ARTICLE

An efficient new algorithm for 2-D line clipping: Its development and analysis

TINA MAN FONG NICHOLL, Western University, London, ON, Canada

D T LEE, Northwestern University, Evanston, IL, United States

ROBIN ALEXANDER NICHOLL, Western University, London, ON, Canada

Published: 01 August 1987

SIGGRAPH '87: Computer graphics and interactive techniques

Conference Sponsors:
SIGGRAPH

SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques (August 1987)
https://doi.org/10.1145/37401.37432
ISBN: 0897912276

AN EFFICIENT NEW ALGORITHM FOR 2-D LINE CLIPPING: ITS DEVELOPMENT AND ANALYSIS

Tina M. Nicholl*, D. T. Lee+ and Robin A. Nicholl*

*Department of Computer Science, The University of Western Ontario, London, CANADA N6A 5B7

and

+Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201, U.S.A.

ABSTRACT

This paper describes a new algorithm for clipping a line in two dimensions against a rectangular window. This algorithm avoids computation of intersection points which are not endpoints of the output line segment. The performance of this algorithm is shown to be consistently better than existing algorithms, including the Cohen-Sutherland and Liang-Barsky algorithms. This performance comparison is machine-independent, based on an analysis of the number of arithmetic operations and comparisons required by the different algorithms. We first present the algorithm using procedures which perform geometric transformations to exploit symmetry properties and then show how program transformation techniques may be used to eliminate the extra statements involved in performing geometric transformations.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Tools and Techniques; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems – geometrical problems and computations; I.3.3 [Computer Graphics] Picture/Image Generation – display algorithms; I.3.5 [Computer Graphics] Computational Geometry and Object Modeling – geometric algorithms, languages and systems; hierarchy and geometric transformations

General Terms: Algorithms, design, measurement, performance

Additional Key Words and Phrases: Clipping, line clipping, program transformation

I. INTRODUCTION

In the most general sense, clipping is the evaluation of the intersection between two geometrical entities. These geometrical entities may be points, line segments, rectangles, polygons, polyhedrons, curves, surfaces and so on, or assemblies of these. In this paper we will restrict ourselves to computing the intersection between a line segment and a rectilinear rectangle (which we call a window) in two dimensions.

Assume that we have a line segment with endpoints (x1,y1) and (x2,y2), and a window represented by four real numbers xleft, ytop, xright and ybottom. The window is defined as the set of all points (x,y) such that xleft $\leq$ x $\leq$ xright and ybottom $\leq$ y $\leq$ ytop. The intersection (if not empty) is a continuous portion of the line segment, and so can be represented by two endpoints. Thus we must determine if the intersection is empty, and if not, compute the coordinates of its endpoints.

II. BACKGROUND

The Cohen-Sutherland (CS) algorithm [1] appears, until recently, to be the only line-clipping algorithm known to the graphics community. (Sproull and Sutherland's midpoint subdivision algorithm [2] is designed for machines with no hardware support for multiplication and division.) The CS algorithm uses an encoding scheme to indicate the positions of the endpoints of line segments.

The Cyrus-Beck (CB) algorithm [3], though published in 1978, is not very well known to most of the graphics community. This algorithm is based on a parametric representation of the line segments. The theoretical model of this algorithm is very general. However, it is rather inefficient. To clip a line segment which is neither vertical nor horizontal and lies entirely within the window it will perform 12 additions, 16 subtractions, 20 multiplications and 4 divisions. The CS algorithm performs *no* arithmetic operations for the same line segment.

The Liang-Barsky (LB) algorithm [4], published in 1984, may be the first attempt to improve on the performance of the CS algorithm. The algorithm is a lot more efficient than the CB algorithm, even though the two use the same theoretical model. Experimental analysis was used to show that this algorithm is faster than the CS algorithm. However the approach of running the algorithms many times, with random line segments and windows, is inadequate. The results of the experiments are dependent on characteristics of the machine used, for example, whether or not a floating point accelerator is used.

A very recent technical report by Sobkow, Pospisil and Yang [5] describes an algorithm (the SPY algorithm) which also uses an encoding scheme for line segments. They also used experimental analysis to show that their algorithm is faster than both of the CS and the LB algorithms. In the experiment they showed that the LB algorithm is slower than the CS algorithm when the windows used are large. This inconsistency demonstrates further the weakness of experimental analysis.

Experimental analysis suffers from other problems too. The probabilistic model used to generate random input data may not be realistic. Even when it is realistic, the experiment provides little insight into *why* one algorithm is faster than another.

