GEOMETRIC MODELING: A First Course

6. Fundamental Algorithms

Fundamental algorithms are the building blocks we use to construct computational solutions to application problems. This chapter covers some of the fundamental algorithms that underly many geometric modeling computations. We begin with a short introduction that emphasizes the connections between algorithms and the representations on which they operate.

6.1 Algorithms and Representations

We consider a very simple example, which is a lower-dimensional version of the familiar problem of computing the image of a 3-D object by orthographically projecting it onto a 2-D screen. Our objective is to design an algorithm for computing the orthographic projections of convex polygons on a 1-D screen. 1-D images are not visually very exciting (to put it mildly), but they are simple, and suffice to illustrate some important concepts.

A convex polygon may be defined as the convex hull of a set on non-collinear points—see Chapter 3. The orthographic projection of a set S on a line L may be defined as follows. First we choose a coordinate system such that L coincides with the x axis. Then, for each point $\mathbf{p} = (x, y)$ of S, we construct the projected point $\mathbf{q} = (x, 0)$. The set of all such $\mathbf{q}$ is the projection we want. With these definitions, we can state our problem in standard mathematical terms:

Given: A convex polygon S
Find: The orthographic projection of S on a line L

This is a well-defined mathematical problem, but it is *not* a well-posed computational problem, because we have not specified how the polygon is to be "given", and what is the format of the result. In other words, we have not specified *how the input and output are represented*.

It is also interesting to note that the definitions of convex polygon and projection are mathematically correct but not computationally effective, in the sense that they cannot be *directly* embodied in algorithms. A convex hull is the smallest convex set that encloses the given points, and the projection is obtained by zeroing the y coordinate of every point of the polygon. Both of these are infinite processes that cannot be implemented directly in computers. We cannot compute all the enclosing convex sets to choose the smallest one, nor can we project and infinite set of points on a line. This does not mean we cannot compute projections of convex polygons. Although a polygon contains an infinite number of points, it can be represented by a finite number of vertices.

Let us specify the input and output representations as follows. The convex polygon is represented in Scheme 2 of Chapter 3, *i.e.*, by a list of its vertex coordinates; the projection is represented by the *x* coordinates of its endpoints. This output representation is unambiguous because the orthographic projection of a convex polygon on a line is a line segment, as shown in Figure 6.1.1. Now we have a well defined computational problem, and can go ahead and design algorithms to solve it. The following are two possible solutions, expressed in a pseudo-C++ language. We assume the existence of classes `PairOfReals` and `VertexList`, with the obvious meanings, and functions `SortX`, which sorts the `VertexList` by *x* coordinate, `FirstX` and `LastX`, which return the *x* coordinates of the first and last element of a `VertexList`, and `MinX` and `MaxX`, which find the minimum and maximum *x* coordinates in a `VertexList.`

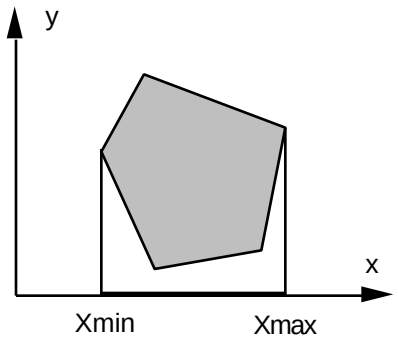


Figure 6.1.1 – Projecting a convex polygon on the *x* axis

Algorithm 1:

```
PairOfReals Project1 (VertexList vl) {
     VertexList Sorted = SortX (vl);
     float Xmin = FirstX (Sorted);
     float Xmax = LastX (Sorted);
     return (Xmin, Xmax);}
```

Algorithm 2:

```
PairOfReals Project2 (VertexList vl) {
     float Xmin = MinX (vl);
     float Xmax = MaxX (vl);
     return (Xmin, Xmax);}
```

Algorithm 1 sorts the vertex list by increasing value of *x* coordinate, and returns the first and last elements of the sorted list. Algorithm 2 traverses the list and extracts the elements with maximum and minimum *x* coordinates. It is clear that both algorithms are correct, and therefore they are functionally equivalent. Is one better than the other? To answer this question we need criteria for comparing algorithms.

The most commonly used criterion is efficiency, which is studied in the Computer Science field of analysis of algorithms. Typically, the worst-case running time for an input of size *n*, as *n* tends to infinity, is taken as the measure of efficiency. Asymptotic worst-case performance for geometric algorithms tends to be very pessimistic, and is of limited

practical importance because the worst cases tend to be "pathologic" and infrequently encountered. A more relevant measure of complexity should capture the cost of computation for "most" cases, or for "average" objects. Unfortunately, this is usually hard to assess because of a lack of meaningful statistical models for geometric entities (e.g., what is an average polygon?), or mathematical intractability of the models. (A promising approach is the use of randomized algorithms, in which randomness is injected in a controlled fashion into a deterministic computation—see e.g. [Mulmuley 1994].) For our example, it is well-known that the worst-case complexity of sorting n real numbers is of order $n \log n$, whereas the complexity of minimum or maximum computations is only of order n. Therefore Algorithm 2 is asymptotically more efficient than Algorithm 1 in the worst case sense.

But efficiency is not the only relevant criterion. It is the easiest to characterize formally, and also the best understood. Other criteria include:

- Robustness in the presence of numerical errors, such as those introduced by floating point computation.
- Extensibility, for example, when the geometric domain increases.
- Suitability for hardware implementation.

The customary approach to efficiency analysis assumes that the representations for the input and output are given, and are fixed. However, a designer of geometric modeling systems has the freedom to choose which representations to use. This causes serious problems, because no theory is available to compare algorithms that are functionally equivalent from the mathematical viewpoint, but operate on different representations—e.g., two algorithms that compute the volume of a solid, one of them for CSG and the other for BReps. The difficulties arise primarily because of input-size considerations. For example, an object may be "small" when represented in CSG, but have a "large" BRep, while just the opposite may be true for a different object—see [Tilove 1981] for other examples and a general discussion.

For our example of convex-polygon projections we could also have represented the polygons in Scheme 4 of Section 3.1, *i.e.*, as intersections of half-spaces. This representation for the input would lead to different algorithms for solving the same mathematical problem.

In summary, here we applied the methodology described in Section 1.3 to a simple problem. We began with an application (graphics in 1-D), formulated it mathematically, selected suitable representations for the input and output, and designed algorithms. We showed that there are many computational approaches that are functionally equivalent, in the sense that they all solve the same mathematical problem. We will continue to apply this methodology in the remainder of this course. In the next few sections we ignore specific applications, and focus on low-level algorithms that serve as computational utilities for the application algorithms discussed in Chapter 7.

6.2 Point-Solid Classification

6.2.1 Point Membership Classification

The *characteristic function* of a set S is usually defined as follows:

$$\chi(\mathbf{p}) = \begin{cases} 1 & \text{if } \mathbf{p} \in S \\ 0 & \text{otherwise} \end{cases}$$

This function is also called the *indicator* of the set, because it tells us if a point **p** belongs or not to the set. It turns out that in geometric modeling we need a slightly different function, which provides us with more information. It is called *point membership classification* (abbreviated PMC) and defined thus:

$$M(\mathbf{p}) = \begin{cases} in & \text{if } \mathbf{p} \in iS \\ on & \text{if } \mathbf{p} \in \partial S \\ out & \text{if } \mathbf{p} \in icS \end{cases}$$

The strings *in, on* or *out* returned by M tell us whether the point **p** is inside, on the boundary, or outside of S.

Point membership classification algorithms are closely related to unambiguous representations for sets. To see why, suppose that we have a representation r for a set S, and an algorithm that evaluates $M(\mathbf{p})$ for the representation r. Then, for any point **p** in Euclidean space we can find whether the point belongs to S or not. In other words, we know which points constitute the set. This implies that r must be an unambiguous representation for S.

In the next two subsections we discuss PMC algorithms for the most common representations for solids, CSG and BReps.

6.2.2 Point Classification for CSG Solids

A CSG representation is a tree. This immediately suggests a *divide and conquer*, or *recursive descent* algorithm for computing PMC. A first cut at the design of such an algorithm is as follows. `PtClassResult` is an enumeration data type with string values *in, on* or *out.* `ClassPtSolC` is the procedure that classifies a point with respect to a solid represented in CSG (hence the C at the end).

