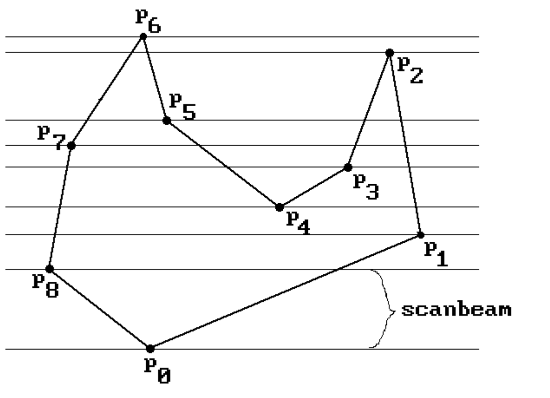
Vatti Polygon Clipping

Quite a few polygon clipping algorithms have been published. We have discussed several. The Liang-Barsky and Maillot algorithms are better than the Sutherland-Hodgman algorithm, but these algorithm only clip polygons against simple rectangles. This is adequate for many situations in graphics. On the other hand, the Sutherland-Hodgman and Cyrus-Beck algorithms are more general and allow clipping against any convex polygon. The restriction to convex polygons is caused by the fact that the algorithm clips against a sequence of half-planes and therefore only applies to sets that are the intersection of half-planes, in other words, convex (linear) polygons. There are situations however where the convexity requirement is too restrictive. The Weiler algorithm is more general yet and works for non-convex polygons. The final two algorithms we look at, the Vatti and Greiner-Hormann algorithms, are also extremely general. Furthermore, they are the most efficient of these general algorithms. The polygons are not constrained in any way now. They can be concave or convex. They can have self-intersections. In fact, one can easily deal with lists of polygons. We begin with Vatti's algorithm ([Vatt92]).

Call an edge of a polygon a left or right edge if the **interior** of the polygon is to the right or left, respectively. Horizontal edges are considered to be both left and right edges. A key fact that is used by the Vatti algorithm is that polygons can be represented via a set of left and right bounds which are connected lists of left and right edges, respectively, that come in pairs. Each of these bounds starts at a local minimum of the polygon and ends at a local maximum. Consider the "polygon" with vertices $\mathbf{p}_0$, $\mathbf{p}_1$, …, $\mathbf{p}_8$ shown in Figure 1(a). The two left bounds have vertices $\mathbf{p}_0$, $\mathbf{p}_8$, $\mathbf{p}_7$, $\mathbf{p}_6$ and $\mathbf{p}_4$, $\mathbf{p}_3$, $\mathbf{p}_2$, respectively. The two right bounds have vertices $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_4$, $\mathbf{p}_5$, $\mathbf{p}_6$.

Note: In this section the y-axis will be pointing up (rather than down as usual for a viewport).

Here is an overview of the Vatti algorithm. The first step of the algorithm is to determine the left and right bounds of the clip and subject polygons and to store this information in a local minima list (LML). This list consists of a list of matching pairs of left-right bounds and is sorted in ascending order by the y-coordinate of the corresponding local minimum. It does not matter if initial horizontal edges are put into a left or right bound. Figure 1(b) shows the LML for the polygon in Figure 1(a). The algorithm for constructing the LML is a relatively straightforward programming exercise and will not be described here. It can be done with a single pass of the clip and subject polygons.



(a)

LML = ((B_1, B_2) , (B_3, B_4))

where the bounds B_i are defined by

B_1 = ($\mathbf{p}_0\mathbf{p}_8$, $\mathbf{p}_8\mathbf{p}_7$, $\mathbf{p}_7\mathbf{p}_6$) ,
B_2 = ($\mathbf{p}_0\mathbf{p}_1$, $\mathbf{p}_1\mathbf{p}_2$) ,
B_3 = ($\mathbf{p}_4\mathbf{p}_5$, $\mathbf{p}_5\mathbf{p}_6$) , and
B_4 = ($\mathbf{p}_4\mathbf{p}_3$, $\mathbf{p}_3\mathbf{p}_2$) .

(b)

Figure 1 Polygon bounds

The bounds on the LML were specified to have the property that their edges are either **all** left edges or **all** right edges. However, it is convenient to have a more general notion of a left or right bound. Therefore, from now on, a left or right bound will denote any connected sequence of edges only whose **first** edge is required to be a left or right edge, respectively. We still assume that a bound starts at a local minimum and ends at a local maximum. For example, we shall allow the polygon in Figure 1(a) to be described by one left bound with vertices $\mathbf{p}_0$, $\mathbf{p}_8$, $\mathbf{p}_7$, $\mathbf{p}_6$, $\mathbf{p}_5$, $\mathbf{p}_4$, $\mathbf{p}_3$, $\mathbf{p}_2$ and one right bound with vertices $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$.