A common weakness of all the previous algorithms is the need to evaluate intersection points which are not part of the result. For example, given the line and window shown in Figure 1 both the CS and SPY algorithms will compute the intersection point I_L before rejecting the entire line segment (i.e. concluding that the intersection is empty), whereas the LB algorithm will compute parametric values of all the intersection points I_L, I_R, I_B and I_T before rejecting the line segment.

There are many trivial cases in two-dimensional line clipping, and they should be handled with as little computation as possible. For example, when both endpoints lie within the boundaries of the window the LB algorithm computes parametric values of the intersection points with all four window boundaries before deciding that the line needs no clipping. In fact, the LB algorithm always computes these four parametric values when the intersection is non-empty. This explains why the LB algorithm is slower than the CS algorithm when the window is large and there are more line segments with non-empty intersection with the window.

Both the CS and SPY algorithms use an encoding system. This approach divides the problem into a manageable number of cases and thus probably leads to a shorter program. However an encoding system can be the cause of inefficiency. To encode the location of a line segment some comparisons must be performed. Further comparisons on the encoding are then needed before the appropriate calculation of an intersection is done. If we can manage all possible cases without going through an encoding scheme then the appropriate calculations can be done immediately after the initial comparisons.

III. MACHINE-INDEPENDENT EVALUATION OF EFFICIENCY

To evaluate the efficiency of a clipping algorithm, we count up the number of times particular operations are executed in all possible cases. This is possible because clipping is a particularly simple and symmetrical problem. The operations to be counted should be generally recognised as most time-consuming and should also be indispensible in a clipping algorithm. What these operations should be depends on the level of abstraction we think about the algorithm and the assumptions we make of the computing environment.

The operations we have chosen are comparisons and the four arithmetic operations: addition, subtraction, multiplication and division. However, a clipping algorithm which relies heavily on operations other than these will be compared rather unjustly favourably in our evaluation. For example, the CS algorithm relies heavily on set operations on codes which are absent in the LB algorithm. This is indeed the most difficult hurdle in designing a machine-independent evaluation, namely, all operations are different and we are not comparing like with like.

To be able to progress from an evaluation scheme to the design of an efficient clipping algorithm, we also need to make some assumptions on the relative speeds (or costs) of these operations to be counted. The assumptions should be true in most computing environments. We have assumed that the operations comparison, addition, subtraction, multiplication and division are decreasing in speed in that order. This is true in most computing environments without a floating point accelerator. With a floating point accelerator, addition and subtraction are about the same speed with the latter slightly slower, multiplication and division are about the same speed with the latter slightly slower, and addition and subtraction are slower than multiplication and division. So, the most general assumption that applies to both kinds of environments is that subtraction is slower than addition and division is slower than multiplication. Even though we designed our algorithm with the initial assumption in mind, our algorithm is shown to be faster than the LB algorithm in all cases with the most general assumption, and faster than the CS algorithm in all cases averaged over symmetry with the most general assumption if the difference in speed between division and multiplication is greater than the difference between subtraction and addition.

IV. DEFINITIONS

The lines x = xleft, x = xright, y = ybottom and y = ytop are called the left, right, bottom and top boundaries of the window respectively. A point, a line segment or a portion of a line segment which lies entirely inside the window is *visible*. A point, a line segment or a portion of a line segment which lies entirely outside the window is *invisible*. A line segment which lies partly inside the window and partly outside is *partially visible*. If a line segment is invisible then no part of the line segment appears in the output, and the line segment is said to be *rejected* by the clipping algorithm. The boundaries of the window divide the 2-dimensional Cartesian plane into 9 *regions*. We call regions which are bounded by only 2 boundaries the *corner regions*, and the regions which are bounded by 3 boundaries, the *edge regions*. (See Figure 2)

V. A HIERARCHICAL ANALYSIS OF THE PROBLEM

Given a line segment with endpoints P1=(x1,y1) and P2=(x2,y2), P1 is located in the window, an edge region or a corner region. In each of these cases, P2 may be at any point on the 2-dimensional Cartesian plane. We can divide all possible positions of P2 (i.e. the whole plane) into regions each of which corresponds to intersection points at the same boundaries (or none) to be output. Figures 3 to 5 show the subdivisions. In those figures, if P2 lies in a region filled with character L, R, B or T, the line P1P2 will intersect the left, right, bottom or top boundary respectively. Combinations of two characters indicate the need for two intersection points. Unfilled regions require calculation of no intersection points. These three figures are sufficient to represent all cases, because all other cases are the same as one of them up to a rotation of a multiple of 90° about the origin or a reflection about the line x = −y, or a combination of the two.

