CHAPTER 3

Clipping

Prerequisites: Basic vector algebra

3.1 Introduction

Planar clipping algorithms rank as probably the second most important type of algorithm in computer graphics, following right behind line-drawing algorithms in importance. Mathematically, to *clip* one set against another means to find their intersection. In practice, one usually wants also to get this intersection in terms of some predefined data structure.

This chapter discusses some of the well-known clipping algorithms along with some newer and more efficient ones. The algorithms fall into two types: the *line-clipping* algorithms, which clip single line segments against rectangular or convex regions, and *polygon-clipping* algorithms, which clip whole polygons against other polygons. The following terminology is used:

Definition. The polygon being clipped is called the *subject* polygon and the polygon that one is clipping against is called the *clip* polygon.

The choice of algorithms to discuss was motivated by the following considerations:

(1) It is currently one of the best algorithms of its type.
(2) It is not the best algorithm but still used a lot.
(3) The algorithm was interesting for historical reasons and easy to describe.
(4) It involved the use of some interesting techniques, even though it itself is no longer a recommended method.

Below we list the algorithms described in this chapter and categorize them by considerations (1)–(4) above. We also state any assumption regarding their clip polygon and make some comments about them. Some of the algorithms will be discussed in great detail. Others are only described very briefly, especially if they fall under heading (3) or (4) above.

Line-clipping algorithms:

	Category	Clip Polygon	Comments
Cohen-Sutherland	(2)	rectangular	**The** classic line-clipping algorithm. Still popular because it is so easy to implement.
Cyrus-Beck	(4)	convex	
Liang-Barsky	(2)	rectangular	Faster than Cohen-Sutherland. Still popular. Easy to implement.
Nicholl-Lee-Nicholl	(1)	rectangular	Purely two-dimensional.

Polygon-clipping algorithms:

	Category	Clip Polygon	Comments
Sutherland-Hodgman	(3)	convex	
Weiler	(3), (4)	arbitrary	
Liang-Barsky	(4)	rectangular	
Maillot	(1)	rectangular	
Vatti	(1)	arbitrary	Fast, versatile, and can generate a trapezoidal decomposition of the intersection.
Greiner-Hormann	(1)	arbitrary	As general as Vatti. Simpler and potentially faster, but no trapezoidal decomposition.

Line-clipping algorithms fall into two types: those that use encoding of the endpoints of the segment (Cohen-Sutherland) and those that use a parameterization of the line determined by the segment (Cyrus-Beck, Liang-Barsky, and Nicholl-Lee-Nicholl). In Section 4.6 we discuss a hybrid of the two approaches that works well for the clipping needed in the graphics pipeline.

Frequently, one needs to clip more than one edge at a time, as is the case when one wants to clip one polygon against another. One could try to reduce this problem to a sequence of line-clipping problems, but that is not necessarily the most efficient way to do it, because, at the very least, there would be additional bookkeeping involved. The clipped figure may involve introducing some vertices that were not present in the original polygon. In Figure 3.1 we see that the corners **A** and **B**

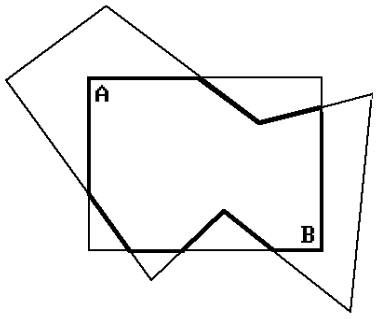


Figure 3.1. Turning points in polygon clipping.

of the window need to be added. These corners are called *turning points*. The term was introduced in [LiaB83] and refers to the point at the intersection of two clipping region edges that has to be added to preserve the connectivity of the original polygon. This is the reason that polygon clipping is treated separately from line clipping.

Polygon-clipping algorithms fall into roughly two categories: *turning-point-based algorithms* like the Liang-Barsky and Maillot algorithms, which rely on quickly being able to find turning points explicitly, and the rest. Turning-point-type algorithms scan the segments of the subject polygon and basically clip each against the **whole** window. The rest tend to find turning points implicitly, in the sense that one does not look for them directly but that they are generated "automatically" as the algorithm proceeds. The Sutherland-Hodgman algorithm treats the clip polygon as the intersection of halfplanes and clips the **whole** subject polygon against each of these halfplanes one at a time. The Weiler, Vatti, and Greiner-Hormann algorithms find the turning points from the clip polygon in the process of tracing out the bounding curves of the components of the polygon intersection, although they trace the boundaries in different ways.

The chapter ends with some comments on clipping text. Some additional comments on clipping when homogeneous coordinates are used can be found in the next chapter in Sections 4.6 and 4.10.

3.2 Line-Clipping Algorithms

3.2.1 Cohen-Sutherland Line Clipping

This section describes an algorithm that solves the following planar clipping problem:

> Given a segment [$\mathbf{P}_1$, $\mathbf{P}_2$], clip it against a rectangular window and return the clipped segment [$\mathbf{Q}_1$, $\mathbf{Q}_2$] (which may be empty if the original segment lies entirely outside the window).

The Cohen-Sutherland line-clipping algorithm is probably the most popular of such algorithms because of its simplicity. It starts out by encoding the nine regions into which the boundary lines of the window divide the whole plane with a 4-bit binary code. See Figure 3.2. If $\mathbf{P}$ is an arbitrary point, then let $c(\mathbf{P}) = x_3x_2x_1x_0$, where x_i is either 0 or 1, define this encoding. The bits x_i have the following meaning:

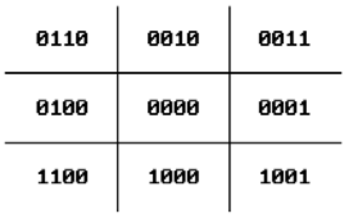


Figure 3.2. Cohen-Sutherland point codes.

$x_0 = 1$ if and only if **P** lies strictly to the right of the right boundary line.
$x_1 = 1$ if and only if **P** lies strictly above the top boundary line.
$x_2 = 1$ if and only if **P** lies strictly to the left of the left boundary line.
$x_3 = 1$ if and only if **P** lies strictly below the bottom boundary line.

The algorithm now has three steps:

Step 1. Encode $\mathbf{P}_1$ and $\mathbf{P}_2$. Let $c_1 = c(\mathbf{P}_1)$ and $c_2 = c(\mathbf{P}_2)$.

Step 2. Check if the segment can be trivially rejected, that is, using the bitwise logical **or** and **and** operators, test whether

(a) c_1 **or** $c_2 = 0$, or
(b) c_1 **and** $c_2 \neq 0$.

In case (a), the segment is entirely contained in the window since both endpoints are and the window is convex. Return $\mathbf{Q}_1 = \mathbf{P}_1$ and $\mathbf{Q}_2 = \mathbf{P}_2$.
In case (b), the segment is entirely outside the window. This follows because the endpoints will then lie in the halfplane determined by a boundary line that is on the other side from the window and halfplanes are also convex. Return the empty segment.

Step 3. If the segment cannot be trivially rejected, then we must subdivide the segment. We clip it against an appropriate boundary line and then start over with Step 1 using the new segment. Do the following to accomplish this:

(a) First find the endpoint **P** that will determine the line to clip against.

If $c_1 = 0000$, then $\mathbf{P}_1$ does not have to be clipped and we let **P** be $\mathbf{P}_2$ and **Q** be $\mathbf{P}_1$.
If $c_1 \neq 0000$, then let **P** be $\mathbf{P}_1$ and **Q** be $\mathbf{P}_2$.

(b) The line to clip against is determined by the left-most nonzero bit in c(**P**). For the example in Figure 3.3, $\mathbf{P} = \mathbf{P}_1$, $\mathbf{Q} = \mathbf{P}_2$, and the line to clip against is the left boundary line of the window. Let **A** be the intersection of the segment [**P**,**Q**] with this line.
(c) Repeat Steps 1–3 for the segment [**A**,**Q**].

The algorithm will eventually exit in Step 2.

With respect to the efficiency of the Cohen-Sutherland algorithm, note that the encoding is easy since it simply involves comparing a number to some constant (the

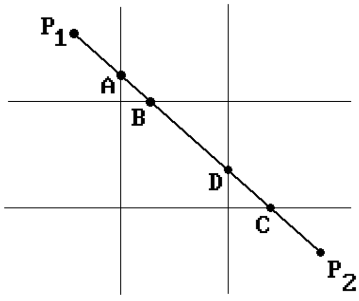


Figure 3.3. Cohen-Sutherland line-clipping example.

boundary lines of the window are assumed to be horizontal and vertical). Step 3 is where the real work might have to be done. We shall have to clip four times in the worst case. One such worst case is shown in Figure 3.3 where we end up having to clip successively against each one of the window boundary lines generating the intersection points **A**, **B**, **C**, and **D**.

Algorithm 3.2.1.1 is an implementation of the just-described algorithm. To be more efficient, all function calls should be replaced by inline code.

Finally, note that the encoding found in the Cohen-Sutherland line-clipping algorithm is driven by determining whether a point belongs to a halfplane. One can easily generalize this to the case where one is clipping a segment against an arbitrary convex set **X**. Assume that **X** is the intersection of halfplanes $\mathbf{H}_1, \mathbf{H}_2, \ldots, \mathbf{H}_k$. The encoding of a point **P** is now a k-bit number $c(\mathbf{P}) = X_K X_{K-1} \ldots X_1$, where

$$X_i \text{ is 1 if } \mathbf{P} \text{ lies in } \mathbf{H}_i \text{ and 0 otherwise.}$$

Using this encoding one can define a clipping algorithm that consists of essentially the same steps as those in the Cohen-Sutherland algorithm. One can also extend this idea to higher dimensions and use it to clip segments against cubes. See Section 4.6.

