, then $(o_P \dot{+} o_Q) = o_P$ and $(i_P \dot{-} o_Q) = i_P$. When the two planes share bounding lines, the inner boundary of P remains an inner boundary in the union of the two planes, because any inner boundary of P and the outer boundary of Q at most share boundary lines.

Suppose P has more than one inner boundary. That is, P is given by the set of boundaries, $\{o_P, i_{P,1}, i_{P,2}, \ldots, i_{P,j}, \ldots\}$. Then,

$$P+Q = \{(o_P \dot{+} o_Q), (i_{P,1} \dot{-} o_Q), (i_{P,2} \dot{-} o_Q), \ldots, (i_{P,j} \dot{-} o_Q), \ldots\}.$$

Some of the differences will be empty and these can be ignored. We can prune the number of differences that need to be performed if the inner boundaries of P are suitably ordered.

Suppose Q is also nonsimple. That is, Q is given by the set of boundaries, $\{o_Q, i_{Q,1}, i_{Q,2}, \ldots, i_{Q,k}, \ldots\}$. Then,

$$\begin{aligned} P+Q = \{&(o_P \dot{+} o_Q), \\ &(i_{P,1} \dot{-} o_Q), (i_{P,2} \dot{-} o_Q), \ldots, (i_{P,j} \dot{-} o_Q), \ldots \\ &(i_{Q,1} \dot{-} o_P), (i_{Q,2} \dot{-} o_P), \ldots, (i_{Q,k} \dot{-} o_P), \ldots \\ &(i_{P,1} \dot{\times} i_{Q,1}), (i_{P,1} \dot{\times} i_{Q,2}), \ldots, (i_{P,j} \dot{\times} i_{Q,k}), \ldots\}, \end{aligned}$$

with the proviso that, if an inner boundary and an outer boundary do not overlap, their intersection is empty and their difference is the inner boundary. The outer boundary of the shape union is given by the union of the outer boundaries of the two planes together with shared lines that are identically classified. The inner boundaries are given by the relative complements of the holes in one plane and the other plane, and by the holes common to both planes. Again, some of the differences and intersections will be empty and these can be ignored. Here, the number of differences and intersections to be performed can be reduced by ordering the boundaries in P and Q in a suitable fashion.

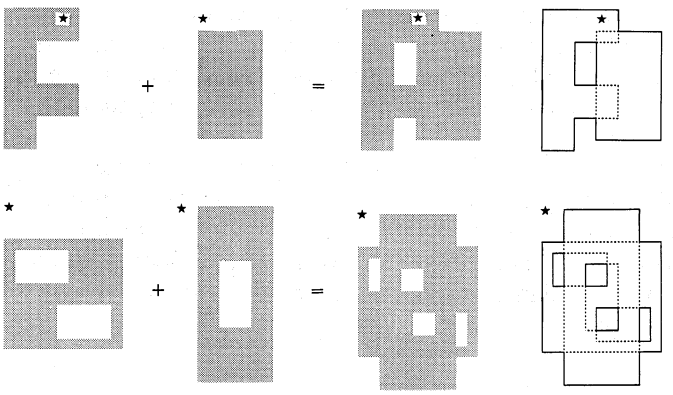


Figure 23. Examples of shape union of two arbitrary planes. ⋆ denotes the reference point.

Thus, the shape union of two arbitrary planes can be expressed in terms of union, difference, and intersection operations on simple planes.

From the discussion above it is clear that the union of two arbitrary planes can be treated in an analogous fashion to the union of two simple planes. Consider the plane formed by the sum $(o_P \dot{+} o_Q)$. Its boundary is made up of lines in o_P outside o_Q, and of lines in o_Q outside o_P. Equally, the hole formed by $(i_{P,j} \dot{-} o_Q)$ is made up of lines in $i_{P,j}$ outside o_Q, and of lines in o_Q inside $i_{P,j}$. The hole formed by $(i_{Q,k} \dot{-} o_P)$ is similarly defined. Any common holes $(i_{P,j} \dot{\times} i_{Q,k})$ are formed by lines in $i_{P,j}$ inside $i_{Q,k}$ and by lines in $i_{Q,k}$ inside $i_{P,j}$. In other words, we split the bounding lines of each plane into inner, outer, and shared lines with respect to the other plane, bearing in mind that the region inside an inner boundary is outside the plane. Thus,