This analysis leads naturally to an algorithm. Given the input P1 and P2, characterize the location of P1 among the 9 regions. According to that, characterize the location of P2 among the appropriate subdivisions. Then, calculate the intersection points according to the characterization. In this way, only the intersection points required for output are calculated.

VI. DESIGN OF THE ALGORITHM

1. Characterization of the endpoints

Characterization of P1 is straightforward. Characterization of P2 could involve a lot of arithmetic operations, because not all boundaries of the subdivision are vertical or horizontal. Observe that all oblique parts of the subdivision boundaries are formed by a line joining P1 to a corner of the window. For example, consider the line joining P1 to the left-top corner in Figure 4. P2 is above this line if and only if

(ytop-y1) * (x2-x1) < (xleft-x1) * (y2-y1).

Note that the arithmetic operations involved in making this comparison must be performed if intersection points at the top and left boundaries are to be calculated, so the effort involved in this comparison can be useful.

2. Geometric Transformations

Since P1 may be in any one of the 9 regions, each of which corresponds to a different subdivision, a different procedure can be written to handle each case. However, this can be very error-prone because the programmer will be writing similar but not identical code many times. A more satisfactory way is to exploit the symmetry of the problem as much as possible so that all cases symmetrical to each other are transformed geometrically to the same case and are handled by only one piece of code. This approach raises the fear that geometric transformations may involve a lot of arithmetic operations. However, a closer examination of the required geometric transformations revealed that the only operations involved are unary minus and assignment. A complete list of the required geometric transformations and their Pascal implementations are as follows:

Rotation of 90° clockwise about the origin:

```
procedure rotate90c (var x,y : real);
  var t: real; begin t := x; x := y ; y := -t end;
```

Rotation of 180° clockwise about the origin:

```
procedure rotate180c (var x,y : real);
begin x := -x; y := -y end;
```

Rotation of 270° clockwise about the origin:

```
procedure rotate270c (var x,y : real);
  var t: real; begin t := x; x := -y; y := t end;
```

Reflection about the line x = −y :

```
procedure reflectxminusy (var x,y : real);
  var t: real; begin t := x; x := -y; y := -t end;
```

Reflection about the x-axis:

```
procedure reflectxaxis (var x,y: real);
begin y := -y end;
```

By employing these simple transformations, the number of different cases becomes manageable and we can optimize each of them with confidence. However, we are not limiting ourselves from bringing the algorithm to its most efficient form – a form that is free from these geometrical transformations and procedure calls. As will be explained in a later section, a mechanical software transformation can be applied to the code to bring it to such a form.

VII. DETAILS OF THE ALGORITHM

P1 can be beyond the left boundary, beyond the right boundary or between the two boundaries. The first two cases are symmetrical up to a rotation of 180° about the origin, and can be handled together. A Pascal implementation of the main procedure is:

```
procedure clip
  (xleft, ytop, xright, ybottom,
   x1,y1,x2,y2: real);
 var display : boolean;
begin
 if x1 < xleft then
   leftcolumn (xleft, ytop, xright, ybottom,
          x1, y1, x2, y2, display)
 else if x1 > xright then begin
   rotate180c (x1, y1); rotate180c (x2, y2);
   leftcolumn (-xright, -ybottom, -xleft, -ytop,
          x1, y1, x2, y2, display);
   rotate180c (x1, y1); rotate180c (x2, y2);
 end else
   centrecolumn (xleft, ytop, xright, ybottom,
          x1, y1, x2, y2, display);

 if display then
   {display the visible part of the line segment,
    which is now between the current values of
    (x1,y1) and (x2,y2)}
end { clip } ;
```

Now we need consider only the cases where P1 is beyond the left boundary or between the left and right boundaries.

1. P1 is beyond the left boundary

Now, it is pointless to further characterize P1, if P2 is also beyond the left boundary. So, we should check that before proceeding on. P1 can be beyond the top boundary, beyond the bottom boundary or between the two boundaries. Again, the first two cases are symmetrical up to a certain transformation, and can be handled together. The transformation should not only map the left-bottom corner region to the left-top corner region, but should also preserve the fact that P2 is not beyond the left boundary. A reflection about the x-axis satisfies the criteria. A Pascal implementation of the procedure is:

```
 procedure leftcolumn
   (xleft, ytop, xright, ybottom: real;
    var x1,y1,x2,y2: real; var display: boolean);
 begin
   if x2 < xleft then display := false
   else if y1 > ytop then
     topleftcorner (xleft, ytop, xright, ybottom,
            x1, y1, x2, y2, display)
   else if y1 < ybottom then begin
     reflectxaxis (x1, y1); reflectxaxis (x2, y2);
     topleftcorner (xleft, -ybottom, xright, -ytop,
            x1, y1, x2, y2, display);
     reflectxaxis (x1, y1); reflectxaxis (x2, y2)
   end else leftedge (xleft, ytop, xright, ybottom,
            x1, y1, x2, y2, display)
 end { leftcolumn } ;
```