3.2.2 Cyrus-Beck Line Clipping

The Cyrus-Beck line-clipping algorithm ([CyrB78]) clips a segment **S** against an arbitrary convex polygon **X**. Let $\mathbf{S} = [\mathbf{P}_1,\mathbf{P}_2]$ and $\mathbf{X} = \mathbf{Q}_1\mathbf{Q}_2 \ldots \mathbf{Q}_k$. Since **X** is convex, it is the intersection of halfplanes determined by its edges. More precisely, for each segment $[\mathbf{Q}_i,\mathbf{Q}_{i+1}]$, $i = 1,2,\ldots,k$, ($\mathbf{Q}_{k+1}$ denotes the point $\mathbf{Q}_1$) we can choose a normal vector $\mathbf{N}_i$, so that **X** can be expressed in the form

$$\mathbf{X} = \bigcap_{i=1}^{k} \mathbf{H}_i,$$

where $\mathbf{H}_i$ is the halfplane

$$\mathbf{H}_i = \{\mathbf{Q} \mid \mathbf{N}_i \bullet (\mathbf{Q} - \mathbf{Q}_i) \geq 0\}.$$

With this choice, the normals will point "into" the polygon. It follows that

$$\mathbf{S} \cap \mathbf{X} = \bigcap_{i=1}^{k} (\mathbf{S} \cap \mathbf{H}_i).$$

In other words, we can clip the segment **S** against **X** by successively clipping it against the halfplanes $\mathbf{H}_i$. This is the first basic idea behind the Cyrus-Beck clipping algorithm. The second, is to represent the line **L** determined by the segment **S** parametrically in the form $\mathbf{P}_1 + t\mathbf{P}_1\mathbf{P}_2$ and to do the clipping with respect to the parameter t.

```
{ Constants }
RIGHTBDRY    = 1;
TOPBDRY      = 2;
LEFTBDRY     = 4;
BOTTOMBDRY = 8;

boolean function CS_Clip (ref real x0, y0, x1, y1;  real xmin, ymin, xmax, ymax)
{ This function clips the segment from (x0, y0) to (x1, y1) against the window
  [xmin, xmax]×[ymin, ymax].  It returns false if the segment is entirely outside the
  window and true otherwise.  In the latter case the variables x0, y0, x1, and y1 will
  also have been modified to specify the final clipped segment. }
begin
    byte c0, c1, c;
    real x, y;

    { First encode the points }
    c0 := RegionCode (x0,y0);
    c1 := RegionCode (x1,y1);

    { Next the main loop }
    while  (c0 or c1) ≠ 0  do
        if  (c0 and c1) ≠ 0
            then    return (false);
            else
                begin
                    { Choose the first point not in the window }
                    c := c0;
                    if  c = 0  then  c := c1;

                    { Now clip against line corresponding to first nonzero bit }
                    if  (LEFTBDRY and c) ≠ 0
                        then
                            begin                  { Clip against left bdry }
                                x := xmin;
                                y := y0 + (y1 − y0)*(xmin − x0)/(x1 − x0);
                            end
                        else  if  (RIGHTBDRY and c) ≠ 0
                            then
                                begin              { Clip against right bdry }
                                    x := xmax;
                                    y := y0 + (y1 − y0)*(xmax − x0)/(x1 − x0);
                                end
                            else  if  (BOTTOMBDRY and c) ≠ 0
                                then
```

Algorithm 3.2.1.1. The Cohen-Sutherland line-clipping algorithm.

```
                                       begin      { Clip against bottom bdry }
                                          x := x0 + (x1 – x0)*(ymin – y0)/(y1 – y0);
                                          y := ymin;
                                       end
                                   else if (TOPBDRY and c) ≠ 0
                                       then
                                          begin   { Clip against top bdry }
                                             x := x0 + (x1 – x0)*(ymax – y0)/(y1 – y0);
                                             y := ymax;
                                          end;

                          { Next update the clipped endpoint and its code }
                          if c = c0
                              then
                                 begin
                                    x0 := x;  y0 := y;
                                    c0 := RegionCode (x0,y0);
                                 end
                              else
                                 begin
                                    x1 := x;  y1 := y;
                                    c1 := RegionCode (x1,y1);
                                 end
                      end;  { while }

    return (true);
end;

byte function RegionCode (real x,y);
{ Return the 4-bit code for the point (x,y) }
begin
    byte c;

    c := 0;
    if x < xmin
        then c := c + LEFTBDRY
        else if x > xmax
                then c := c + RIGHTBDRY;
    if y < ymin
        then c := c + BOTTOMBDRY
        else if y > ymax
            then c := c + TOPBDRY;
    return (c);
end;        { RegionCode }
```

Algorithm 3.2.1.1. *Continued*

Let $\mathbf{L}_i$ be the line determined by the segment $[\mathbf{Q}_i,\mathbf{Q}_{i+1}]$. Define intervals $\mathbf{I}_i = [a_i,b_i]$ as follows:

Case 1: $\mathbf{L}$ is parallel to $\mathbf{L}_i$.

(a) If $\mathbf{L}$ lies entirely in $\mathbf{H}_i$, then let $\mathbf{I}_i = (-\infty, +\infty)$.
(b) If $\mathbf{L}$ lies entirely outside of $\mathbf{H}_i$, then let $\mathbf{I}_i = \phi$.

Case 2: $\mathbf{L}$ is not parallel to $\mathbf{L}_i$.

In this case $\mathbf{L}$ will intersect $\mathbf{L}_i$ in some point $\mathbf{P} = \mathbf{P}_1 + t_i\mathbf{P}_1\mathbf{P}_2$. We distinguish between the case where the line $\mathbf{L}$ "enters" the halfplane $\mathbf{H}_i$ and where the line "exits" $\mathbf{H}_i$.

(a) (Line enters) If $\mathbf{P}_1\mathbf{P}_2 \bullet \mathbf{N}_i \geq 0$, then let $\mathbf{I}_i = [t_i, +\infty)$.
(b) (Line exits) If $\mathbf{P}_1\mathbf{P}_2 \bullet \mathbf{N}_i < 0$, then let $\mathbf{I}_i = (-\infty, t_i]$.

See Figure 3.4, where a segment $\mathbf{S} = [\mathbf{P}_1,\mathbf{P}_2]$ is being clipped against a triangle $\mathbf{Q}_1\mathbf{Q}_2\mathbf{Q}_3$. Note that finding the intersection point $\mathbf{P}$ in Case 2 is easy. All we have to do is solve the equation

$$\mathbf{N}_i \bullet (\mathbf{P}_1 + t\,\mathbf{P}_1\mathbf{P}_2 - \mathbf{Q}_i) = 0$$

for t.

Now let $\mathbf{I}_0 = [a_0,b_0] = [0,1]$. The interval $\mathbf{I}_0$ is the set of parameters of points which lie is $\mathbf{S}$. It is easy to see that the interval

$$\begin{aligned} I &= \bigcap_{i=0}^{k} I_i \\ &= \left[\max_{0\le i\le k} a_i, \min_{0\le i\le k} b_i\right] \\ &= [a,b] \end{aligned}$$

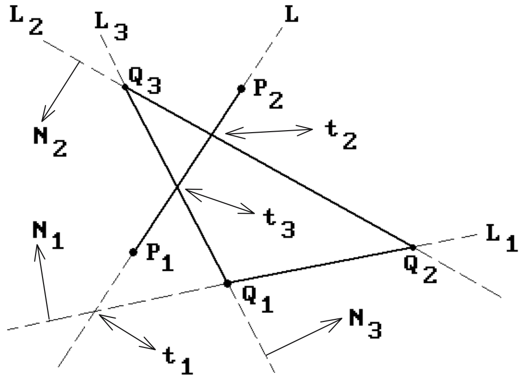


Figure 3.4. Cyrus-Beck line clipping.

is the set of parameters for the points in $\mathbf{S} \cap \mathbf{X}$. In other words, if **I** is not empty, then

$$\mathbf{S} \cap \mathbf{X} = [\mathbf{P}_1 + a\mathbf{P}_1\mathbf{P}_2, \mathbf{P}_1 + b\mathbf{P}_1\mathbf{P}_2].$$

We shall explain this process with the example in Figure 3.4. In this example,

$$\mathbf{I} = [0,1] \cap [t_1,+\infty) \cap (-\infty,t_2] \cap [t_3,+\infty) = [t_3,t_2],$$

which clearly gives the right answer.

3.2.3 Liang-Barsky Line Clipping

The Liang-Barsky line-clipping algorithm ([LiaB84]) optimizes the Cyrus-Beck line-clipping algorithm in the case where we are clipping against a rectangle. It starts by treating a segment as a parameterized set. Let $\mathbf{P}_1 = (x_1,y_1)$ and $\mathbf{P}_2 = (x_2,y_2)$. A typical point $\mathbf{P} = (x,y)$ on the oriented line **L** determined by $\mathbf{P}_1$ and $\mathbf{P}_2$ then has the form $\mathbf{P}_1 + t\mathbf{P}_1\mathbf{P}_2$. See Figure 3.5. If we let $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$, then

$$\begin{aligned} x &= x_1 + \Delta x\, t \\ y &= y_1 + \Delta y\, t. \end{aligned}$$

If the window **W** we are clipping against is the rectangle [xmin,xmax] × [ymin,ymax], then **P** belongs to **W** if and only if

$$\begin{aligned} \text{xmin} &\le x_1 + \Delta x\, t \le \text{xmax} \\ \text{ymin} &\le y_1 + \Delta y\, t \le \text{ymax} \end{aligned}$$

that is,

$$\begin{aligned} -\Delta x\, t &\le x_1 - \text{xmin} \\ \Delta x\, t &\le \text{xmax} - x_1 \\ -\Delta y\, t &\le y_1 - \text{ymin} \\ \Delta y\, t &\le \text{ymax} - y_1. \end{aligned}$$

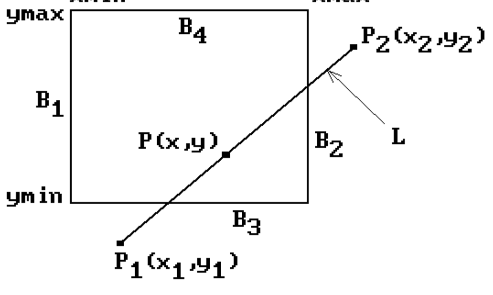


Figure 3.5. Liang-Barsky line clipping.

To simplify the notation, we introduce variables c_k and q_k and rewrite these equations as

$$\begin{aligned} c_1 t &\le q_1 \\ c_2 t &\le q_2 \\ c_3 t &\le q_3 \\ c_4 t &\le q_4. \end{aligned}$$

Set $t_k = q_k/c_k$ whenever $c_k \neq 0$. Let $\mathbf{B}_1$, $\mathbf{B}_2$, $\mathbf{B}_3$, and $\mathbf{B}_4$ denote the left, right, bottom, and top boundary lines of the window, respectively. With this notation we can make the following observations:

(1) If $c_k > 0$, then **L** goes from the "inside" to the "outside" of the boundary line $\mathbf{B}_k$ as t increases and we shall call t_k an *exit* value.
(2) If $c_k < 0$, then **L** goes from the "outside" to the "inside" of the boundary line $\mathbf{B}_k$ as t increases and we shall call t_k an *entry* value.
(3) If $c_k = 0$, the **L** is parallel to $\mathbf{B}_k$, which is outside the window if $q_k < 0$.

The clipping algorithm now proceeds by analyzing the three quantities q_k, c_k, and t_k. Algorithm 3.2.3.1 is a high-level version of the Liang-Barsky algorithm. Algorithm 3.2.3.2 gives the code for the actual Liang-Barsky algorithm.