combine(P, Q, *Planes*) ←
  ¶ split the boundaries of P and Q into inner, outer, left-shared, and right-
  ¶ shared fragments
  split_plane(P, Q, I_P, O_P, L_P, R_P) ∧
  split_plane(Q, P, I_Q, O_Q, L_Q, R_Q) ∧
  ¶ consider just the common left-shared and right-shared fragments for union
  shape_intersection_in_U$_1$(L_P, L_Q, L_{PQ}) ∧
  shape_intersection_in_U$_1$(R_P, R_Q, R_{PQ}) ∧
  MERGE(L_{PQ}, R_{PQ}, L')[9] ∧
  ¶ boundary of the union is made up of outer fragments and
  ¶ shared fragments of the same type
  shape_union_in_U$_1$(O_P, O_Q, L) ∧
    shape_union_in_U$_1$(L, L', *Lines*) ∧
  maximal_planes(*Lines*, *Planes*).

Split_plane is similar to *split_boundary* except each line of each boundary polygon in a plane is split into fragments that lie inside or outside the other plane which may not be simple, or into fragments that are left-shared or right-shared.

split_plane($\varnothing$, Q, $\varnothing$, $\varnothing$, $\varnothing$, $\varnothing$). (SpP11)

split_plane($OB_P + P$, $OB_Q + Q$, I, O, L, R) ←
  split_boundary(OB_P, OB_Q, I_P, O_P, L_P, R_P) ∧
  split_lines_wrt_holes(I_P, Q, I'_P, O'_P, L'_P, R'_P) ∧
  split_plane(P, $OB_Q + Q$, I''_P, O''_P, L''_P, R''_P) ∧
  APPEND(O_P, I'_P, O''_P, O)[10] ∧
  APPEND(O'_P, I''_P, I) ∧
  APPEND(L_P, R'_P, L''_P, L) ∧
  APPEND(R_P, L'_P, R''_P, R). (SpP12)

The rules need a little explanation. Essentially the algorithm iterates for each boundary of one of the planes, say P, and compares it (using *split_boundary*) with the outer boundary of the other, say Q. The lines of the boundary fall into three categories: those outside the outer boundary of Q, those inside the outer boundary of Q, and those that share boundary fragments. The inner bounding lines may now be further fragmented into lines that lie inside, outside, and on the inner boundaries of Q. This is done by invoking the routine *split_lines_wrt_holes*.

[9] *MERGE* merges two lists into a single list and a naive version is described in Krishnamurti (1992).
[10] *APPEND*(X, Y, Z, W) is shorthand for the conjunction *APPEND*(X, Y, U) ∧ *APPEND*(U, Z, W).

There are two points to note. First, the inner boundaries of Q are, at best, point connected and consequently, any fragment of a bounding lines inside a hole in Q cannot also be inside another hole in Q. Second, any fragment of an inner line inside a hole in Q lies outside Q. This is why the inner fragments (with respect to the holes) of the bounding lines are concatenated with the outer fragments (with respect to the outer boundary of Q) as indicated by the first *APPEND* statement. For the same reason, the fragments outer to the holes are concatenated with the inner fragments with respect to the outer boundary, indicated by the second *APPEND* statement. Similarly, the left-shared fragments of the boundaries of P (with respect to the outer boundary of Q) are concatenated with the right-shared fragments of the boundaries of P with respect to the holes of Q (and vice versa). The first rule (SpPl1) is a terminating rule which only applies when we have iterated through the boundaries of P.

split_lines_wrt_holes$(\varnothing, Q, \varnothing, \varnothing, \varnothing, \varnothing)$. (SpLH1)

split_lines_wrt_holes$(Lines, \varnothing, \varnothing, Lines, \varnothing, \varnothing) \leftarrow$
$Lines \neq \varnothing$. (SpLH2)

split_lines_wrt_holes$(Lines, H+Q, I, O, L, R) \leftarrow$
$Lines \neq \varnothing \wedge$
split_boundary$(Lines, H, I_H, O_H, L_H, R_H) \wedge$
split_lines_wrt_holes$(O_H, Q, I', O, L', R') \wedge$
APPEND$(I_H, I', I) \wedge$
APPEND$(L_H, L', L) \wedge$
APPEND(R_H, R', R). (SpLH3)

The rules are straightforward. The first two rules are terminating rules which apply either when there are no more lines to be fragmented or when Q has no inner boundaries. The third rule splits the current set of lines into four categories: fragments inside a hole, fragments outside a hole, left-shared fragments, and right-shared fragments. As a fragment cannot be inside or on two holes, we need only consider the outer fragments with the respect to the remaining holes. The process is repeated until all the holes in Q are considered.