1.1. P1 is in the left-top corner region and P2 is not beyond the left boundary

Now, it is pointless to proceed if P2 is beyond the top boundary. We check that first, and then start to characterize P2 among the subdivisions in Figure 5. To reduce the number of different cases by half (approximately), we compare P2 against the line joining P1 and the left-top corner. The case in which P2 is on one side of this line is symmetrical to the case in which it is on the other side. So, we only need to handle one of the two cases. Notice that up to this point, we still do not know whether the subdivisions are as in Figure 5 or its reflection about the line x = -y. If the line segment P1P2 is partially visible then this comparison is unavoidable. A Pascal implementation of the procedure is:

```
 procedure topleftcorner
   (xleft, ytop, xright, ybottom: real;
    var x1,y1,x2,y2: real; var display: boolean);
   var relx2, rely2, topproduct, leftproduct: real;
 begin
   if y2 > ytop then display := false
   else begin
    relx2 := x2 - x1; rely2 := y2 - y1;
    topproduct := (ytop - y1) * relx2 ;
    leftproduct := (xleft - x1) * rely2;
    if topproduct > leftproduct then
      lefibottomregion
        (xleft, ytop, xright, ybottom,
         x1, y1, x2, y2, display,
         relx2, rely2, leftproduct)
    else begin
      reflectxminusy (x1, y1);
      reflectxminusy (x2, y2);
      leftbottomregion
        (-ytop, -xleft, -ybottom, -xright,
         x1, y1, x2, y2, display,
         -rely2, -relx2, topproduct);
      reflectxminusy (x1, y1);
      reflectxminusy (x2, y2)
    end
   end
 end { topleftcorner } ;
```

1.1.1. P1 is in the left-top corner, P2 is not beyond the left boundary, and P2 is to the right of the vector from P1 to the left-top corner

If P2 is above the bottom boundary, then no more oblique boundaries need to be compared. The intersection point(s) that should be calculated are as shown in Figure 5. If P2 is below the bottom boundary, then there are 3 possibilities. The boundary formed by the line joining P1 to the left-bottom corner has to be compared no matter which side of the right boundary P2 happens to be on. So, we do this comparison first. Now, if P2 is to the left of the right boundary, we have no more oblique boundaries to be compared. Otherwise, one more comparison with an oblique boundary will decide which subdivision P2 is in. A Pascal implementation is as follows:

```
procedure leftbottomregion
  (xleft, ytop, xright, ybottom: real;
   var x1,y1,x2,y2: real; var display: boolean;
   relx2, rely2, leftproduct : real);
  var bottomproduct, rightproduct : real;
begin
  if y2 >= ybottom then begin
    if x2 > xright then begin
      y2 := y1 + (xright - x1) * rely2/relx2;
      x2 := xright
    end;
    y1 := y1 + leftproduct/relx2; x1 := xleft;
    display := true
  end else begin
    bottomproduct := (ybottom - y1) * relx2;
    if bottomproduct > leftproduct then
      display := false
    else begin
      if x2 > xright then begin
        rightproduct := (xright - x1) * rely2;
        if bottomproduct > rightproduct then begin
          x2 := x1 + bottomproduct/rely2;
          y2 := ybottom;
        end else begin
          y2 := y1 + rightproduct/relx2;
          x2 := xright;
        end;
      end else begin
        x2 := x1 + bottomproduct/rely2;
        y2 ;= ybottom;
      end;
      y1 := y1 + leftproduct/relx2; x1 := xleft;
      display := true
    end
  end
end { leftbottomregion } ;
```

1.2. P1 is in the left edge region

If P2 is beyond the left boundary, then the line segment should be rejected. Otherwise, P2 can be above the top boundary, below the bottom boundary, or between the two boundaries. The first two cases are symmetrical to each other and should be handled together. The last case is quite obvious from Figure

