Chapter 3

Boolean Operations on Boundary Representation

Algorithms for determining the regularized union, intersection, or set difference of two solids can be used in B-rep and dual-representation modelers. They can be used also to convert solids represented by CSG trees to an equivalent B-rep. Thus, algorithms for Boolean operations on B-rep are sometimes called *boundary evaluation and merging* algorithms.

These algorithms are not difficult conceptually, but their implementation requires substantial work for several reasons. Layers of primitive geometric and topological operations to implement them have to be designed. Finding a good structure for these layers is not simple, and accounting for the many special positions of incident structures in three dimensions can be tedious. Moreover, the presence of curved surfaces introduces nontrivial mathematical problems, and, since most of the algorithms require numerical techniques, there is inevitably the problem of numerical precision and stability of the calculations.

In this chapter, we consider the regularized intersection of two nonmanifold polyhedral solids given in B-rep. This avoids presenting up front the mathematical problems arising from nonlinear geometric objects. The algorithm presented here has not been specialized to the point where its structure makes it unsuitable for extension to the curved case. Moreover, the majority of geometric operations to be formulated will, with certain extensions, apply to the curved-surface case. However, the need for high precision is more exacting in the curved-surface case, and the possibility of curve and surface singularities is a new dimension that necessitates additions to the algorithm.

It is desirable to use a B-rep in which the topological information is separated from the geometric representation of the surfaces elements. Moreover, as far as possible, the algorithm ought to be made independent of the specific details of the geometric representation, since then extensions of the coverage or alternatives to the chosen representation can be explored with minimum programming. This is especially important in the curved-surface domain, where different geometric representations offer different advantages.

3.1 Chapter Organization

The intersection algorithm to be described has many details and depends on many conventions. We begin the description by explaining the representation, and introducing a number of low-level operations used repeatedly throughout.

The heart of the algorithm is a method for intersecting two polyhedra, A and B, each of which has one shell. Conceptually, the method subdivides the faces of the two polyhedra along the curves in which their surfaces intersect. This subdivision is refined to faces of the output polyhedron, and the adjacencies of these faces are determined. Thereafter, the surface of the intersection is completed by adding certain faces of A and of B. Two aspects complicate this description:

1. The analysis and subsequent transfer of results must account for many special cases that come about when surface elements on the two polyhedra align in specific ways.

2. The face subdivision does not proceed independently, face by face, on each solid. Rather, each intersecting face pair is subjected to a neighborhood analysis whose results are immediately transferred to all adjacent surface elements.

The first complication is intrinsic to the problem. Often, descriptions in the literature will omit many of these details to simplify the narrative, leaving the reader to invent his or her own methods for handling them. The second complication trades programming effort against robustness, and is discussed later on. Our organization increases robustness, but at the price of additional programming.

After the description of how to intersect single-shell polyhedra, we explain how to intersect polyhedra with multiple shells. This requires fairly easy extensions. We also mention how the union and difference operations can be implemented by a simple modification of the intersection algorithm.

In the form first sketched, the intersection algorithm requires testing of each face pair for intersection. This is not efficient, and there should be a preceding computation that filters out face pairs that cannot intersect. A simple way to do this is to enclose each face in a box whose sides are aligned with the coordinate axes, and to construct a list of intersecting boxes. Clearly, if two enclosing boxes do not intersect, the faces inside them cannot intersect and need not be considered together. A fast algorithm for box intersection is described at the end of the chapter.

Note that such preprocessing steps cannot speed up certain cases of intersecting polyhedra. However, they do speed up the algorithm on average and should therefore be incorporated.

At first reading, it may be advisable to skim or skip parts of the chapter. Section 3.4, which describes how to intersect single-shell polyhedra, is the key part. It requires a conceptual understanding of the representation and of some of the geometric subroutines; this understanding can be obtained by skimming Sections 3.2 and 3.3. Section 3.4.4 may be skipped, postponing the various situations arising in neighborhood analysis. On subsequent reading, Sections 3.5 and 3.6 explain multishell polyhedra and the reduction of other Boolean operations to intersection.

Section 3.7 is self-contained. It presents box-intersection techniques without assuming any background in computational geometry. The algorithm is developed in stages, reviewing the needed data structures and discussing first the simpler problems of interval and rectangle intersection.

Subsection 3.4.4, on neighborhood analysis, maps out the many positional special cases that are encountered when implementing Boolean operations. It is included for the less experienced system developer who may become stymied by the many details. There is a second, less obvious purpose to this subsection. When we study the various cases carefully, we form a conceptual understanding of positional degeneracy as a cumulative impression, and we develop a valuable ability to organize the algorithm concisely.

3.2 Representation Conventions

We assume that polyhedra have nonmanifold surfaces of *finite* area. The topological representation fixes the following information:

1. For each vertex, the adjacent edges and faces are given.

2. For each edge, the bounding vertices and the adjacent faces are specified. Moreover, the adjacent faces are cyclically ordered according to how they intersect a plane normal to the edge, and pairs of adjacent faces that enclose volume in between are identified.

3. For each face, the bounding edges and vertices are given. They are organized in a set of cycles locally enclosing the face area to the right, as seen from the outside. Ordering information is given that specifies how the boundary graph is embedded in the face plane.

The logical structure of this information is as described later. In the description, we do not distinguish between, for example, a face and a *reference* to that face, since this distinction is unnecessary for understanding the algorithms.

For the sake of specificity, we describe how the geometric information is stored. However, in the subsequent algorithm description, the exact format of the geometric data is not essential. The geometric information is *irredundant*, specifying only the equations for the planes containing faces. These equations are oriented by the convention that the normal direction points locally to the solid exterior. Edges are defined geometrically as the line segment connecting the bounding vertices. Note that an edge may be adjacent to more than two faces. Vertices are given as the intersection of specific planes containing incident faces.

Irredundancy of the geometric information reduces the possibility of contradictory data and therefore increases robustness. Moreover, since the planes containing the faces of $A \cap^* B$ are a subset of the face planes of A and B, no new geometric data are ever constructed. Hence, no inaccuracy can be introduced through computed geometric information. On the other hand, irredundancy of geometric data in the curved-surface domain must be weighed against the computational cost of deriving coordinates for vertices.

Note that faces may consist of disjoint areas, provided these areas lie in the same plane with the same orientation. Some Boolean algorithms have been proposed that require that each face be a connected region, or a region homeomorphic to a disk, or a convex polygon, or a triangle. Typically, such constraints simplify the algorithms. However, one would then have to devise an algorithm that subdivides certain surface areas of the result polyhedron, since inputs with restricted face topologies do not always yield results that satisfy these constraints as well. Thus, the work is shifted from the intersection algorithm to a postprocessing step that constructs legal faces for the result object.

In the polyhedral case, this strategy is not without merit, even though it may lead to unnecessary edges in repeated Boolean operations on an object. It is unclear, however, whether the approach remains viable in the curved-surface domain, because in that case triangulation or other forms of face subdivision can be quite difficult.

3.2.1 Face Representation

A *face* is a finite, nonzero area in a plane, bounded by one or more cycles of vertices and edges. Edges are directed such that the face area locally lies to the right, as seen from the exterior of the solid. Face planes have been written such that the normal vector points locally to the solid's exterior. We refer to this as the convention of *outward-pointing normals*, and use it, for example, when determining the face direction vector as explained in Section 3.3.1.

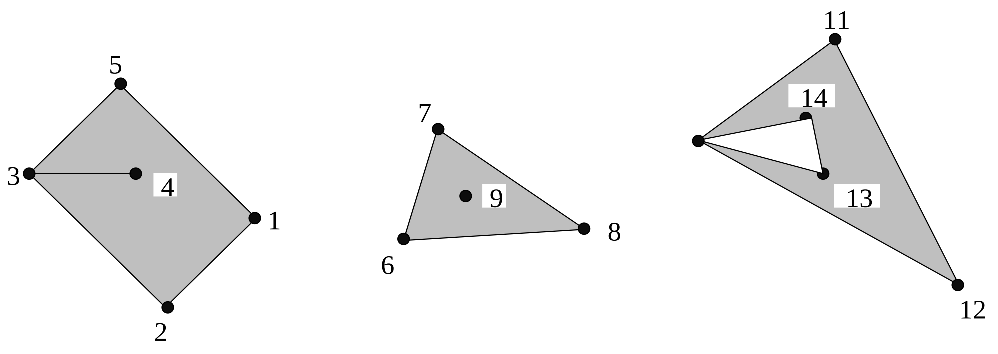


Figure 3.1 Legal Face Cycles $(1, 2, 3, 4, 3, 5)$, $(6, 7, 8)$, (9), and $(10, 11, 12, 10, 13, 14)$

A bounding cycle may be *degenerate*, containing the same vertex more than once, or containing only one or two vertices. In these cases, the cycle should not enclose zero face area. Thus, a single vertex may bound a zero-area puncture in a face, but may not lie outside the face area. In particular, edges must bound a nonzero area immediately to the right.

Example 3.1: Figure 3.1 shows several legal degenerate face cycles; Figure 3.2 shows illegal ones. $\diamond$

Together, the edge cycles form an embedded directed, planar graph. Separate connected components may be nested. This nesting structure is recorded in a separate forest of trees. Each tree node corresponds to a connected boundary component. Nested components are in subtrees. For example, for the face shown in Figure 3.1, we have three trees in the forest. Two trees consist of a single vertex each, and represent the cycles $(1, 2, 3, 4, 3, 5)$ and $(10, 11, 12, 10, 13, 14)$, respectively. A third tree has two nodes. Its root represents the cycle $(6, 7, 8)$, and its descendant represents the cycle (9).

Consider a vertex u of the face f. The incident edges of u define sectors in the face plane that are alternately inside and outside the face f. Since edges are oriented, the incident edges must alternate in direction, one being oriented away from u, the other being oriented toward u. We order the edges

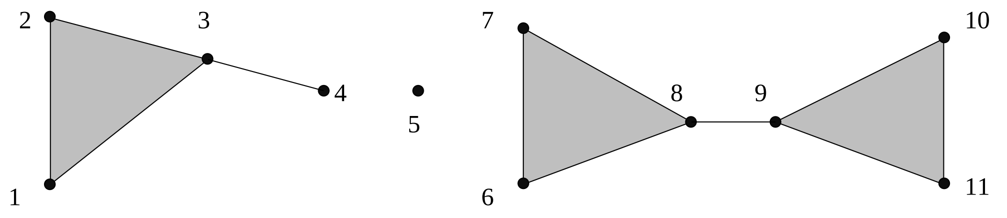


Figure 3.2 Illegal Face Cycles $(1, 2, 3, 4, 3)$, (5) and $(6, 7, 8, 9, 10, 11, 9, 8)$

clockwise about u and consider them paired such that each pair encloses a sector in f. Each pair is called an *area-enclosing pair.* Note that the clockwise orientation is with respect to the solid exterior; that is, it depends on the face-plane normal. We think of this pairing as a representation of the two-dimensional neighborhood of u in the face plane.

Example 3.2: In the face cycle $(1, 2, 3, 4, 3, 5)$ shown in Figure 3.1, the vertex 3 is incident to the four directed edges $(3, 5)$, $(4, 3)$, $(3, 4)$, and $(2, 3)$. There are two area-enclosing pairs at u; namely, $(3, 5), (4, 3)$ and $(3, 4), (2, 3)$. $\diamond$

Recall from Section 2.4 that the surface of a solid should be orientable. Since all faces are planar, the orientation of the edge cycles ensures that every triangulation of a face can be coherently oriented, assuming the orientation is consistent. Here, consistency is exactly analogous to consistency of orientation of the boundary components of topological solids.

3.2.2 Edge Representation

Consider an oriented edge $e = (u, v)$. The adjacent faces define wedges of volume that are alternately inside and outside the solid. We order the adjacent faces clockwise about e as seen in the direction (u, v), and pair adjacent faces that enclose a wedge of solid interior. Each pair is called a *volume-enclosing pair.* Again, we think of this pairing as a representation of the three-dimensional neighborhood of points in the *interior* of the edge. Representing this information explicitly, we thus give the following information for each edge:

1. Beginning and ending vertices, establishing a default orientation

2. An ordered, circular list of adjacent faces

3. A pairing establishing which face pairs enclose volume

The circular ordering and pairing can be represented by a single structure.

Note that the adjacent faces use the edge in alternating direction. We explicitly annotate the face reference, indicating whether the edge is used in the default orientation (u, v) or in the opposite orientation (v, u). This alternation of edge directions implies that locally the surface of the solid can be oriented coherently in the sense of Section 2.4 of Chapter 2.

3.2.3 Vertex Representation

Given a vertex, we must know all incident edges and adjacent faces. As discussed in the previous chapter, the adjacent faces form cones that can be nested, and the logical structure is a spherical map. Instead of representing this structure explicitly, we let the algorithm infer it when needed; see also Section 3.3.4.

3.2.4 Surface Structure

We require finite surface area for polyhedra, but permit infinite volume. This is convenient for reducing the union operation to an intersection and several complementation operations. A polyhedron can have internal voids, and is thus, in general, an object with several surface components. Each surface component is a collection of vertices, edges, and faces that are adjacent. These components are its *shells* and are organized into a forest of trees reflecting spatial containment and shell orientation.[1] A face consisting of two or more disjoint areas must not belong to different shells.

Each shell is given as a list of faces and an indication of whether the shell, taken separately, represents a polyhedron with finite or infinite volume. This information is stored at the nodes of the shell trees. A node s is a descendant of another node t if the shell stored at s is spatially contained by the shell stored at t. Consider the shell structure of a cube with two internal voids. The exterior surface is the shell s represented at the root. By itself, it describes a solid of finite volume, and is marked as such. The two shells bounding the interior voids are t_1 and t_2. Separately, each describes a polyhedron of infinite volume. Both shells are nested in s, but not within each other. In consequence, the shell forest is a single tree whose root is s and whose two leaves are t_1 and t_2.