The rules for *combine* hold even when the two maximal planes are simple, that is, when the planes have no holes.

Shape difference

Expressions can be derived for shape difference of two arbitrary maximal planes in much the same way as shape union. The shape difference of two planes $P-Q$ can be constructed by iteratively removing holes from the plane formed by the relative difference of the outer boundary of P and the plane Q. Examples of shape difference are shown in figure 24. The holes correspond to the relative difference of the holes in P and the outer boundary of Q. Note that the difference between a plane and a hole is a plane whereas that between a hole and a plane is a hole. Suppose $P = \{o_P, i_P\}$ and $Q = \{o_Q\}$. Then, the shape difference is given by,

$$P-Q = \{(o_P \dot{-} o_Q) \dot{-} (i_P \dot{-} o_Q)\}$$

If P has more than one hole, then the shape difference is given by

$$P-Q = \{(o_P \dot{-} o_Q) \dot{-} (i_{P,1} \dot{-} o_Q) \dot{-} (i_{P,2} \dot{-} o_Q) \ldots (i_{P,j} \dot{-} o_Q) \ldots\}$$

Here the individual differences $(i_{P,j} \dot{-} o_Q), j \geqslant 1$, are all disjoint. Hence, if for some $j > 0$, $(o_P \dot{-} o_Q) \dot{-} (i_{P,j} \dot{-} o_Q)$ is nonsimple, that is, we may suppose its boundaries are given by $\{O_P, I_{P,1}, I_{P,2}, \ldots\}$, then, $(o_P \dot{-} o_Q) \dot{-} (i_{P,j} \dot{-} o_Q) \dot{-} (i_{P,j+1} \dot{-} o_Q)$ is given by

one of the following sets of boundaries:

$$\{O_P \dot{-} (i_{P,j+1} \dot{-} o_Q), I_{P,1}, I_{P,2}, ...\},$$

or

$$\{O_P, I_{P,1} \dot{+} (i_{P,j+1} \dot{-} o_Q), I_{P,2}, ...\},$$

or

$$\{O_P, I_{P,1}, I_{P,2} \dot{+} (i_{P,j+1} \dot{-} o_Q), ...\},$$

and so on depending on which boundary shares a line or overlaps with $(i_{P,j+1} \dot{-} o_Q)$; only one of them will. Note that the above expression holds only if the shape difference does not produce multiple maximal planes.

If Q is nonsimple, then each of its inner boundaries potentially helps define a maximal plane. That is, $i_{Q,k} \dot{\times} o_P$ defines a plane if $i_{Q,k}$ and o_P overlap. However, this plane may overlap with holes in P. That is, for each such nonempty plane $i_{Q,k} \dot{\times} o_P$, we take its difference with the nonempty common holes $(i_{Q,k} \dot{\times} i_{P,j})$, for all $j > 0$.

The expression for shape difference thus becomes complex if we follow the above procedure to completion. With a little bit of shape arithmetical magic(!), it turns out to be an expression of the form:

$$P - Q = \{((o_P \dot{-} o_Q) \dot{+} (o_P \dot{\times} (i_{Q,1} \dot{+} i_{Q,2} \dot{+} ...))) \dot{-} (i_{P,1} \dot{+} i_{P,2} \dot{+} ...)\}$$

However, it is clear that, as in the case of shape union, we can split the boundary lines of the two planes into inner, outer, and shared lines with respect to the other plane. Here, the boundary lines in the shape difference are made up of lines in P outer with respect to Q, lines in Q inner with respect to P, and the shared lines that are oppositely classified. The algorithm for shape difference of two arbitrary