The clipped or output polygons we are after will be built in stages from sequences of "partial" polygons, each of which is a "V-shaped" list of vertices with the vertices on the left side coming from a left bound and those on the right side coming from a right bound and where there is one vertex in common, namely, the one at the bottom of the "V" which is at a local minimum. Let us use the notation $P[\mathbf{p}_0\mathbf{p}_1\ldots\mathbf{p}_n]$ to denote the partial polygon with vertices $\mathbf{p}_0$, $\mathbf{p}_1$, …, $\mathbf{p}_n$, where $\mathbf{p}_0$ is the first point and $\mathbf{p}_n$, the last. The points $\mathbf{p}_0$ and $\mathbf{p}_n$ are the top of the partial left and right bound, respectively. Some vertex $\mathbf{p}_m$ will be the vertex at a local minimum which connects the two bounds but since it will not be used for anything there is no need to indicate this index m in the notation. For example, one way to represent the polygon in Figure 1(a) would be as $P[\mathbf{p}_6\mathbf{p}_7\mathbf{p}_8\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3\mathbf{p}_4\mathbf{p}_5\mathbf{p}_6]$ (with m being 3 in this case). Notice how the edges in the left and right bounds are **not** always to the right or left of the interior of the polygon here. In the case of a "completed" polygon, $\mathbf{p}_0$ and $\mathbf{p}_n$ will be the same vertex at a local maximum, but at all the other intermediate stages in the construction of a polygon the vertices $\mathbf{p}_0$ and $\mathbf{p}_n$ may **not** be equal. However, $\mathbf{p}_0$ and $\mathbf{p}_n$ will always correspond to top vertices of the current left and right partial bounds, respectively. For example, $P[\mathbf{p}_7\mathbf{p}_8\mathbf{p}_0\mathbf{p}_1]$ (with m equal to 2) is a legitimate expression describing partial left and right bounds for the polygon in Figure 1(a). A good way to implement these partial polygons is via a circularly linked list, or cycle, and a pointer that points to the last element of the list.

The algorithm now computes the bounds of the output polygons from the LML by scanning the world from the bottom to the top using what are called scanbeams. A scanbeam is a horizontal section between two scan lines (not necessarily adjacent), so that each of these scan lines contains at least one vertex from the polygons but there are **no** vertices in between them. Figure 1(a) shows the scanbeams and the scan lines that determine them for that particular polygon. The scanbeams are the regions between the horizontal lines. It should be noted here that the scan lines that determine the scanbeams are not computed all at once but incrementally in a bottom-up fashion. The information about the scanbeams is kept in a scanbeam list (SBL) which is an ordered list of the y-coordinates of all the scan lines that define the scanbeams. This list of increasing values will be thought of as a stack. As we scan the world, we also maintain an active edge list (AEL) which is an ordered list consisting of all the edges intersected by the current scanbeam.

When we begin processing a scanbeam, the first thing we do is to check the LML to see if any of its bound pairs start at the bottom of the scanbeam. These bounds correspond to local minima and may start a new output polygon or break one into two depending on whether the local minimum starts with a left-right or right-left edge pair. After any new edges from the LML are added to the AEL we need to check for intersections of edges within a scanbeam. These intersections affect the output polygons and are dealt with separately first. Finally, we process the edges on the AEL. Algorithm 1 summarizes this overview of the Vatti algorithm.

Now let us look at the algorithm in more detail. Data 1 shows a data structure for an edge. Each edge has the x-value of its intersection with the bottom of the scanbeam associated to it and the edges of the AEL are ordered by increasing values of these x-values with ties being broken using the **dx** value of the edge. The x-values are updated as we move from scanbeam to scanbeam. The kind of an edge specifies whether it belongs to the clip or subject polygon. We shall say that two edges are like edges if they have the same kind value and unlike edges otherwise.

The UpdateLMLandSBL procedure shown in Algorithm 2 finds the bounds of a polygon, adds them to LML, and also updates SBL. Finding a bound involves creating the edges which make them up. Each edge has its **bottomX**, **topY**, **dx**, and **kind** fields defined here. The **dx** field for a horizontal edge is simply the signed length of the edge with the value of **dx** negative if the edge is oriented to the left. The

```
{ Global variables }
real list          SBL;  { an ordered list of distinct reals thought of as a stack}
bound pair list    LML;  { a list of pairs of matching polygon bounds }
edge list          AEL;  { a list of nonhorizontal edges ordered by x-intercept
                           with the current scan line}
polygon list       PL;   { the finished output polygons are stored here as algorithm
                           progresses }

polygon list function Vatti_Clip (polygon subjectP; polygon clipP)
{ The polygon subjectP is clipped against the polygon clipP.
  The list of polygons which are the intersection of subjectP and clipP is returned to the
  calling procedure. }
begin
    real yb, yt;

    Initialize LML, SBL to empty;

    { Define LML and the initial SBL }
    UpdateLMLandSBL (subjectP,subject);
    UpdateLMLandSBL (clipP,clip);

    Initialize PL, AEL to empty;

    yb := PopSBL ();                      { bottom of current scanbeam }
    repeat
        AddNewBoundPairs (yb);            { modifies AEL and SBL }
        yt := PopSBL ();                  { top of current scan beam }
        ProcessIntersections (yb,yt);
        ProcessEdgesInAEL (yb,yt);
        yb := yt;
    until Empty (SBL);

    return (PL);
end;
```

Algorithm 1 The Vatti polygon clipping algorithm

side, **contributing**, and **adjPolyPtr** fields are determined in the InsertIntoAEL procedure described later. The pointer **adjPolyPtr** in an edge record points to the polygon associated to the edge. This polygon will also be referred to as the adjacent polygon of the edge.