3.3 Geometric Operations

The intersection algorithm is developed in terms of simpler geometric operations that are used as subroutines. The most trivial ones, such as computing vertex coordinates and testing point/line incidence are not described here. Note, however, that in the context of robustness they will have to be considered in some detail, as discussed in the next chapter.

3.3.1 Face Direction Vector

Given an edge e of a face f, we will need to know the orientation of the edge and the direction in the plane of f in which the interior of f lies. Suppose the edge is defined by two incident vertices u and v, and we know an equation $ax + by + cz + d = 0$ for the plane P containing the face f. The orientation of the edge with respect to f is determined from the topological data that specify how the edge is referenced by f.

The *face direction vector* is a vector fd in the plane P. The vector is perpendicular to the edge tangent vector t_e and points to the interior of f (Figure 3.3). It is defined for every point p on the edge. Since the edge is a line segment, the vector does not depend on p. It is computed from the edge orientation as referenced by f and from the normal vector $n_P = (a, b, c)$ of

[1]Each shell is a connected component of the 2-manifold bounding the solid, as discussed in Section 2.4 of Chapter 2.

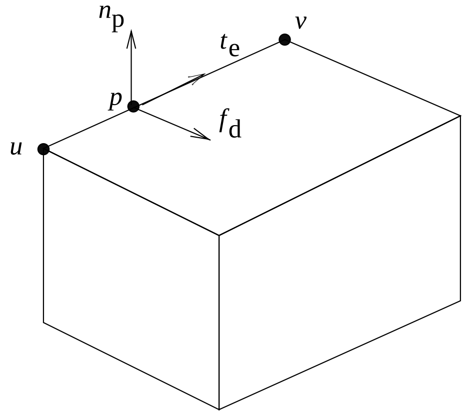


Figure 3.3 Face Direction Vector

the plane P as the cross-product $t_e \times n_P$. Here, t_e is the edge tangent vector, oriented by the edge direction. This computation also works for curved edges, but then the surface normal n_P and the edge tangent t_e depend on the point p.

It is possible that the same edge is referenced twice by f, once in each direction. In this case, we know that there is face interior on both sides of the edge, and which of the two sides is needed must be determined from the context; see also Section 3.3.3.

3.3.2 Splitting an Area- or Volume-Enclosing Pair

In the neighborhood analysis, we have to determine whether a face locally extends to the interior or exterior of the other solid. This question is reduced to determining whether a certain vector splits two paired vectors. Similarly, in two dimensions, we may determine whether a line segment extends into the interior of a face, which we do by determining whether the segment splits an area-enclosing pair of edges.

Given an area-enclosing pair of edges (v_1, u) and (u, v_2), we wish to determine whether a vector $t = \overline{(u, w)}$ lies in the area enclosed by the pair. If so, we say that t *splits an area-enclosing pair*; see Figure 3.4 for an example. To determine this, we order clockwise the three vectors t, $\overline{(u, v_1)}$, and $\overline{(u, v_2)}$ about u, beginning with (u, v_2). If, after sorting, t lies between the pair, then it splits the pair.

Given a volume-enclosing pair of adjacent faces, we determine whether the vector t lies inside the volume so bounded. If so, we say that t *splits a volume-enclosing pair*. The test can be reduced to a test for splitting area-enclosing pairs by projecting t onto the plane spanned by the face direction vectors of the faces.

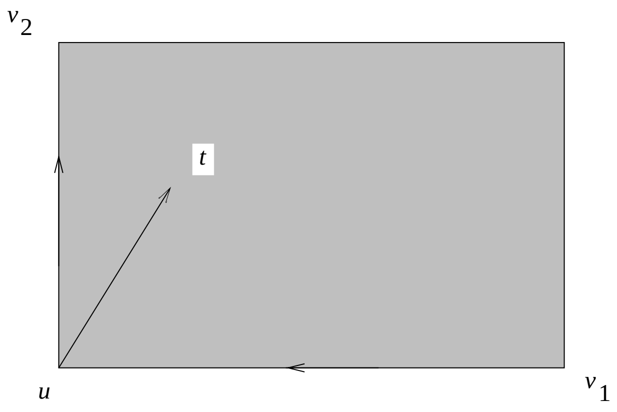


Figure 3.4 Splitting an Area-Enclosing Pair

3.3.3 Ordering Points Along a Line and Pairing Them

The transversal intersection of two faces is a set of segments on a line. The segments are obtained by first intersecting each face with the plane containing the other face, followed by intersecting the segments. We address how to intersect one face with the plane containing the other face.

Given a line representing the intersection of two face planes P and Q, containing, respectively, the faces f and g, we order sequentially the points in which the bounding edges of g intersect the plane P. Assuming no arithmetic problem, the ordering of points is straightforward and can be done by sorting them by one of the coordinates, depending on the slope of the line.

Having ordered the points, we now pair consecutive points such that the line segment bounded by each pair represents an intersection of the face g with the plane P (Figure 3.5). Since faces have finite area, there cannot be infinite line segments, and so we pair consecutive points in sorted order. For curved surfaces P and Q, the problem is much more complicated because their intersection may be a complicated space curve.

We orient the line $P \cap Q$ (arbitrarily) by the cross-product of the plane normals, $t = n_P \times n_Q$, and order the intersection points accordingly. Complications arise from special positions where vertices of g lie on the line $P \cap Q$, and from intersections with edges that must be considered in both orientations.

Let (u, v) be an *oriented* edge of g. If u is below P while v is above it, then the intersection point is paired with the subsequent point in sorted order; otherwise, it is paired with the previous point. If both (u, v) and (v, u) must be considered, then the two intersection points must not be paired with each other.

If a vertex of g is on the line $P \cap Q$, it is considered as a double intersection point. Call the two intersections u_1 and u_2, in sorting order. Then, u_1 is paired with the preceding point if $-t$ splits an area-enclosing pair of edges incident to u, and u_2 is paired with the succeeding point if t splits an area-

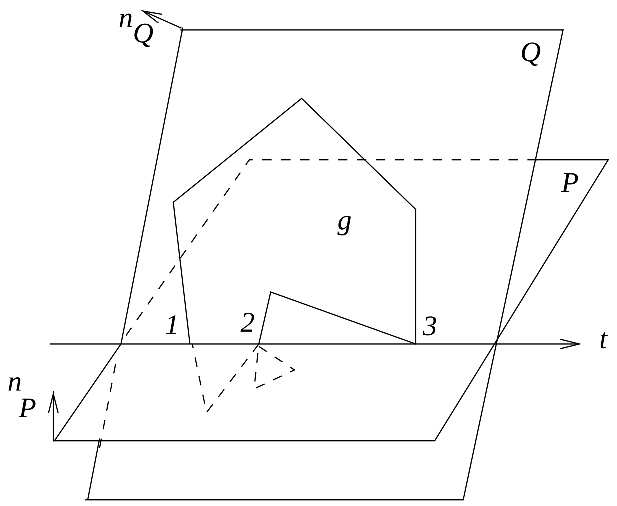


Figure 3.5 Intersection-Point Sorting and Pairing

enclosing pair of incident edges. If $-t$ or t is collinear to an incident edge of g, then that edge is used to connect the respective copy of u with its predecessor or successor. If neither t nor $-t$ split an area-enclosing pair, then u is an isolated point.

For example, in Figure 3.5, the first copy 2_1 of point 2 is paired with the preceding point 1, but the second copy is not paired and is therefore ignored. The intersection point 3 is isolated. Note that isolated points must be recalled in later stages of the algorithm, and cannot be discarded outright.

3.3.4 Line/Solid Classification

A general method for determining possible containment of two nonintersecting bounding structures A and B is to connect a point on the boundary of A by a line segment with a point on the boundary of B, and then to analyze how this line segment intersects the boundaries of the two structures. However, the structure A may consist of several disconnected components, as may B. Therefore, any containment conclusions drawn apply to only those components of A and B that the line segment actually intersects. For example, the line segment (p, q) in Figure 3.6 allows us to conclude that both the component containing p and the component containing q will bound the intersection of the two areas A and B, but it cannot reveal that the other component of A is not part of the final boundary. For the final boundary, the components are assembled by a sequence of the tests now described; see also Section 3.5 on multishell objects. We explain the test first in the case of faces.

Consider a face f and a face g, both in the same plane and oriented the same way. We assume that the boundaries of f and g do not intersect, and wish to test whether one face boundary contains the other. We select a point

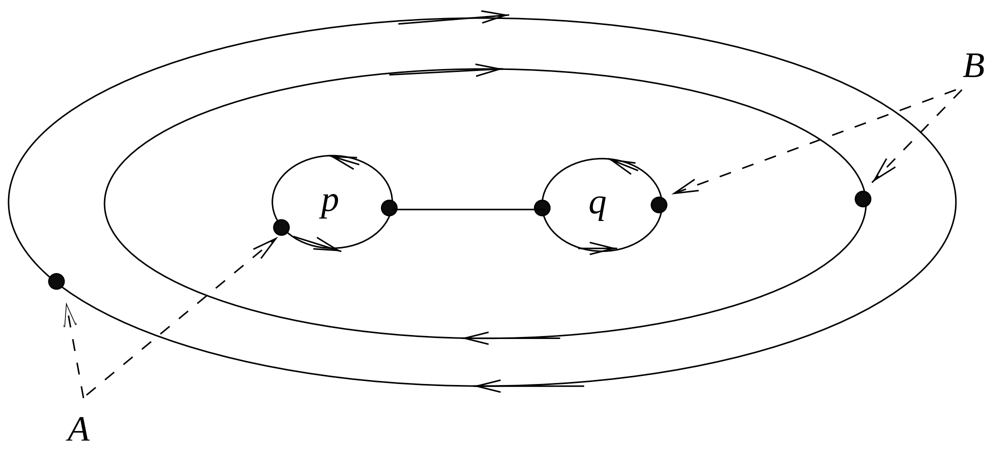


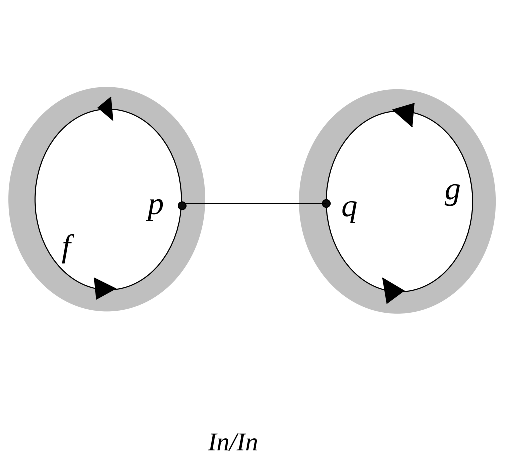
Figure 3.6 Classification of Multiple-Boundary Components

p on the boundary of f and a point q on the boundary of g, and connect them with a line segment (p, q). The segment intersects the boundaries of f and g in a number of points that must be ordered linearly and that then partition the line (p, q) into intervals. Each interval is classified as being outside of f or in f, and, likewise, as being outside of g or in g. Since p is on the boundary of f and q is on the boundary of g, there is some interval with one endpoint that is an intersection of the line (p, q) with the boundary of f, and the other endpoint that is an intersection with the boundary of g. We pick the first such interval encountered, scanning the intervals in order beginning at p. There are four possible classifications of this interval, as shown in Table 3.1. Associated with each is a position of the two faces relative to each other. Figures 3.7 and 3.8 illustrate the four cases, with the interior as implied by the orientation of the bounding cycles. The interval classification is essentially the same operation that was done in the intersection-point pairing along the line $P \cap Q$ described previously. As with that operation, care has to be exercised when the segment (p, q) intersects a boundary at a vertex.

Now consider testing the possible containment of two solids A and B

Test	Example	Action
in f, in g	f and g intersect	both components are kept
in f, out g	g is contained in f	the g component is kept
out f, in g	f is contained in g	the f component is kept
out f, out g	f and g do not intersect	neither component is kept

Table 3.1 Containment Classification for f and g



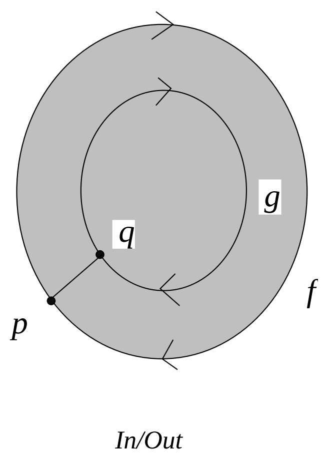
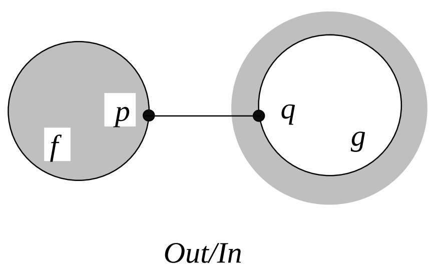


Figure 3.7 The Classifications "in *f*, in *g*" and "in *f*, out *g*"

whose boundaries do not intersect. In spirit, we proceed exactly as for faces, selecting a point p on the surface of A and a point q on the surface of B, and connecting these two points with a line segment. As before, the intersection of (p, q) with the boundaries of A and B induces an interval partition, and the individual intervals are again classified as being inside or outside of the solids. Again, Table 3.1 governs the outcome of the test, based on the first interval encountered, going from p to q, that is bounded by an intersection with the boundary of A and an intersection with the boundary of B.

The classification of intervals in the solid case is more complicated, however, and we explain it further. Conceptually, it is an analysis of the neighborhood of the intersection points. Thus, it is easiest for interior face points, more complicated for interior edge points, and hardest for vertices.