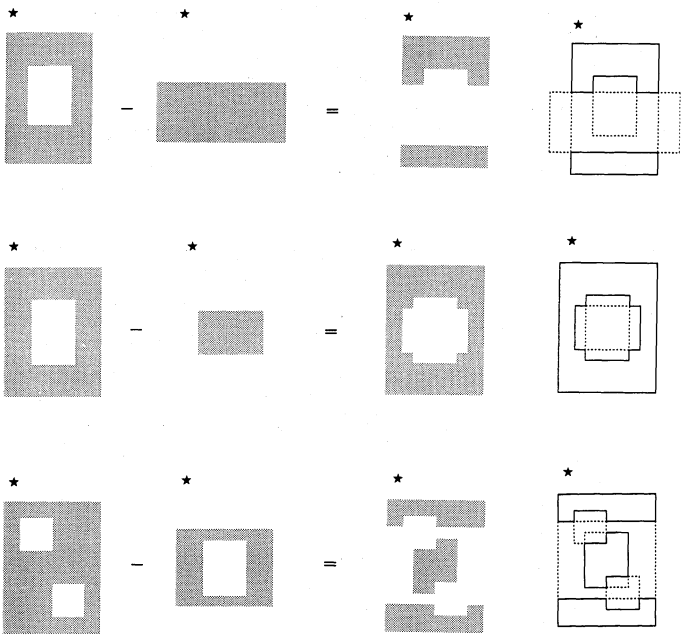


Figure 24. Examples of shape difference of two maximal planes. ⋆ denotes the reference point.

planes is:

complement(P, Q, *Planes*) ←
¶ split the boundaries of P and Q into inner, outer, left-shared, and right-
¶ shared fragments
split_plane(P, Q, I_P, O_P, L_P, R_P) ∧
split_plane(Q, P, I_Q, O_Q, L_Q, R_Q) ∧
¶ consider just the oppositely classified shared fragments for difference
shape_intersection_in_U$_1$(L_P, R_Q, L_{PQ}) ∧
shape_intersection_in_U$_1$(R_P, L_Q, R_{PQ}) ∧
MERGE(L_{PQ}, R_{PQ}, L') ∧
¶ boundary of the union is made up of outer fragments of P and inner
¶ fragments of Q and shared fragments of the opposite types
shape_union_in_U$_1$(O_P, I_Q, L) ∧
shape_union_in_U$_1$(L, L', *Lines*) ∧
maximal_planes(*Lines*, *Planes*).

The rules for *complement* also hold for simple maximal planes.

Shape intersection

The shape intersection of two arbitrary maximal planes P and Q can be constructed by iteratively adding holes to the plane formed by the intersection of the regions defined by the outer boundaries of P and Q. The holes that are added correspond to the intersection of the inner boundaries of the one plane and the outer boundary of the other. At the end of this procedure, we may have to postprocess the resulting outer, inner, and shared boundaries, because the holes may lie outside the outer boundaries, in which case they can be ignored. It should be noted that the intersection of a hole with any plane or hole is a hole. Examples of shape intersection are shown in figure 25.

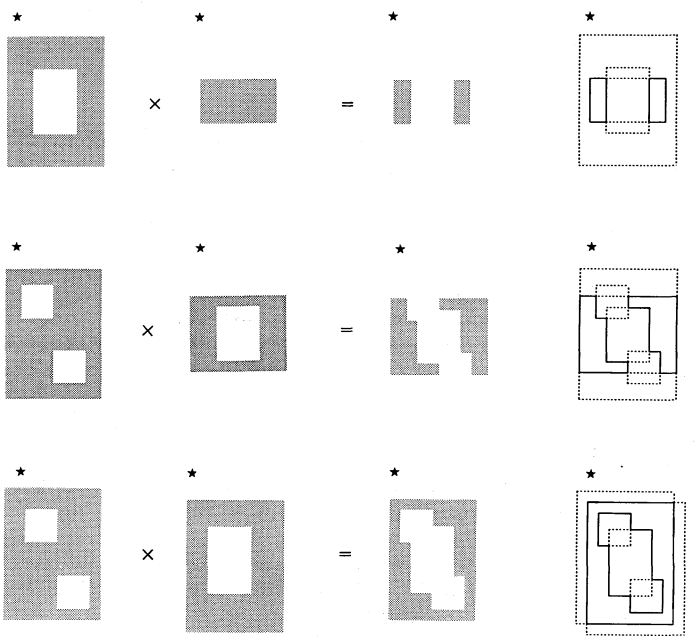


Figure 25. Examples of shape intersection of two maximal planes. ⋆ denotes the reference point.