4. A Pascal implementation is as follows:

```
procedure leftedge
  (xleft, ytop, xright, ybottom : real;
   var x1,y1,x2,y2: real; var display: boolean);
  var relx2, rely2: real;
begin
  if x2 < xleft then display := false
  else if y2 < ybottom then
    p2bottom (xleft, ytop, xright, ybottom,
          x1, y1, x2, y2, display)
  else if y2 > ytop then begin
    reflectxaxis (x1, y1); reflectxaxis (x2, y2);
    p2bottom (xleft, -ybottom, xright, -ytop,
          x1, y1, x2, y2, display);
    reflectxaxis (x1, y1); reflectxaxis (x2, y2)
  end else begin
    relx2 := x2 - x1; rely2 := y2 - y1;
    if x2 > xright then begin
      y2 := y1 + rely2 * (xright - x1)/relx2;
      x2 := xright
    end;
    y1 := y1 + rely2 * (xleft - x1)/relx2 ;
    x1 := xleft;
    display := true
  end
end { leftedge } ;
```

1.2.1. P1 is in the left edge region, P2 is not beyond the left boundary and P2 is beyond the bottom boundary

Comparing with the boundary joining P1 to the left-bottom corner cannot be avoided by comparing with any vertical or horizontal boundaries first. We, therefore, perform the comparison first before comparing with the right boundary which may save us from further comparisons. A Pascal implementation is as follows:

```
procedure p2bottom
  (xleft, ytop, xright, ybottom :real;
   var x1,y1,x2,y2: real; var display: boolean);
  var leftproduct, bottomproduct, rightproduct,
      relx2, rely2: real;
begin
  relx2 := x2 - x1; rely2 := y2 - y1;
  leftproduct := (xleft - x1) * rely2;
  bottomproduct := (ybottom - y1) * relx2;
  if bottomproduct > leftproduct then
    display := false
  else begin
    if x2 <= xright then begin
      x2 := x1 + bottomproduct/rely2;
      y2 := ybottom;
    end else begin
      rightproduct := (xright - x1) * rely2;
      if bottomproduct > rightproduct then begin
```

```
        x2 := x1 + bottomproduct/rely2;
        y2 := ybottom;
      end else begin
        y2 := y1 + rightproduct/relx2;
        x2 := xright;
      end;
    end;
    y1 := y1 + leftproduct/relx2; x1 := xleft;
    display := true
  end
end { p2bottom } ;
```

2. P1 is between the left and right boundaries

If P1 is above the top boundary or below the bottom boundary, the case is symmetrical to that if P1 is in the left edge region, and we will use the same procedure. A Pascal implementation is as follows:

```
procedure centrecolumn
  (xleft, ytop, xright, ybottom: real;
   var x1,y1,x2,y2: real; var display: boolean);
begin
  if y1 > ytop then begin
    rotate270c (x1, y1); rotate270c (x2, y2);
    leftedge (-ytop, xright, -ybottom, xleft,
          x1, y1, x2, y2, display);
    rotate90c (x1, y1); rotate90c (x2, y2)
  end else if y1 < ybottom then begin
    rotate90c (x1, y1); rotate90c (x2, y2);
    leftedge (ybottom, -xleft, ytop, -xright,
          x1, y1, x2, y2, display);
    rotate270c (x1, y1); rotate270c (x2, y2)
  end else
    inside (xleft, ytop, xright, ybottom,
        x1, y1, x2, y2, display)
end { centrecolumn } ;
```

2.1. P1 is in the window

If P2 is in an edge region then the intersection point to be calculated is obvious from Figure 3. If P2 is in a corner region then comparison with one oblique boundary is necessary before the appropriate intersection points are calculated. Again, symmetry is used to reduce the number of different cases. A Pascal implementation is as follows:

```
  procedure inside
    (xleft, ytop, xright, ybottom: real;
     var x1,y1,x2,y2: real; var display: boolean);
    procedure p2left
      (xleft, ytop, xright, ybottom: real;
       var x1,y1,x2,y2: real);
      procedure p2lefttop
        (xleft, ytop, xright, ybottom: real;
         var x1,y1,x2,y2: real);
        var relx2, rely2,
            leftproduct, topproduct: real;
      begin relx2 := x2 - x1; rely2 := y2 - y1;
        leftproduct := rely2 * (xleft - x1);
        topproduct := relx2 * (ytop - y1);
        if topproduct > leftproduct then begin
          x2 := x1 + topproduct / rely2; y2 := ytop
        end else begin
          y2 := y1 + leftproduct / relx2; x2 := xleft
        end
      end { p2lefttop } ;
    begin
      if y2 > ytop then
        p2lefttop (xleft, ytop, xright, ybottom,
              x1, y1, x2, y2)
      else if y2 < ybottom then begin
        rotate90c (x1, y1); rotate90c (x2,y2);
        p2lefttop (ybottom, -xleft, ytop, -xright,
              x1, y1, x2, y2);
        rotate270c (x1, y1); rotate270c (x2,y2)
      end else begin
        y2 := y1 + (y2-y1) * (xleft-x1)/(x2-x1);
        x2 := xleft
      end
    end { p2left } ;
  begin
    display := true;
    if x2 < xleft then
      p2left (xleft, ytop, xright, ybottom,
          x1, y1, x2, y2)
    else if x2 > xright then begin
      rotate180c (x1, y1); rotate180c (x2, y2);
      p2left (-xright, -ybottom, -xleft, -ytop,
          x1, y1, x2, y2);
      rotate180c (x1, y1); rotate180c (x2, y2)
    end else if y2 > ytop then begin
      x2 := x1 + (x2 -x1) * (ytop - y1)/(y2 - y1);
      y2 := ytop
    end else if y2 < ybottom then begin
      x2 := x1 + (x2-x1) * (ybottom-y1)/(y2-y1);
      y2 := ybottom
    end
    {else P2 is inside,
     just display, no need to clip}
  end { inside } ;
```