First, if the segment intersects in the interior of a face f of A, then the direction vector $t = \overline{(p, q)}$ is compared in angle against the face normal n_f of f. If the angle is less than 90° — that is, if the dot product $n_f \cdot t$



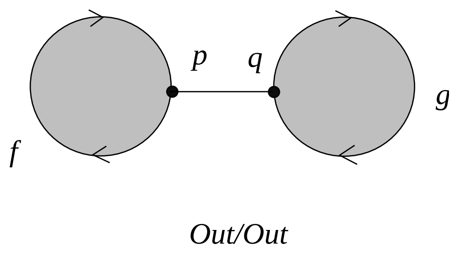


Figure 3.8 The Classifications "out *f*, in *g*" and "out *f*, out *g*"

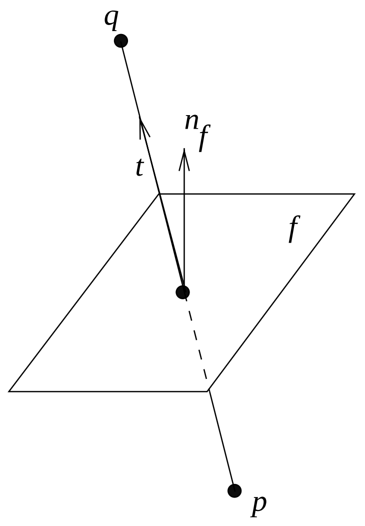


Figure 3.9 Classification for Face Interior

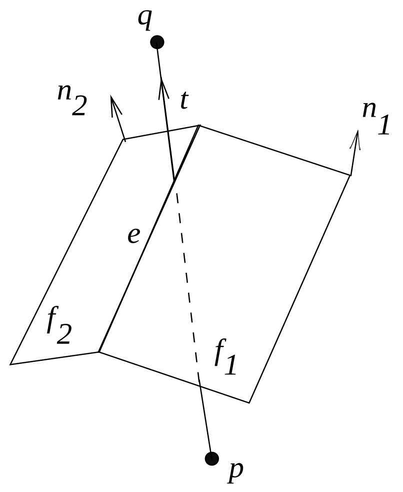


Figure 3.10 Classification for Edge Interior

is positive — then the interval preceding the intersection is in A, and the interval succeeding it is outside of A. Otherwise, the preceding interval is outside, and the succeeding interval is inside, of A; see also Figure 3.9.

Next, assume that (p, q) intersects at the interior of an edge e of A. Here we must determine whether t and $-t$ split volume-enclosing pairs. That is, if t splits a volume-enclosing pair of the faces adjacent to e, then the succeeding interval is inside of A. If not, it is outside of A. The same analysis of the vector $-t$ classifies the preceding interval with respect to A; see also Figure 3.10. Note that both t and $-t$ must be classified.

Finally, assume that (p, q) intersects at the vertex u of A. Here we must determine whether t, and $-t$, are in the interior of a solid cone defined by the faces incident to u. Since this neighborhood is not explicitly represented, we investigate the structure by intersecting it with a suitable plane. We pick a plane R that contains the line (p, q) and an interior point w of some face adjacent to u. Since u is in the plane R, the vertex appears on R as a point with lines radiating outward. Each line represents the intersection of some adjacent face of u with R.

The sectors defined by these lines are classified as inside or outside of A. Since R contains w, there is at least one nonempty sector. Then t and $-t$ are classified by the sector in which they lie. Accordingly, we now classify the interval on (p, q). See Figure 3.11 for an illustration of the process.

Since the classification is fairly complex and expensive for vertex intersections, we may wish to choose a different point q to avoid this case. Thus, it is a good idea to choose p and q to lie in the interior of faces. Although this does not guarantee that all other intersection points are located favorably, subsequent random perturbations of the positions of p and q will usually

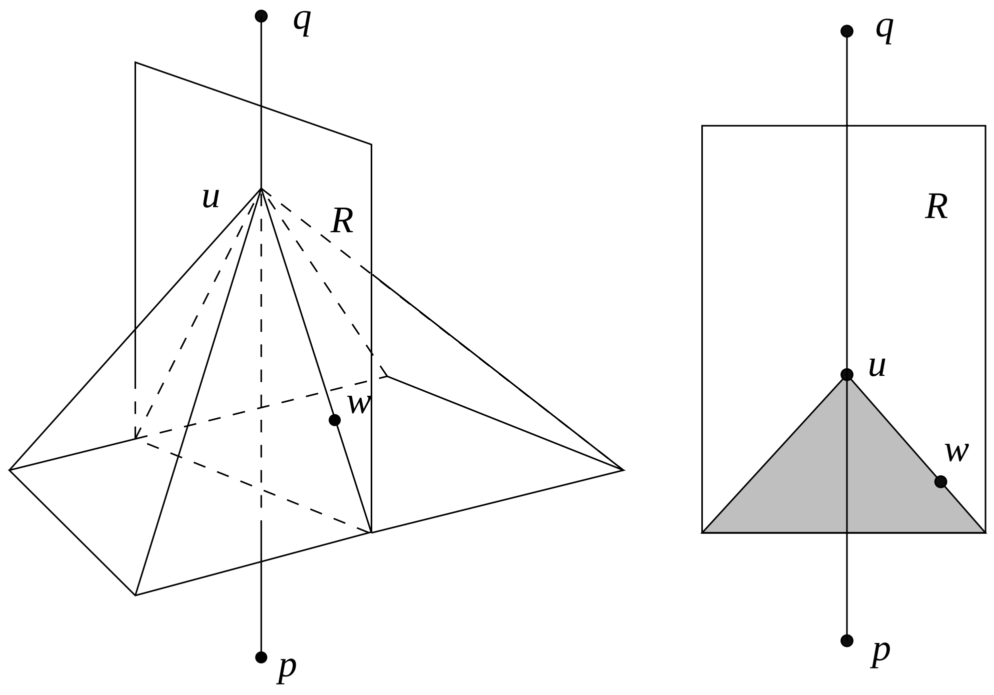


Figure 3.11 Classification for Vertex Intersection

succeed in creating well-behaved intersections everywhere. Typically, one or two perturbations suffice.

Recall from Section 2.4 that the connected components of the boundary of a multishell solid should be consistently oriented. The shell-containment test just described can be used to test whether the components are consistently oriented. It is not difficult to prove that two components are consistently oriented iff the containment test determines an “in/in” or an “out/out” classification.

3.4 Intersection of Two Shells

We consider first the intersection of two polyhedra, A and B, each with a surface consisting of a single shell. The intersection of multishell polyhedra will be considered subsequently. We conceptualize the process of intersection as follows:

1. Determine which pairs of faces $f \in A$ and $g \in B$ intersect. If there are none, test shell containment only and skip steps 2 through 4.

2. For each face f of A that intersects a face of B, construct the cross-section of B with the plane containing f. Then determine the surface area of $A \cap^* B$ that is contained in f.

3. By transferring the relevant line segments discovered in step 2, determine the faces of B that contain some of the surface area of $A \cap^* B$

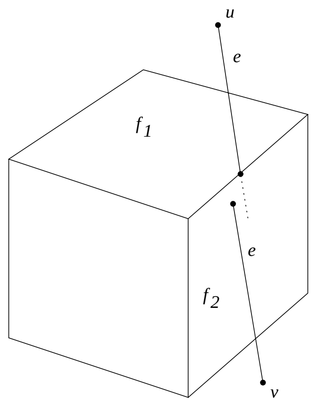


Figure 3.12 Locally Inconsistent Intersection Analysis

and must be subdivided. Subdivide these faces, and by exploring the face adjacencies of B, find and add all those faces of B contained in the interior of A. Likewise, find and addall faces of A contained in the interior of B.

4. Assemble all faces so found into the solid $A \cap^* B$.

This conceptual structure is also suited to the curved-surface domain.

3.4.1 Robustness Considerations

A drawback of the organization just presented is its sensitivity to failure because of numerical imprecision. If step 2 considers each face of A independently, certain intersection structures may be inconsistently analyzed for adjacent faces. In consequence, the algorithm could fail for legitimate inputs.

Example 3.3: Figure 3.12 illustrates the problems numerical error could cause when implementing the proposed conceptual algorithm. We analyze whether and how the edge $e = (u, v)$ of B intersects two adjacent faces f_1 and f_2 of A. When considering these faces, the edge e is intersected with the respective face planes. When this is done independently, the intersection-point coordinates could have different precision. In consequence, it is possible that e will be judged to intersect the edge between f_1 and f_2 when considering face f_1, but that e will not be recognized to intersect the edge when considering f_2. This apparent inconsistency could cause a catastrophic failure of the implemented algorithm. $\diamond$

Similar robustness problems are endemic to the following popular algorithm for Boolean operations. First, mark on the surface of A the curves in which B intersects A. Then, reverse the roles of A and B, and repeat this step. Thereafter, assemble the surface of $A \cap^* B$ by adjacency exploration. This method is attractive because it reduces the programming effort. However, the independent subdivision of the surfaces of A and B gives many opportunities for inconsistencies due to numerical error. Thus, this paradigm is also not robust.

We refine the conceptual structure of our algorithm with the goal of achieving local consistency and avoiding problems such as the one illustrated in Figure 3.12. Our strategy is to avoid asking the same geometric question more than once. That is, all intersection information is immediately posted to all adjacent faces:

1. Determine which pairs of faces $f \in A$ and $g \in B$ intersect. If there are none, then do a shell-containment test only and skip steps 2 through 4.

2. For each intersecting pair of faces, f of A and g of B, construct the points and curves in which they intersect. For each intersection, analyze the three-dimensional neighborhood and transfer its elements to all adjacent faces of A and of B.

3. By exploring the face adjacencies of A and of B, find and add all those faces of either solid that are in the interior of the other.

4. Assemble all faces into the solid $A \cap^* B$.

3.4.2 Intersecting-Pairs Determination and Shell Containment

We elaborate on step 1 of the algorithm, detecting pairs of intersecting faces. The obvious method for determining whether the face $f \in A$ intersects the face $g \in B$ is actually to intersect these faces. Since all face pairs would be so tested, there is no hope that the algorithm could perform well in those situations where A and B have many faces but only a few of them actually intersect. Instead, it is convenient to enclose each face in a box whose sides are parallel to the coordinate planes, and to ask whether the box containing f intersects the box containing g. If the boxes do not intersect, then the faces cannot intersect. If the boxes do intersect, then the faces may or may not intersect, and we continue with the remaining steps.

There is an $O(n \log^2(n) + J)$ algorithm for intersecting the boxes, where n is the number of boxes and J is the number of intersecting box pairs found. Since J may be quadratic in n, we cannot improve the worst-case running time for polyhedral intersection, but we can significantly improve the average running time. A box-intersection algorithm with this time performance is described later in this chapter.

Since box intersection can determine only that two faces do not intersect, we may arrive at a situation where some boxes intersect, yet later in step 2, we determine that the surfaces of A and B do not intersect after all. In this case, we must run the shell-containment test following step 2.

Assuming that the surfaces of A and B do not intersect, we distinguish four possibilities:

1. The surface of $A \cap^* B$ consists of both the surface of A and the surface of B.

2. $A \cap^* B = B$.

3. $A \cap^* B = A$.

4. $A \cap^* B$ is empty.

These four situations are precisely the four possibilities of the line/solid classification, discussed previously, corresponding, respectively, to "in A, in B," "in A, out B," "out A, in B," and "out A, out B."

3.4.3 Face Intersection and Neighborhood Analysis

Having found all candidates of intersecting face pairs, we proceed to step 2 of the algorithm and construct their intersection. Although this step is conceptually not hard, the details tend to get in the way. Moreover, it requires intermediate data structures that are incomplete edge and face subdivisions. Conceptually, we repeat the following tasks for the intersecting faces f of A and g of B:

1. Construct and analyze the points and line segments of their intersection.

2. Transfer the results to all adjacent faces of A and of B.

3. Link up the intersection elements into complete face and edge subdivisions.

After this operation has been performed for all intersecting faces, we know implicitly the surface area of $A \cap^* B$ that lies on the surface of A and of B, except the faces of A that lie inside B and the faces of B that lie inside A.

Example 3.4: Consider the intersecting boxes A and B shown in Figure 3.13. Four faces of B and one face of A are subdivided in the process of face-pair intersection. In the end, we have discovered five of the six faces bounding $A \cap^* B$. The sixth face is discovered in a later step of the algorithm. The important point is that, when intersecting the faces f and g, both f and g are subdivided by a line segment. A technical difficulty is that the intersection line segment introduced on f does not yet define a valid subdivision of f. The subdivision of f is completed only after f has been intersected with four faces of B. $\diamond$

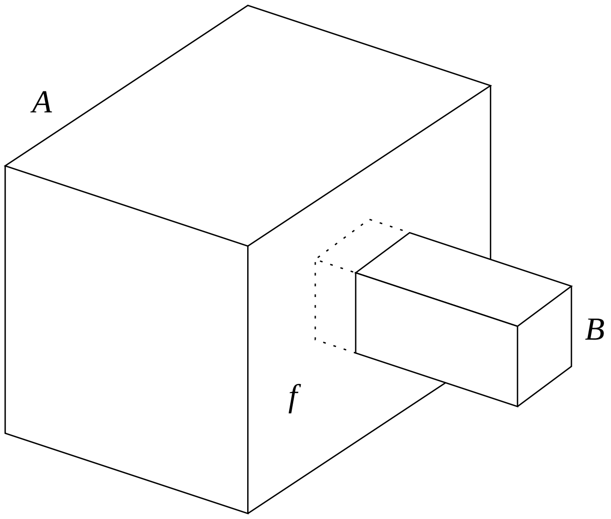


Figure 3.13 Two Intersecting Boxes

Determining and Placing Intersecting Elements

Let g be a face of B that we suspect intersects the face f of A. We intersect the bounding edges of g with the plane P containing f. Excluding for the moment the case that g is contained in P (i.e., that every edge of g is contained in the plane), the edges will intersect P in a number of points that lie on a line l. The line l is the intersection of the plane Q containing g and the plane P. If an edge e of g is contained in P but g is not, then the vertices of e are considered the intersection points of the edge with P. The intersection points are sorted along l and are paired as described previously. Thereafter, we intersect the line l with f, obtaining a second set of segments. The two sets are then intersected, and the resulting segments are placed on f, g, and, possibly, other adjacent faces, as appropriate.