As with shape difference, the expressions for shape intersection when Q is simple are easily defined. The expressions become complex when both P and Q are nonsimple. For completeness, the expression for shape intersection of two planes is given and has the form:

$$P \times Q = (o_P \dot{\times} o_Q) \dot{-} ((i_{P,1} \dot{+} i_{P,2} \dot{+} \ldots) \dot{+} (i_{Q,1} \dot{+} i_{Q,2} \dot{+} \ldots)) .$$

An easier approach is to look at the bounding lines of the two planes and determine which fragments correspond to the bounding lines of the resulting shape intersection. According to arguments similar to those given for shape union and shape difference, the fragments correspond to precisely those lines in the boundary of each plane that lie inside the other plane and the shared lines that are identically classified. That is,

common(P, Q, *Planes*) ←
  ¶ split the boundaries of P and Q into inner, outer, left-shared, and right-
  ¶ shared fragments
  split_plane(P, Q, I_P, O_P, L_P, R_P) ∧
  split_plane(Q, P, I_Q, O_Q, L_Q, R_Q) ∧
  ¶ consider just the common left-shared and right-shared fragments for
  ¶ intersection
  shape_intersection_in_U$_1$(L_P, L_Q, L_{PQ}) ∧
  shape_intersection_in_U$_1$(R_P, R_Q, R_{PQ}) ∧
  MERGE(L_{PQ}, R_{PQ}, L') ∧
  ¶ boundary of the union is made up of inner fragments and shared fragments
  ¶ of the same type
  shape_union_in_U$_1$(I_P, I_Q, L) ∧
    shape_union_in_U$_1$(L, L', *Lines*) ∧
  maximal_planes(*Lines*, *Planes*) .

As in the case of *combine* and *complement*, *common* also works for simple maximal planes.

Spatial conditions on shapes in U_2

All that remains is to define the predicates *disjoint*, *contain*, *overlap*, and *share_boundary*. Each of these can be defined in terms of the bounding lines of the planes.

Consider two planes P and Q. Then, P and Q are disjoint whenever every bounding line of one plane lies outside the other and there are no shared fragments. However, as the simple example in figure 26 demonstrates, the converse situation does not hold. Hence,

disjoint(P, Q) ← *split_plane*(P, Q, ∅, O_P, ∅, ∅) ∧ *split_plane*(Q, P, ∅, O_Q, ∅, ∅)

That is, the bounding lines of P inner with respect to Q is empty and the converse also applies.

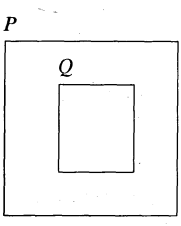


Figure 26. A simple counterexample to show that disjointedness requires examining the boundaries of both planes.

P contains Q whenever every bounding line in Q is inside P and every inner bounding line of P is outside Q. As in the case of disjointedness of planes, it is not sufficient to consider just the outer boundary of the planes as illustrated by figure 27.

In figure 27(a), the outer boundary of Q lies inside P, whereas part of its inner boundary lies outside P. In figure 27(b), the boundaries of Q lie inside P, whereas the inner boundary of P lies inside Q. That is,

$$\begin{aligned}&\mathit{contain}(P, Q) \leftarrow\\ &\quad \mathit{split\_plane}(P, Q, \varnothing, O_P, L_P, R_P) \wedge\\ &\quad \mathit{split\_plane}(Q, P, I_Q, \varnothing, L_Q, R_Q) \wedge\\ &\quad [L_P \cap R_Q] = \varnothing \wedge [R_P \cap L_Q] = \varnothing .\end{aligned}$$

The last line ensures that P and Q do not share boundary segments that are oppositely oriented with respect to the other; for example, this occurs when Q matches a hole in P exactly.

As containment includes all fragments of a bounding line of one plane that are coincident with bounding lines of the other, it follows that

$$\mathit{equal}(P, Q) \leftarrow \mathit{contain}(P, Q) \wedge \mathit{contain}(Q, P),$$

which is what we expect.

Overlapping of two planes is likewise defined. In the case of overlapping planes, the bounding lines will split into two nonempty sets of lines that lie inside and outside the other plane. That is,

$$\begin{aligned}&\mathit{overlap}(P, Q) \leftarrow\\ &\quad \mathit{split\_plane}(P, Q, I_P, O_P, L_P, R_P) \wedge\\ &\quad \mathit{split\_plane}(Q, P, I_Q, O_Q, L_Q, R_Q) \wedge\\ &\quad [[I_P \neq \varnothing \wedge O_P \neq \varnothing] \vee [I_Q \neq \varnothing \wedge O_Q \neq \varnothing]] .\end{aligned}$$