Because horizontal edges complicate matters, in order to make dealing with horizontal edges easier, we assume that the matching left and right bound pairs in the LML list are "normalized." A normalized left and right bound pair satisfies the following properties:

(1) All consecutive horizontal edges are combined into one so that bounds do not have two horizontal edges in a row.
(2) No left bound has a bottom horizontal edge (any such edges are shifted to the right bound).
edge = **record**

```
    real               bottomX,      { initially the x-coodinate of the bottom vertex, but once the
                                       edge is on the AEL, then it is the x-intercept of the edge with
                                       the line at the bottom of the current scanbeam }
                       topY,         { y-coordinate of top vertex }
                       dx;           { the reciprocal of the slope of the edge }
    (clip,subject)     kind;         { does edge belong to clip or subject polygon? }
    (left,right)       side;         { is it a left or right bound edge? }
    boolean            contributing; { does edge contribute to output polygons? }
    polygon pointer    adjPolyPtr;   { pointer to partial polygon associated to the edge }
end;
```

Data 1 The edge data structure

```
procedure UpdateLMLandSBL (closed polygonal curve P; (clip,subject) type)
{ Finds the bounds of P and adds them to LML. Only the edge fields bottomX, topY,
  dx, and kind are defined here. The side and adjPolyPtr fields are defined later. Add
  all local minima and the tops of the first nonhorizontal edge of every bound to SBL. }
begin
    for each  bound pair BP = (B1,B2) of P  do
        begin
            Add BP to LML;

            { Add the minimum y-value of BP to SBL }
            InsertIntoSBL (StartY (BP));

            { Add the top of the first nonhorizontal edges to SBL }
            Let ei denote the first nonhorizontal edge of Bi;
            InsertIntoSBL (TopY (e1));
            InsertIntoSBL (TopY (e2));
        end
end;

real function StartY (bound pair BP)
{ Returns the minimum y-value of the vertices in BP }

procedure InsertIntoSBL (real y)
{ If y is not yet in SBL, then insert it at the correct place in the ordered list SBL }

real function PopSBL ()
{ Delete the smallest (first) element of SBL and return its value }

edge function Succ (edge e)
{ The edge after e in the polygon bound to which e belongs }
```

Algorithm 2 Updating the LML and SBL

(3) No right bound has a top horizontal edge (any such edges are shifted to the left bound).

Let us introduce some more terminology. Some edges and vertices that one encounters or creates for the output polygons will belong to the bounds of the clipped polygon, others will not. Let us call a vertex or an edge a contributing or noncontributing vertex or edge depending on whether or not it belongs to the output polygons. With regard to vertices, if a vertex is not a local minimum or maximum, then it will be called a left or right intermediate vertex depending on whether it belongs to a left or right bound, respectively. Because the overall algorithm proceeds by taking the appropriate action based on the vertices that are encountered, we shall see that it therefore basically reduces to a careful analysis of the three cases:

(1) The vertex is a local minimum.
(2) The vertex is a left or right intermediate vertex.
(3) The vertex is a local maximum.

Local minima are encountered when elements on the LML become active. Intermediate vertices and local maxima are encountered when scanning the AEL. Intersections of edges also give rise to these three cases.

Returning to Algorithm 1, the first thing that happens in the main loop is to check for new bound pairs that start at the bottom of the current scanbeams. If any such pairs exist, then we have a case of two bounds starting at a vertex which is a local minimum. We add their first nonhorizontal edges to the AEL and the top y-values of these to the SBL. The edges are flagged as being a **left** or **right** edge using the side field. We determine if the edges are **contributing** by a parity test. An edge of the subject polygon is contributing if there are an odd number of edges from the clip polygon to its left in the AEL. Similarly, an edge of the clip polygon is contributing if there are an odd number of edges from the subject polygon to its left in the AEL. If the vertex is contributing, then we create a new partial polygon P[**p**] and make the **adjPolyPtr** field in both edge records point to this polygon. If the vertex is noncontributing, then we set their **adjPolyPtr** fields to **nil**. Note that to determine whether or not an edge is contributing or noncontributing we actually have to look at the geometry only for the first nonhorizontal edge of each bound. After that, we simply check the **contributing** field. Algorithm 3 describes the details.

The central task of the main loop in the Vatti algorithm is to process the edges on the AEL. If edges intersect, we shall have to do some preprocessing (procedure ProcessIntersections), but right now let us skip that and describe the actual processing, namely, procedure ProcessEdgesInAEL. Because horizontal edges cause substantial complications, we describe two versions of the procedure. The first, shown in Algorithm 4, assumes that the polygons have **no** horizontal edges. The general case, where we allow horizontal edges is shown in Algorithm 6. We shall discuss the nonhorizontal version first.

If an edge does not end at the top of the current scanbeam, then we simply update its **bottomX** field to the x-coordinate of the intersection of the edge with the scan line at the top of the scanbeam. If an edge **does** end at the top of the scanbeam, then the action we take is determined by the type of the top end vertex **p**. The vertex can either be an intermediate vertex or a local maximum. Issues associated to horizontal edges are treated later.

If the vertex **p** is a left or right intermediate vertex, then the vertex is added at the beginning or end of the vertex list of its adjacent polygon, depending on whether it is a left or right edge, respectively. The edge is replaced by its successor edge and the polygon pointer and left/right flag are passed on to the new edge.