Algorithm 6.2.2.1

```
PtClassResult ClassPtSolC(Pt p; Sol S) {
 if Prim(S) then return ClassPtPrimSolC(p,S) // S is a leaf
 else return CombClassPtSolC(ClassPtSolC(p,S.Left),
                             ClassPtSolC(p,S.Right),  ); }
```

In words: first we check if the node of the CSG tree that corresponds to `S` is a leaf (and hence a primitive solid) by evaluating the predicate `Prim(S)`. If so, we call a primitive-specific procedure `ClassPtPrimSolC`. Since in practice a system has only a few primitives, and these are relatively simple, it is not hard to write the primitive classifiers. If `S` is not a primitive, it must be a Boolean operator. Then we call `ClassPtSolC` recursively on the left and right arguments, `S.Left` and `S.Right`, of the operator node, and combine the results by means of the procedure `CombClassPtSolC`. This combining procedure depends on the Boolean operator, denoted by .

The combining procedure is the crucial component of Algorithm 1. Let us construct a table to guide the design of `CombClassPtSolC`. Suppose that $S = A \quad B$, and the operator is regularized intersection. By examining Figure 6.2.2.1 we conclude that if a point **p** classifies inside *A*, or *inA*, and also *inB*, then it must be inside their intersection, i.e., *inS*. The other possible cases are shown in the following table.

$S = A \cap^{*} B$	*inB*	*onB*	*outB*
inA	*inS*	*onS*	*outS*
onA	*onS*	*?*	*outS*
outA	*outS*	*outS*	*outS*

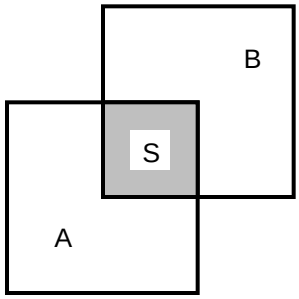


Figure 6.2.2.1 – Intersection of two sets

There is an ambiguity when the point classifies both *onA* and *onB*. Figure 6.2.2.2 depicts the regularized intersection of an L-shaped polygon *A* with a rectangle *B*. In the figure, points **p** and **q** both are *onA* and *onB*, but **p** is *onS*, whereas **q** is *outS*.

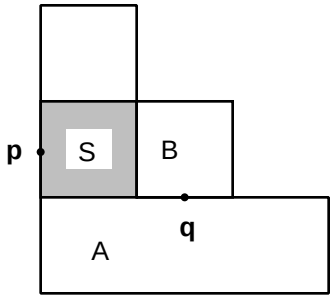


Figure 6.2.2.2 – On/on ambiguity

Analogous tables for the difference and union operators also exhibit on/on ambiguities:

$S = A -^* B$	*inB*	*onB*	*outB*
inA	*outS*	*onS*	*inS*
onA	*outS*	*?*	*onS*
outA	*outS*	*outS*	*outS*

$S = A \cap^* B$	*inB*	*onB*	*outB*
inA	*inS*	*inS*	*inS*
onA	*inS*	*?*	*onS*
outA	*inS*	*onS*	*outS*

This discussion shows that the classification values for the two arguments of a Boolean operator do not always provide sufficient information for determining the classification of a point with respect to the solid which results from the operation. Figure 6.2.2.2 shows that the result depends on whether the two solids locally are on the same or on opposite sides of the overlapping boundary. That is, the resulting classification depends on the *neighborhoods* of the point with respect to the two solid arguments *A* and *B*. The regularized intersection

$$N(\mathbf{p}, A) \cap^* N(\mathbf{p}, B)$$

is a half disk, and therefore **p** is a boundary point of *S*, whereas

$$N(\mathbf{q}, A) \cap^* N(\mathbf{q}, B) =$$

and **q** is outside of *S*—see Figure 6.2.2.3.

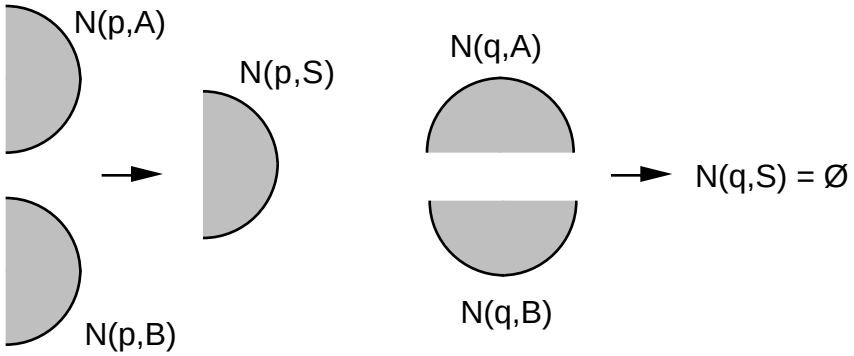


Figure 6.2.2.3 – The regularized intersections of the neighborhoods of points **p** and **q** with respect to *A* and *B* are their neighborhoods with respect to *S*

PMC can be done by divide and conquer, provided that we take into consideration neighborhood information in on/on cases. We define the *augmented* PMC function as

$$M^*(\mathbf{p}) = \begin{cases} in & \text{if } \mathbf{p} \in iS \\ [on, N(\mathbf{p}, S)] & \text{if } \mathbf{p} \in \partial S \\ out & \text{if } \mathbf{p} \in icS \end{cases}$$

To evaluate the augmented classification function it is convenient to first compute the neighborhood, and then infer the result, as shown by the following pseudo-code.

Algorithm 6.2.2.2

```
NbhdSol NbhdPtSolC(Pt p; Sol S) {
 if Prim(Sol)then return NbhdPtPrimSolC(p,S)
 else return CombNbhdPtSolC(NbhdPtSolC(S.Left)
                            NbhdPtSolC(S.Right), ); }

PtClassResult ClassPtSolC(Pt p; Sol S) {
 NbhdSol N = NbhdPtSolC(p,S);
 if Full(N)then return in
 else if Empty(N) then return out
      else return (on, N); }
```

Observe that we did not change the names of the classification procedure and of the data type it returns, but their meanings in Algorithm 6.6.6.2 and Algorithm 6.2.2.1 are slightly different. Now `ClassPtSolC` computes the augmented point classification and `PtClassResult` contains a neighborhood for *on* points.

The classification procedure invokes the neighborhood evaluator, and then simply looks at the results and produces the correct output. The predicates `Full` and `Empty` check if the neighborhood is a complete ball or is null. When neither of these is true, the neighborhood contains a portion of the solid and of its complement, and therefore the point **p** is on the boundary of *S*.

The neighborhood evaluator takes the familiar divide and conquer form. It relies on a neighborhood combiner, `CombNbhdPtSolC`, which computes Boolean operations on neighborhoods.

Algorithm 6.2.2.2 provides a complete high-level procedure for classifying a point with respect to a solid represented in CSG. It uses a few predicates, which are very easy to implement, and `NbhdPtPrimSolC` and `CombNbhdPtSolC`, two procedures which depend strongly on how neighborhoods are represented. Issues of neighborhood representation and manipulation are discussed in a separate section in this chapter.

With minor modifications, Algorithm 6.2.2.2 can also be used in 2-D problems, to classify points with respect to polygons, or, more generally, with respect to surface segments that are defined constructively in terms of 2-D primitive patches. But constructive representations for surface segments are seldom used in geometric modeling.

6.2.3 Point Classification for BRep Solids

Classification algorithms for solids represented by their boundaries are very different from their CSG counterparts. Conceptually, the simplest algorithm for PMC with respect to a BRep solid consists of casting a ray (*i.e.*, a semi-infinite line) from the point, and counting how many times it intersects the solid's boundary. If the number of intersection points is odd, the point is *in*; if it is even, the point is *out.* Figure 6.2.3.1 shows a 2-D example. Note that the choice of ray used to test a point is arbitrary, and different rays may have

different numbers of intersections with the boundary, but for each point all of these numbers have the same parity, *i.e.*, they are all odd or all even. For example, if we had cast a ray from point **p** to the right in Figure 6.2.3.1, the number of intersections would be 1 instead of 5. Computationally, instead of a semi-infinite line we use a line segment that is sufficiently long to finish outside of a box that encloses the solid completely.