3.2.3.1 Example. Consider the segment $[\mathbf{P}_1,\mathbf{P}_2]$ in Figure 3.6. We see that $c_1, c_4 < 0$ and $c_2, c_3 > 0$. In other words, t_1 and t_4 are entry values and t_2 and t_3 are exit values. The picture bears this out. One can also easily see that $t_1 < 0 < t_2 < t_4 < 1 < t_3$. Therefore, there is an entry value (t_4) that is larger than an exit value (t_2). Using the algorithm we conclude that the segment lies entirely outside the window, which is correct.

```
Reject the segment as soon as

    an entry value is larger than 1 or
        an exit value is less than 0 or
            an entry value is larger than an exit value

Otherwise, the segment meets the window.  We need to compute an intersection
only if  t0 > 0  and  t1 < 1, where

            $t0 = \max(0, \max\{\text{entry values } t_k\})$, and
            $t1 = \min(1, \min\{\text{exit values } t_k\})$.

(The case where  t0 = 0  or  t1 = 1  means that we should use the endpoint, that is,
no clipping is necessary.)
```

Algorithm 3.2.3.1. High-level Liang-Barsky line-clipping algorithm.

```
boolean function LB_Clip (ref real  x0, y0, x1, y1;  real xmin, ymin, xmax, ymax)
{ This function clips the segment from (x0, y0) to (x1, y1) against the window
[xmin, xmax]×[ymin, ymax].  It returns false if the segment is entirely outside the
window and true otherwise.  In the latter case the variables x0, y0, x1, and y1 will
also have been modified to specify the final clipped segment. }
begin
    real      t0, t1, dx, dy;
    boolean more;

    t0 := 0;  t1 := 1;  dx := x1 – x0;

    Findt (–dx,x0 – xmin,t0,t1,more);          { left bdry }
    if  more  then
        begin
            Findt (dx,xmax – x0,t0,t1,more);       { right bdry }
            if  more  then
                begin
                    dy := y1  –  y0;
                    Findt (–dy,y0 – ymin,t0,t1,more);  { bottom bdry }
                    if  more  then
                        begin
                            Findt (dy,ymax – y0,t0,t1,more);  { top bdry }
                            if  more  then
                                begin                  { clip the line }
                                    if  t1 < 1  then
                                        begin          { calculate exit point }
                                            x1 := x0 + t1*dx;
                                            y1 := y0 + t1*dy;
                                        end;
                                    if  t0 > 0  then
                                        begin          { calculate entry point }
                                            x0 := x0 + t0*dx;
                                            y0 := y0 + t0*dy;
                                        end;
                                end
                        end
                end
        end;
    return (more);
end;

procedure Findt (real denom, num; ref real t0, t1; ref boolean more)
begin
    real r;

    more := true;
```

Algorithm 3.2.3.2. The Liang-Barsky line-clipping algorithm.

```
        if denom < 0
            then
                begin                   { line from outside to inside }
                    r := num/denom;
                    if r > t1
                        then more := false
                        else if r > t0 then t0 := r;
                end
            else if denom > 0
                then                    { line from inside to outside }
                    begin
                        r := num/denom;
                        if r < t0
                            then more := false
                            else if r < t1 then t1 := r;
                    end
                else if num < 0         { line parallel to boundary }
                    then more := false;
    end; { Findt }
```

Algorithm 3.2.3.2. *Continued*

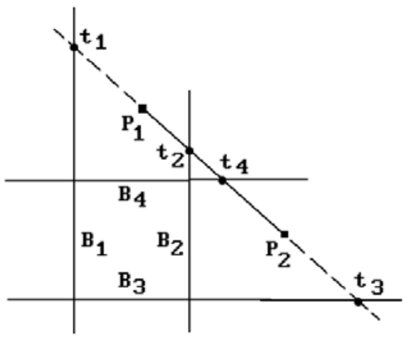


Figure 3.6. Liang-Barsky line-clipping example.

3.2.3.2 Example. Consider the segment [$\mathbf{P}_1$,$\mathbf{P}_2$] in Figure 3.7. In this example, c_1, $c_3 < 0$ and c_2, $c_4 > 0$, so that t_1 and t_3 are entry values and t_2 and t_4 are exit values. Furthermore, $t_1 < 0 < t_3 < 1 < t_2 < t_4$. This time we cannot reject the segment and must compute t0 = max (0, c_1, c_3) = c_3 and t1 = min (1, c_2, c_4) = 1. The algorithm tells us that we must clip the segment at the $\mathbf{P}_1$ end to get $\mathbf{Q}$ but do not need to clip at the $\mathbf{P}_2$ end. Again this is clearly what had to be done.

In conclusion, the advantage of the Liang-Barsky algorithm over the Cohen-Sutherland algorithm is that it involves less arithmetic and is therefore faster. It needs only two subtractions to get q_k and c_k and then one division.

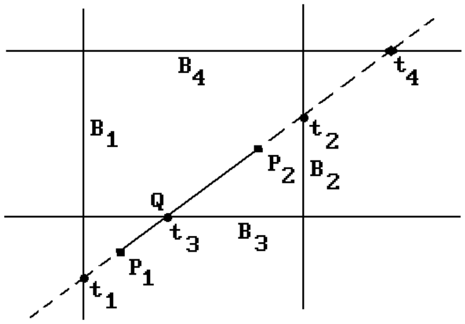


Figure 3.7. Liang-Barsky line-clipping example.

3.2.4 Nicholl-Lee-Nicholl Line Clipping

One of the problems common to both the Cohen-Sutherland and the Liang-Barsky algorithm is that more intersections are computed than necessary. For example, consider Figure 3.6 again where we are clipping line segment [$\mathbf{P}_1$,$\mathbf{P}_2$] against the window. The Cohen-Sutherland algorithm will compute the intersection of the segment with the top boundary at t_4 even though the segment is later rejected. The Liang-Barsky algorithm will actually compute **all** the parameter values corresponding to the intersection of the line with the window. Avoiding many of these wasted computations is what the Nicholl-Lee-Nicholl line-clipping algorithm ([NiLN87]) is all about. These authors also make a detailed analysis of the deficiencies of the Cohen-Sutherland and Liang-Barsky algorithms. Their final algorithm is much faster than either of these. It is not really much more complicated conceptually, but involves many cases. We describe one basic case below.

Assume that we want to clip a segment [$\mathbf{P}_1$,$\mathbf{P}_2$] against a window. The determination of the exact edges, if any, that one needs to intersect, reduces, using symmetry, to an analysis of the three possible positions of $\mathbf{P}_1$ shown in Figure 3.8. The cases are

(1) $\mathbf{P}_1$ is in the window (Figure 3.8(a)),
(2) $\mathbf{P}_1$ is in a "corner region" (Figure 3.8(b)), or
(3) $\mathbf{P}_1$ is in an "edge region" (Figure 3.8(c)).

For each of these cases one determines the regions with the property that no matter where in the region the second point $\mathbf{P}_2$ is, the segment will have to be intersected with the **same** boundaries of the window. These regions are also indicated in Figure 3.8. As one can see, these regions are determined by drawing the rays from $\mathbf{P}_1$ through the four corners of the window. The following abbreviations were used:

T – ray intersects top boundary	LT – ray intersects left and top boundary
L – ray intersects left boundary	LR – ray intersects left and right boundary
B – ray intersects bottom boundary	LB – ray intersects left and bottom boundary
R – ray intersects right boundary	TR – ray intersects top and right boundary
	TB – ray intersects top and bottom boundary

For example, suppose that the segment [$\mathbf{P}_1$,$\mathbf{P}_2$] is as shown in Figure 3.8(c). Here are the computations one has to perform. Let $\mathbf{P}_i = (x_i,y_i)$ and let $\mathbf{C}$ = (xmax,ymin) be the corner of the window also indicated in Figure 3.8(c). After checking that $y_2 <$ ymin, we must determine whether the vector $\mathbf{P}_1\mathbf{P}_2$ is above or below the vector $\mathbf{P}_1\mathbf{C}$. This reduces to determining whether the ordered basis ($\mathbf{P}_1\mathbf{C}$,$\mathbf{P}_1\mathbf{P}_2$) determines the standard orientation of the plane or not. Since

$$\det\begin{pmatrix}\mathbf{P}_1\mathbf{C}\\ \mathbf{P}_1\mathbf{P}_2\end{pmatrix} = (x\max - x_1)(y_2 - y_1) - (y\min - y_1)(x_2 - x_1) < 0,$$

$\mathbf{P}_1\mathbf{P}_2$ is below $\mathbf{P}_1\mathbf{C}$. We now know that we will have to compute the intersection of [$\mathbf{P}_1$,$\mathbf{P}_2$] with both the left and bottom boundary of the window.

Algorithm 3.2.4.1 is an abstract program for the Nicholl-Lee-Nicholl algorithm in the edge region case ($\mathbf{P}_1$ in the region shown in Figure 3.8(c)). We assume a window [xmin,xmax] × [ymin,ymax].

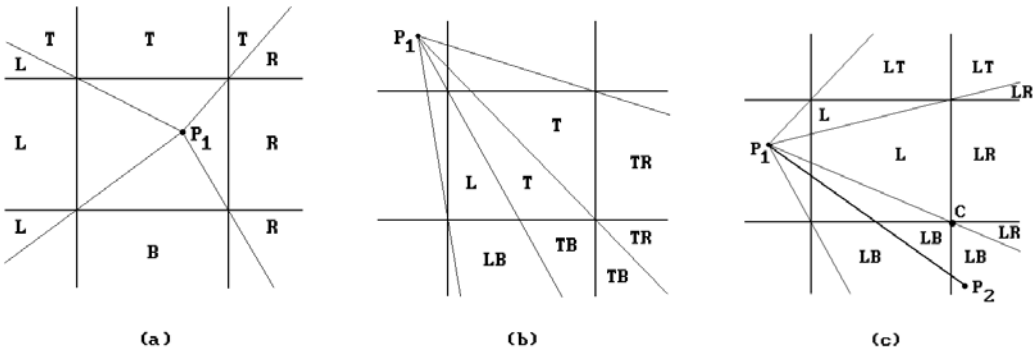


Figure 3.8. Nicholl-Lee-Nicholl line clipping.

```
procedure LeftEdgeRegionCase (ref real x1, y1, x2, y2; ref boolean visible)
begin
    real dx, dy;

    if x2 < xmin
        then visible := false
        else if y2 < ymin
            then LeftBottom (xmin,ymin,xmax,ymax,x1,y1,x2,y2,visible)
            else if y2 > ymax
                then
                    begin
```