Now, we have handled all possible cases.

VIII. ANALYSIS OF THE ALGORITHM

To analyse the efficiency of the algorithm, we insert extra variables in our Pascal implementation to count up the number of comparisons, additions, subtractions, multiplications and divisions in all the different cases. For the sake of comparison, we do the same to the CS algorithm and the LB algorithm. To

normalize the effect of the different order in which the window boundaries are considered in the algorithms, we average the result over symmetry. For the cases in which P1 is in the window or in an edge region, we average over the 4 rotations of a multiple of 90°. For the case in which P1 is in a corner region, we average over the 4 rotations of a multiple of 90° and their reflections about the line x = -y. The results are displayed in Figures 6 to 8. From the analysis, we can draw the following conclusions (and these conclusions are machine independent) :

1. Our algorithm has the fewest number of divisions, equal to the number of intersection points for output.

2. Our algorithm has fewest comparisons, in most cases about 1/3 of the CS algorithm and about 1/2 of the LB algorithm.

3. If we assume only that (i) subtraction is slower than addition, (ii) division is slower than multiplication and (iii) the difference in speed between subtraction and addition is smaller than that between division and multiplication, then our algorithm is the most efficient.

4. The LB algorithm requires the largest number of multiplications + divisions in all cases, and the largest number of arithmetic operations in most cases.

5. The CS algorithm requires the largest number of comparisons in most cases. (To allow for possible *short-circuit evaluation* we treat the while condition as 2 comparisons instead of 3.)

6. The LB algorithm is the slowest in most trivial reject cases.

7. When the window is large, the dominating case is when P2 is in the window in Figure 6. In this case, the CS algorithm does very few set operations, and so the LB algorithm is the slowest.

8. When the window is small, the dominating cases are the four corner regions of Figure 8. If P1 and P2 are in corner regions at opposite ends of a diagonal of the window, then the CS algorithm may require twice as many comparisons as the LB algorithm. In this case the CS algorithm performs many set operations, so it may be slower than the LB algorithm. A floating point accelerator makes this more likely, because the CS algorithm has more additions and subtractions than the LB algorithm in this case.

The LB algorithm clips vertical and horizontal line segments faster than oblique line segments, hence a detailed analysis of the LB algorithm has been performed. This detailed analysis does not alter the conclusions.

IX. TRANSFORMATION OF THE ALGORITHM

The algorithm presented above uses explicit geometric transformations (rotations and reflections) to exploit the symmetries present in the problem. Thus sections of the algorithm have the form:

1) transform the positions of the line and the window ;
2) clip the line to that window ;
3) transform the clipped line to its original position ;

illustrated by the following statements from the main procedure "clip":

```
rotate180c (x1, y1) ; rotate180c (x2, y2) ;
leftcolumn (-xright, -ybottom, -xleft, -ytop,
        x1, y1, x2, y2, display) ;
rotate180c (x1, y1) ; rotate180c (x2, y2) ;
```

In addition to the arithmetic operations and comparisons of procedure leftcolumn, we have additional assignments and negations in precedure rotate180c, as well as the negations (and passing) of parameters to procedure leftcolumn. We will now explain how correctness-preserving program transformations are used to eliminate these extra operations. In this way we can construct a highly efficient algorithm from a form in which it is less efficient, but easier to follow.