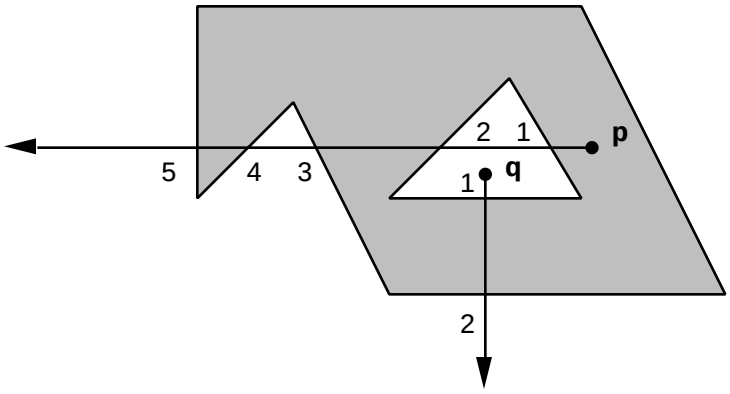


Figure 6.2.3.1 – The ray cast from *in* point **p** has 5 intersections with the polygon's boundary, whereas the ray from *out* point **q** has 2.

Despite its apparent simplicity, there are subtleties associated with this algorithm. First, we also need to consider points that are *on* the boundary. How do we count intersections when the endpoint of a ray is *on*? Second, and more perniciously, numerical errors associated with the representations or with the intersection calculations may produce wrong counts. Third, a ray may intersect an edge or a vertex, or partially lie in a face or an edge. How do we count intersections in such singular cases? Figure 6.2.3.2 illustrates some of the difficulties. Numerical errors in the endpoint coordinates for the edges of the square are grossly exagerated in the figure, to show what can happen in actual computations. The point **p** is *in* the polygon, yet a ray emanating from **p** may not intersect any edge, or intersect two edges, or a vertex. In the first two cases the computed number of intersections is even, and the point will be erroneously classified *out.* When the ray goes through a vertex, it intersects two edges; should we count one or two intersections?

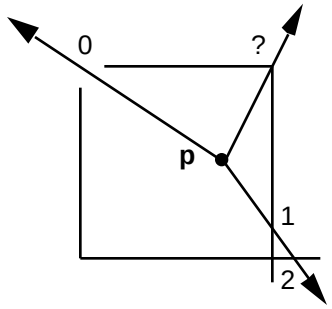


Figure 6.2.3.2 – Classifying a point with respect to a square whose edge representations have numerical errors.

It is possible to resolve all of these difficulties, but the algorithm becomes considerably more complicated. Instead, we can avoid many of them. Since the ray we cast is arbitrary, we choose a ray that does not pass near singularities, *i.e.*, a ray that does not intersect any vertices or edges and does not lie (totally or partially) in any faces or edges. In case of

doubt, *e.g.*, when an intersection point is very close to a vertex, it is safer to assume that the intersection is singular. The endpoint of the ray, however, is the given point **p** to be classified, and cannot be moved. Therefore we need to check if **p** is itself a point on the boundary. In practice, it is difficult to select a ray that is *a priori* known to be free of singularities. It is easier to select a random ray and check if it is singularity-free; if not,we select another ray, and repeat the process. For complex objects, several iterations may be needed.

We discussed in some detail the intersection-counting PMC algorithm not because it is a very good algorithm, but because it illustrates many of the difficulties that arise in geometric computation. How to deal with singularities and the effects of numerical errors are issues that do not affect the asymptotic complexity of the algorihms, and therefore are usually ignored in the theoretical literature. However, if such issues are not addressed carefully, the resulting systems suffer from severe robustness problems and are unusable for real-world applications.

We now turn to another PMC algorithm that is also based on ray casting but does not count intersections. First we check if the given point **p** is a vertex or lies in an edge or a face. If not, we cast a random ray from **p**, intersect it with all the faces of the solid, and select the intersection point closest to **p**. If we cannot decide with certainty that this first intersection point is not singular, we select another ray and repeat the procedure. Finally, we examine the first intersection and infer the point classification from the type of transition at the intersection. The algorithm in pseudo-code is as follows. We assume that the ray is parameterized, starting with $u = 0$ for the given point **p**.

Algorithm 6.2.3.1

```
PtSolClassResult ClassPtSolB(Pt p; Sol S) {

 for each face F of solid S do {

  if PtSurf(p,Surf(F)) then
  // Surf(F) is the host surface of F. For p to be on S
  // it must belong to some F and therefore to some Surf(F)
  // PtSurf(p,G) is a predicate that returns true if p  G

   if ClassPtFace(p,F) == (in or on) then return on;
   // Exit if p is in the 2-D interior or boundary of a face
   // Otherwise we know p is not on S; it is in or out
  };

 do forever {

  R = CastRay(p); // Pick R randomly
  IntList = Ø;
  // Initialize a list of intersection points
  // Each point is represented by its u parameter value
  // plus a Boolean flag, Singular, which is true when the
  // point is in the 1-D interior of an edge or is a vertex

  for each face F of solid S do  // Intersection loop
```

```
   if not RaySurf(R,Surf(F)) then {
   // Predicate RaySurf(R,G) is true if R   G
   PList = IntRaySurf (R,Surf(F));
   // Compute the intersections
   // If Surf(F) is curved there may be several intersections
   // Put them in a list PList

   for each point q in PList do {
    Cvalue = ClassPtFace(q,F);
    if Cvalue == on then q.Singular = true
                  // q on edge or vertex
                   else q.Singular = false;
    if Cvalue == (in or on) then Append(q,IntList);
   }
   } // End of intersection loop

  if IntList == Ø then return out // No intersections
  else {
   r = FirstPoint(IntList); // Minimum u value
   if not r.Singular then {
    Tvalue = TransitionSol(r);
    // This function determines if the ray at r is going
    // from inside the solid to outside or vice-versa
    if Tvalue == (in,out) then return in else return out;
    }
    // Otherwise the first point is singular and we go
    // back and cast another random ray
  } // End of forever loop
 }
```

The first loop of this algorithm determines if **p** is on the boundary of *S*. To do this we check if **p** belongs to any of the faces *F* of *S* in two steps. First we check if **p** lies in the host surface of *F*. If it does, we classify **p** with respect to *F* by means of a 2-D analog of Algorithm 6.2.3.1. (`ClassPtFace` is described later in this section.)

If **p** is not *on S*, we enter the `do forever` loop. We need to find all the intersections of the ray *R* with the boundary of the solid. Therefore we search for intersections between the ray and the faces *F* of *S*. First we ensure that the number of intersections is finite, *i.e.*, *R* does not lie in the host surface of a face *F*. Then we invoke `IntRaySurf` to obtain the intersection points between the ray and the host surface. (Line/surface intersection routines are described in a later section in this chapter.) A ray may intersect a host surface without intersecting its associated face(s), as shown by the 2-D example of Figure 6.2.2.3. Therefore we need to classify the intersection points in 2-D with respect to the faces themselves. The output of `ClassPtFace(q,F)` tells us if there is an intersection, if it is a non-singular intersection, *i.e.*, if the intersection point is in the 2-D interior of the face, or if the intersection is singular, *i.e.*, lies on an edge or is a vertex. We remember if the intersection is singular or not by associating a `Singular` flag with each intersection point. After all intersections are calculated and stored in `IntList`, we extract the intersection point closest to **p**. If this first point is singular we simply cast another ray and repeat the procedure. It does not matter if intersection points other than the first are singular. Therefore only one iteration is usually needed, even for complex objects.

Finally, we invoke the `TransitionSol` function and infer from its value the classification of **p**. `TransitionSol` is described in detail later in this chapter, in the section on neighborhood representation and manipulation. Here we simply note that the value returned by this function specifies the type of transition encountered as an imaginary observer travels along the ray in the positive direction (*i.e.*, the direction of increasing parameter *u*). For example, the transition type for the first point **r** in Figure 6.2.3.3 is *out* to *in*, or *(out,in.)*. This value implies that **p** is *out* of *S*. `TransitionSol` is especially easy to implement robustly when intersections are non-singular. Thus it is preferable to cast additional rays until a first non-singular intersection is found than to attempt to compute transition types at singular intersections.

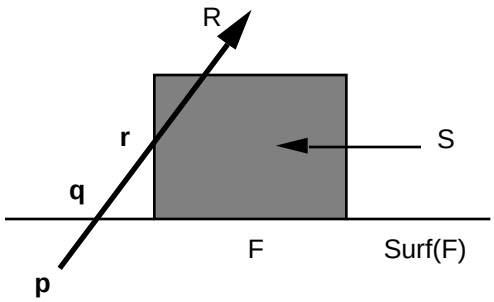


Figure 6.2.3.3 – A ray *R* intersects the host surface of a face *F* of a 2-D solid *S* at a point **q** but does not intersect the face *F* itself

There are still several issues to be addressed in Algorithm 1.