Algorithm 3.2.4.1. The edge region case of the Nicholl-Lee-Nicholl line-clipping algorithm.

```
                    { Use symmetry to reduce to LeftBottom case }
                    y1 := −y1;   y2 := −y2;   { reflect about x-axis }
                    LeftBottom (xmin,−ymax,xmax, −ymin,x1,y1,x2,y2,visible);
                    y1 := −y1;   y2 := −y2;   { reflect back }
                end
            else
                begin
                    dx := x2 − x1;   dy := y2 − y1;
                    if  x2 > xmax  then
                        begin
                            y2 := y1 + dy*(xmax − x1)/dx;    x2 := xmax;
                        end;
                    y1 := y1 + dy*(xmin − x1)/dx;    x1 := xmin;
                    visible := true;
                end
end;

procedure LeftBottom (      real  xmin, ymin, xmax, ymax;
                        ref real  x1, y1, x2, y2; ref boolean  visible)
begin
    real dx, dy, a, b, c;

    dx := x2 − x1;              dy := y2 − y1;
    a   := (xmin − x1)*dy;    b   := (ymin − y1)*dx;
    if  b > a
        then  visible := false  { (x2,y2) is below ray from (x1,y1) to bottom left corner }
        else
            begin
                visible := true;
                if  x2 < xmax
                    then
                        begin   x2 := x1 + b/dy;    y2 := ymin;    end
                    else
                        begin
                            c := (xmax − x1)*dy;
                            if  b > c
                                then   { (x2,y2) is between rays from (x1,y1) to
                                                bottom left and right corner }
                                    begin   x2 := x1 + b/dy;    y2 := ymin;    end
                                else
                                    begin   y2 := y1 + c/dx;    x2 := xmax;   end
                        end;
            end;
    y1 := y1 + a/dx;   x1 := xmin;
end;
```

Algorithm 3.2.4.1. *Continued*

To deal with symmetry only rotations through 90, 180, and 270 degrees about the origin and reflections about the lines x = –y and the x-axis are needed. These operations are extremely simple and involve only negation and assignment. See [NiLN87] for further details.

This finishes our survey of line-clipping algorithms. Next, we turn our attention to polygon-clipping algorithms.

3.3 Polygon-Clipping Algorithms

3.3.1 Sutherland-Hodgman Polygon Clipping

One of the earliest polygon-clipping algorithms is the Sutherland-Hodgman algorithm ([SutH74]). It is based on clipping the **entire** subject polygon against an edge of the window (more precisely, the halfplane determined by that edge which contains the clip polygon), then clipping the new polygon against the next edge of the window, and so on, until the polygon has been clipped against all of the four edges. An important aspect of their algorithm is that one can avoid generating a lot of intermediate data.

Representing a polygon as a sequence of vertices $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_n$, suppose that we want to clip against a single edge **e**. The algorithm considers the input vertices $\mathbf{P}_i$ one at a time and generates a new sequence $\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_m$. Each $\mathbf{P}_i$ generates 0, 1, or 2 of the $\mathbf{Q}_j$, depending on the position of the input vertices with respect to **e**. If we consider each input vertex **P**, except the first, to be the terminal vertex of an edge, namely the edge defined by **P** and the immediately preceding input vertex, call it **S**, then the **Q**'s generated by **P** depend on the relationship between the edge [**S**,**P**] and the line **L** determined by **e**. There are four possible cases. See Figure 3.9. The window side of the line is marked as "inside." The circled vertices are those that are output. Figure 3.10 shows an example of how the clipping works. Clipping the polygon with vertices labeled $\mathbf{P}_i$ against edge $\mathbf{e}_1$ produces the polygon with vertices $\mathbf{Q}_i$. Clipping the new polygon against edge $\mathbf{e}_2$ produces the polygon with vertices $\mathbf{R}_i$.

Note that we may end up with some bogus edges. For example, the edge $\mathbf{R}_5\mathbf{R}_6$ in Figure 3.10 is not a part of the mathematical intersection of the subject polygon with

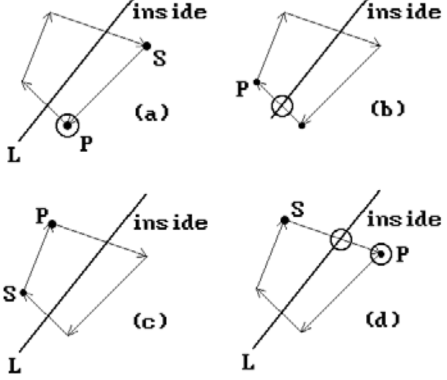


Figure 3.9. The four cases in Sutherland-Hodgman polygon clipping.

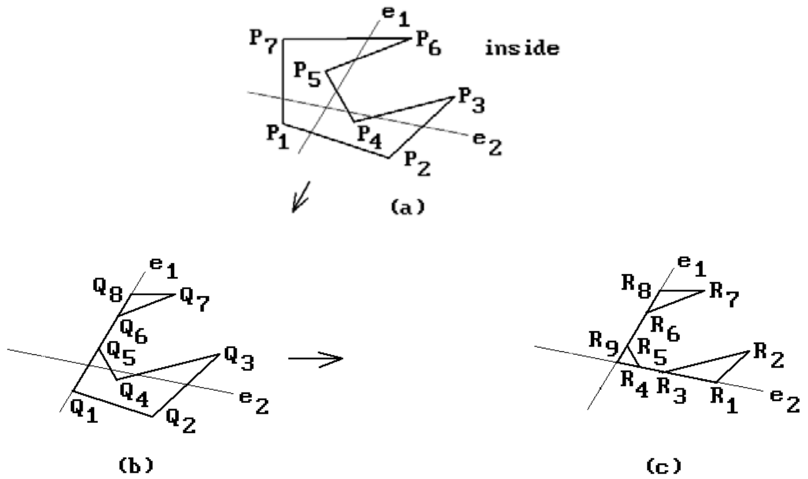


Figure 3.10. A Sutherland-Hodgman polygon-clipping example.

the clip polygon. Eliminating such edges from the final result would be a nontrivial effort, but normally they do not cause any problems. We run into this bogus edge problem with other clipping algorithms also.

An implementation of the Sutherland-Hodgman algorithm can be found in [PokG89].

3.3.2 Weiler Polygon Clipping

Another early polygon clipping algorithm was developed in the context of the visible surface determination algorithm in [WeiA77]. Weiler and Atherton needed a new algorithm because the Sutherland-Hodgman algorithm would have created too many auxiliary polygons. An improved version of the algorithm can be found in [Weil80]. Here is a very brief description of the algorithm:

> The boundaries of polygons are assumed to be oriented so that the inside of the polygon is always to the right as one traverses the boundary. Note that intersections of the subject and clip polygon, if any, occur in pairs: one where the subject *enters* the inside of the clip polygon and one where it *leaves*.
>
> **Step 1:** Compare the borders of the two polygons for intersections. Insert vertices into the polygons at the intersections.
> **Step 2:** Process the nonintersecting polygon borders, separating those contours that are outside the clip polygon and those that are inside.
> **Step 3:** Separate the intersection vertices found on all subject polygons into two lists. One is the *entering list,* consisting of those vertices where the polygon edge enters the clip polygon. The other is the *leaving list,* consisting of those vertices where the polygon edge leaves the clip polygon.
> **Step 4:** Now clip.

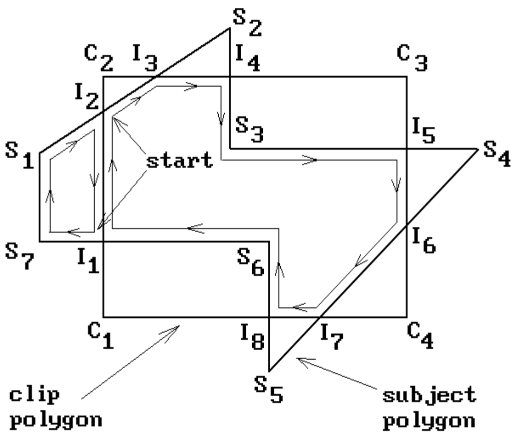


Figure 3.11. Weiler polygon clipping.

(a) Remove an intersection vertex from the entering list. If there is none, then we are done.
(b) Follow the subject polygon vertices to the next intersection.
(c) Jump to the clip polygon vertex list.
(d) Follow the clip polygon vertices to the next intersection.
(e) Jump back to the subject polygon vertex list.
(f) Repeat (b)–(e) until we are back to the starting point.

This process creates the polygons inside the clip polygon. To get those that are outside, one repeats the same steps, except that one starts with a vertex from the leaving list and the clip polygon vertex list is followed in the **reverse** direction. Finally, all holes are attached to their associated exterior contours.

3.3.2.1 Example. Consider the polygons in Figure 3.11. The subject polygon vertices are labeled $\mathbf{S}_i$, those of the clip polygon are labeled $\mathbf{C}_i$, and the intersections are labeled $\mathbf{I}_i$. The entering list consists of $\mathbf{I}_2$, $\mathbf{I}_4$, $\mathbf{I}_6$, and $\mathbf{I}_8$. The leaving list consists of $\mathbf{I}_1$, $\mathbf{I}_3$, $\mathbf{I}_5$, and $\mathbf{I}_7$. Starting Step 4(a) with the vertex $\mathbf{I}_2$ will generate the inside contour

$$\mathbf{I}_2\mathbf{I}_3\mathbf{I}_4\mathbf{S}_3\mathbf{I}_5\mathbf{I}_6\mathbf{I}_7\mathbf{I}_8\mathbf{S}_6\mathbf{I}_1\mathbf{I}_2.$$

Starting Step 4(a) with vertices $\mathbf{I}_1$, $\mathbf{I}_3$, $\mathbf{I}_5$, and $\mathbf{I}_7$ will generate the outside contours

$$\mathbf{I}_1\mathbf{S}_7\mathbf{S}_1\mathbf{I}_2\mathbf{I}_1,\ \mathbf{I}_3\mathbf{S}_2\mathbf{I}_4\mathbf{I}_3,\ \mathbf{I}_5\mathbf{S}_4\mathbf{I}_6\mathbf{I}_5,\ \text{and}\ \mathbf{I}_7\mathbf{S}_5\mathbf{I}_8\mathbf{I}_7.$$

3.3.3 Liang-Barsky Polygon Clipping

This section gives a brief outline of the Liang-Barsky polygon-clipping algorithm ([LiaB83]). The algorithm is claimed to be twice as fast as the Sutherland-Hodgman clipping algorithm. This algorithm and the next one, the Maillot algorithm, base their success on their ability to detect turning points efficiently. Before we get to the algorithm, some comments on turning points are in order.