If f and g are in the same plane, then they must be intersected as polygons. The result area is on the surface of $A \cap^* B$, provided both faces have the same orientation. If the faces have opposite normals, then $f \cap g$ will be a lower-dimensional structure that is eliminated by regularization. See Figure 3.14 for an example.

The intersection of f and g consists of line segments and points. Segments are either edges or edge segments, or they represent the intersection of two face interiors. Note that there may be isolated vertices. Points and lines both are analyzed. In particular, the points bounding a segment are analyzed, as is the segment interior. The results of this analysis are

1. The placement of oriented segments on faces

2. The segmentation of edges

3. The placement of certain points on faces and edges

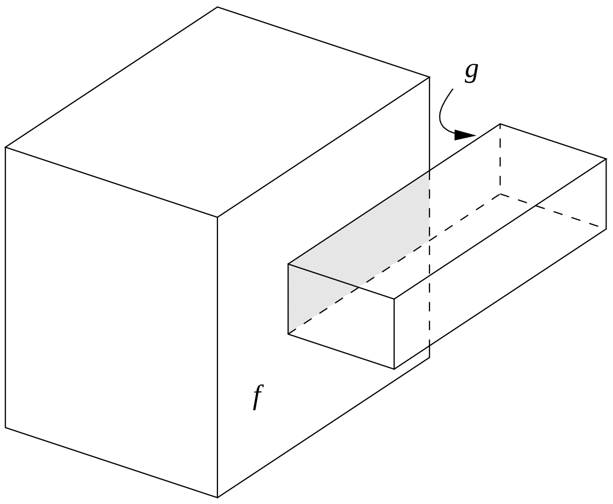


Figure 3.14 Coplanar Faces with Opposite Orientation

4. The creation of adjacency constraints between points and segments

Placing points on an edge defines a segmentation of the edge. Segments so defined may have to be refined later, when other points on the edge are discovered.

Example 3.5: Consider Figure 3.15, assuming that the faces f and g are intersected first. Here, the segment (w_1, w_2) is placed on f in the orientation (w_2, w_1) and on g in the opposite orientation. The edge (u_1, v_1) of g is subdivided by placing the point w_1 on it. The resulting segment

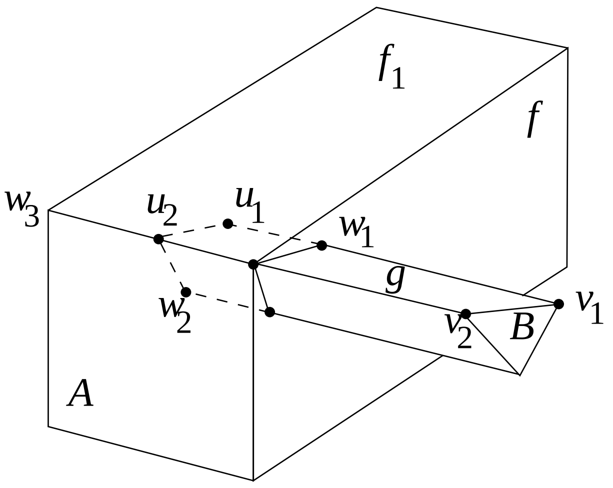


Figure 3.15 Placing Points and Segments

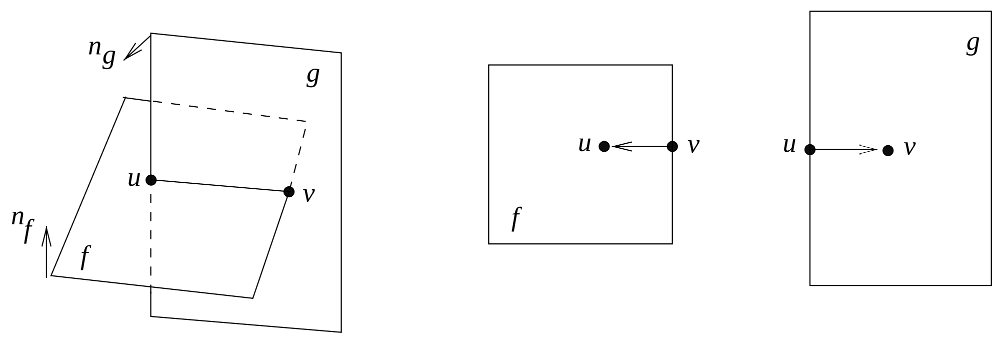


Figure 3.16 Line-Segment Orientation

(u_1, w_1) would have to be further subdivided if another face of A intersected it. This is not the case in the figure. Similarly, the segment (u_2, w_2) is created by placing w_2, and is not further subdivided later.

Now assume we intersect next the faces f_1 and g. Then the edge (u_2, v_2) of g is collinear with the edge (w_3, w_2) of f_1. Both edges are subdivided by placing the segment (w_2, u_2). Here we know that the segment cannot be further subdivided, since the two edges overlap. On g, the segment is placed in the orientation (w_2, u_2), consistent with the orientation in which g uses the edge (u_2, v_2). An analysis of the three-dimensional neighborhood reveals, moreover, that the segment should not be placed on f_1, since the area of f_1 adjacent to the segment lies in the exterior of B. $\diamond$

Segment Orientation

A segment that is placed on a face f represents part of the boundary of the intersection $A \cap^* B$. It will be oriented such that the interior of the face of $A \cap^* B$ bounded by it is locally to the right. The correct orientation is deduced from the orientation of the two face planes. Figure 3.16 shows a simple example. In some cases, two oriented line segments are placed on f, indicating that the final bounding cycle of edges is degenerate.

3.4.4 Neighborhood Analysis

To analyze the intersection of face pairs, we consider the three-dimensional neighborhoods of the intersection line segments and points in the subdivision. We recall that the interior and the endpoints of a segment must be analyzed. There are six major cases, indexed by the intersecting structures:

1. A face of one solid intersects the face of another solid at a point interior to both.

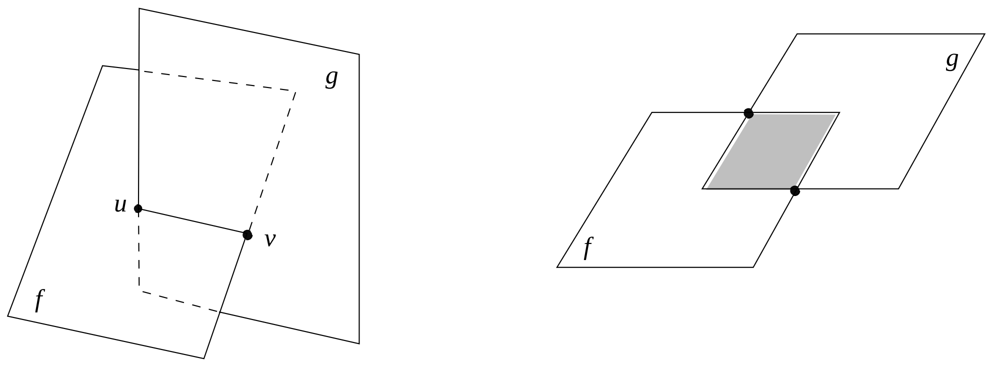


Figure 3.17 Face/Face Intersection

2. An edge of one solid intersects a face of the other solid at a point interior to both edge and face.
3. An edge of one solid intersects an edge of the other solid at a point interior to both edges.
4. A vertex of one solid intersects a face of the other solid in the interior.
5. A vertex of one solid intersects an edge of the other solid in the interior.
6. A vertex of one solid intersects a vertex of the other solid.

All cases exhibit a conceptual similarity in that the progression from face to edge and then to vertex entails similar processing, but involves more and more faces.

Face/Face Intersection

The generic case arises from a transversal intersection of two faces, shown on the left in Figure 3.17. The segment (u, v) generates two oriented line segments, one on f, the other on g. The orientation is computed from face normals. See also Figure 3.16. The interior of (u, v) will be only on f and on g, but the endpoints require further analysis as described later, and this analysis involves additional faces.

For the degenerate case, arising when f and g are in the same plane, we analyze the interior of a region by comparing the face normals. Normals of equal direction mean that the area is on the surface of $A \cap^* B$. Normals of opposite direction mean that the area is not on the surface of $A \cap^* B$. See also Figure 3.16.

Edge/Face Intersection

Assume that the edge $e = (u, v)$ of B intersects the interior of the face f of A. Ordinarily, the edge intersects the face in one point, as shown in Figure

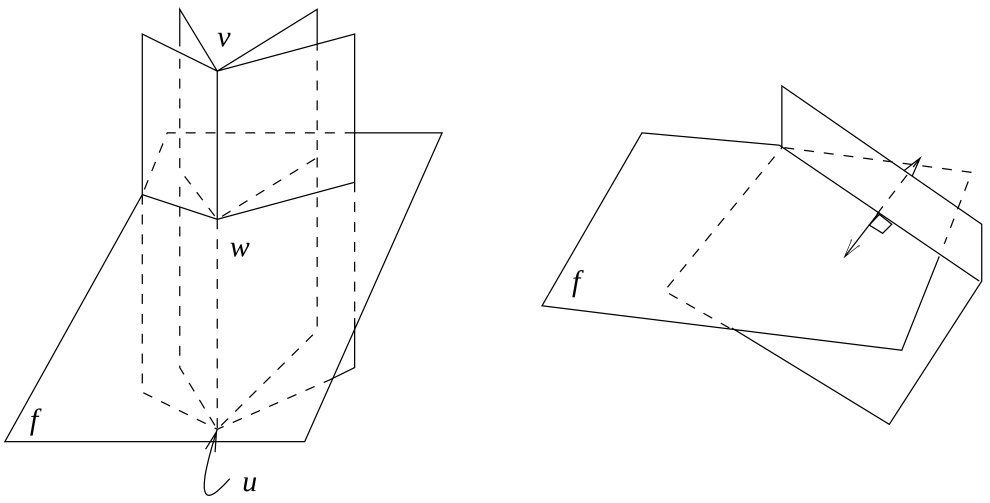


Figure 3.18 Edge/Face Intersection

3.18 on the left. We assume this point is in the interior of *f*. The edge is subdivided by w. There must be a segment of e, bounded by w, that lies inside A. Whether this segment is contained in (u, w) or $\underline{\text{in } (w, v)}$ is determined by computing the dot product of the direction vector $\overline{(u, v)}$ and the face normal n_f. We also know that the faces of B adjacent to e all intersect *f*, so we must make sure that the respective segments are recognized as adjacent to w.

In the degenerate case, the edge e lies in the plane of *f*, as shown in Figure 3.18 on the right. The edge is then subdivided by the boundary of f into a number of segments. These segments must be transferred to f and to all faces g adjacent to e. Transfer and orientation is determined as follows.

For the face *f*, we consider vectors perpendicular to e in the plane of *f*, in each direction. We ask whether these vectors split a volume-enclosing pair of

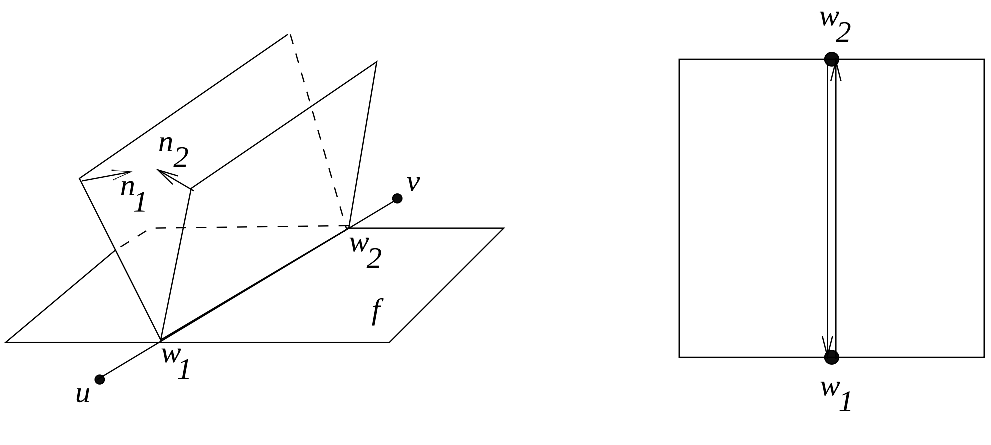


Figure 3.19 Transfer for Degenerate Edge/Face Intersection to f

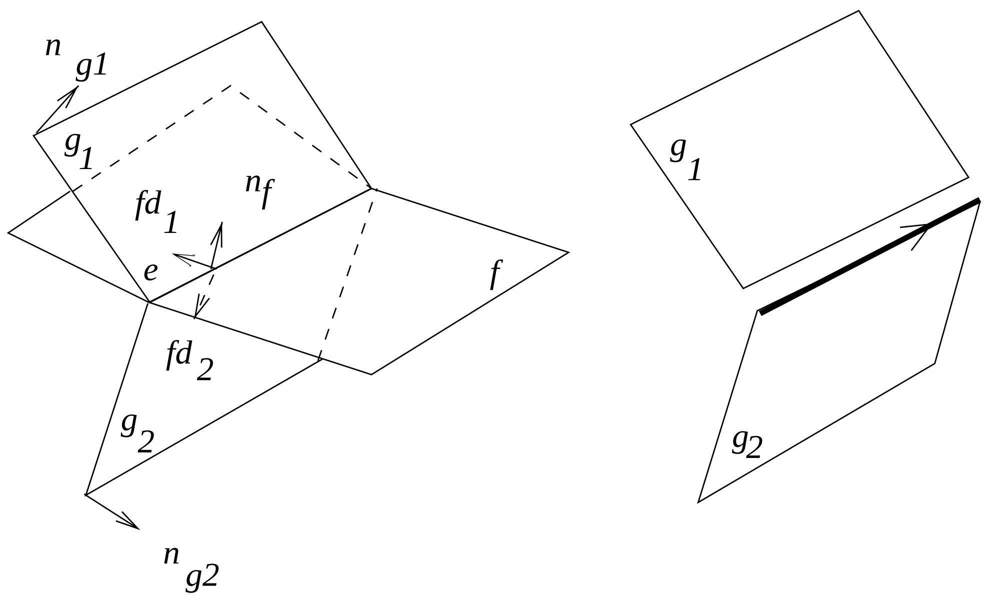


Figure 3.20 Transfer for Degenerate Edge/Face Intersection to g_i

faces adjacent to e. If so, the segment is transferred to f with the appropriate orientation. In consequence, f receives zero, one, or two directed lines for each segment of e contained in f. Figure 3.19 shows an example in which two segments of opposite orientation are transferred.