1. How do we find if a point belongs to a surface? This is also a classification problem and we could define a function `ClassPtSurf` to solve it. Instead we define a predicate `PtSurf`, which is trivially computed when the implicit equation of the surface is known. We simply plug the coordinates of the point in the equation of the surface and check if it is satisfied. However, if only a parametric equation of the surface is known, the problem is much more complicated. We can solve it by computing the distance between the point and the surface, as explained later in this chapter. If the distance is zero, the point is on the surface.

2. How do we know if a ray lies in a surface, *i.e.*, how do we compute the predicate `RaySurf`? This must be done by routines specific to each primitive surface used in the modeling system. For planar surfaces, it suffices to check if two points of the ray belong to the surface. As another example, consider a cylinder. Here a ray lies in the cylindrical surface if two of its points belong to the surface *and* the ray is aligned with the cylinder's axis.

3. What happens when a ray lies in a face's surface? This issue is very easy to dispatch: nothing happens. We can ignore in the intersection loop any *F* for which $R \quad Surf(F)$.To see why, consider the 2-D example of Figure 6.2.3.4. The ray lies in the host surface of the two top faces of the notched object. Transitions between *in*, *on*, or *out* states along the ray can occur only at the points **a**, **b** and **c**. But these points are computed in the loop when we intersect the ray with the host surfaces of the vertical faces *A*, *B* and *C*. Typical intersection routines fail (or crash) when the number of intersections is not finite, and therefore we need to know if the ray lies in the surface before attempting to compute intersection points.

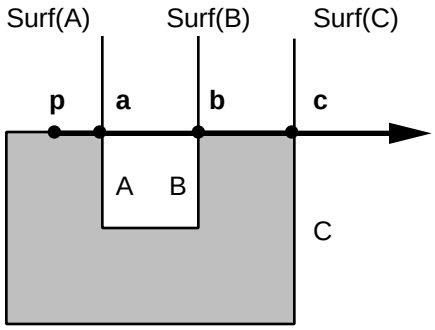


Figure 6.2.3.4 – Ray lying on a host surface

4. How do we classify points with respect to faces? This is done by the 2-D analog of the 3-D classification algorithm we have been discussing. We work on the 2-D host surface of the face, and *cast a ray lying in this surface*. For curved surfaces a (curved) ray can be computed, for example, by intersecting the surface with a plane containing the point to be classified. We present the point/face algorithm below in pseudo-code.

Algorithm 6.2.3.2

```
PtFaceClassResult ClassPtFace(Pt p; Face F) {
 // p is assumed to lie in Surf(F)

 for each edge E of face F do {

  if PtCurve(p,Curve(E)) then
  // Predicate PtCurve(p,C) returns true if p    C
  // Curve(E) is the host curve of E. For p to be on F
  // it must belong to some E and therefore to some Curve(E)

   if ClassPtEdge(p,E) == (in or on) then return on;
   // Exit if p is in the 1-D interior or boundary of E.
   // (The 1-D boundary of E are its vertices.)
   // Otherwise we know p is not on F; it is in or out
  }

 do forever {

  R = CastRay(p);
  // Pick R randomly on the Surf(F)
  IntList = Ø;
  // Initialize a list of intersection points
  // Each point is represented by its u parameter value
  // plus a Boolean flag, Singular, which is true when the
  // point is a vertex

  for each edge E of face F do // Intersection loop
```

```
    if not RayCurve(R,Curve(E)) then {
   // Predicate RayCurve(p,C) is true if R    C
   PList = IntRayCurve(R,Curve(E));
   // If Curve(E) is not a straight line there may be
   // several intersections. Put them in a list PList

   for each point q in PList do {
     Cvalue = ClassPtEdge(q,E);
     if Cvalue == on then q.Singular = true
                    // q is a vertex
                     else q.Singular = false;
     if Cvalue == (in or on) then Append(q,IntList);
    }
   } // End of intersection loop

  if IntList == Ø then return out // No intersections
  else {
   r = FirstPoint(IntList); // Minimum u value
   if not r.Singular then {
    Tvalue = TransitionFace(r);
    // This function determines if the ray at r is going
    // from inside the face to outside or vice-versa
    if Tvalue == (in,out) then return in else return out;
    }
    // Otherwise the first point is singular and we go
    // back and cast another random ray
  } // End of forever loop
 }
```

We need to be able to determine if a point lies in a host curve. Host curves are intersections of primitive surfaces. Therefore we test the point for membership in each of the surfaces to see if it belongs to both, and hence to the curve. (Point/surface algorithms were described above.) Deciding if a ray is a subset of a host curve is treated similarly: we check the ray for membership in each of the surfaces by the techniques described earlier. The `TransitionFace` function is very similar to `TransitionSol`; both are discussed later in this chapter. Finally, `ClassPtEdge` is trivial for points and edges represented parametrically. It amounts to comparing the parameter value of the point with the values that correspond to the edge's enpoints. (Note that we only invoke the point/edge classifier for points that lie in the edge's host curve.)

Let us collect together here the main low-level routines needed by `ClassPtSolB`.

- `A PtSurf` predicate, which often is implemented as in-line code since it is so simple.
- `ClassPtFace` for points lying in Surf(F).
- A `RaySurf` membership predicate to determine if a ray lies in a surface.
- `IntRaySurf` to compute the parameter values of the intersection points. This can be done be invoking `IntCurveSurf` (discussed later in this chapter) with the host curve of the ray as one of the arguments, and retaining those points that belong to the ray, *i.e.*, for which $u \geq 0$.
- `TransitionSol` for determining the type of transition at an intersection point.

In addition, `ClassPtFace` needs the following routines, which are 2-D analogs of those just listed above.

- `A PtCurve` predicate, often implemented as in-line code since it is so simple.
- `ClassPtEdge` for points lying on Curve(*E*). This is a very simple routine that compares parameter values along the host curve.
- A `RayCurve` membership predicate to determine if the ray lies in the curve.
- `IntRayCurve` to compute the parameter values of the intersection points. This can be done be invoking `IntCurveCurve` (discussed later in this chapter) with the host curve of the ray as one of the arguments, and retaining those points that belong to the ray, *i.e.*, for which $u \geq 0$.
- `TransitionFace` for determining the type of transition at an intersection point.

6.3 Curve-Solid Classification

6.3.1 General Membership Classification

A point **p** is either inside, outside, or on the boundary of a *reference set S*. But a *candidate set X* that contains a continuum of points generally is not entirely contained in either *S* or its complement. Therefore, we need a more general notion of membership classification. First let *L* be a straight-line segment and *S* a solid. We define the membership classification of *L* with respect to *S* as the function

$$M(L,S) = (LinS, LonS, LoutS)$$
$$LinS = r_1(L \cap iS)$$
$$LonS = r_1(L \cap \partial S)$$
$$LoutS = r_1(L \cap cS)$$

Therefore *M* divides the line segment into three subsets *LinS*, *LonS*, and *LoutS*, which are, respectively, inside, on the boundary, and outside of *S*. In the formulas above, r_1 denotes regularization in 1-D. It implies that the results should be composed of closed line segments with no dangling, or isolated, points.

More generally, we consider a candidate set X that is regular in the topology associated with a space W', and a reference set S that is regular in the topology of a space $W \quad W'$. Then we define membership classification as

$$M(X,S) = (XinS, XonS, XoutS)$$
$$XinS = r'(X \cap iS)$$
$$XonS = r'(X \cap \partial S)$$
$$XoutS = r'(X \cap cS)$$

where the regularizations are in the topology of W'. We focus on line classification in the next subsections, but we will later also use classification of 2-D surface segments with respect to solids.

6.3.2 Line Classification for CSG Solids

Not surprisingly, M must be augmented with neighborhood information to be computed by divide and conquer, because of on/on ambiguities that arise for points lying on overlapping solid boundaries. The augmented classification function contains also the neighborhood $N(LonS,S)$ of a generic point of $LonS$ with respect to S. The augmented segmentation of L is denoted $LwrtS$ (pronounced "L with respect to S"), and equals

$$LwrtS = [LinS,(LonS,N(LonS,S)),LoutS] \ .$$

$LwrtS$ may be computed by the following divide and conquer procedure for a solid S represented by CSG.

Algorithm 6.3.2.1

```
LineSolClassResultC ClassLineSolC(LineSeg L; Sol S) {
 if Prim(S) then return ClassLinePrimSolC(p,S);
 else return CombLineSolClass(ClassLineSolC(L,S.Left),
                              ClassLineSolC(L,S.Right),  );}
```

This algorithm is very similar to Algorithm 6.2.2.1 for point classification.