If the vertex **p** is a local maximum of the original clip or subject polygons. In that case, a pair of edges from two bounds meet in a point **p**. If **p** is a contributing vertex, then the two edges may belong either to the same or different (partial) polygons. If they belong to the same polygon, then this polygon will now be closed once the point **p** is added. If they belong to different polygons, say **P** and **Q**, respectively, then we need to merge these polygons. Let $\mathbf{e}_1$ and $\mathbf{e}_2$ be the edges for **P** and $\mathbf{f}_1$ and $\mathbf{f}_2$, the

```
procedure AddNewBoundPairs (real y)
while  not (Empty (LML)) and (first bound pair BP on LML starts at y)  do
    begin
        AddEdgesToAEL (BP,y);
        Delete BP from LML;
    end;

procedure AddEdgesToAEL (bound pair (B1,B2), real yb)
{ Bound pair (B1,B2) starts at yb }
begin
    { Since bounds are normalized, only B2 can have a horizontal first edge }
    Let (e1,e2) denote the first nonhorizontal edges of B1 and B2, respectively;
    InsertIntoSBL (TopY (e1));
    InsertIntoSBL (TopY (e2));

    { Now add edges to AEL and finish their definition }
    InsertIntoAEL (e1,e2);

    Let p be the vertex at which both e1 and e2 start;
    if  Contributing (p)  then  AddLocalMin (e1,e2,p);
end;     { AddEdgesToAEL }

procedure InsertIntoAEL (ref edge e1, e2)
{ Add nonhorizontal edges e1 and e2 to active edge list maintaining  increasing x order.
  Also set side and contributing fields of e1 and e2 using a parity argument. }

procedure AddLocalMin (edge e1, e2;  point p);
{ e1 and e2 are the first nonhorizontal edges of a local minimum. }
begin
    Create a new adjacent polygon P[p] for e1;
    Make the adjPolyPtr field in edge e1 and e2 point to this polygon;
end;

edge function Next (edge e)
{ The edge after e in the AEL. }

edge function Previous (edge e)
{ The edge which precedes e in the AEL. }
```

Algorithm 3 Adding new bound pairs

```
procedure ProcessEdgesInAEL (real yb, yt)
    { Assumption:  Polygons have no horizontal edges. }
    begin
        real        dy;
        boolean   moreEdges;

        if  Empty (AEL)  then  return;

        dy := yt – yb;
        Let e denote the first edge of AEL;
        moreEdges := true;
        while  moreEdges  do
            begin
                if  e terminates at level yt in a vertex p
                    then
                        case  Type (p)  of
                            left intermediate:  begin
                                    if  Contributing (e)  then  AddLeft (e,p);
                                    Replace e in AEL by Succ (e);
                                end;
                            right intermediate:  begin
                                    if  Contributing (e)  then  AddRight (e,p);
                                    Replace e in AEL by Succ (e);
                                end;
                            local maximum:     begin
                                    Let nexte denote the edge of AEL after e;
                                    if  Contributing (e)  then  AddLocalMax (e,nexte,p);
                                    Delete e and nexte from AEL;
                                end;
                        end
                    else  SetBottomX (e,TopX (e,dy));    { Update e's bottomX value }

                Update e to denote the next edge in AEL or
                    set moreEdges to false if there is none;
            end
    end;  { ProcessEdgesInAEL }

    procedure AddLeft (edge e;  point p);
    If P[$p_0p_1...p_n$] is the polygon associated to e, replace it by P[$pp_0p_1...p_n$];

    procedure AddRight (edge e;  point p);
    If P[$p_0p_1...p_n$] is the polygon associated to e, replace it by P[$p_0p_1...p_np$];

    procedure AddLocalMax (edge e1, e2; point p);
    begin
        if  Side (e1) = left  then  AddLeft(e1,p)
                                else   AddRight (e1,p);
```

```
      if  e1 and e2 have different output polygons
         then  AppendPolygon(e1,e2)
         else  Add polygon to PL;
      end;

   procedure AppendPolygon (edge e1, f1);
   { Let  P1 = P[$\mathbf{p}_0\mathbf{p}_1\ldots\mathbf{p}_n$]  and  P2 = P[$\mathbf{q}_0\mathbf{q}_1\ldots\mathbf{q}_s$]  be the polygons adjacent
     to e1 and f1, respectively.  Let e2 and f2 be the other top edges of P1 and P2,
     respectively. }
   if  e1 is a left edge
      then
         begin
            Add vertex list of P2 to the left of vertex list of P1, that is,
               replace P1 by P[$\mathbf{q}_s\mathbf{q}_{s-1}\ldots\mathbf{q}_0\mathbf{p}_0\mathbf{p}_1\ldots\mathbf{p}_n$]
            Make P1 the adjacent polygon of f2;
         end
      else
         begin
            Add vertex list of P1 to the right of vertex list of P2, that is,
               replace P2 by P[$\mathbf{q}_0\mathbf{q}_1\ldots\mathbf{q}_s\mathbf{p}_n\mathbf{p}_{n-1}\ldots\mathbf{p}_0$]
            Make P2 the adjacent polygon of e2;
         end;

   real function TopX (edge e; real dy)
   { This function returns the x-coordinate of the intersection of the edge with
     a scan line a height dy above the current one which intersects the edge
     in a point with x-coordinate BottomX (e). }
   begin
      return (BottomX (e) + Dx (e)*dy);
   end;
```