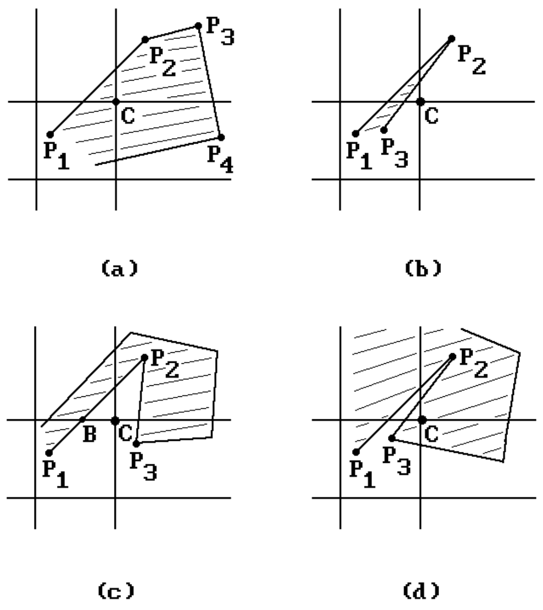


Figure 3.12. Different types of turning points.

Assume that [$\mathbf{P}_1$,$\mathbf{P}_2$] is an edge of a polygon. It is easy to see that the only time that this edge is relevant to the issue of turning points is if it enters or exits a corner region associated to the window. Figure 3.12 shows some cases of polygons (the shaded regions) and how the exiting edge [$\mathbf{P}_1$,$\mathbf{P}_2$] affects whether or not the corner **C** becomes a turning point and needs to be added to the output. One sees the following:

(1) The analysis divides into two cases: whether the polygon is to the right or left of the edge. (Figure 3.12(a,b) versus Figure 3.12(c,d))
(2) There are two subcases that depend on which side of the ray from $\mathbf{P}_2$ to **C** the segment [$\mathbf{P}_2$,$\mathbf{P}_3$] is located.
(3) The decision as to whether a turning point will be needed cannot be made on the basis of only a few edges. In principle one might have to look at all the edges of the polygon first.

It is observation (3) that complicates life for polygon-clipping algorithms that process edges sequentially in one pass. One could simplify life and generate a turning point **whenever** we run into an edge that enters, lies in, or exits a corner region. The problem with this approach is that one will generate bogus edges for our clipped polygon. The polygon in Figure 3.12(c) would generate the "dangling" edge [**B**,**C**]. Bogus edges were already encountered in Sutherland-Hodgman clipping (but for different reasons). These edges might not cause any problems, as in the case where one is simply filling two-dimensional regions. On the other hand, one would like to minimize the number of such edges, but avoiding them entirely would be very complicated with some algorithms like the Liang-Barsky and Maillot algorithm.

With this introduction, let us describe the Liang-Barsky algorithm. We shall be brief because it does not contain much in the way of new insights given the fact that

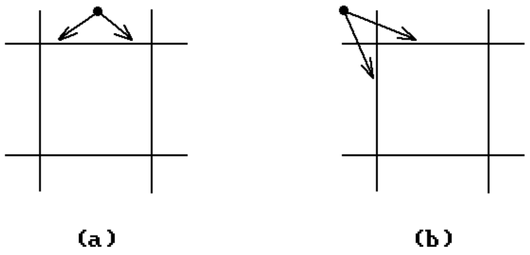


Figure 3.13. Testing for turning points.

it is built on the approach they use in their line-clipping algorithm. Furthermore, another algorithm, the Maillot algorithm, is better. Our discussion here will follow [FVFH90], who have modified the algorithm so that it is easier to follow although it has the potential disadvantage of creating more bogus edges.

Consider Figure 3.13. We extend the edges of our rectangular window to divide the plane into nine regions. If we are at a point of the polygon that lies in one of the four outside regions that meet the window in an edge, then the next edge of the polygon can only meet the window in that edge. See Figure 3.13(a). On the other hand, if the point is in one of the four corner regions, then the next edge could meet the window in one of two possible edges. Without look-ahead, we really cannot tell whether the adjacent corner of the window will become a turning point and will have to be added to the output polygon. In such a situation we shall play it safe and always include the corner point (even though this may create some of these unwanted edges that we have been talking about). This is the first point to make about the algorithm.

The other point is that the algorithm rests on the idea of entry and exit points for the edges of the polygon that correspond to the entry and exit values used in the line-clipping algorithm described in Section 3.2.3. By analyzing these points one can tell if an edge intersects the window or if it gives rise to a turning point. As we work our way through the edges of the polygon, assume that the current edge $\mathbf{e} = [\mathbf{p}_i,\mathbf{p}_{i+1}]$ is neither vertical nor horizontal. Then **e** will intersect all the boundary lines of our window. If we parameterize the line containing **e** in the form $\mathbf{p}_i + t\mathbf{p}_i\mathbf{p}_{i+1}$, then the four intersection points correspond to four parameter values and can again be classified as two entry and two exit points. We shall denote these associated parameter values by t_in1, t_in2, t_out1, and t_out2, respectively. It is easy to see that the smallest of these is an entry value and we shall let t_in1 be that one. The largest is an exit value and we shall let t_out2 be that one. Nothing can be said about the relative size of the remaining two values in general. From Section 3.2.3 we know, however, that if t_in2 < t_out1, then the line intersects the window and if t_out2 < t_in1, then the line intersects a corner region.

If a line does not intersect the window, then it must intersect three corner regions. The conditions for that are that 0 < t_out1 ≤ 1 and 0 < t_out2 ≤ 1. The last statement also holds if the line intersects the window. Putting all these facts together leads to Algorithm 3.3.3.1. However, we were assuming in the discussion that edges were neither horizontal nor vertical. We could deal with such lines by means of special cases, but the easiest way to deal with them and preserve the structure of Algorithm 3.3.3.1 is to use a trick and assign dummy $\pm\infty$ values to the missing entering and leaving parameters. See [FVFH90].

```
for each edge e of polygon do
    begin
        Determine the direction of e;    { Used to tell in what order the bounding
                                           lines of the clip region will be hit }
        Find exit t values;
        if t_out2 > 0 then find t_in2;
        if t_in2 > t_out1
            then                         { No visible segment }
                begin
                    if 0 < t_out1 ≤ 1 then OutputVertex (turning vertex);
                end
            else
                begin
                    if (0 < t_out1) and (t_in2 ≤ 1) then
                        begin            { Part of segment is visible }
                            if 0 ≤ t_in2
                                then OutputVertex (appropriate side intersection)
                                else OutputVertex (starting vertex);
                            if t_out1 ≤ 1
                                then OutputVertex (appropriate side intersection)
                                else OutputVertex (ending vertex);
                        end
                end;
        if 0 < t_out2 ≤ 1 then OutputVertex (appropriate corner);
    end;
```

Algorithm 3.3.3.1. Overview of a Liang-Barsky polygon-clipping algorithm.

3.3.4 Maillot Polygon Clipping

The Maillot clipping algorithm ([Mail92]) clips arbitrary polygons against a rectangular window. It uses the well-known Cohen-Sutherland clipping algorithm for segments as its basis and then finds the correct turning points for the clipped polygon by maintaining an additional bit of information. As indicated earlier, it is speedy determination of turning points that is crucial for polygon clipping, and this algorithm does it very efficiently. We shall use the same notation that was used in Section 3.2.1. We also assume that the **same** encoding of points is used. This is very important; otherwise, the tables Tcc and Cra below **must** be changed.

Let **P** be a polygon defined by the sequence of vertices $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n, \mathbf{p}_{n+1} = \mathbf{p}_0$. Algorithm 3.3.4.1 gives a top-level description of the Maillot algorithm.

In addition to the Cohen-Sutherland trivial rejection cases, Maillot's algorithm subjects all vertices of the polygon to one extra test, which he calls the "basic turning point test." This test checks for the case where the current point lies in one of the four

```
p := p0;   cp := c(p);
for i:=1 to n+1 do
    begin
        q := pi;   cq := c(q);

        { Clip the segment [p,q] as in Cohen-Sutherland algorithm }
        DoCSClip ();

        if segment [p,q] is outside clipping region then TestForComplexCase;

        DoBasicTurningPointTest ();

        p := q;   cp := cq;
    end;
```

Algorithm 3.3.4.1. Overview of Maillot polygon-clipping algorithm.

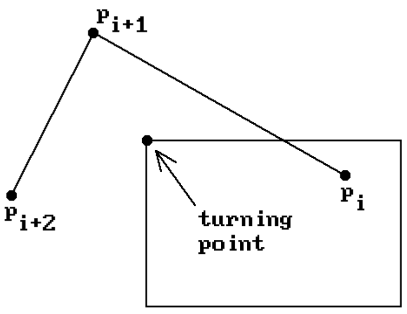


Figure 3.14. A turning point case.

corners outside the window. One probably needs to add a turning point to the clipped polygon in this case. See Figure 3.14. We said "probably" because if the current point is considered in isolation (without looking at its predecessors), then to always automatically add the point may cause us to add the same corner several times in a row. See points $\mathbf{p}_i$, $\mathbf{p}_{i+1}$, and $\mathbf{p}_{i+2}$ in Figure 3.14. In the implementation of Maillot's algorithm, we do not try to eliminate such redundancies. If this is not desired, then extra code will have to be added to avoid it.

If all of a polygon's edges meet the window, then the basic turning point test is all that is needed to clip it correctly. For polygons that have edges entirely **outside** the clipping region, one needs to do more. Figure 3.15 shows all (up to symmetry) generic cases that need to be handled in this more complex situation. The following terminology is useful for the case of edges outside the clipping region.

Notation. A point that lies in a region with code 0001, 0010, 0100, or 1000 will be called a *1-bit point*. A point that lies in a region with code 0011, 0110, 1100, or 1001 will be called a *2-bit point*. A segment is called an *x-y segment* if its start point is an x-bit point and its endpoint is a y-bit point.

Knowing the type of segment that one has is important for the algorithm. This is why an extra bit is used in the encoding of points. It is stuck at the left end of the original Cohen-Sutherland code. Below is an overview of the actions that are taken in the TestForComplexCase procedure. Refer to Figure 3.15.

The 1–1 Segment Cases (Segments **a** and **b).** Either the two points have the same code (segment **a**) and no turning point needs to be generated or they have different codes (segment **b**). In the latter case there is one turning point that can be handled by the basic turning point test. The code for the corner for this turning point is computed from the **or** of the two codes and a lookup table (the Tcc table in the code).

The 2–1 and 1–2 Segment Cases (Segments **c** and **d).** In this case one point of the segment has a 1-bit code and the other, a 2-bit code.

(a) The endpoint is the point with the 1-bit code (segment **c**): If both codes **and** to a nonzero value (segment [**P**,**R**] in Figure 3.16(a)), there is no turning point. If both

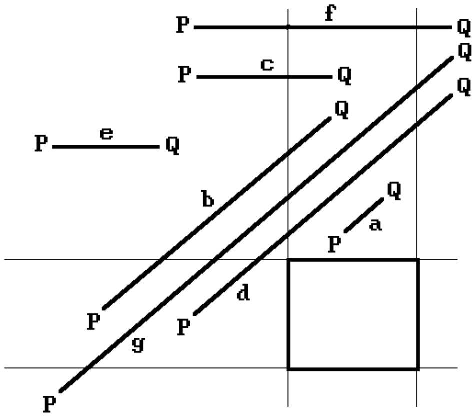


Figure 3.15. Turning point cases in Maillot algorithm.