The sharing of boundaries is a little trickier. Suppose we split the bounding lines of one plane into inner and outer lines with respect to the other. Then, the two planes share boundaries if and only if there are no inner bounding lines for either plane and there are some outer bounding lines and some shared fragments. In other words,

$$\begin{aligned}&\mathit{share\_boundary}(P, Q) \leftarrow\\ &\quad \mathit{split\_plane}(P, Q, \varnothing, O_P, L_P, R_P) \wedge\\ &\quad \mathit{split\_plane}(Q, P, \varnothing, O_Q, L_Q, R_Q) \wedge\\ &\quad [L_P \neq \varnothing \vee R_P \neq \varnothing] \wedge\\ &\quad [L_P \cap L_Q] = \varnothing \wedge [R_P \cap R_Q] = \varnothing .\end{aligned}$$

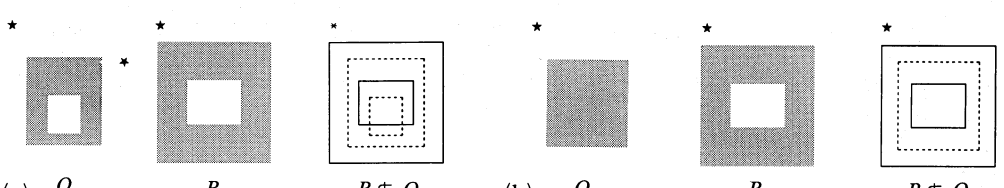


Figure 27. Counterexamples to show that containment requires examining the bounding lines of both planes.

Conclusion

The geometry of shapes made up of finite planes of nonzero area has been considered with a view to being able to compare such shapes and to produce new shapes thereof. By doing so, we have completed the computational framework necessary

to construct systems based on maximal representations of shapes (Krishnamurti, 1992) for shapes in U_2. This complements the framework, previously established, for shapes in U_0 and U_1.

We have shown that the Boolean operations on shapes made up of planes can be expressed in terms of shape operations on lines which in turn can be expressed as set operations on points. Further, we have shown that nonmanifold shapes such as those with holes require no special treatment. In other words, all shapes in U_2 can be handled in a uniform and consistent fashion.

The geometry of spatial objects is encapsulated in four basic relations: *disjoint*, *overlap*, *share_boundary*, and *contain*; and by three basic operations: *combine*, *complement*, and *common*. Algorithms to test for these relations and to perform these operations have been developed and presented. The algorithms are expressed in logic, via a resolution-based programming notation through unification of variables, and thereby, demonstrate their own correctness. In one sense, no additional proof is required.

Postscript—the GRAIL project

The maximal representation of shapes described in Krishnamurti (1992) and the maximal planes algorithms presented in this paper form the kernel of the GRAIL[11] project which is in progress at Carnegie Mellon University. GRAIL is a quest for an environment for the interactive representation, modelling, and generative composition (shape editing) of two-dimensional and three-dimensional geometric objects with or without nonspatial attributes. Currently, work in GRAIL focuses on efficient implementations of the algorithms[12] based on the maximal representation of shapes, on user-interaction issues related to rule-based generative and drawing systems, on radiosity-based rendering techniques, on semantic interface issues, and on shape grammar applications with or without associated descriptions (Stiny, 1990). Planned work on GRAIL includes the maximal representation of solids and subshape recognition in three dimensions for line, plane, and solid shapes.

Acknowledgement. I would like to thank George Stiny for convincing me that the time is ripe to tackle the subject matter of this paper. I would like to add my appreciation to Chris Earl for pointing out serious and careless errors in earlier versions of this paper and for his invaluable suggestions to improve its content. I am indebted to Rudi Stouffs for his inspired enhancements to earlier versions of the algorithms and for implementing and testing all the algorithms described in this paper. Lastly but not least, I would like to thank the original GRAIL team consisting of Churn Der Hwang, Marc Schindewolf, Rudi Stouffs, and Robert Ching Ping Tseng who share my faith in this quest.

References

Krishnamurti R, 1992, "The maximal representation of a shape" *Environment and Planning B: Planning and Design* **19** 267 - 288

Preparata F P, Shamos I, 1985 *Computational Geometry: An Introduction* (Springer, New York)

Stiny G, 1990, "What is a design?" *Environment and Planning B: Planning and Design* **17** 97 - 103

Stiny G, 1991, "The algebras of design" *Research in Engineering Design* **2** 171 - 181

[11] GRAIL does not stand for any one thing and can be expanded in several ways. The current favourite is 'generative representation of artefacts illustrated by lines'. Work on GRAIL will be documented in various reports both individually and collectively.

[12] Computational complexity of maximal plane algorithms is the subject of a forthcoming paper by Stouffs and Krishnamurti.