Algorithm 4 Algorithm for processing the AEL edges (no horizontal edges)

edges for **Q**, so that $\mathbf{e}_1$ and $\mathbf{f}_1$ meet in **p** with $\mathbf{f}_1$ the successor to $\mathbf{e}_1$ in the AEL. See Figure 2. Figure 2(a) and (c) show specific examples and (b) and (d) generic cases. If $\mathbf{e}_1$ is a left edge of **P** (Figure 2(a) and (b)), then we append the vertices of **Q** to the beginning of the vertex list of **P**. If $\mathbf{e}_1$ is a right edge of **P** (Figure 2(c) and (d)), then we append the vertices of **P** to the end of the vertex list of **Q**. Note that each of the polygons has two top contributing edges. In either case, after combining the vertices of **P** and **Q**, the two edges $\mathbf{e}_1$ and $\mathbf{f}_1$ become noncontributing. If $\mathbf{e}_1$ was a left edge, then $\mathbf{f}_2$ will be contributing to **P**. Therefore, we will need to modify the polygon pointer of $\mathbf{f}_2$ to point to **P**. If $\mathbf{e}_1$ was a right edge, then $\mathbf{e}_2$ will be contributing to **Q**. Therefore, we will need to modify the polygon pointer of $\mathbf{e}_2$ to point to **Q**.

When we find a local maximum we get two edges. If these belong to different polygons, then how do we find the other two top edges for these polygons. There are two ways to handle this. One could maintain pointers in the polygons to their current top edges, or one could do a search of the AEL. The first method gives us our edges without a search, but one will have to maintain the pointers as we move from one edge to the next. Which method is better depends on the number of edges versus the number of local

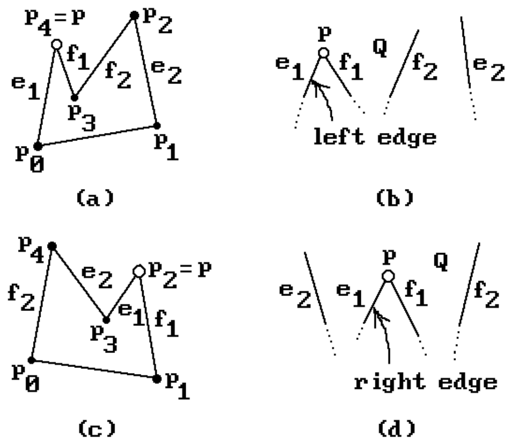


Figure 2 Merging polygons

maxima. Since there probably are relatively few local maxima, the second method is the recommended one.

Finally, we look at how one deals with intersections of edges within a scanbeam. The way that these intersections are handled depends on whether we have like or unlike edges. Like intersections need only be considered if both edges are contributing and in that case the intersection point should be treated as both a left and right intermediate vertex. (Note that in the case of like intersections, if one edge is contributing, then the other one will be also.) Unlike intersections must always be handled. How their intersection point is handled depends on their type, side, and relative position in the AEL.

It is possible to give some precise rules on how to classify intersection points. The rules are shown in Table 1 in an encoded form. Edges have been specified using the following two-letter code: The first letter indicates whether the edge is a left (L) or right (R) edge, and the second letter specifies whether it belongs to the subject (S) or clip (C) polygon. The resulting vertex type is also specified by a two-letter code: local minimum (MN), local maximum (MX), left intermediate (LI), and right intermediate (RI). Edge codes are listed in the order in which their edges appear in the AEL!

For example, Rule 1 translates into the following: The intersection of a left clip edge and a left subject edge, or the intersection of a left subject edge and a left clip edge, produces a left intermediate vertex. Rules 1-4 are shown graphically in Figure 3(a). Figure 3(b) shows an example of how the rules apply to some real polygon intersections.

As one moves from scanbeam to scanbeam, one updates the **bottomX** values of all the edges (unless they end at the top of the scanbeam). Although the AEL is ordered as one enters a new scanbeam, if any intersections are found in a scanbeam, the AEL will no longer be sorted after the **bottomX** values are updated. The list must therefore be resorted, but this can be done in the process of dealing with the intersections. Vatti used a temporary <u>sorted edge list</u> (SEL) and an <u>intersection list</u> (IL) to identify and

Unlike edges:	**Like edges**:
(1) (LC $\cap$ LS) **or** (LS $\cap$ LC) $\rightarrow$ LI	(5) (LC $\cap$ RC) **or** (RC $\cap$ LC) $\rightarrow$ LI **and** RI
(2) (RC $\cap$ RS) **or** (RS $\cap$ RC) $\rightarrow$ RI	(6) (LS $\cap$ RS) **or** (RS $\cap$ LS) $\rightarrow$ LI **and** RI
(3) (LS $\cap$ RC) **or** (LC $\cap$ RS) $\rightarrow$ MX	
(4) (RS $\cap$ LC) **or** (RC $\cap$ LS) $\rightarrow$ MN	