Next, consider the face g adjacent to e. By computing the dot product of the face direction vector of g with the normal of f, we determine whether g extends locally into the interior of A. If so, every segment of e is transferred to g, in the same orientation as e has on the boundary of g. Otherwise, no segment is transferred. See also Figure 3.20.

Edge/Edge Intersection

Assume that edge e of A intersects edge e' of B. In the generic case, shown on the left of Figure 3.21, the edges intersect in a single point w that subdivides both edges. This case can be considered as several edge/face intersections, one for each face f adjacent to e. At most two of these cases can be degenerate. Thereafter, we determine whether e and/or e' contain segments bounded by w that lie in the interior of the other solid. This requires the line/solid classification described previously.

The degenerate edge/edge intersection case arises when the two edges are collinear and overlap, as shown in Figure 3.21 to the right. The case generalizes degenerate edge/face intersection. The segment of overlap is transferred in the appropriate direction to each of the adjacent faces whose face direction vector(s) split a volume-enclosing pair of faces of the other solid. See also Figure 3.22.

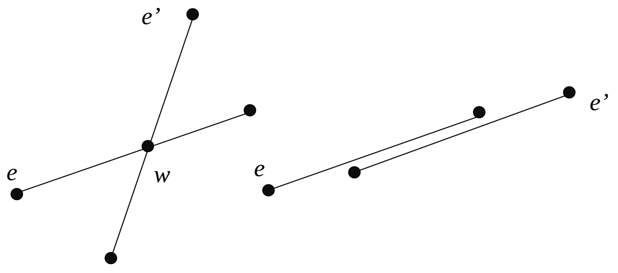


Figure 3.21 Edge/Edge Intersection

Vertex/Face Intersection

When a vertex u is in the interior of a face f, we have to satisfy adjacency constraints. Figure 3.23 shows an example. The faces g_k adjacent to u may intersect f in segments that must be incident to u. In addition, certain edges incident to u will extend into the interior of f. They are determined by computing dot products, and define edge segments interior to the other solid. Note that such segments may be further subdivided, as discussed before. Here u plays the role of w in the edge/face intersection analysis.

Vertex/Edge Intersection

Assume that vertex u of A intersects the interior of edge e of B. This case is conceptually a collection of vertex/face intersections, one for each face

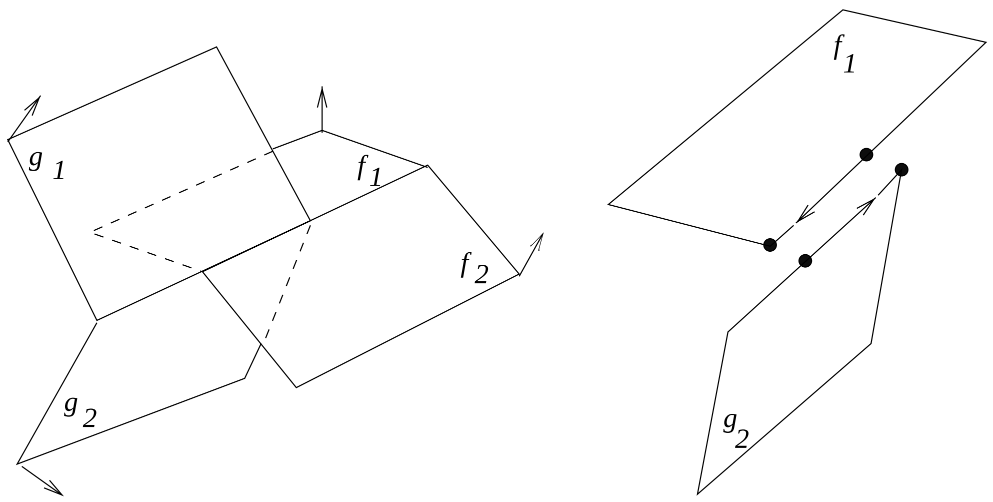


Figure 3.22 Transfer for Degenerate Edge/Edge Intersection

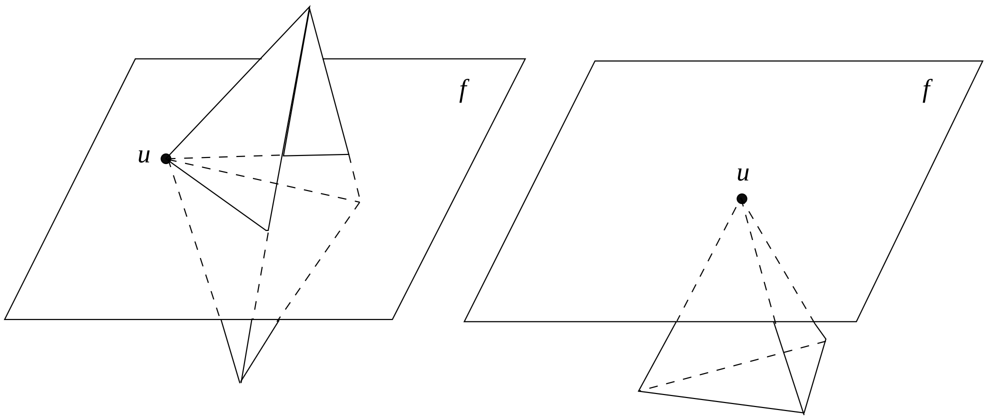


Figure 3.23 Vertex/Face Intersection

adjacent to the edge. The analysis of which edges of u extend into the interior of B is more complicated and is done as described in the edge/edge intersection analysis. In addition, the edge $e = (v, w)$ is subdivided by u, and we need to determine whether the segments (v, u) and (u, w) extend into the interior of A. See also Figure 3.24.

Vertex/Vertex Intersection

Assume that vertex v of A coincides with vertex u of B, as in Figure 3.25. We determine all edges incident to u that extend into the interior of B and, conversely, all edges incident to v that extend into the interior of A. This is essentially a line/solid classification; see Section 3.3.4. However, since the line is induced by the position of the intersecting solids, we cannot simplify the classification procedure by perturbing its position. In addition, we must

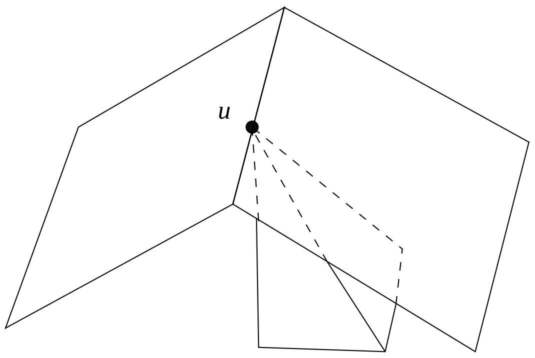


Figure 3.24 Vertex/Edge Intersection

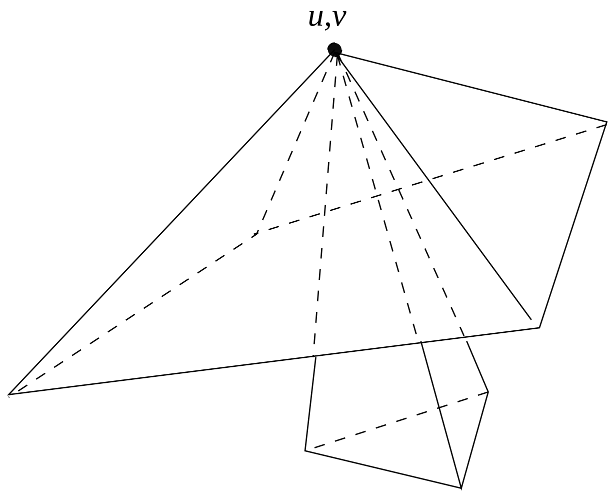


Figure 3.25 Vertex/Vertex Intersection

make sure that all intersection segments of adjacent faces are incident to u and v.

3.4.5 Face Subdivision

We have placed points and line segments subdividing edges and faces. We think of the points and segments as vertices and edges of a graph. We continue to refer to the graph vertices as points and to the graph edges as segments, to distinguish them from the edges and vertices of A and B. The purpose of neighborhood analysis has been to embed the graph consistently on both surfaces, and to obtain correct incidences at its points. After all face pairs have been intersected, the graph has the following properties:

1. If (u, v) is a segment that is not a complete edge, then the incidences at the points u and v are completely known.

2. If (u, v) is a segment and the incidences at u are not completely known, then u is a vertex of, say, A, and the missing incidences at u are initial segments of all edges of A incident to u.

Property 1 follows from the neighborhood analysis. For property 2, observe that (u, v) must be an edge of A that is included in the graph because v intersects the surface of B and (u, v) extends into the interior of A. Since (u, v) is not subdivided further, no other point of it can intersect the boundary of B. Hence the vertex u must be an interior point of B. Let (u, w) be any edge of A. Since u is in the interior of B, either (u, w) is contained in B, or an initial segment (u, w_1) of it is in B. In the latter case, the point w_1 is an intersection with the surface of B, and has already been discovered.

Thus, we can subdivide all faces of A and of B that intersect the boundary of the other solid by exploring the subgraph consisting of all points and segments contained in a face f, adding, when needed, undivided edges (u, w) of f adjacent to a vertex u that is not a point in the subgraph:

1. Initialize a list L of all points in f.
2. If L is empty, stop. Otherwise, initialize the stack S to contain a point u in L.
3. If S is empty, return to step 2. Otherwise, pop u from S and delete it from L. Mark u as explored.
4. Let E_1 be the set of all segments incident to u contained in f. If u is not a point, then let E_2 be all edges of f not containing a point; otherwise, E_2 is empty.
5. Order the edges and segments in $E_1 \cup E_2$ cyclically about u in the plane of f, and construct area-enclosing pairs.
6. For each (u, w) or (w, u) in $E_1 \cup E_2$, stack w if it is unexplored. Then return to step 3.

When the exploration is finished, we have a complete subdivision of f from which we obtain faces of $A \cap^* B$ by organizing into cycles the segments and edges considered and determine how they may be nested. Note that the algorithm is organized as a depth-first search.

3.4.6 Adjacencies in the Result

After completion of step 2, we are now ready to find the missing faces of $A \cap^* B$. The missing faces are those faces of A that are in the interior of B, and, vice versa, the faces of B that are in the interior of A. They are found by considering edge adjacencies.

We recall from the neighborhood analysis that the face adjacencies of each segment must be known, since at least one point of the segment is on the boundary of one of the solids. Hence, missing faces are precisely those faces that are edge adjacent to an undivided edge (u, w), added in the face-subdivision phase just described, or faces that are vertex adjacent to an undivided edge (u, v) that is a segment with u not being a point. Again, by exploration of the adjacency structure, all such faces can be found:

1. Let F_1 be the set of all faces of $A \cap^* B$ constructed by the subdivision given previously, and mark them as unprocessed. Set F_2 to the empty set.
2. If all faces in $F_1 \cup F_2$ have been processed, then stop. We have found all faces of $A \cap^* B$.
3. For all unprocessed faces f in $F_1 \cup F_2$, mark f as processed. For each edge (u, v) of f where u is not a point, add to F_2 all faces incident

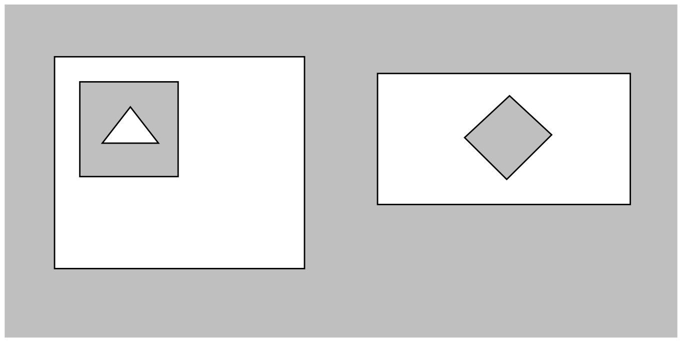


Figure 3.26 A Multishell Polyhedron in Two Dimensions

to u in A or in B that have not been subdivided, and mark them as unprocessed.

Note that F_2 accumulates all faces of A and of B that are contained in the interior of the solid. When implementing the algorithm, it is useful to remember which vertices u have already been considered, to avoid redundant processing.

3.4.7 Single-Shell Intersection Summary

By analyzing the three-dimensional neighborhoods of face-pair intersections, we have found segments and points making up the curves of intersection. The complete neighborhood analysis implies that the graph is consistently embedded on the two surfaces, and that the adjacencies have been correctly determined, except for certain adjacencies that are inherited from the boundary cycles of the faces. Considering each face intersecting the boundary of the other solid, we have completed the graph and have constructed a consistent subdivision of those faces of each solid that are not completely in the interior of the other solid. Finally, by considering certain undivided edges, we completed the surface of $A \cap^* B$, adding those faces that are completely in the interior of the other solid.

3.5 Multishell Objects

In general, a multishell polyhedron A consists of several disjoint polyhedra $P_1, P_2, \ldots P_m$. At most one of these polyhedra has infinite volume, and the remaining ones have finite volume, since we required that solids have a finite, bounded surface. Figure 3.26 illustrates this in two dimensions.

Consider the infinite polyhedron P_1. It fills the entire space except for a finite number r of voids that contain the remaining polyhedra $P_2, \ldots P_m$.