To classify a line with respect to a primitive solid, consider the primitive as a CSG combination of half spaces, classify the line with respect to each half space, and then combine the results. For example, a solid block is the intersection of 6 planar half spaces. Therefore, we classify the line with respect to the block by using Algorithm 6.3.2.1, with procedure `ClassLinePrimSolC` replaced by `ClassLineHalfSpace`. Classification with respect to a half space amounts mainly to intersecting the line with the bounding surface of the half space, which is discussed later, in the section on curve and surface intersections.

How do we combine line classifications? Figure 6.3.2.1 illustrates the combine procedure with a simple example. The solid S is the regularized union of two subtree solids A and B. (Each of these two may itself be composed of several primitives, but this is irrelevant for the combining procedure.) In this example there are no *on* components, and the resulting classification can be written as follows.

$$XinS = XinA \ {}^{*}_{1} \ XinB$$

$$XonS =$$

$$XoutS = X \ -^{*}_{1} \ XinB$$

Here the operations have a subscript to denote that they are regularized in the 1-D topology of the line. We can see that the combine procedure consists essentially of *Boolean operations in 1-D*. In a more complicated example, we may encounter *on/on* ambiguities when object boundaries overlap, and these require also neighborhood manipulations.

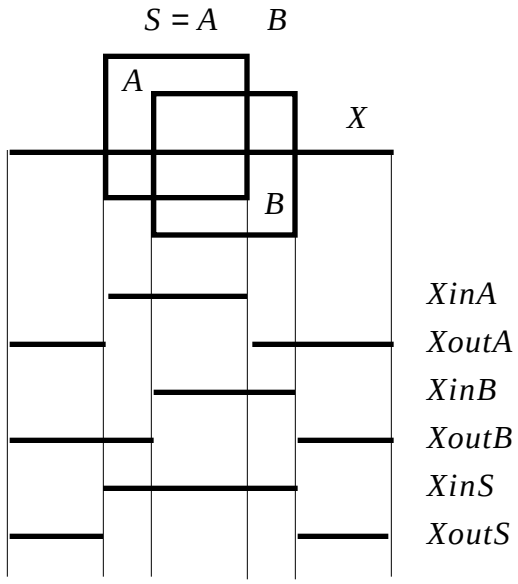


Figure 6.3.2.1 – Combining line classifications.

The implementation of `CombLineSolClass` depends on how we choose to represent classification results. A convenient representation for `LineSolClassResult` is depicted in Figure 6.3.2.2. It consists of a sorted list of pairs (*u*, *Nu*), where *u* is a parameter value that corresponds to a transition point (*i.e.*, to a classification change), and *Nu* is the neighborhood with respect to a solid of a point $\mathbf{p}(u + \varepsilon)$, where $\varepsilon > 0$ is an infinitesimal distance. Intuitively, this is a point immediately to the right of the transition point $\mathbf{p}(u)$. The list is sorted in ascending order of parameter values. In the figure, *E* stands for an *Empty* and *F* for a *Full* (*i.e.*, an entire ball) neighborhood. The transition points segment the line into subsets such that all the points in a subset have the same neighborhood. For example, all the points in the segment a_1a_2 have a full neighborhood (associated with the left endpoint of the segment). The classification values *in*, *on* or *out* for each segment need not be stored because they can be inferred trivially from the associated neighborhoods. The last neighborhood value is meaningless and can be ignored.

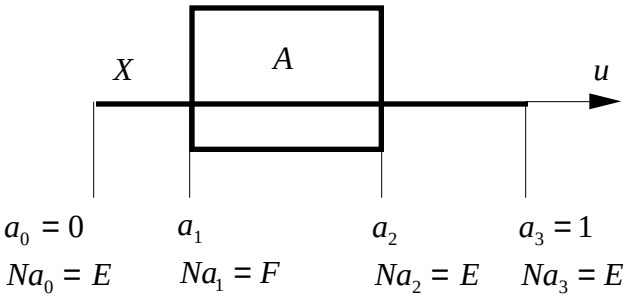


Figure 6.3.2.2 – Representation for classification results.

Observe that the partial results of line-solid classification at each stage of the recursion in Algorithm 6.3.2.1 are segmentations of the same line segment that is being classified. Therefore, the representation for `LineSolClassResult` illustrated in Figure 6.3.2.2 is appropriate for all these intermediate results, and, together with a representation for the original `LineSeg` being classified, defines the results unambiguously.

The combine procedure can be divided conceptually into the three following phases:

- Merge the two input sorted lists of parameter values. The classification of a line segment with respect to the resulting solid $S = A \quad B$ can only change at the transition points of the input classifications with respect to the solids A and B.

- Compute the neighborhoods with respect to S by combining the input neighborhood representations, according to the operator in $S = A \quad B$.

- Clean up the results, by merging segments with the same classification

The procedure is illustrated in Figure 6.3.2.3 for the union of two simple objects. The classification of line segment X with respect to A is the sorted list of pairs (a, Na) shown in the figure. The other input is the classification of X with respect to B, also represented by a sorted list of pairs. The output is the classification with respect to S shown at the bottom of the figure. The two sorted lists of parameter values a and b are merged to produce a list of s parameter values. A neighborhood Ns is computed as the regularized union of the corresponding Na and Nb. For example, $Ns_1 = Na_1 \quad Nb_0 = F \quad E = F$. Note that there are three consecutive segments with full neighborhoods. These are merged in the clean-up phase of the algorithm.

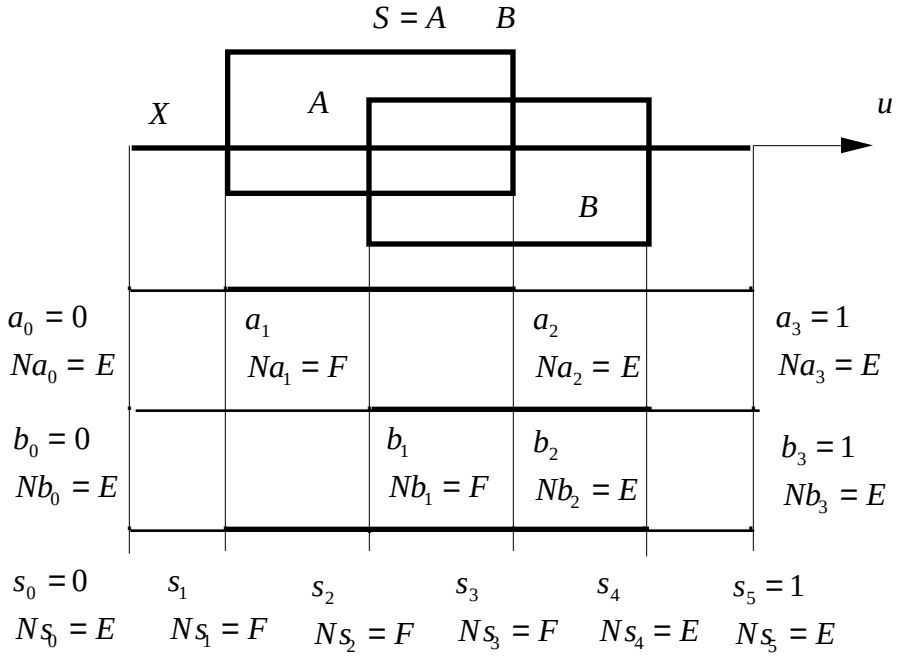


Figure 6.3.2.3 – Combining two classification result representations.

In practice, the three phases of the combine procedure can be merged into a single scan of the two sorted lists. When a new point is placed in the output list, we compute its associated neighborhood, and check whether the point corresponds to a transition, or is redundant and need not be stored. In the example of Figure 6.3.2.3, points s_2, s_3 are redundant.

The combine procedure just described works with parameter values, not with the coordinates of the transition points. This implies that *the procedure can be applied to any parameterized curve* that is homeomorphic to a line segment. It can also be applied to a circle parameterized by the central angle θ if we modify the calculations slightly to take into account that $0 = 2\pi = \ldots = 2n\pi$. This modified procedure is then applicable to any parameterized closed curve homeomorphic to a circle. Most geometric modeling systems parameterize all their curves, and segment them into subsets that are connected, compact 1-manifolds (*i.e.*, they are homeomorphic to either circles or line segments). Therefore our combine procedure has wide applicability.