Table 1 Rules which classify the intersection point between edges

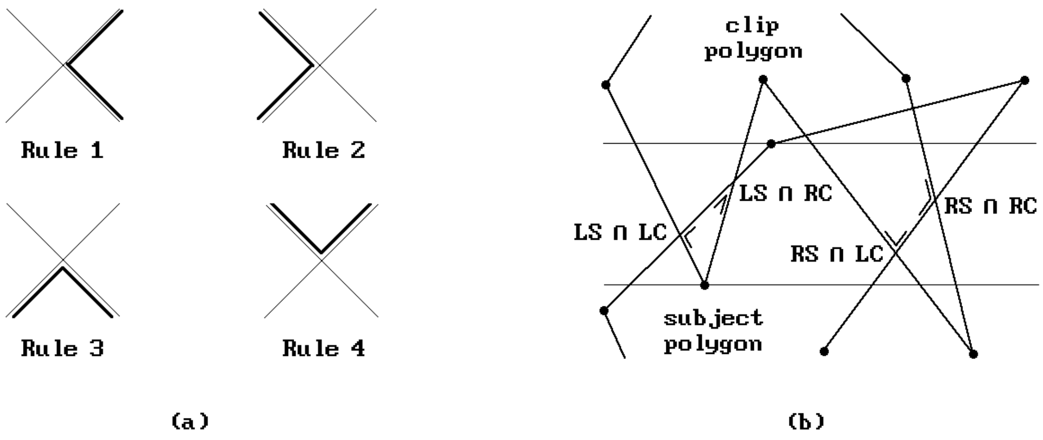


Figure 3 Intersection rules

store all the intersections in the current scanbeam. The SEL is ordered by the x-coordinate of the intersection of the edge with the **top** of the scanbeam similarly to the way that the AEL is ordered by the **bottomX** value of its edges. The IL is a list of nodes specifying the two intersecting edges and also the intersection itself. It is sorted in an increasing order by the y coordinate of the intersection. The SEL is initialized to empty. One then makes a pass over the AEL comparing the top x value of the current edge with the top x values of the edges in the SEL starting at the **right** of the SEL. There will be an intersection each time the AEL edge has a smaller top x value than the SEL edge. Note that the number of intersections that are found is the same as the number of edge exchanges in the AEL it takes to bring the edge into its correct place at the **top** of the scanbeam. Algorithm 5 has a more detailed description of this process.

Intersection points of edges are basically treated as vertices. Such "vertices" will be classified in a similar way as the regular vertices. If we get a local maximum, then there are two cases. If two unlike edges intersect, then a contributing edge becomes a noncontributing edge and vice versa. This is implemented by simply swapping the output polygon pointers. If two like edges intersect, then a left edge becomes a right edge and a right edge becomes a left edge. One needs to swap the intersecting edges in the AEL to maintain the x-sort.

This finishes our discussion of the Vatti algorithm in the case where there are no horizontal edges. Now we address the more complicated general case which allows horizontal edges to exist. (However, we never allow edges to overlap, that is, where they share a common segment.) The only changes we have to make are in procedure ProcessEdgesInAEL. On an abstract level, it is easy to see how horizontal edges should be handled. The classification of vertices described above should proceed as if such edges were absent (had been shrunk to a point). Furthermore, if horizontal edges do not intersect any other edge, then for all practical purposes they could be ignored. The problems arise when intersections exist.

Imagine that the polygons were rotated slightly so that there were no horizontal edges. The edges that used to be horizontal would now be handled without any problem. This suggests how they should be treated when they are horizontal. One should handle horizontal edges the same way that intersections are handled. Note that horizontal edge intersections occur only at the bottom or top of a scanbeam. Horizontal edges at local minima should be handled in the AddNewBoundPairs procedure. The others are handled as special cases in that part of the algorithm which tests whether or not an edge ends in the current scanbeam. At that point we also need to look for horizontal edges at the top of the current scanbeam and the "Type(**p**)" classification should return three other cases for horizontal edges in a local maximum, left intermediate, or right intermediate. The corresponding procedures need to continue scanning the AEL for edges that intersect the horizontal edge until one gets past it. One final problem occurs with horizontal edges that are oriented to the left. These would be detected too late, that is, by the time one finds the edge

```
intNode = record
    edge  e1, e2;
    point intP;
end;

edge list       SEL;
intNode list  IL;

procedure ProcessIntersections (real yb, yt)
begin
    if  Empty (AEL)  then  return;

    BuildIL (yt – yb);
    ProcessIL ();
end;

procedure BuildIL (real dy);
begin
    real       topX1;
    edge       e1;
    boolean  moreEdges;
    point      p;

    Initialize IL to empty;
    SEL := { first edge of AEL };

    for each  edge e1 on AEL after the first  do
        begin
            topX1 := TopX (e1);            { top x value of e1 }

            { Starting with the rightmost node of SEL we shall now move from right
              to left through the nodes of SEL checking for an intersection with e1 }
            Let e2 denote the rightmost edge of SEL;
            moreEdges := true;
            while  moreEdges and (topX1 < TopX (e2)  do
                begin
                    p := IntersectionOf (e1,e2);
                    Insert intNode (e1,e2,p) into IL;
                    if  e2 is the left-most edge in SEL
                        then  moreEdges := false;
                        else   update e2 to denote edge to its left in SEL;
                end;

            { Now insert e1 into SEL at the point where we quit the while loop.
              If moreEdges is false then e2 had reached the left end of SEL. }
            if  moreEdges
                then  Insert e1 to the right of e2 in SEL
                else  Insert e1 at the left end of SEL;
        end
end;  { BuildIL }

procedure ProcessIL ()
begin
    for each  intNode (e1,e2,p) in IL  do
        begin
            { e1 precedes e2 in AEL and p is point of intersection }
            case  IntersectionType (p)  of
                like edge intersection :  if  Contributing (e1)  then
                                              begin
                                                  AddLeft (e1,p);
                                                  AddRight (e2,p);
                                                  Exchange side values of edges;
                                              end;
                local maximum       :  AddLocalMax (e1,e2,p);
                left intersection   :  AddLeft (e2,p);
                right intersection  :  AddRight (e1,p);
                local minimum       :  AddLocalMin (e1,e2,p);
            end;
            Swap e1 and e2 position in AEL;
            Exchange adjPolyPtr pointers in edges;
        end
end;  { ProcessIL }

point function IntersectionOf (edge e1, e2)
{ This function returns the intersection of the two edges e1 and e2. }
```

Algorithm 5 An algorithm for finding intersecting edge pairs

to which they are the successor we would have already scanned past the AEL edges which intersected them. To avoid this, the simplest solution probably to make an initial scan of the AEL for all such edges before one checks events at the top of the scanbeam and put them into a special left-oriented horizontal edge list (LHL) ordered by the x values of their left end points. Then as one scans the AEL one needs to constantly compare the top x value of an edge for whether it lies inside one of these horizontal edges.