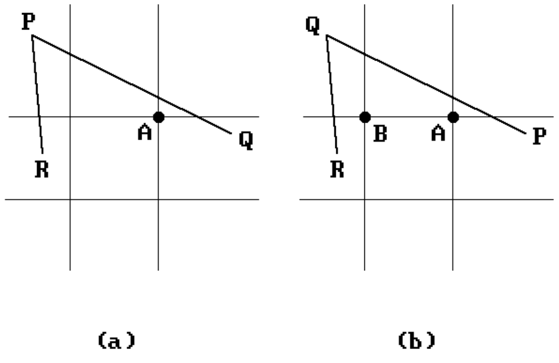


Figure 3.16. Turning point tests.

codes **and** to zero, then we need to generate a turning point that depends on the two codes. A lookup table (Tcc in the code) is used for this.

(b) The endpoint has the 2-bit code (segment **d**): The case where the **and** of both codes is nonzero is handled by the basic turning point test (segment [**R**,**Q**] in Figure 3.16(b). If both codes **and** to zero, we need two turning points. The first one depends on the two codes and is determined by again using a lookup table (Tcc in the code). The other is generated by the basic turning point test (segment [**P**,**Q**] in Figure 3.16(b)).

As an example of how the Tcc table is generated, consider the segment [**P**,**Q**] in Figure 3.16(b). In the figure there are two turning points **A** and **B**. The basic turning point test applied to **Q** will generate **B**. Let us see how **A** is generated. How can one compute the code, namely 3, for this turning point? Maillot defines the sixteen element Tcc table in such a way that the following formula works:

$$\text{newCode} = \text{code}(\mathbf{Q}) + \text{Tcc}[\text{code}(\mathbf{P})]$$

For the 1–1, 2–1, and 1–2 segment cases only four entries of Tcc are used in conjunction with this formula. Four other entries are set to 1 and used in the 2–2 segment case discussed below when it runs into a 1–1 segment. The remaining eight of the entries in Tcc are set to 0.

The 2–2 Segment Case (Segments **e, f** and **g).** There are three subcases.

(a) Both points have the same code (segment **e**): No turning point is needed here.

(b) Both codes **and** to a nonzero value (segment **f**): Apply the basic turning point test to the end point.

(c) Both codes **and** to a zero value (segment **g**): There will be two turning points. One of them is easily generated by the basic turning point test. For the other one we have a situation as shown in Figure 3.17 and we must decide between the two possible choices **A** or **B**. Maillot uses a midpoint subdivision approach wherein the edge is successively divided into two until it can be handled by the previous cases. The

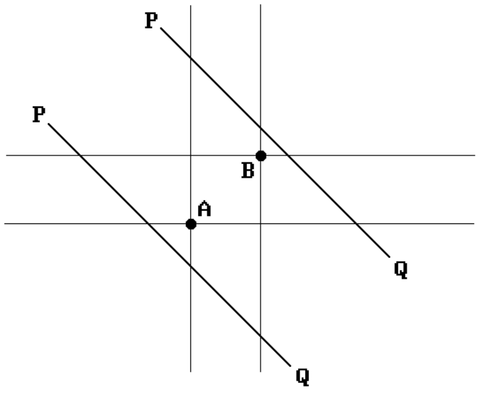


Figure 3.17. 2-2 segment case turning points.

number of subdivisions required depends on the precision used. For 32-bit integers, there will be less than 32 subdivisions.

Maillot presents a C implementation of his algorithm in [Mail92]. Our version of this algorithm is Algorithm 3.3.4.2 below. The main difference is that we tried to be as clear as possible by using extra auxiliary functions and procedures. To be efficient, however, all these calls should be eliminated and the code put inline.

As mentioned earlier, Maillot's algorithm uses the Cohen-Sutherland clipping algorithm. One can use the implementation in Section 3.2.1 for this except that the extended encoding function (ExtendedCsCode) shown in Algorithm 3.3.4.3 should be

```
{ Constants }
MAXSIZE  = 1000; { maximum size of pnt2d array }
NOSEGM   = 0;    { segment was rejected }
SEGM     = 1;    { segment is at least partially visible }
CLIP     = 2;    { segment was clipped }
TWOBITS  = $10;  { flag for 2-bit code }

{ Two lookup tables for finding turning point.
   Tcc is used to compute a correct offset.
   Cra gives an index into the clipRegion array for turning point coordinates. }
integer array [0..15] Tcc = (0, −3, −6,1,3,0,1,0,6,1,0,0,1,0,0,0);
integer array [0..15] Cra = (−1, −1, −1,3, −1, −1,2, −1, −1,1, −1, −1,0, −1, −1, −1);

pnt2d = record
    real x, y;
end;

pnt2ds = pnt2d array [0..MAXSIZE];

{ Global variables }
{ The clipping region [xmin,xmax]×[ymin,ymax] bounds listed in order:
   (xmin,ymin),(xmax,ymin),(xmin,ymax),(xmax,ymax) }
array [0..3] of pnt2d clipRegion;

pnt2d   startPt;  { start point of segment }
integer startC;   { code for start point }
integer startC0;  { saves startC for next call to CS_EndClip }
pnt2d   endPt;    { endpoint of segment }
integer endC;     { code for endpoint }
integer aC;       { used by procedure TwoBitEndPoint }
```

Algorithm 3.3.4.2. The Maillot polygon-clipping algorithm.

```
procedure M_Clip (ref pnt2ds inpts; integer numin);
                  ref pnt2ds outpts; ref integer numout)
{ inpts[0..numin–1] defines the input polygon with inpts[numin–1] = inpts[0] .
  The clipped polygon is returned in outpts[0..numout–1]. It is assumed that
  the array outpts is big enough. }
begin
    integer i;

    numout := 0;

    { Compute status of first point. If it is visible, it is stored in outpts array. }
    if CS_StartClip () > 0 then
        begin
            outpts[numout] := startPt;
            Inc (numout);
        end;

    { Now the rest of the points }
    for i:=1 to numin-1 do
        begin
            cflag := CS_EndClip (i);

            startC0 := endC; { endC may get changed }
            if SegMetWindow (cflag)
                then
                    begin
                        if Clipped (cflag) then
                            begin
                                outpts[numout] := startPt;
                                Inc (numout);
                            end;
                        outpts[numout] := endPt;
                        Inc (numout);
                    end
                else if TwoBitCase (endC)
                    then TwoBitEndPoint ()
                    else OneBitEndPoint ();

            { The basic turning point test }
            if TwoBitCase (endC) then
                begin
                    outpts[numout] := clipRegion[Cra[endC and not (TWOBITS)]];
                    Inc (numout);
                end;

            startPt := inpts[i];
        end;
```

Algorithm 3.3.4.2. *Continued*

```
        { Now close the output }
        if  numout > 0  then
            begin
                outpts[numout] := outpts[0];
                Inc (numout);
            end
end;  { M_Clip }


boolean function SegMetWindow (integer cflag)
return ( (cflag and SEGM) $\neq$ 0 );

boolean function Clipped (integer cflag)
{ Actually, this function should return true only if the first point is clipped;
   otherwise we generate redundant points. }
return ( (cflag and CLIP) $\neq$ 0 );

boolean function TwoBitCase (integer cflag)
return ( (cflag and TWOBITS) $\neq$ 0 );

procedure TwoBitEndPoint ()
{ The line has been rejected and we have a 2-bit endpoint. }
if  (startC and endC and (TWOBITS – 1)) = 0  then
    begin
        { The points have no region bits in common.  We need to generate
           an extra turning point - which one is specified by Cra table. }
        if  TwoBitCase (startC)
            then  BothAreTwoBits ()              { defines aC for this case }
            else  aC := endC + Tcc[startC];      { 1-bit start point, 2-bit endpoint }

        outpts[numout] := clipRegion[Cra[aC and not (TWOBITS)]];
        Inc (numout);
    end;  { TwoBitEndPoint }

procedure BothAreTwoBits ()
{ Determines what aC should be by doing midpoint subdivision. }
begin
    boolean notdone;
    pnt2d Pt1, Pt2, aPt;

    notdone := true;
    Pt1 := startPt;
    Pt2 := endPt;
```

Algorithm 3.3.4.2. *Continued*

```
    while notdone do
        begin
            aPt.x := (Pt1.x + Pt2.x)/2.0;
            aPt.y := (Pt1.y + Pt2.y)/2.0;
            aC := ExtendedCsCode (aPt);
            if TwoBitCase (aC)
                then
                    begin
                        if aC = endC
                            then Pt2 := aPt
                            else
                                begin
                                    if aC = startC
                                        then Pt1 := aPt
                                        else notdone := false
                                end
                    end
                else
                    begin
                        if (aC and endC) ≠ 0
                            then aC := endC + Tcc[startC and not (TWOBITS)]
                            else  aC := startC + Tcc[endC and not (TWOBITS)];
                        notdone := false;
                    end
        end
end; { BothAreTwoBits }

procedure OneBitEndPoint ()
{ The line has been rejected and we have a 1-bit endpoint. }
if TwoBitCase (startC)
    then
        begin
            if (startC and endC) = 0 then
                endC := startC + Tcc[endC];
        end
    else
        begin
            endC := endC or startC;
            if Tcc[endC] = 1 then endC := endC or TWOBITS;
        end; { OneBitEndPoint }
```

Algorithm 3.3.4.2. *Continued*

```
integer function ExtendedCsCode (pnt2d p)
{ The Maillot extension of the Cohen-Sutherland encoding of points }
begin
   if p.x < clipRegion[0].x then
      begin
         if p.y > clipRegion[3].y then return (6 or TWOBITS);
         if p.y < clipRegion[0].y then return (12 or TWOBITS);
         return (4);
      end;
   if p.x > clipRegion[3].x then
      begin
         if p.y > clipRegion[3].y then return (3 or TWOBITS);
         if p.y < clipRegion[0].y then return (9 or TWOBITS);
         return (1);
      end;
   if p.y > clipRegion[3].y then return (2);
   if p.y < clipRegion[0].y then return (8);
   return (0);
end;
```

Algorithm 3.3.4.3. An extended clipping code function.

used. This function adds the extra bit (TWOBITS), which we talked about. Within the Cohen-Sutherland clipping the extra bit should be ignored.