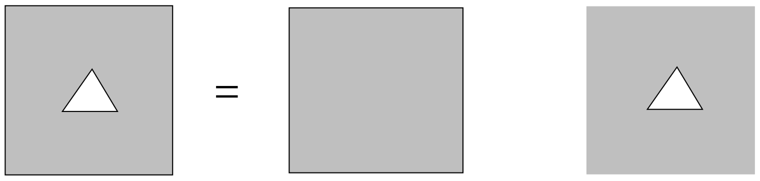


Figure 3.27 Polyhedron as Intersection of Single-Shell Polyhedra

Each void is bounded by a shell S that, taken separately, bounds a single-shell infinite polyhedron; see also Figure 3.27. Moreover, the forest of shell trees of A contains r trees, each root representing one of the voids of P_1. We think of such a polyhedron as the intersection of r single-shell polyhedra with disjoint shells.

The finite polyhedra P_i are bounded by an *exterior*, positive shell and may have internal voids in turn. Again, we think of each polyhedron as the intersection of several single-shell polyhedra. One of them, bounded by the external shell, has finite volume. The others are of infinite volume and are bounded by the shells of the internal voids.

We therefore write $A = P_1 \cup P_2 \cup \ldots \cup P_m$, where each component is the intersection of single-shell polyhedra. Likewise, $B = Q_1 \cup Q_2 \cup \ldots \cup Q_r$. Since the P_i are pairwise disjoint, as are the Q_i, the intersection of A and B is

$$A \cap^* B = (P_1 \cap^* Q_1) \cup (P_1 \cap^* Q_2) \cup \ldots \cup (P_1 \cap^* Q_r) \cup (P_2 \cap^* Q_1) \cup \ldots \cup (P_m \cap^* Q_r)$$

Each intersection $C_{ik} = P_i \cap^* Q_k$ is the intersection of a set of single-shell polyhedra. Moreover, the polyhedra C_{ik} must be disjoint; thus, their union is trivially determined. It follows that the intersection of multishell polyhedra reduces to the simultaneous intersection of several single-shell polyhedra.

Now it is simple to modify the single-shell intersection algorithm to intersect more than two shells at once, because the algorithm makes no essential use of the fact that the surface of single-shell polyhedra is connected, except for the conclusion that two shells either intersect or else must be analyzed with a shell-containment test. In the more general setting, therefore, the algorithm is run as before, subdividing the appropriate faces of intersecting shells and merging these shells through surface exploration. Thereafter, we consider the remaining, nonintersecting shells, classifying them by the shell-containment test, adding or deleting shells as required.

3.6 Complement, Union, and Difference

To complement an object, we must reverse the surface orientation. This is done as follows:

1. Multiply each face equation by -1, thereby inverting the orientation of the face interior.
2. Reverse the orientation of every boundary vertex cycle of a face.
3. Change the pairing of volume-enclosing pairs by moving it over by one entry, reverse the edge direction, and reverse the cyclic order of adjacent faces. Thus, the pairing $((a, b), (c, d))$ in the cyclical order (a, b, c, d) is changed to $((a, d), (c, b))$.

The forest of shell trees is modified by complementing at each node the indication of whether the shell at that node encloses finite or infinite volume. Note that complementation takes time proportional to the size of the boundary representation; that is, it is linear in the number of faces.

To compute the union of two objects, we apply de Morgan's law and compute instead $\neg(\neg A \cap^* \neg B)$. The difference $A -^* B$ is computed as $A \cap^* \neg B$.

3.7 Face-Boxing Techniques

An *iso-oriented box* is a parallelepiped whose sides are parallel to the coordinate planes. We seek a fast algorithm for reporting all intersecting pairs among a set of iso-oriented boxes, so as to reject as nonintersecting certain face pairs in polyhedral intersection. For each face, the smallest iso-oriented box is used that completely contains the face. For a planar face, the box is found by determining the maximum and minimum coordinates of the vertices of the face, and each box can be specified as three intervals, $[x_0, x_1]$, $[y_0, y_1]$, $[z_0, z_1]$, that specify the extreme coordinate values. The algorithm developed here solves the following problem:

> **Problem**
> Given n iso-oriented boxes, report in $O(n \log^2(n) + J)$ steps all intersecting pairs of boxes, where J is the number of pairs reported.

First, we develop the algorithm by considering one- and two-dimensional versions. In the two-dimensional case, we seek a fast algorithm for reporting intersections among iso-oriented rectangles. This problem can be solved in $O(n \log(n) + J)$ steps, as can the one-dimensional interval-intersection problem.

After giving an $O(n \log(n) + J)$ algorithm for rectangle intersection, we then develop an $O(n \log^2(n) + J)$ algorithm for the same problem. Although slower, this algorithm then allows us to intersect boxes within the same time bound. Various data structures will be needed, and these are explained first.

3.7.1 The Static Interval Tree

This section develops an algorithm for the interval-intersection problem:

> **Problem**
> Given a set S of intervals $I_1, ..., I_n$ of the real line, store them in a data structure such that, for any given query interval Q, we can determine all intervals in S that intersect Q. Moreover, the time needed to find the intersecting intervals is proportional to $\log(n)$ and to the number of intersecting intervals.

This problem is *static* in the sense that the set S is fixed. Eventually, the solution is adapted to a situation in which the set S changes, thus solving a *dynamic* version of the problem. This change will be easy after the static solution has been understood.

The algorithm for interval intersection uses a tree as basic data structure, to direct the search for intersecting pairs. The nodes of the tree are annotated with various additional data structures. The underlying tree will be called a *range tree*. After suitable embellishments, we obtain from it an *interval tree*. We explain the structure and construction of these trees in stages. The algorithm will integrate these stages.

Assume that the intervals are given as the pairs of numbers

$$[a_1, b_1], \ldots, [a_n, b_n]$$

The range tree T carrying the interval information is a binary tree whose leaves are the distinct values a_k and b_k, in ascending order. Let u be an interior node of T, with left subtree T_1 and right subtree T_2. Then the node u contains a number x that lies between the maximum leaf value in T_1 and the minimum leaf value in T_2. This number is called the *split value* of u, and will be denoted $\texttt{split}(u)$.

Example 3.6: Assume we have intervals $I_1 = [-1, 6]$, $I_2 = [-2, 3]$, $I_3 = [0, 4]$, $I_4 = [2, 7]$, $I_5 = [3, 4]$, and $I_6 = [-2, -1]$. The range tree T is shown in Figure 3.28. As split values, we have chosen the arithmetic mean of the maximum and minimum leaf values in the left and right subtree, respectively. ◇

Recall that the half-open interval $(x, y]$ is the set $\{z | x < z \leq y\}$. Each node in the tree represents a half-open interval $(x, y]$ of the real line, called its *range*. The tree root represents the half-open real line $(-\infty, \infty]$. Let u be a node in the tree representing the range $(x, y]$, and assume that $\texttt{split}(u) = s$. Then the left child of u represents the range $(x, s]$, and the right child represents the range $(s, y]$. In the preceding example, the root represents the range $(-\infty, \infty]$. The left child, with split value -0.5, represents the range $(-\infty, 2.5]$, and the right child, with split value 5, represents the range $(2.5, \infty]$. The interior node with split value 3.5 represents the range $(2.5, 5]$.

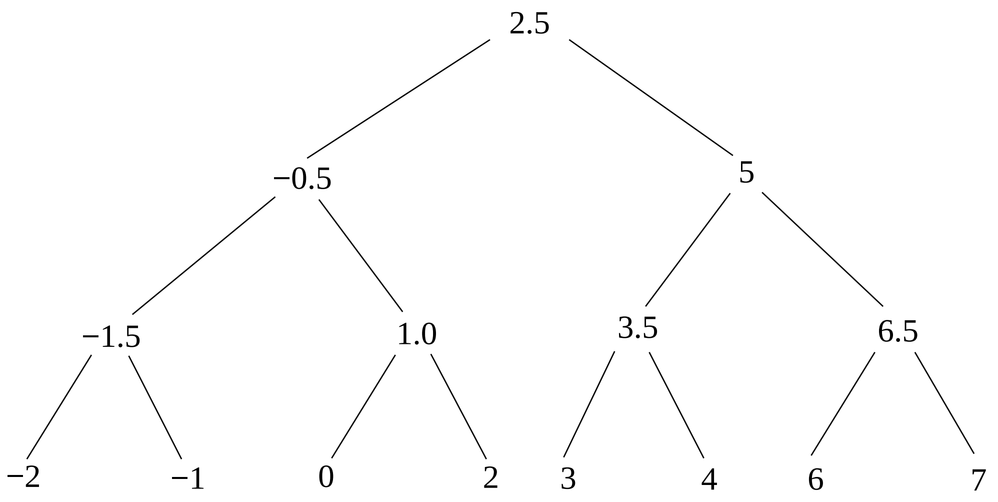


Figure 3.28 Range Tree T

Given a set S of intervals, the range tree is easy to construct: Sort all interval endpoints in ascending order into a list $L = (x_1, x_2, \ldots, x_m)$, where $m \leq 2n$. For L we construct a range tree top-down by splitting L into two lists of equal size, $L_1 = (x_1, \ldots, x_p)$ and $L_2 = (x_{p+1}, \ldots, x_m)$, where $p = \lceil m/2 \rceil$. The tree's root has the split value $(x_p + x_{p+1})/2$. The left and right subtrees are now constructed from L_1 and L_2, respectively, in the same way. Note that the range-tree construction requires $O(n \log(n))$ steps.

At the nodes of the range tree, we store the intervals of S. The interval $[a, b]$ is stored at the unique node u satisfying the following:

1. $[a, b]$ is contained in the range of u.
2. $[a, b]$ contains $\texttt{split}(u)$, but does not contain the split value of any ancestor of u.

All intervals at u are stored in two lists, with the left endpoints in the list $\texttt{left\_search}(u)$, sorted in ascending order, and the right endpoints in the list $\texttt{right\_search}(u)$, sorted in descending order.

Example 3.7: In the range tree of Example 3.6, the intervals I_1, I_2, I_3, and I_4 are all stored at the tree root, the interval I_5 is stored at the node with split value 3.5, and the interval I_6 at the node with split value -1.5. $\diamond$

The intervals are stored in the tree in two phases. In the first phase, the left search lists are constructed. Then, by an obvious modification, the right search lists are constructed.

1. Sort all intervals of S by the left endpoint a_k in ascending order into a list L.
2. Propagate the list down into the tree, beginning at the root.

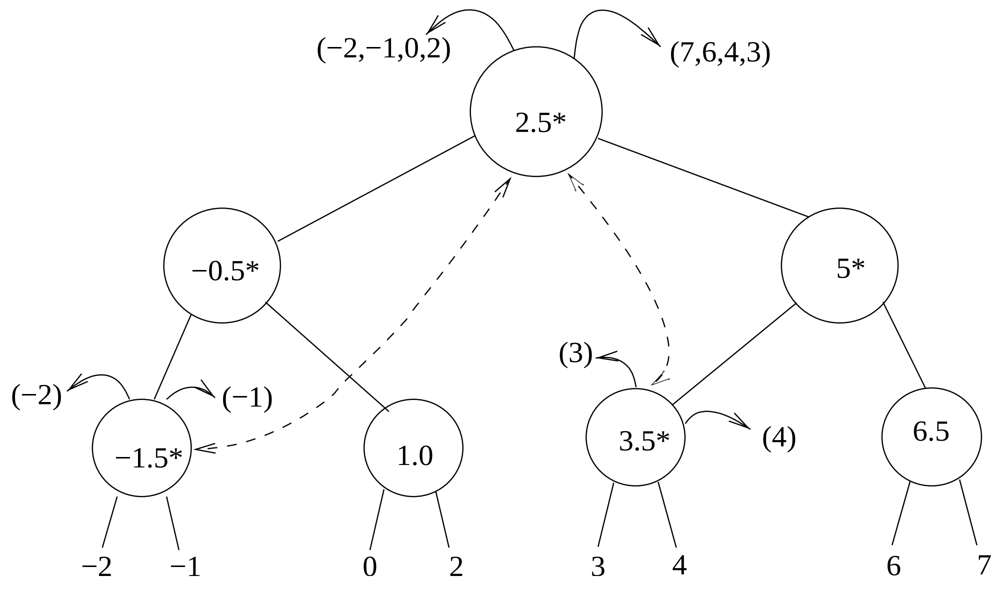


Figure 3.29 Interval Tree T

3. At each node u, break the incoming list L into the lists L_1, L_2 and L_3, where L_1 contains the intervals $[a, b]$ with $b <$ `split`(u), L_2 contains the intervals containing `split`(u), and L_3 contains the intervals $[a, b]$ with `split`$(u) < a$.

4. Propagate the list L_1 to the left descendant of u, propagate the list L_3 to the right descendant of u, and store the intervals in L_2 at u.

The list L is split by sequentially scanning it. Each entry in it is scanned once at each node at which the containing list is considered. Thus, it is scanned at most $\log(n)$ times. Therefore, all intervals are added to the range tree in $O(n\log(n))$ steps.

Recall that in a *preorder* traversal of a binary tree, the root of a subtree is visited, followed by its left and right subtrees being visited. By applying this rule recursively, beginning with the root of the tree, all nodes are visited in *preorder*.

The data structure, as developed, is not sufficiently flexible. The problem, briefly, is that the intervals may be clustered at very few tree nodes, so locating nodes with stored intervals may require too much searching. Thus, we link all nodes containing stored intervals in a doubly linked list. In this list, the nodes are in preorder. Furthermore, each tree node is marked if it, or any descendant of it, contains stored intervals. It is clear that these annotations can be added in $O(n\log(n))$ steps. After we add these annotations, we have now constructed an *interval tree*. The interval tree for the intervals of the example is shown in Figure 3.29. The node marks are represented by asterisks. The doubly linked list of nonempty tree nodes is shown by dashed arrows.