To deal with horizontal edges as just described, replace Algorithm 4 by Algorithm 6. Algorithm 6 contains three new procedures, LeftHoriz, RightHoriz, and HorizLocalMax. The first two handle the cases where a horizontal edge follows a left or right intermediate vertex, respectively. The last deals with a horizontal edge at the top of a local maximum. What we have to do is scan down the AEL for any edges that intersect the current horizontal edge and deal with this intersection. The algorithms for these three procedures are very similar and we shall only give more details in the case of the RightHoriz procedure. See Algorithm 7. The function IntersectionType determines the intersection type as specified in Table 1. One should keep in mind one simplification we made at the very beginning, namely, that bounds are normalized so that we never have more than one horizontal edge in a row. The nastiest horizontal edges

```
procedure ProcessEdgesInAEL (real yb, yt)
{ This procedure deals with horizontal edges }
begin
    edge list  LHL;                    { thought of as queue }
    real       dy, topX;
    boolean    moreEdges;

    if  Empty (AEL)  then  return;

    { Look for top left-directed horizontal intermediate edges and put them on LHL list }
    Initialize LHL to empty;
    for each  edge e in AEL  do¦
        if  (e ends at yt) and HasSucc (e) then
            begin
                Let succe denote Succ (e);
                if  Horiz (succe) and (Dx (succe) < 0)  then  Append (LHL,succe);
            end;

    dy := yt – yb;
    Let e1 denote the first edge of AEL;
    moreEdges := true;
    while  moreEdges  do
        begin
            topX := TopX (e1,dy);
            if  not (Empty (LHL)) and (topX > OtherEndX (First element of LHL))
                then
                    begin
                        he := Pop (LHL);     LeftHoriz (yt,he,e1);
                    end
                else  if  e terminates at level yt in vertex p
                    then
                        case  Type (p)  of
                            local maximum        :   { same as in Algorithm 3.12 }
                            left intermediate    :   { same as in Algorithm 3.12 }
                            right intermediate  :   { same as in Algorithm 3.12 }
                            horizontal local maximum : begin
                                    Let e2 denote the edge on AEL after e1;
                                    if  e1 and e2 come from matching bounds
                                        then  { no intersections in horizontal edge }
                                            begin
                                                if  Contributing (e1)  then
                                                    AddLocalMax (e1,e2,p);
                                                Delete e1 and e2 from AEL;
                                            end
                                        else  HorizLocalMax (e1,e2,p);
                                end;
                        horizontal right intermediate :  RightHoriz (yt,e1,p);
                     end
                 else  { Update e1's bottomX value }
                     SetBottomX (e1,topX);

            { Note that if we encountered intersections of horizontal edges, then we may
              have scanned past a number of edges on the AEL after e1 at this point. }
            Update e1 to denote the next "untouched" edge in the AEL or
                 set moreEdges to false if there are none;
        end;
end;

real function OtherEndX (edge e)
{ The function returns the x-coordinate of the other end of the horizontal edge }
begin
    return (BottomX (e) + Dx (e));
end;
```

Algorithm 6 Algorithm for processing the AEL edges (with horizontal edges)

```
procedure RightHoriz (real yt; edge e1; point p)
{ We assume that  Succ (e1)  is a right-oriented horizontal edge which is not a
  local maximum. }
begin
    real        hrx, nextTop;
    boolean   hasNext;

    if  Contributing (e1)  then
        begin
            if  Side (e1) = left  then  AddLeft (e1,yt)
                                  else   AddRight (e1,yt);
        end;

    Let he denote the horizontal edge Succ (e1);
    Copy e1's side, contributing, and adjPolyPtr field values to he;
    hrx := OtherEndX (he);                    { hrx is the right x value of the edge he }

    hasNext := HasNextEdge (e1);
    if  hasNext  then
        begin
            Let e2 denote the edge on AEL after e1;
            nextTop := TopX (e2);         { top x value for e2 }
        { loop through all edges on AEL after e which intersect he }
        while  hasNext and (nextTop ≤ hrx)  do
            begin
                Let p be the point where he and e2 intersect;
                case  IntersectionType (he,e2)  of
                    like edge intersection :  if  Contributing (he)  then
                                                  begin
                                                      AddLeft (e2,p);
                                                      SetSide (he,Side (e2));
                                                  end;
                    local maximum   :
                        begin
                            if  Contributing (he)  then  AddLocalMax (he,e2,p);
                            SetContributing (he,false);
                            SetContributing (e2,false);
                        end;
                    left intermediate   :  if  Contributing (he)  then  AddLeft (e2,yt);
                    right intermediate :  if  Contributing (he)  then
                                              begin
                                                  AddRight (he,p);
                                                  SetAdjPolyPtr (he,AdjPolyPtr (he));
                                              end;
                    local minimum :
                        begin
                            if  Contributing (he)  then
                                begin
                                    AddRight (he,p);
                                    SetAdjPolyPtr (he,AdjPolyPtr (he));
                                end;
                            SetContributing (he,true);
                            SetContributing (e2,true);
                        end
                end;

                Exchange value of contributing fields for he and e1;
                if  TopY (e2) = yt                    { edge ends at top of scanbeam }
                    then  Replace e2 in AEL by Succ (e2)
                    else  SetBottomX (e2,nextTop);

                hasNext := HasNextEdge (e2);
                if  hasNext  then
                    begin
                        nextTop := TopX (e2);  { top x value for next edge }
                        Update e2 to next edge on AEL;
                    end
            end
    end;
```

```
        Delete e1 from AEL;
        Insert Succ (he) into AEL;
        InsertIntoSBL (TopY (Succ (he)));
    end;  { RightHoriz }

    boolean function HasNextEdge (edge e)
    { Return true if there is an edge to the right of e on the AEL and false otherwise. }
```