Two functions in the Maillot algorithm, Algorithm 3.3.4.2, make use of Cohen-Sutherland clipping:

CS_StartClip: This function defines global variables

startPt – the first point of the input polygon
startC – the extended Cohen-Sutherland code for startPt

and returns values SEGM or NOSEGM, where

SEGM means that the point is inside the clipping region
NOSEGM means that the point is outside the clipping region

CS_EndClip (**integer** i): This function uses the global variables startC0 and startPt, clips the segment [startPt,ith point of polygon], and defines the global variables

startC, endC – the extended Cohen-Sutherland code for the start and end-point, respectively
startPt, endPt – these are the original endpoints if there was no clipping or are the clipped points otherwise

The function returns values SEGM, SEGM **or** CLIP, or NOSEGM, where

SEGM means that the segment is inside the clipping region
CLIP means that the segment is only partly inside the clipping region
NOSEGM means that the segment is outside the clipping region

In conclusion, Maillot claims the following for his algorithm and implementation:

(1) It is up to eight times faster than the Sutherland-Hodgman algorithm and up to three times faster than the Liang-Barsky algorithm.
(2) It can be implemented using only integer arithmetic.
(3) It would be easy to modify so as to reduce the number of degenerate edges.

With regard to point (3), recall again that the Sutherland-Hodgman and Liang-Barsky algorithms also produce degenerate edges sometimes. The Weiler and Vatti algorithm are best in this respect.

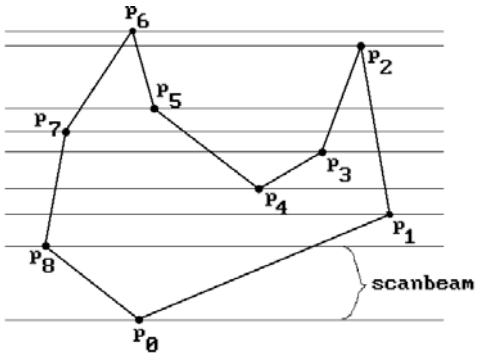
3.3.5 Vatti Polygon Clipping

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 algorithms 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 halfplanes and therefore only applies to sets that are the intersection of halfplanes, 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 $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_8$ shown in Figure 3.18(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



(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 3.18. Polygon bounds.

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 3.18(b) shows the LML for the polygon in Figure 3.18(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.

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 3.18(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 with the two bounds having 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 that 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 3.18(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 3.18(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 *scan beams*. A scan beam 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 3.18(a) shows the scan beams and the scan lines that determine them for that particular polygon. The scan beams are the regions between the horizontal lines. It should be noted here that the scan lines that determine the scan beams are not computed all at once but incrementally in a bottom-up fashion. The information about the scan beams is kept in a *scan beam list* (SBL), which is an ordered list ordered by the y-coordinates of all the scan lines that define the scan beams. 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 scan beam.

When we begin processing a scan beam, the first thing we do is to check the LML to see if any of its bound pairs start at the bottom of the scan beam. 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 scan beam. These intersections affect the output polygons and are dealt with separately first. Finally, we process the edges on the AEL. Algorithm 3.3.5.1 summarizes this overview of the Vatti algorithm.

To understand the algorithm a little better we look at some more of its details. The interested reader can find a much more thorough discussion with abstract programs and explicit data structures in the document VattiClip on the accompanying CD. The UpdateLMLandSBL procedure in Algorithm 3.3.5.1 finds the bounds of a polygon, adds them to LML, and also updates SBL. Finding a bound involves finding the edges that make them up and initializing their data structure that maintains the information that we need as we go along. For example, we keep track of the x-coordinate of their intersection with the bottom of the current scan beam. We call this the *x-value* of the edge. The edges of the AEL are ordered by these values with ties being broken using their slope. We also record the *kind* of an edge which refers to whether it belongs to the clip or subject polygon. Two edges are called *like* edges if they are of the same kind and *unlike* edges otherwise. The partial polygons that are built and that, in the end, may become the polygons that make up the clipped polygon are called the *adjacent polygons* of their edges.

Because horizontal edges complicate matters, in order to make dealing with horizontal edges easier, one assumes 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).
(3) No right bound has a top horizontal edge (any such edges are shifted to the left bound).

```
{ 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); { subject and clip specify a subject }
    UpdateLMLandSBL (clipP, clip);       { or clip polygon, respectively }

    Initialize PL, AEL to empty;

    yb := PopSBL ();                     { bottom of current scan beam }
    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 3.3.5.1. The Vatti polygon-clipping algorithm.

We 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 following 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 3.3.5.1, the first thing that happens in the main loop is to check for new bound pairs that start at the bottom of the current scan beams. If any such pairs exist, then we have a case of two bounds starting at a vertex **p** that 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. We determine if the edges are **contributing** by a parity test and flag them accordingly. 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 associate this polygon to both edges. 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. The bound's other edges will be of the same type as that one.

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 separate the discussion into two cases. We shall discuss the case where there are no horizontal edges first.

If an edge does not end at the top of the current scan beam, then we simply update its x-value to the x-coordinate of the intersection of the edge with the scan line at the top of the scan beam. If an edge **does** end at the top of the scan beam, 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.

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 on the AEL by its successor edge which inherits the adjacent polygon and left/right flag of the old edge.

If the vertex **p** is a local maximum of the original clip or subject polygons, then a pair of edges from two bounds meet in the point **p**. If **p** is a contributing vertex, then the two edges may belong either to the same or different (partial) polygons. If they have the same adjacent polygons, 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 top edges for **P** and $\mathbf{f}_1$ and $\mathbf{f}_2$, the top 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 3.19. Figures 3.19(a) and (c) show specific examples and (b) and (d) generic cases. If $\mathbf{e}_1$ is a left edge of **P** (Figures 3.19(a) and (b)), then we append the vertices of **Q** to the begin-

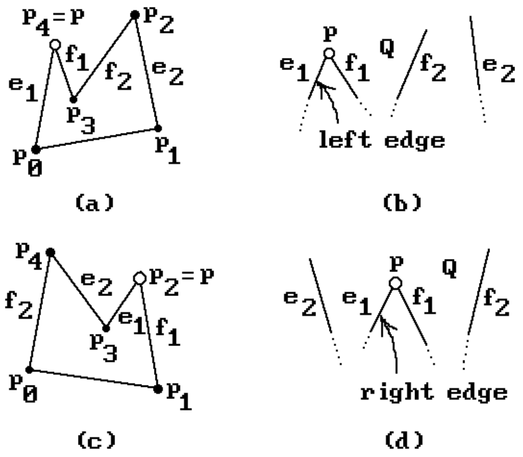


Figure 3.19. Merging polygons.

ning of the vertex list of **P**. If $\mathbf{e}_1$ is a right edge of **P** (Figures 3.19(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** and the adjacent polygon of $\mathbf{f}_2$ will become **P**. If $\mathbf{e}_1$ was a right edge, then $\mathbf{e}_2$ will be contributing to **Q**. Therefore, the adjacent polygon of $\mathbf{e}_2$ will become **Q**.

When we find a local maximum we know two top edges right away, but if these have different adjacent polygons, then we need to 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 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 scan beam. 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 **classification rules** are shown in Table 3.3.5.1 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

Table 3.3.5.1 Rules that Classify the Intersection Point Between Edges

Unlike edges:	Like edges:
(1) (LC ∩ LS) **or** (LS ∩ LC) → LI	(5) (LC ∩ RC) **or** (RC ∩ LC) → LI **and** RI
(2) (RC ∩ RS) **or** (RS ∩ RC) → RI	(6) (LS ∩ RS) **or** (RS ∩ LS) → LI **and** RI
(3) (LS ∩ RC) **or** (LC ∩ RS) → MX	
(4) (RS ∩ LC) **or** (RC ∩ LS) → MN	

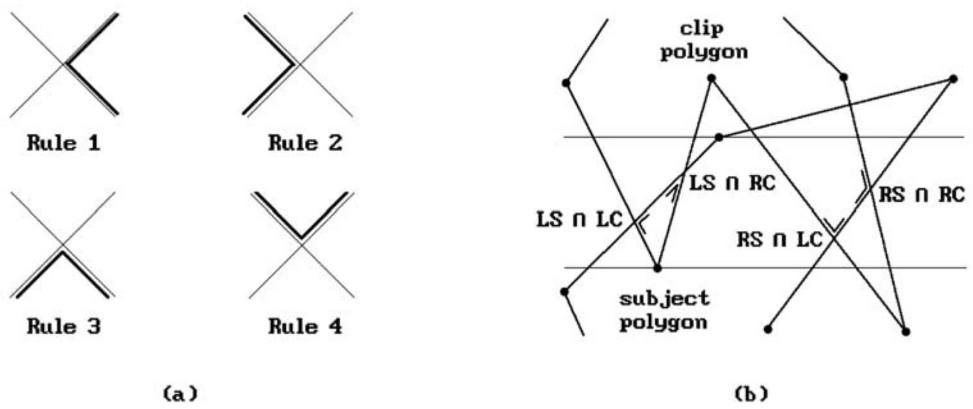


Figure 3.20. Intersection rules.

a left intermediate vertex. Rules 1–4 are shown graphically in Figure 3.20(a). Figure 3.20(b) shows an example of how the rules apply to some real polygon intersections.

As one moves from scan beam to scan beam, one updates the x-values of all the edges (unless they end at the top of the scan beam). Although the AEL is sorted as one enters a new scan beam, if any intersections are found in a scan beam, the AEL will no longer be sorted after the x-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 *sorted edge list* (SEL) and an *intersection list* (IL) to identify and store all the intersections in the current scan beam. The SEL is ordered by the x-coordinate of the intersection of the edge with the **top** of the scan beam similarly to the way that the AEL is ordered by the intersection values with the bottom of the scan beams. 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 scan beam.

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 that 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 scan beam. 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 that tests whether or not an edge ends in the current scan beam. If it does, we also need to look for horizontal edges at the top of the current scan beam and the type classification of a vertex should then distinguish between a local maximum, left intermediate vertex, or right intermediate vertex cases. 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 to which they are the successor, we would have already scanned past the AEL edges that intersected them. To avoid this, the simplest solution probably is to make an initial scan of the AEL for all such edges before one checks events at the top of the scan beam and put them into a special left-oriented horizontal edge list (LHL) ordered by the x-values of their left endpoints. Then as one scans the AEL one needs to constantly check the top x-value of an edge for whether it lies inside one of these horizontal edges.

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. 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. 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 update the trapezoid pointers of the relevant edges to the two new trapezoids. 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 improvement) and substantially faster if one does both clipping **and** filling. Because Section 14.4 will describe a special case of the trapezoid form of the Vatti algorithm for use with trimmed surfaces, we postpone any further details on how to deal with trapezoids to 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

(1) (LC ∪ LS) **or** (LS ∪ LC) → LI
(2) (RC ∪ RS) **or** (RS ∪ RC) → RI
(3) (LS ∪ RC) **or** (LC ∪ RS) → MN
(4) (RS ∪ LC) **or** (RC ∪ LS) → MX

Local minima of the subject polygon that lie outside the clip polygon and local minima of the clip polygon that 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

(1) (RC – LS) **or** (LS – RC) → LI
(2) (RS – LC) **or** (LC – RS) → RI
(3) (RS – RC) **or** (LC – LS) → MN
(4) (RC – RS) **or** (LS – LC) → MX

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

3.3.6 Greiner-Hormann Polygon Clipping

The last polygon-clipping algorithm we consider is the Greiner-Hormann algorithm ([GreH98]). It is very much like Weiler's algorithm but simpler. Like the Weiler and Vatti algorithm it handles any sort of polygons including self-intersecting ones. Furthermore, it, like Vatti's algorithm, can be modified to return the difference and union of polygons, not just their intersections.

Suppose that we want to clip the subject polygon **S** against the clip polygon **C**. What we shall do is find the part of the boundary of **S** in **C**, the part of the boundary of **C** in **S**, and then combine these two parts. See Figure 3.21. Since we allow self-intersecting polygons, one needs to be clear about when a point is considered to be inside a polygon. Greiner-Hormann use the winding number $\omega(\mathbf{p},\gamma)$ of a point **p** with respect to a parameterized curve γ. They define a point **p** to be in a polygon **P** if the winding number of the point with respect to the boundary curve of **P** is odd. (The oddness or evenness of the winding number with respect to a curve is independent of how the curve is parameterized.)

Polygons are represented by doubly-linked lists of vertices. The algorithm proceeds in three phases. One will find it helpful to compare the steps with those of the Weiler algorithm as one reads.

Phase 1. We compare each edge of the subject polygon with each edge of the clip polygon, looking for intersections. If we find one, we insert it in the appropriate place

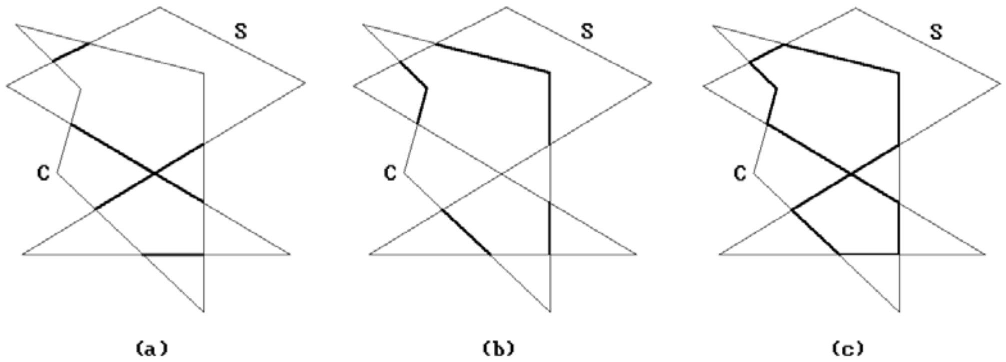


Figure 3.21. Greiner-Hormann polygon clipping.