6.3.3 Line Classification for BRep Solids

Algorithms for line/solid classification for BRep objects are very similar to the point classification algorithms discussed earlier in Section 6.2.3. This is not surprising, since point classification was done essentially by casting a ray and classifying it. However, in the PMC case we were free to choose a convenient ray that avoided singularities, whereas now we must classify the given input line segment. In addition, it is not sufficient to examine only the first point of intersection because we need to segment the input into its *in, on* and *out* subsets.

The BRep is assumed to contain neighborhood representations for all of its faces and edges. We represent classification results by a simplified form of the representation in the previous section. We store in ascending order the parameter value u for each point where the classification changes along the line, plus the classification value at a point $u+\varepsilon$. Recall that in Section 6.3.2 we also stored a neighborhood representation at $u+\varepsilon$.

We need a slightly modified point/face classifier `ClassPtFaceN` that returns not only an *in/on/out* value, but also the neighborhood of the point of intersection with respect to the solid (hence the suffix `N`). The neighborhoods for intersection points that are in the 2-D interior of a face or in the 1-D interior of an edge can be copied directly from the solid's BRep. If the intersection occurs at a vertex, the neighborhood is assigned the special value `Unknown`. The algorithm in pseudo-code follows.

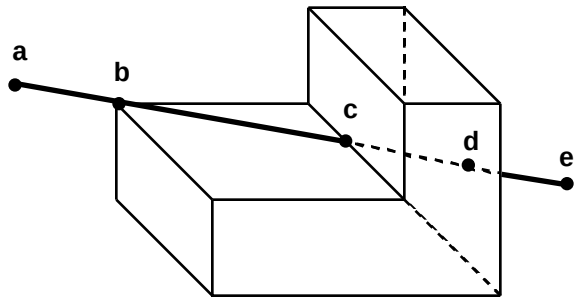
Algorithm 6.3.3.1

```
LineSolClassResultB ClassLineSolB(LineSeg L; Sol S) {

 IntListNC = Ø;
 // Initialize a list of intersection points
 // Each point is represented by its u parameter value and
 // the Nbhd and ClassVal associated with it

 for each face F of solid S do {
```

```
 if not LineSurf(L,Surf(F)) then {
 // Predicate LineSurf(L,G) is true if L    G
 PList = IntLineSurf (L,Surf(F));
 // Compute the intersections
 // If Surf(F) is curved there may be several intersections
 // Put them in a list PList

 for each point p in PList do {
  C = ClassPtFaceN(p,F);
  // Returns a classification value plus a neighborhood
  // If p is a vertex the Nbhd of C = Unknown
   if C.Value == (in or on) then Append(p,IntListNC);
  }
 } // End of face loop

if IntListNC == Ø then { // No intersections
 // L is entirely in, on or out
 q = SelectPt(L);
 // Pick an arbitrary point of L and classify it
 return ClassPtSolB(q,S);
 }

else { // Intersection list processing
 Append(EndPoint1,IntListNC); Append(EndPoint2,IntlistNC);
 // Add the endpoints and set their neighborhoods to Unknown
 IntListNC = Sort(IntListNC);
 // Sort by u value and merge coincident points

 until end of IntListNC do {
  if p.Nbhd == Unknown then p.ClassVal = Unknown
  else {
   Tvalue = TransitionSol(p);
   // Tvalue is a pair of elements such as (in,out)
   Previous(p).ClassVal = FirstElem(Tvalue);
   p.ClassVal = SecondElem(Tvalue);
   } // End else
  Next(p);
  } // End of list traversal

 // At this point we may miss classification values for some
 // segments because of intersections at vertices. Classify
 // the mid-points of such segments

 until last but one element of IntListNC do {
  if p.ClassVal == Unknown then {
   q = MidPoint(p,Next(p));
   p.ClassVal = ClassPtSolB(q,S);
   } // End then
  Next(p);
  } // End do
 } // End of intersection list processing
```

```
return IntListNC without the Nbhd field
}
```

Intersection list processing is illustrated in Figure 6.3.3.1. Points **b**, **c** and **d** are computed by intersecting the line segment *L* with the faces of the solid. (Actually **b** will be found twice, as the intersection of *L* with the left and back faces of the solid; duplicates are merged.) The endpoints of the line segment are assigned an `Unknown` neighborhood. Since **b** is a vertex, its neighborhood is also set to `Unknown`. After `TransitionSol` returns, we know that the transitions at **c** and **d** are (*on,in*) and (*in,out*), respectively, but `TransitionSol` is unable to determine the transition types of points **a**, **b** and **e**, which have `Unknown` neigborhoods. This situation is shown schematically in the first labelled horizontal line of the figure. The transition information at **c** implies that segment **bc** is *on* and segment **cd** is *in*. Therefore we can attach a `ClassVal` of *on* to **b** and a `ClassVal` of *in* to **c**. (Recall that in our representation the classification value of a segment is attached to the initial point of the segment.) The transition type of **d** implies a `ClassVal` of *out* for **d** and *in* for **c**. Note that we have found **c**'s value twice. This redundancy occurs most of the time in this algorithm, and can be avoided through more sophisticated programming, but contributes little to the overall cost. The value of **b** was found by propagation of the transition type of **c**, although the neighborhood of **b** was `Unknown` because it is a vertex. The second horizontal line in the figure shows the situation at the end of the first list traversal. The `ClassVal` for **a** is still `Unknown`, and so is **e**'s, but this latter is irrelevant and can be ignored. We classify the midpoint **q** of the segment **ab** and find that **q** is *out*, and hence so is the whole segment **ab**, as shown in the third horizontal line.



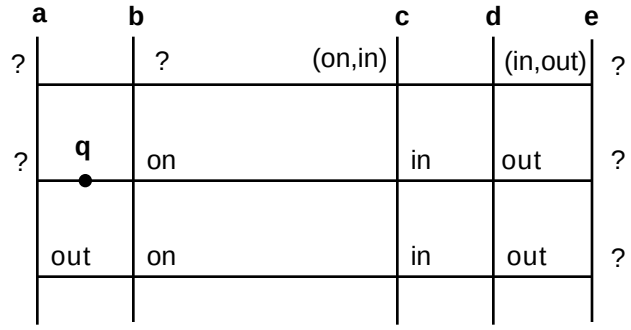


Figure 6.3.3.1 – Intersection list processing in `ClassLineSolB`

We avoided the difficult problems of representing vertex neighborhoods and reasoning about the transitions at vertices by using the strategy of testing an interior point of a segment by point classification. We could similarly avoid dealing with neighborhoods of points lying on edges (e.g., **c** in Figure 6.3.3.1), but these are considerably easier than vertex neihborhoods, and it is generally more effective to handle them directly. Note that midpoint classification amounts essentially to another line classification (without singularities), which costs almost as much as the original one, and therefore should be used sparingly.

6. 4 Neighborhoods

This section discusses how to represent and combine neighborhoods, and how to infer transition types from neighborhood information.

6.4.1 Representation and Combination

The neigborhood of a point with respect to a geometric entity (*e.g.*, a solid) can be represented *explicitly,* as we will see below, or *implicitly*, by pointers to entities that are adjacent to the point. For example, if a point is in the interior of an edge that is shared by two faces of a solid, pointers to these two faces suffice to define completely the local geometry of the solid in the neighborhood of the point. Implicit representations are used in many geometric modeling systems, but often lead to complicated algorithms for combining neighborhood information. In this text we focus on explicit neighborhood representations, which are relatively simple to combine.

The representations discussed below assume that geometric entities have associated coordinate systems that can be used as references, for example, for measuring angles. These coordinate systems can be defined as follows. We assume that all curves and surfaces are oriented at construction time. Thus, a surface constructor associates a *reference normal* ν with the surface being instantiated. It does not matter which of the two possible orientations for a surface normal is chosen, but the system must be able to know which is which. Reference normal assignment may be done explicitly, by attaching to the surface representation a normal vector to the surface at a given point, *i.e.*, an applied vector. This vector must then be carried along with the surface representation and updated if the surface is moved in space. Alternatively, the normal may be defined implicitly. For example, a system may define a reference normal to spherical and cylindrical surfaces as the inward pointing normal, in the direction of decreasing radial coordinate.

Similarly, a curve constructor associates a *reference tangent* vector τ with the curve instance. Again, this vector can be represented expicitly and associated with the curve representation, or implicitly. Often, the reference tangent for a parametric curve is represented implictly by agreeing that it points in the direction of increasing parameter values. For example, a line segment is usually represented by two endpoints **p** and **q** such that **p** corresponds to $u = 0$ and **q** to $u = 1$. The reference tangent then points from **p** to **q**. When a curve lies on a surface we use the curve's tangent τ and the surface's normal ν to define a binormal $\beta = \tau \times \nu$.