1. Basic transformations

Let x and y be lists of variable names, with no names in common. Let E be a list of expressions, of the same length as x. Let F(x) be a list of expressions, of the same length as x and possibly involving some of the names in x. Then:

(a) (x,y := E,y) = (x := E)
For example,
(x1,y1,x2,y2 := -x1,-y1,x2,y2) = (x1,y1 := -x1,-y1)

(b) (x := E ; x := F(x)) = (x := F(E))
For example,
(x1,y1 := -x1,-y1; x1,y1 := x1,-y1)
= (x1,y1 := -x1,-(-y1)) = (x1,y1 := -x1,y1)

(c) (x := E ; if p(x) then S1 else S2) =
(if p(E) then begin x := E ; S1 end
else begin x := E ; S2 end)
For example,
(x2,y2 := -x2,-y2; if x2 < x1 then S1 else S2)
= (if -x2 < x1 then begin x2,y2 := -x2,-y2; S1 end
else begin x2,y2 := -x2,-y2; S2 end)

(d) (x := x) = () — the empty statement

(e) (if b then S1 else S2 ; x := E) =
(if b then begin S1 ; x := E end
else begin S2 ; x := E end)
For example,

```
(if tp > lp then S1 else S2 ; x1,y1 := -y1,x1)
= (if tp > lp then begin S1 ; x1,y1 := -y1,x1 end
  else begin S2 ; x1,y1 := -y1,x1 end)
```

These transformations (identified by Hoare *et al.* in [6]) may be used to eliminate many assignments by modifying statements following the assignments, as seen particularly in transformations (b) and (c). We illustrate this process by transforming the text of the procedure p2left.

2. Example: Transformation of p2left

p2left has the form:

```
if y2 > ytop then p2lefttop ...
else if y2 < ybottom then
  (rotate90c ... ; p2lefttop ... ; rotate270c ...)
else (y2 := ... ; x2 := ...)
```

The section of text to be transformed is

```
rotate90c ... ; p2lefttop ... ; rotate270c ...
```

which may be written as

```
x1,y1,x2,y2 := y1,-x1,y2,-x2 ;
p2lefttop ... ;
x1,y1,x2,y2 := -y1,x1,-y2,x2 ;
```

which is, expanding the parameter substitution involved in the call to p2lefttop,

```
x1,y1,x2,y2 := y1,-x1,y2,-x2 ;
xleft,ytop,xright,ybottom :=
          ybottom,-xleft,ytop,-xright ;
... body of procedure p2lefttop ... ;
x1,y1,x2,y2 := -y1,x1,-y2,x2 ;
```

Using transformation rules (a) and (b) and simplifying arithmetic expressions, the first four assignments of p2lefttop, which read

```
relx2 := x2 - x1 ; rely2 := y2 - y1 ;
leftproduct := rely2 * (xleft - x1) ;
topproduct := relx2 * (ytop - y1) ;
```

will become simply

```
relx2 := y2 - y1 ; rely2 := x1 - x2 ;
leftproduct := rely2 * (ybottom - y1) ;
topproduct := relx2 * (x1 - xleft) ;
```

Similarly transformation rules (c) and (e) allow us to rewrite the if statement of p2lefttop as

```
if topproduct > leftproduct then begin
  x1,y1,x2,y2 := y1,-x1,y2,-x2 ;
  xleft,ytop,xright,ybottom :=
          ybottom,-xleft,ytop,-xright ;
  x2 := x1 + topproduct / rely2 ; y2 := ytop ;
  x1,y1,x2,y2 := -y1,x1,-y2,x2
end else begin
  x1,y1,x2,y2 := y1,-x1,y2,-x2 ;
  xleft,ytop,xright,ybottom :=
          ybottom,-xleft,ytop,-xright ;
  y2 := y1 + leftproduct / relx2 ; x2 := xleft ;
  x1,y1,x2,y2 := -y1,x1,-y2,x2
end
```

which can be further transformed by rules (a) and (b) and then simplified to

```
if topproduct > leftproduct then begin
  x2 := xleft ; y2 := y1 + topproduct / rely2
end else begin
  x2 := x1 - leftproduct / relx2 ; y2 := ybottom
end ;
```