Algorithm 7 Algorithm for dealing with right-oriented horizontal edges

are the left-oriented ones. These will always follow a left intermediate vertex. We cannot eliminate them, but at least we have reduced their number.

This completes our description of the basic Vatti algorithm. The algorithm can be optimized in the common case of rectangular clip bounds. Another optimization is possible if the clip polygon is fixed (rectangular or not) by computing its bounds only once and initializing the LML to these bounds at the beginning of a call to the clip algorithm.

An attractive feature of Vatti's algorithm is that it can easily be modified to generate trapezoids. This is particularly convenient for scan line oriented rendering algorithms. A trapezoid can be represented by a record as shown in Data 2. Each local minimum starts a trapezoid or breaks an existing one into two depending on whether the local minimum starts with a left-right (contributing case) or right-left (noncontributing case) edge pair. At a contributing local minimum we create a trapezoid setting all of its fields except for **topY**. Trapezoids are output at local maxima and left or right intermediate vertices. A noncontributing local minimum should output the trapezoid it is about to split and the trapezoid pointers of the relevant edges updated to the two new trapezoids. The following simple loop fills a trapezoid:

```
for  y:=bottomY to topY  do
    begin
        DrawLine (leftX,y,rightX,y);
        leftX   := leftX + leftDx;
        rightX := rightX + rightDx;
    end;
```

Vatti compared the performance of the trapezoid version of his algorithm to the Sutherland-Hodgman algorithm and found it to be roughly twice as fast for clipping (the more edges, the more the

```
trapezoid = record
    real  leftX,          { the x-coordinate of the bottom left vertex }
          leftDx,         { the reciprocal of the slope of the left edge }
          rightX,         { the x-coordinate of the bottom right vertex }
          rightDx,        { the reciprocal of the slope of the right edge }
          bottomY,        { the y-coordinate of the bottom edge }
          topY;           { the y-coordinate of the top edge }
end;
```

Data 2 The trapezoid data structure

improvement) and substantially faster if one does both clipping **and** filling.

A special case of the trapezoid form of the Vatti algorithm will be very useful when we deal with trimmed surfaces in Section 14.8. The reader can find more details on how to deal with trapezoids there.

Finally, we can also use the Vatti algorithm for other operations than just intersection. All we have to do is replace the classification rules. For example, if we want to output the union of two polygons, use the rules

$$\begin{array}{llll}
(1) & (\mathrm{LC} \cup \mathrm{LS}) \ \textbf{or} \ (\mathrm{LS} \cup \mathrm{LC}) & \rightarrow & \mathrm{LI} \\
(2) & (\mathrm{RC} \cup \mathrm{RS}) \ \textbf{or} \ (\mathrm{RS} \cup \mathrm{RC}) & \rightarrow & \mathrm{RI} \\
(3) & (\mathrm{LS} \cup \mathrm{RC}) \ \textbf{or} \ (\mathrm{LC} \cup \mathrm{RS}) & \rightarrow & \mathrm{MN} \\
(4) & (\mathrm{RS} \cup \mathrm{LC}) \ \textbf{or} \ (\mathrm{RC} \cup \mathrm{LS}) & \rightarrow & \mathrm{MX}
\end{array}$$

Local minima of the subject polygon which lie outside the clip polygon and local minima of the clip polygon which lie outside the subject polygon should be treated as contributing local minima.

For the difference of two polygons (subject polygon minus clip polygon) use the rules

$$\begin{array}{llll}
(1) & (\mathrm{RC} - \mathrm{LS}) \ \textbf{or} \ (\mathrm{LS} - \mathrm{RC}) & \rightarrow & \mathrm{LI} \\
(2) & (\mathrm{RS} - \mathrm{LC}) \ \textbf{or} \ (\mathrm{LC} - \mathrm{RS}) & \rightarrow & \mathrm{RI} \\
(3) & (\mathrm{RS} - \mathrm{RC}) \ \textbf{or} \ (\mathrm{LC} - \mathrm{LS}) & \rightarrow & \mathrm{MN} \\
(4) & (\mathrm{RC} - \mathrm{RS}) \ \textbf{or} \ (\mathrm{LS} - \mathrm{LC}) & \rightarrow & \mathrm{MX}
\end{array}$$

Local minima of the subject polygon which lie outside the clip polygon should be treated as contributing local minima.