The most important types of neighborhoods needed in geometric modeling algorithms are the following.

3-D face neighborhood : neighborhood of a point that lies in the interior of a face with respect to a solid. The solid S is known, and so is the face F on which **p** lies. In addition, the face's reference normal ν is also known. Since the point is not on an edge and is not a vertex, there are only two possible situations: either the solid's material locally is on the side towards which ν points, or is on the opposite side. One bit of data, called a *side bit*, suffices to distinguish between these two situations. For example, we can represent the first case by a 1, and the second by a 0. A complete 3-D face neighborhood representation for a point **p** on a face F with respect to a solid S is then a pair

```
Nbhd(p,S) = (SideBit(p,S), RefNormal(p,S)).
```

Alternatively we can combine the two entities into just one normal vector that points towards the material side:

```
InwardNormal(p,S) = if SideBit(p,S) then RefNormal(p,S)
                    else –RefNormal(p,S)
```

Figure 6.4.1.1 shows a point on a cube's face, and the associated reference normal ν'. The `SideBit` in the representation for the neighborhood of **p** with respect to S is a 0, signifying that ν' points away from the material. The corresponding `InwardNormal` representation is the vector $-\nu$. Note that both of these representation schemes also apply to curved surfaces.

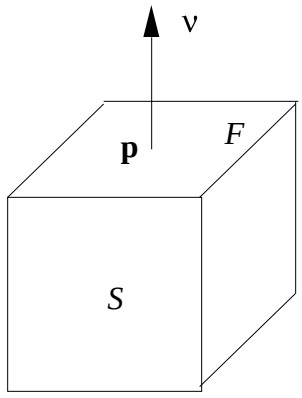


Figure 6.4.1.1 – The side bit for **p** is 0.

3-D face neighborhood representations are easy to combine by comparing normal directions. We define a predicate

```
SameSide = (InwardNormal(p,A) == InwardNormal(p,B))
```

which is `true` when the two material sides coincide and `false` when they are opposite. Neighborhoods represented by their `InwardNormal` values can be combined as follows. We are given the neighborhoods of a point with respect to solids A and B and we want to combine these so as to obtain the neighborhood of the same point with respect to solid $C = A \quad B$, where denotes a regularized Boolean operation.

Algorithm 6.4.1.1

```
NbhPtSol CombNbhdPtSol (NbhPtSol nA, nB) {

 // nA is the neighborhood of a point p with respect to solid
 // A and nB the neighborhood of the same point with respect
 // to B; p is in the interior of faces fA and fB of A and B.
 // Neighborhoods are represented by their inward normals
 // or the special values Full and Empty.Input neighborhoods
 // are assumed neither Full nor Empty

 if SameSide then

  case   of {

   Union:           nC = nA;
   Difference:      nC = Empty;
   Intersection:    nC = nA;
   }

  else

   case   of {

    Union:          nC = Full;
    Difference:     nC = nA;
    Intersection:   nC = Empty;
    }
}
```

Figure 6.4.1.2 illustrates neighborhood combination in 2-D when the operator is union. The same algorithm applies to objects with curved surfaces, provided that the normals are computed at the same point **p**.

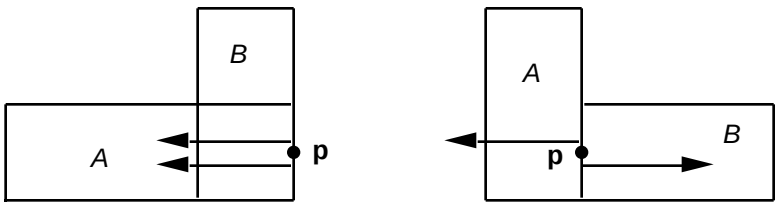


Figure 6.4.1.2 – 3-D face neighborhood combination for the union operator.
On the left the rectangles *A* and *B* overlap, and on the right they just touch.

3-D edge neighborhood: neighborhood of a point that lies in the interior of an edge with respect to a solid. The situation now is more complicated. We work in the plane normal to the reference tangent vector τ at **p**. This normal plane intersects the point neighborhood in a sector subtended by the intersections of the normal plane and the faces adjacent to the given edge—see Figure 6.4.1.3. All that we need is to represent this sector, as a pair of angles. To obtain an origin for measuring the angles we construct a local coordinate system with basis vectors (ξ,η,τ) as follows:

$$\xi = \boldsymbol{y} \times \tau$$
$$\eta = \tau \times \xi$$

Here τ is the tangent vector to the edge and $\boldsymbol{y}$ is the basis vector that corresponds to the y axis of the lab coordinate frame. If these two vectors are parallel we choose

$$\xi = \boldsymbol{z}$$
$$\eta = \boldsymbol{x}$$
$$\tau = \boldsymbol{y}$$

We use the ξ axis as origin, and measure the angles in the $\xi\eta$ plane with the positive sense being given by the motion of a right-handed screw as it advances in the positive τ direction. For example, the sector shown on the right in Figure 6.4.1.3 is represented by the angular interval in degrees (–90,0).

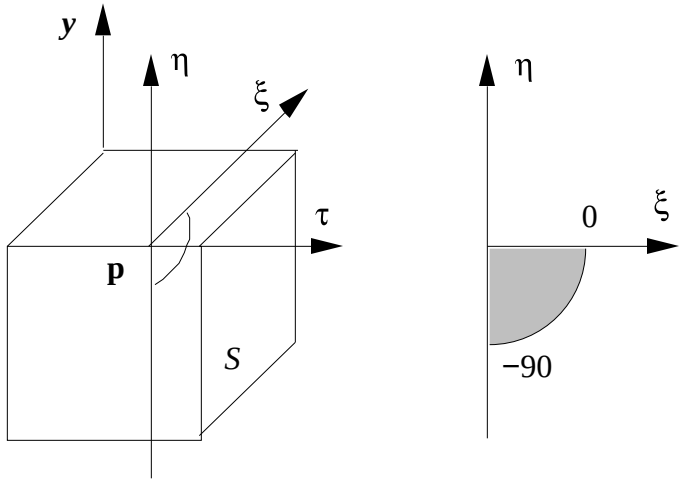


Figure 6.4.1.3 – Representation for a 3-D edge neighborhood.

Sector representations are easy to combine through regularized Boolean operations, as shown in Figure 6.4.1.4. If we replace the sectors by the corresponding arcs of, say, the unit circle, the combine procedure required is precisely the same as for combining circular arc classifications, discussed in Section 6.3.2.

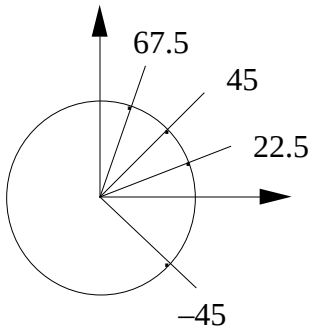


Figure 6.4.1.4 – The regularized intersection of sectors (–45,45) and (22.5, 67.5) is the sector (22.5,45), which can be computed by intersecting the circular arcs that correspond to the argument sectors.

Edge neighborhoods for non-manifold solids can be represented by lists of angular sectors, instead of single sectors. The circle classification procedures used for combining neighborhoods can handle such lists.

For objects with curved surfaces, a sector representation for edge neighborhoods can still be constructed, by using the tangents to the curves of intersection between the curved surfaces and the $\xi\eta$ plane normal to the edge. But this representation is incomplete. It must be supplemented, for example, with pointers to the actual curved surfaces. The tangent approximation can be used to combine neighborhoods in most cases. We use the standard circle classification combine procedure and produce a new sector list. We also carry along with each angle the corresponding surface pointer and output a list of angular intervals, each with two associated surface pointers. This procedure fails when two distinct surfaces have the same tangent approximation in the normal plane to the curve. Figure 6.4.1.5 provides an example. The two circles in the figure have the same tangent. When this occurs, we resort to higher-order approximations for the curves of intersection of the surfaces with the $\xi\eta$ plane. (This approach was used successfully in the PADL-2 system.) In essence we must sort the surfaces (or the curves of intersection of the surfaces with the normal plane) around the edge, so as to properly identify and process the (curved) sectors that lie between pairs of surfaces. We omit the details, because they are not very instructive and the first-order approximations fail very seldom.