```
vertex = record
  float          x, y;
  vertex pointer next, prev;
  boolean        intersect;
  boolean        entry;
  vertex pointer neighbor;
  float          alpha;
  vertex pointer nextPoly;
end;
```

Data 3.3.6.1. The Greiner-Hormann vertex structure.

in **both** polygons' vertex lists. If there are no intersections, then either one polygon is contained in the other or they are disjoint. These cases are checked for easily and we then exit the algorithm in this case with our answer.

Phase 2. We traverse each polygon's new vertex lists marking any intersection points as either entry or exit points. This is done by checking whether the first vertex of each polygon lies inside the other polygon or not using the winding number. The rest of the tagging as entry or exit points is then easy.

Phase 3. This stage actually creates the intersection polygons. We start at an intersection point of the subject polygon and then move along its point list either forward or backward depending on its entry-exit flag. If we are at an entry point, then we move forward, otherwise, backward. When we get to another intersection point, we move over to the other polygon's list.

The data structure used for vertices is shown in Data 3.3.6.1. In the case of an intersection point, if the **entry** field is false, then the point is an exit point. At an intersection vertex in one of the polygon's vertex lists the field **neighbor** points to the corresponding vertex in the other polygon's vertex list. The field **alpha** for an intersection point specifies the position of the intersection relative to the two endpoints of the edge

```
vertex pointer current;

while more unprocessed subject intersection points do
    begin
        current := pointer to first remaining unprocessed subject intersection point;
        NewPolygon (P);
        NewVertex (current);
        repeat
            if current→entry
                then
                    repeat
                        current := current→next;
                        NewVertex (current);
                    until current→intersect
                else
                    repeat
                        current := current→prev;
                        NewVertex (current);
                    until current→intersect
            current := current→neighbor;
        until Closed (P);
    end;
```

Algorithm 3.3.6.1. Algorithm for Greiner-Hormann's Phase 3.

containing this intersection point. Because the intersection polygon may consist of several polygons, these polygons are linked with this field. The first vertex of each intersection polygon list has its **nextPoly** field point to the first vertex of the next intersection polygon.

The basic steps for Phase 3 are shown in Algorithm 3.3.6.1. The procedure NewPolygon starts a new polygon P and NewVertex creates a new vertex for this polygon and adds it to the end of its vertex list. Figure 3.22 shows the data structure that is created for a simple example.

The algorithm uses an efficient edge intersection algorithm and handles degenerate cases of intersections by perturbing vertices slightly.

The advantage of the Greiner-Horman algorithm is that it is relatively simple and the authors claim their algorithm can be more than twice as fast as the Vatti algorithm. The reason for this is that Vatti's algorithm also checks for self-intersections which is not done here. Of course, if one knows that a polygon does not have self-intersections, then the extra work could be avoided in Vatti's algorithm also. The disadvantage of the algorithm is that one does not get any trapezoids but simply the boundary curve of the intersection. In conclusion, the Greiner-Horman algorithm is a good one if all one wants is boundaries of polygons because it is simple and yet fast.

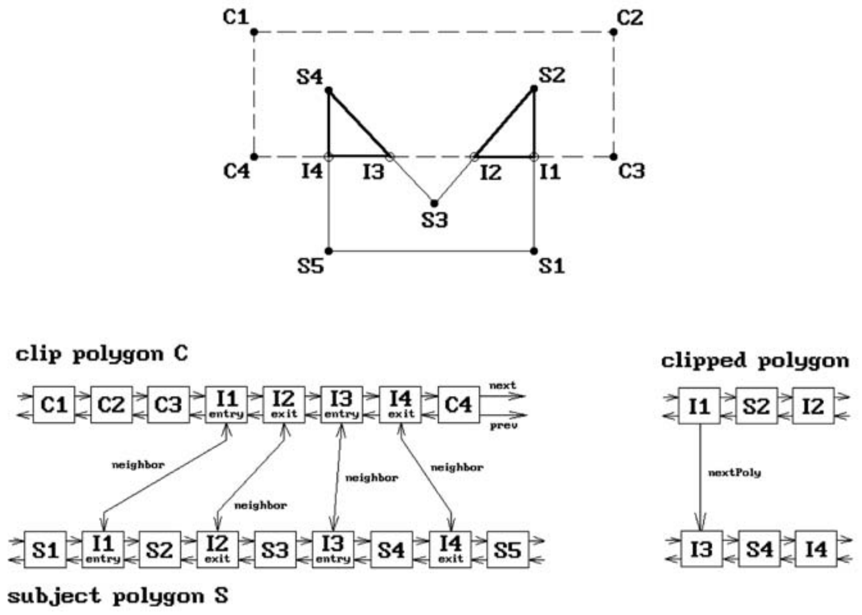


Figure 3.22. The Greiner-Hormann data structures.

3.4 Text Clipping

The topic of text generation and display is a very complex one. We shall barely scratch the surface here.

Characters can be displayed in many different styles and sizes and each such overall design style is called a *typeface* or *font*. Fonts are defined in one of several ways:

Bit-Mapped Fonts. Each character is represented by a rectangular bitmap. All the characters for a particular font are stored in a special part of the graphics memory and then mapped to the frame buffer when needed.

Vector Fonts. Each character is represented by a collection of line segments.

Outline Fonts. Each character's outline is represented by a collection of straight line segments or spline curves. This is more general than vector fonts. An attractive feature of both vector and outline fonts is that they are device independent and are easily scaled, rotated, and transformed in other ways. In either case, one has the option of scan converting them into the frame buffer on the fly or precomputing them and storing the bitmaps in memory. Defining and scan converting outline fonts gets very complicated if one wants the result to look nice and belongs to what is called *digital typography*.

Two overall strategies that are used to clip text are:

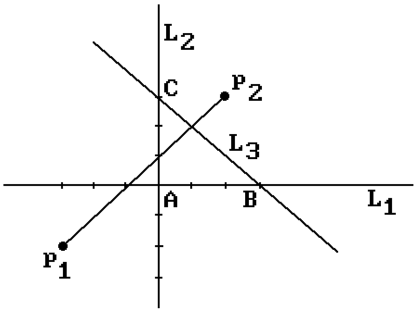


Figure 3.23. A Cyrus-Beck clipping example.

All-or-Nothing String Clipping. Here one computes the size of the rectangle that contains the string and only maps the string to the frame buffer if the rectangle fits entirely into the window in which the string is to be displayed.

All-or-Nothing Character Clipping. Here one clips on a character-by-character basis. One computes the size of the rectangle that contains a given character and only maps the character to the frame buffer if the rectangle fits entirely into the window in which it is to be displayed.

The all-or-nothing approaches are easy to implement because it is easy to check if one rectangle is inside another. The all-or-nothing character clipping approach is often quite satisfactory. A more precise way to clip is to clip on the bit level. What this means in the bit-mapped font case is that one clips the rectangular bitmap of each character against the window rectangle and displays that part which is inside. In the vector or outline font case, one would clip the curve that defines a character against the window using one of the line-clipping algorithms and then scan converts only the part of the character that lies in the window.

3.5 Exercises

Section 3.2.2.

3.2.2.1 Let $\mathbf{p}_1 = (-4,-2)$ and $\mathbf{p}_2 = (2,3)$. Let $\mathbf{A} = (0,0)$, $\mathbf{B} = (3,0)$, and $\mathbf{C} = (0,3)$. Work out the steps of the Cyrus-Beck clipping algorithm and compute the $[a_i,b_i]$s that are generated when clipping the segment $[\mathbf{p}_1,\mathbf{p}_2]$ against triangle **ABC**. See Figure 3.23. Assume that the lines $\mathbf{L}_1$, $\mathbf{L}_2$, and $\mathbf{L}_3$ are defined by equations $y = 0$, $x = 0$, and $x + y = 3$, respectively.

3.6 Programming Projects

1. Clipping (Section 3.3.5 and 3.3.6)

Implement either the Vatti or Greiner-Hormann clipping algorithm in such a way so that it handles all three set operations $\cap$, $\cup$, and $-$.