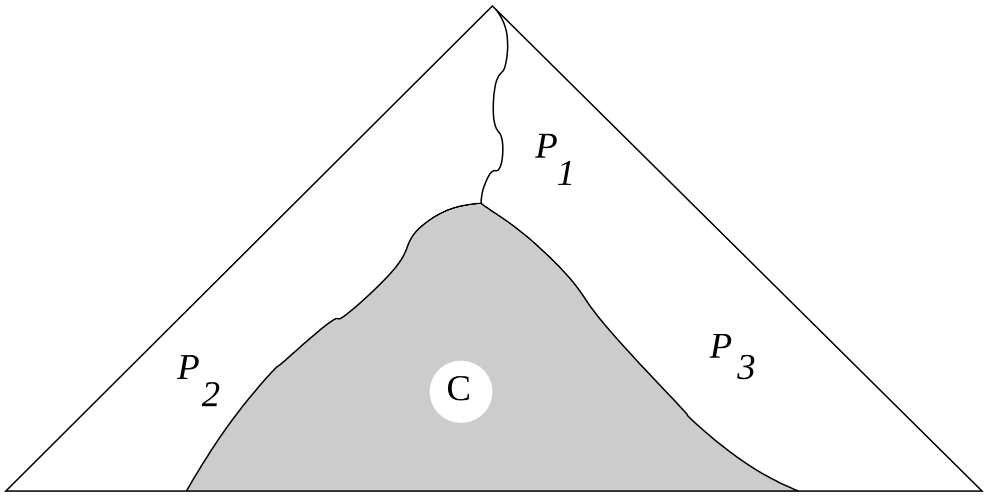


Figure 3.30 Node Sets Used for Querying for Interval Intersection

3.7.2 Static Interval Query

The problem of static interval query is solved using an interval tree. Let S be the set of intervals, $Q = [a, b]$ the query interval. We construct an interval tree for S.

In the interval tree, we identify a node set P, consisting of all nodes u whose range has a nonempty intersection with the query interval but is not completely contained in Q. We also identify a set C of nodes each of whose range is contained in Q. Since, at each level in the tree, the set of represented ranges is a partition of the root range, the set P has at most two nodes at each depth, and consists of three paths, P_1, P_2, and P_3. Here P_1 consists of all nodes whose ranges contain Q. The path P_1 begins at the root and ends at the node u_1 whose split value is in Q; the path P_2 consists of all nodes whose range contains the left endpoint a of Q, but not all of Q; and the path P_3 consists of all nodes whose range contains the right endpoint b of Q, but not all of Q. Figure 3.30 shows these node sets schematically. Note that the nodes of C are in subtrees rooted in right descendants of nodes in P_2, or in subtrees rooted in left descendants of nodes in P_3. If the query interval is a point (i.e., if $a = b$) then the sets P_2, P_3, and C are empty.

Example 3.8: Let $Q = [2, 5.5]$ be a query interval to be tested against the intervals I_1 through I_6. In the interval tree of Figure 3.29, the root is the only member of the set P_1, since its split value is in Q. The set P_2 consists of the nodes with split value -0.5 and 1, and the leaf labeled 2. The set P_3 consists of the nodes with split value 5, 6.5, and the leaf labeled 6. The set C contains only those nodes in the subtree whose root has split value 3.5. $\diamond$

The overall structure of the algorithm for reporting all intersections with the query interval is now as follows:

1. Construct an interval tree for S.
2. Identify the path P_1.
3. Identify the paths P_2 and P_3.
4. By processing the path nodes and their descendants appropriately, identify all intersecting intervals.

Step 2 is performed as follows. With u initially the tree root, identify the path P_1 by selecting the left (resp., right) descendant of u provided the split value $s = \texttt{split}(u)$ satisfies $b < s$ (resp., $s < a$). We continue until we encounter the first node u_1 whose split value is contained in the interval Q.

At u_1, we initiate construction of P_2 and P_3. The path P_2 begins at the left descendant of u_1 and is identified as follows. At node u, select the left (resp., right) descendant of u as the next node provided that $a \leq \texttt{split}(u)$ (resp., $a > \texttt{split}(u)$). The path P_3 begins at the right descendant of u_1. To identify the remaining nodes, pick at node u the left (resp., right) descendant provided that $b \leq \texttt{split}(u)$ (resp., $b > \texttt{split}(u)$). For degenerate query intervals $[a, a]$, the paths P_2 and P_3 are empty. In this case, care must be exercised if $a = \texttt{split}(u)$ for an interior node u.

We now discuss how to identify intersecting intervals along the paths P_i, $i = 1, 2, 3$. Let u be a node on one of the paths and $s = \texttt{split}(u)$. If $s < a$, then an interval stored at u intersects Q iff its right endpoint y satisfies $a \leq y$. Hence, by scanning the right endpoints of intervals stored at u, in descending order, we identify all intersecting intervals in time proportional to their number. If $b < s$, then an interval at u intersects Q iff its left endpoint x satisfies $x \leq b$. These intervals are found by scanning the left endpoints in ascending order, in time proportional to their number. Finally, if s is in Q, then all intervals at u intersect Q.

The nodes in C are in subtrees rooted in right descendants from nodes in P_2, and in left descendants of nodes in P_3. They are characterized by the fact that the range represented at any node in C is completely contained within Q. Hence, all intervals stored at such nodes will intersect Q. For the sake of speed, we must avoid searching through all nodes of these subtrees. Intuitively, we find the first node in the set C that contains intervals, using the node marks. To do so, we examine the nodes in P_2, beginning at the leaf and ending at the left descendant of u_1, until we find a marked node that has a marked right descendant not contained in P_2. If no such node can be found, we try the analogous procedure with the nodes in P_3, beginning at the right descendant of u_1 and ending at the leaf, searching for a left descendant not in P_3. If no such node can be found, then C contains no intervals. Otherwise, we have found the leftmost node in C that contains intervals. The remaining nodes in C containing intervals are found by following the linked list. It is not difficult to see that the entire procedure requires $O(n \log(n))$ steps for constructing the interval tree, plus $O(\log(n) + J)$ steps to report all intersections, assuming J intervals intersect Q.

3.7.3 Rectangle Intersection

We are given a set of rectangles whose intersecting pairs we want to find. Each rectangle is given as the pair of intervals $[x_0, x_1] \times [y_0, y_1]$, the rectangle's projection on the x and y axis, respectively. The algorithm will use interval intersection as one of its operations.

Rectangle intersection will be based on the *line-sweep paradigm*: By sweeping a line that is parallel to the x axis, in increasing y direction, any intersecting rectangle pair must appear as an intersecting pair of x intervals. These intervals are the intersection of the rectangles with the line. As the line sweeps upward, the interval $[x_0, x_1]$ of the rectangle $[x_0, x_1] \times [y_0, y_1]$ appears at the line position y_0, and disappears at the line position y_1.

We sort the numbers x_i and build an interval tree from them that initially does not contain any stored intervals. In this way, we will be able to store each interval at some time during rectangle intersection. Next, we sort the numbers y_i in ascending order, and consider them in sequence. For each particular number y, we have a set of rectangles $[x_0, x_1] \times [y, y_1]$ beginning at y, and another set of rectangles $[x'_0, x'_1] \times [y_0, y]$ ending at y.

The beginning rectangles define a set of x intervals that must be inserted into the interval tree. Before insertion, each of these intervals is tested for intersection with the intervals already in the tree. Intersecting intervals correspond to rectangle intersections. Thereafter, the x intervals of ending rectangles are deleted from the tree. It is clear that the outlined algorithm is correct, and that it does not report an intersecting pair of rectangles more than once. Moreover, rectangle intersection can be reported quickly, based on the algorithm for interval intersection, provided we can insert and delete intervals efficiently.

Example 3.9: In Figure 3.31, we are at a y position at which we must insert the interval I_4 corresponding to rectangle 4, and delete the interval I_2 corresponding to rectangle 2. Before deleting I_2, we use I_4 as query interval and report all intersections. Then we delete I_2 and insert I_4. $\diamond$

To support quick interval insertion and deletion, we must modify the `left_search` and `right_search` lists, organizing them as balanced search trees rather than as lists. These trees must support logarithmic-time insertion and deletion, and they must allow linear-time sequential access to the stored values in sorted order. For example, a 2-3 tree will satisfy these requirements.

An interval $Q = [a, b]$ is inserted as follows: The interval is added to the node u_1 that is last in the path P_1. Note that P_1 was found as part of finding all intervals in the tree that intersect Q. Insertion into the `left_search` and `right_search` trees is routine. If there are already intervals stored at u_1, we are now done. Otherwise, we must update the linked list of nonempty tree nodes and, possibly, the node marks on the path from u_1 to the tree root. Clearly, updating the node marks is trivial.

Recall how to visit the nodes of a binary tree in *inorder*. Beginning at the root of the tree, a node v is visited as follows: If v is a leaf, then visit

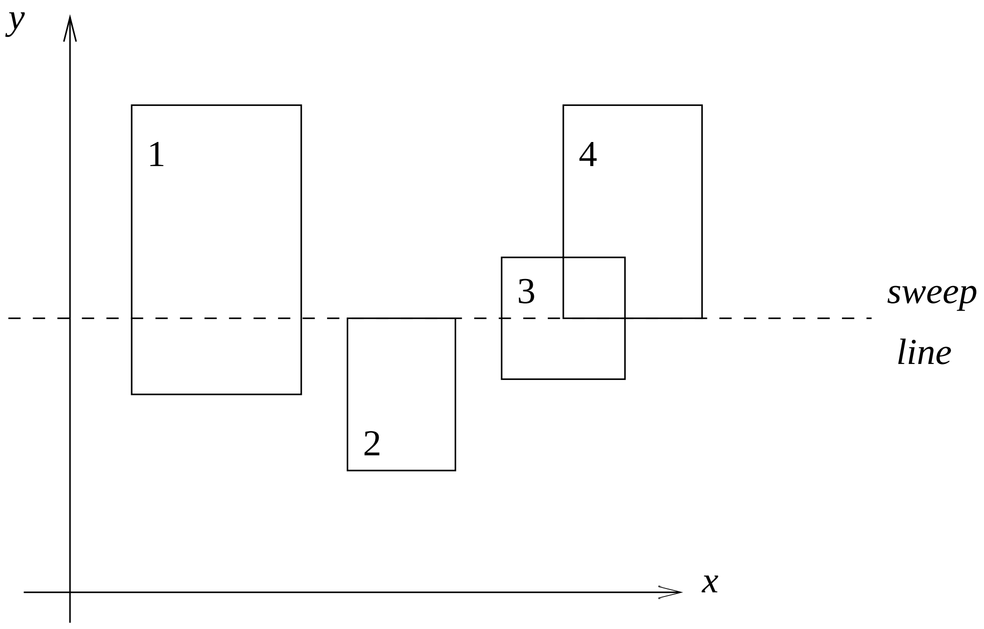


Figure 3.31 Interval Insertion During Rectangle Intersection

it; otherwise, visit recursively all nodes of the left subtree, then visit v, and finally visit the nodes of the right subtree. The order in which the tree nodes are encountered is called inorder.

To locate where to insert u_1 into the list of nonempty tree nodes, we search for the first nonempty node succeeding u_1 in inorder. This node is found using the node marks. If u_1's right descendant u_R is marked, then the needed nonempty tree node is found by exploring the leftmost marked path beginning with u_R. If that node is not marked, we must back up toward the root until we find a marked ancestor who is not empty or whose right descendant is not in P_1 and is marked. Clearly, locating this node requires no more than $O(\log(n))$ steps.

Now consider deleting an interval Q. We first find the node u_1 at which the interval is stored, and delete its endpoints from the search structures unless other intervals at u_1 share that endpoint. If Q was the only interval at u_1, we must delete u_1 from the linked list of nonempty nodes, and, possibly, delete the node mark of u_1 and some of its ancestors. Because the node list is doubly linked, node deletion is simple.

In summary, we have shown that intervals can be inserted and deleted in time proportional to $\log(n)$. In consequence, intersections among a set of n rectangles can be reported in $O(n\log(n) + J)$ steps, where J is the number of intersecting pairs. The simplicity of the algorithm makes its implementation quite practical.

3.7.4 The Segment Tree

The rectangle-intersection algorithm described previously is based on dynamic interval intersection. In principle, a similar extension to three dimensions is possible: Sweep a plane in the z direction and test, at certain positions, whether there are intersections among the rectangles in which the boxes intersect the sweep plane. We do not use the $O(n \log(n) + J)$ rectangle intersection for this purpose, since the recursive line sweeps for finding intersecting rectangles would consume too much time. Instead, we develop a more flexible data structure that supports rectangle intersection without a line sweep. This data structure is called a *segment tree*, and is an annotated balanced binary search tree, recording a set of intervals.

We are given n intervals $[a_k, b_k]$ with distinct endpoints c_i, where $1 \leq i \leq m$ and $m = 2n$. We assume that the c_k are enumerated in ascending order. The underlying binary search tree has $2m + 1$ leaves representing, from left to right, the partition of the real line induced by the endpoints.

$$(-\infty, c_1), [c_1], (c_1, c_2), [c_2], \ldots, [c_m], (c_m, +\infty)$$

The tree is constructed in much the same way as the range tree. Like the range tree, the interior nodes have split values to support binary search, only now we must indicate, at each node, whether the left descendant is chosen if the query value is less than, or not greater than, the split value. An interior node represents a segment that is the union of the segments of its descendants. We store an interval $I_k = [a_k, b_k]$ in this tree at a node u provided I_k contains the range of u but not the range of u's ancestor. When we have completed this annotation, we have constructed a segment tree. Note that I_k can be stored at more than one tree node, but not at more than $2h$ nodes, where h is the tree height. The ranges at which a given interval is stored are a partition of the interval. It is not difficult to see that a segment tree can be constructed in $O(n \log(n))$ steps.