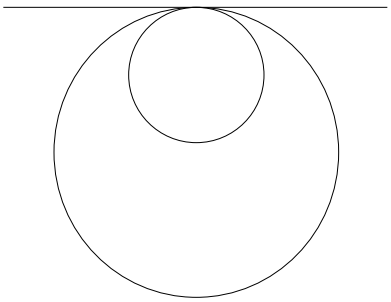


Figure 6.4.1.5 – The linear approximation does not suffice for combining neighborhoods of points on the intersection of two tangent cylinders.

2-D edge neighborhood: neighborhood of a point that lies in the interior of an edge of a solid with respect to a face of the solid. We use the binormal β to define a direction normal to the edge and tangent to the face's surface. Then, a 1-bit representation suffices to distinguish between the two possible cases, much like in 3-D face neighborhoods—see Figure 6.4.1.6. In some geometric modeling systems 2-D neighborhood information is encoded in the orientation of the edges.

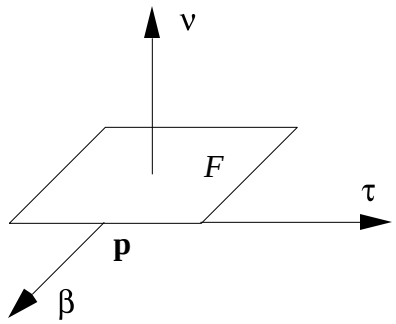


Figure 6.4.1.6 – The representation for the 2-D edge neighborhood of **p** is 0, since *F* is on the side of the binormal β's tail.

3-D vertex neighborhood: neighborhood of a vertex with respect to a solid. 3-D vertex neighborhoods are considerably more difficult to represent and combine than their edge or face counterparts. They are not needed in the algorithms discussed in this text. However, some geometric modeling systems are based primarily on vertex-neighborhood manipulations [Mantyla TOG], [ref on Noodles].

2-D vertex neighborhood: neigborhood of a vertex with respect to a face of a solid. These are very much like 3-D edge neighborhoods, and also are not needed in our algorithms.

6.4.2 Transition Evaluation

The algorithms we presented earlier for set membership classification with respect to BRep solids relied on our ability to infer the type of transition encountered when a line (or, more generally, a curve) intersects the boundary of a solid. We assume throughout this section that an observer travels along the curve in the direction of its reference tangent τ, which usually is the direction of increasing parameter values.

Transition evaluation is very easy when a curve/solid intersection is non-singular, *i.e.*, when the point of intersection lies in the interior of a face of the solid, and the intersection is "clean" (also called *transversal*). A transversal intersection occurs when a curve is not tangent to the solid's face. Figure 6.4.2.1 illustrates the procedure in 2-D. It is clear from the figure that

$$\tau \cdot \nu > 0 \qquad out \rightarrow in$$
$$\tau \cdot \nu < 0 \qquad in \rightarrow out$$

Here we denote by ν the inward normal to a face. If the inner product above is zero, a curve is tangent to a face. This cannot happen for line/plane intersections in our algorithms. If it did, the line would lie entirely in the plane and there would be a continuum of intersection points. Recall that in our algorithms we do not compute intersections of lines with faces in which they lie. However, for general curves and surfaces tangential intersections may occur. They are treated as singular intersections, by techniques similar to those described below.

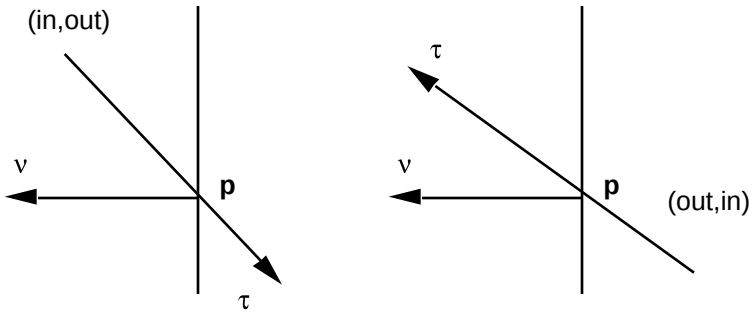


Figure 6.4.2.1 – Transition evaluation for non-singular line/plane intersections

When the intersections are singular, we can take two approaches. Either we *avoid* the singularities, or we *resolve* them. Avoiding singularities is easy but not very efficient. Figure 6.4.2.2 presents a 2-D example. The line/rectangle intersection occurs at a vertex. Instead of reasoning about the vertex neighborhood so as to infer the transition type, we simply select and classify a point in the interior of a line segment between intersection points. The point classification is also the classification of the entire segment. Point classification is done by casting a ray, but the ray is arbitrary and can be selected to avoid singularities. The drawback of this method is that it requires additional ray classifications and therefore can be expensive.

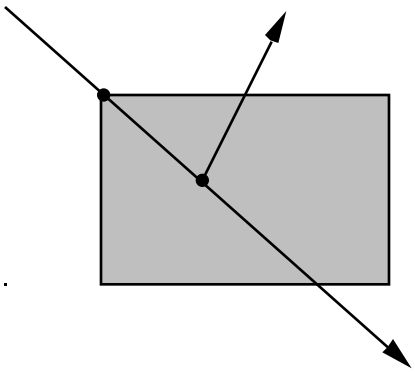


Figure 6.4.2.2 Avoiding singularities by classifying a segment's midpoint.

Resolving vertex singularities is complicated, and will not be addressed in this text. Avoiding such singularities as explained above often leads to more robust and efficient algorithms. Also, vertex singularities occur rarely. Edge singularities are much more common, and can be resolved as follows.

We work in the plane normal to the edge at the point of intersection **p**, as shown in Figure 6.4.2.3. We assume that the line segment *L* to be classified does not lie in the host line of the edge *E*. (If it did, the computed intersection points in our algorithm would be vertices of the solid, and we would have vertex singularities, instead of edge singularities.) We project the line segment on the normal plane of the edge. The projection is another line segment. We represent the 3-D edge neighborhood by an arc (or several arcs, for non-manifold neighborhoods) on the unit circle in the normal plane, and intersect the projected line segment (extending it, if necessary) with the unit circle. Finally, we classify the two points of intersection **q** and **r** with respect to the arc(s) that represent the neighborhhod. The transition type follows directly from these classification values, read in the order of

increasing parameter (or reference tangent) for the line. We consider the unit circle parameterized by the angle about the center, and classify the points **q** and **r** simply by comparing their polar coordinates with the angles that correspond to the neighborhood.

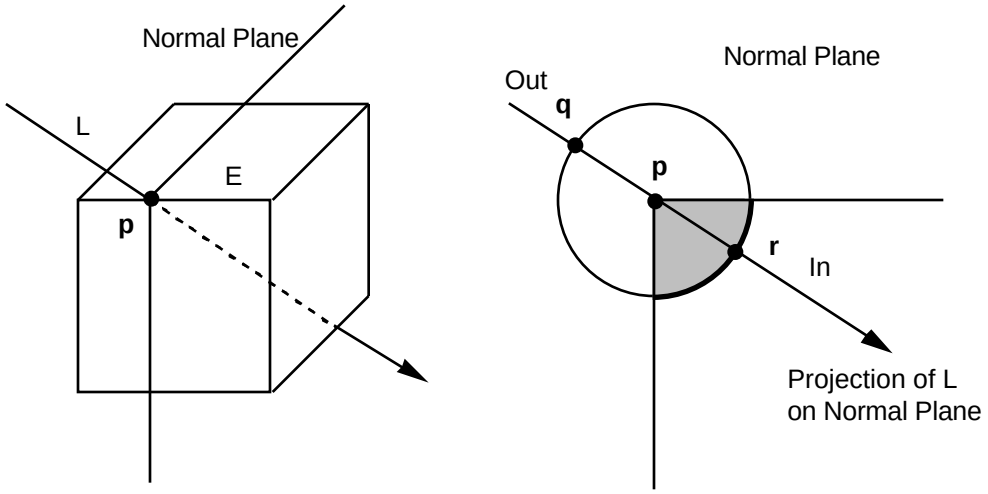


Figure 6.4.2.3 – The line segment *L* intersects the solid at the point **p** in the interior of edge *E*. The neighborhood of **p** with respect to the solid cube is the thick arc in the plane normal to *E* at **p**, shown on the right. The projection of *L* on the normal plane intersects the unit circle at **q** and **r**. Point **q** is *out* of the neighborhood arc, and **r** is *in*. Therefore the transition type is (*out,in*.).