Example 3.10: The segment tree for the intervals $I_1 = [2, 3]$, $I_2 = [5, 7]$, $I_3 = [2, 4]$, and $I_4 = [3, 5]$ is shown in Figure 3.32. The interval [3,5] is stored at the two nodes marked with asterisks. $\diamond$

Now consider the problem of locating all those intervals in a set S that contain a query point q. We proceed as follows. From S, we construct a segment tree. Using this tree as a binary search tree, we locate the leaf in whose range q lies. Clearly, all intervals stored at the nodes on the path from the tree root to the leaf containing q will contain q. Furthermore, since leaves represent disjoint segments, no other interval of S can contain q. Note also that no interval stored along the path is repeated. Therefore, all containing intervals can be found in $O(\log(n) + J)$ steps, ignoring the time for constructing the segment tree.

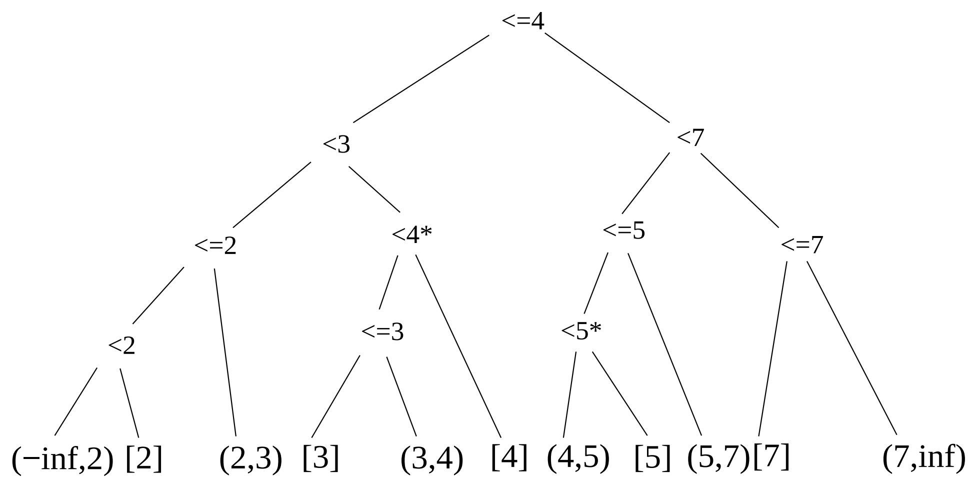


Figure 3.32 Segment Tree T_{seg}; Nodes Storing [3,5] Are Marked with Asterisks

3.7.5 Interval, Rectangle, and Box Intersection

In the one-dimensional case, we are given a set S of n intervals and $Q = [a, b]$, the query interval. We observe that an interval Q' in S intersects Q iff one of the following is true:

1. a is in Q'.
2. The left endpoint of Q' is in the half-open interval $(a, b]$.

We will need two search structures, one for each intersection criterion.

The first search structure is a segment tree T_{seg} for the intervals in S. The second search structure is a balanced binary search tree T_{bin}, built with only the left endpoints of intervals in S. Let x be a left endpoint of an interval in S, and l be the leaf at which x is stored. Then we store the interval belonging to x at every node of the path from the root to l. Thus, at each interior node u, we have stored all intervals whose left endpoints are the leaves of the subtree rooted in u.

We test interval intersection as follows. All intervals intersecting Q by the first criterion are found by searching T_{seg} for the intervals containing a. To find intersections by the second criterion, we locate in T_{bin} the search paths for a and for b. The intervals at the point of path bifurcation intersect Q by the second criterion, excluding those intervals that have the left endpoint a. Since the two intersection criteria are mutually exclusive, we have found all intersecting intervals without duplication. Clearly, the trees can be constructed in $O(n \log(n))$ space and time. So, $O(\log(n) + J)$ steps suffice to report all intersections with a query interval.

Example 3.11: Figure 3.33 shows the segment tree for the interval set S of Example 3.10, and Figure 3.34 shows the binary search tree. Assume

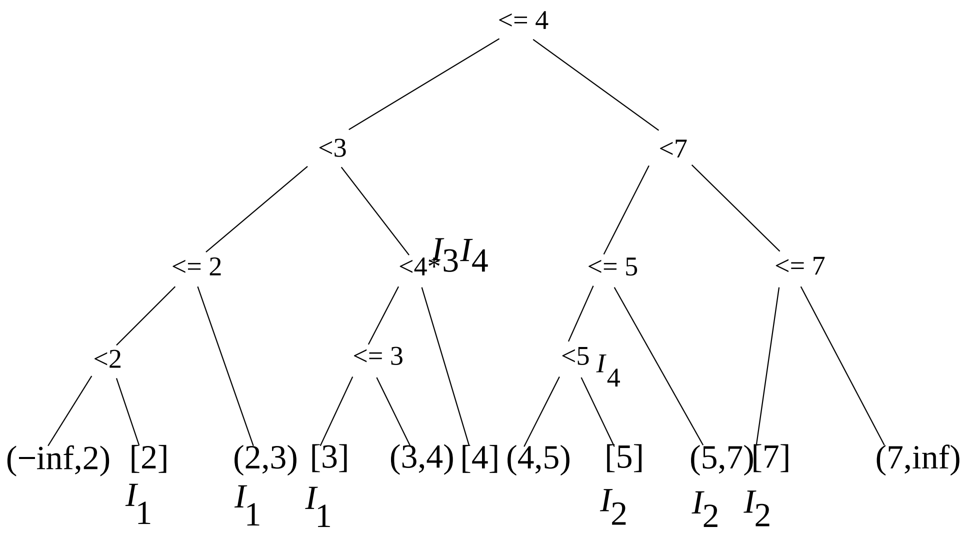


Figure 3.33 Segment Tree T_{seg} with Intervals

a query interval $Q = [4, 6]$. The search of the segment tree with the left endpoint 4 ends at the leaf [4]. So, the intervals I_3 and I_4 intersect Q by the first criterion. Next, we search the tree T_{bin} with the arguments 4 and 6. The search paths bifurcate at the leaf labeled 5; hence, the interval I_2 intersects Q by the second criterion. $\diamond$

Now consider locating all those rectangles in a set S that intersect the query rectangle $Q = [x_0, x_1] \times [y_0, y_1]$. The rectangles in S define a set of y intervals and a set of x intervals. We proceed as follows. For the y intervals, we construct the trees T_{seg} and T_{bin}. Consider the rectangles that correspond to a set of y intervals to be stored at each node in these trees. They induce, for each node u, a set of x intervals X_u. With X_u, we construct at each u a *nested* pair of trees $T_{seg}(u)$ and $T_{bin}(u)$.

We use these data structures as follows. With the y interval $[y_0, y_1]$, we locate all intersecting y intervals induced by S. Then, we solve separately the intersection problem for $[x_0, x_1]$ at each node. Clearly, we thus identify all intersecting rectangles.

Let S contain n rectangles. To understand space and time requirements, we recall that each interval is stored at no more than $O(\log(n))$ different nodes, in each tree. Thus, there are $O(n \log(n))$ x intervals for which we construct the nested pairs of trees initially. From this, it follows that the needed data structures can be built in $O(n \log^2(n))$ space and time, and that the query time for reporting all intersections with the query rectangle is $O(\log^2(n) + J)$.

Finally, consider box intersection. We can build a doubly nested data structure, or adopt the sweep paradigm. The first approach yields an $O(n \log^3(n)+ J)$ box-intersection algorithm; the second one yields the slightly faster $O(n \log^2(n)+$

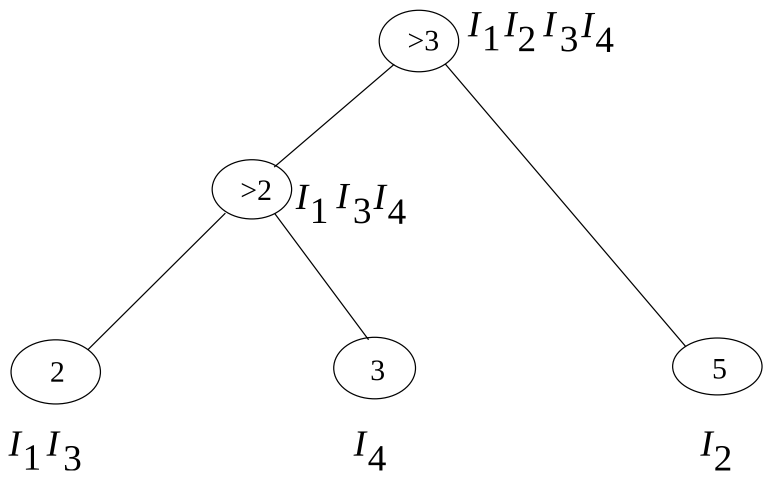


Figure 3.34 Search Tree T_{bin} with Intervals

J) method. In the sweep paradigm, we now have to enter and delete rectangles dynamically into the nested tree pairs. This can be done in a manner analogous to that used for the dynamic version of the interval tree.

3.7.6 Red-Blue Intersection

When enclosing faces with iso-oriented boxes, the boxes enclosing adjacent faces will intersect. However, intersections among boxes enclosing faces of the same polyhedron are not of interest. We discuss a way of excluding such intersecting pairs.

Given the polyhedra A and B, we color the boxes enclosing faces of A red, and color boxes enclosing faces of B blue. The *red-blue intersection* problem is to report intersecting pairs of red and blue boxes without spending time on finding, and ignoring, blue-blue and red-red intersecting pairs. Clearly, rectangle and interval intersection have analogous problem variants. We solve the red-blue box intersection problem using the static version of box intersection. Briefly, from the set of blue boxes we construct the needed data structures. Then, we use the red boxes to query the blue data structure for intersection. The details are straightforward, as are the details for red-blue rectangle intersection, and for red-blue interval intersection.

3.7.7 Implementation Remarks

Some form of box intersection should be incorporated into every solid modeler. However, many asymptotically fast algorithms have a considerable start-up cost and are intricate to program. Thus, simpler versions may be contemplated. For example, it may suffice to implement only rectangle intersection and to accept the lower resolution of nonintersecting face pairs. In our experiments, this strategy was satisfactory.

If we replace the balanced search trees at the nodes of the dynamic interval tree with lists, the programming required is further reduced. In small applications, there is no appreciable running-time difference. However, when many intervals are stored at the same node, then this simplification spoils the performance of the algorithm. Ultimately, the attractiveness of these compromises depends on the mix of applications.

The box-intersection algorithm is somewhat harder to implement than is rectangle intersection based on line sweep. Moreover, the segment tree appears to be less robust because of the many leaves that represent point intervals. For this reason, the boxes to be tested for intersection should be enlarged by a small tolerance ϵ. This will make box intersection robust.

3.8 Notes and References

The conceptual topological data structures explained in Section 3.2 have been designed for convenience of accessing various adjacencies. Irredundant representations will store less information. See Weiler (1986) for a study of irredundant topological data structures. Irredundancy of the geometric data is motivated by robustness considerations. As mentioned, geometric irredundancy for curved solids might not be cost-effective, but perhaps an attractive alternative would be to annotate the data structures with *approximate* geometric data to be made precise by a suitable numerical computation when needed.

Boolean operations on polyhedra in B-rep have been implemented many times. Descriptions include Braid (1975), Hillyard (1982), Mäntylä (1986), Okino et al. (1973), Voelcker et al. (1974), and Wesley et al. (1980).

The brief description in Requicha and Voelcker (1985) stresses the important role of local neighborhood analysis. The description does not assume manifold boundaries. Mäntylä (1988) describes an intersection algorithm in considerable detail, but restricts himself to manifold solids. Moreover, his algorithm is based on a symmetric design in which the role of A and B is interchanged for the purpose of face subdivision; hence, one should expect robustness problems. Chiyokura (1988) describes a similar algorithm, also restricted to manifold polyhedra. Both methods use Euler operators to implement surface subdivision.

Paoluzzi et al. (1986) describe a polyhedral modeler for which all faces must be triangles. Although the surfaces topologically are manifolds, surface structures may coincide geometrically. Thus, the algorithm must disambiguate the topology in the manner described in Section 2.3.1 of Chapter 2. Laidlaw et al. (1986) describe a method in which all faces must be convex polygons, but the solid surfaces need not be manifolds. By carefully observing properties of convex polygons, their algorithm attempts to increase local robustness. However, no global processing is done to ensure consistency of the resulting surface structures, and they experience algorithm failures for certain inputs. The observation that two or three random perturbations suffice to eliminate complicated vertex intersection cases in line/solid classifications is due to Laidlaw et al. (1986).

Hoffmann, Hopcroft, and Karasick (1987) and Karasick (1988) describe an algorithm based on the conceptual structure discussed at the beginning of Section 3.4. For robustness purposes, the algorithm was subsequently modified, adding a form of local neighborhood analysis to achieve a consistent subdivision of adjacent faces. The processing is somewhat more complicated, because the underlying structure of the algorithm requires more special-case processing than the version we give here. However, the Karasick modeler also includes *global* consistency computations that explore the consequences of, say, nonincidence between a vertex v and a face f along a path of edges beginning at v. See Karasick (1988) for a detailed description.

Euler operations are often mentioned as a conceptual infrastructure with which to implement higher-level geometric operations, including solid intersection. In our view, operations such as line/solid classification or sorting and pairing points along a line should also be considered part of the infrastructure. Implementing the placement of points and segments with help of Euler operations has the advantage that at all times A and B have valid boundary description. However, the orientation information needed to later interpret points and segments as part of face subdivisions would require extensions to Euler operations or a subsequent separate neighborhood classification. We prefer to combine this classification with the placement of points and lines, for robustness reasons.

Box intersection and related techniques are described in the computational geometry literature. Our description of these algorithms follows Mehlhorn (1984). Many of the technical concepts in that section are standard in the literature on algorithms. See, for example, Aho, Hopcroft, and Ullman (1974) for tree traversals, balanced binary search trees, and other basic techniques. The method to solve the red-blue version of the problem was suggested to me by M. Sharir. Rectangle intersection was implemented by J. Sasaki in 1986 for the Karasick modeler. Karasick (1988) reports that it speeds up polyhedral intersection to almost an $O(n\log(n))$ behavior when intersecting several hundred randomly positioned cubes.