Thus after transformation, including removal of procedure calls, the text of p2left appears as:

```
procedure p2left
  (xleft, ytop, xright, ybottom: real ;
   var x1, y1, x2, y2: real) ;
 var relx2, rely2, leftproduct, topproduct: real ;
 begin
  if y2 > ytop then begin
    ... text of p2lefttop ...
  end else if y2 < ybottom then begin
    relx2 := y2 - y1 ; rely2 := x1 - x2 ;
    leftproduct := rely2 * (ybottom - y1) ;
    topproduct := relx2 * (x1 - xleft) ;
    if topproduct > leftproduct then begin
      x2 := xleft ; y2 := y1 + topproduct / rely2
    end else begin
      x2 := x1 - leftproduct / relx2 ;
      y2 := ybottom
    end
  end else begin
    y2 := y1 + (y2-y1) * (xleft-x1) / (x2-x1) ;
    x2 := xleft
  end
 end { p2left } ;
```

3. Observations on the transformations

Performing the above transformations reduces the number of assignments and negations executed in clipping a line. However the size of the program text increases considerably, so that it is not practical to provide the complete text here. Furthermore, since considerable effort is required to apply the transformations throughout the program, the transformations should be applied by a program (and not manually). Program transformation systems do exist (for example [7]), but we have not used such software to support development of this algorithm.

X. CONCLUSIONS AND FUTURE WORK

We have developed a new 2-dimensional line clipping algorithm. Using a machine-independent analysis, we have shown that this algorithm is faster than both the Cohen-Sutherland algorithm and the Liang-Barsky algorithm. Using program transformation techniques we showed how geometric transformations on the line to be clipped could be performed by modified program text without the overhead of actually executing statements to perform the transformation. Currently, we are investigating the possibility of extending it to polygons and to the 3-dimensional case.

Acknowledgements The authors would like to acknowledge the useful discussions with colleagues and students at the University of Western Ontario, especially Tim Walsh. We are very grateful for the reviewers' suggestions, which we have incorporated, within the space allowed. Financial upport for this research was provided by the National Science and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] Newman, W. M. and R. F. Sproull. Principles of Interactive Computer Graphics, 2nd ed., McGraw-Hill, New York, 1979.

[2] Sproull, R. F. and I. E. Sutherland. A Clipping Divider, FJCC 1968, Thompson Books, Washington, D.C., pp. 765-775.

[3] Cyrus, M. and J. Beck. Generalised Two- and Three-Dimensional Clipping, Computers and Graphics, Vol. 3, No. 1, 1978, pp. 23-28.

[4] Liang, Y.-D. and B. A. Barsky. A New Concept and Method for Line Clipping, ACM Transactions on Graphics, Vol. 3, No. 1, 1984, pp. 1-22.

[5] Sobkow, M. S., Pospisil, P. and Y.-H. Yang. A Fast Two-Dimensional Line Clipping Algorithm, University of Saskatchewan Technical Report, 86-2.

[6] Hoare, C. A. R. et al. Laws of Programming: a tutorial paper, Oxford University Programming Research Group PRG-45, 1985.

[7] Arsac, J. J. Syntactic Source to Source Transforms and Program Manipulation, Communications of the ACM, Vol. 22, No. 1, 1979, pp. 43-54.

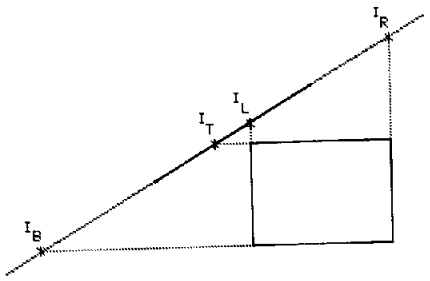

Fig. 1 Intersection points computed by previous algorithms

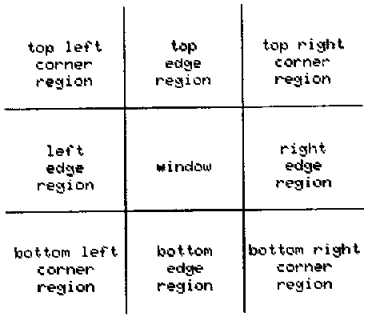

Fig. 2 Subdivision of the plane into regions

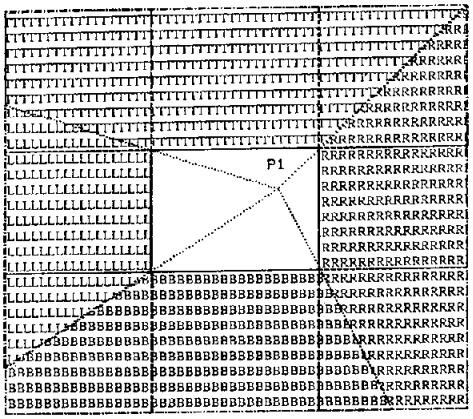

Fig. 3 Subdivision of the plane when P1 is in the window

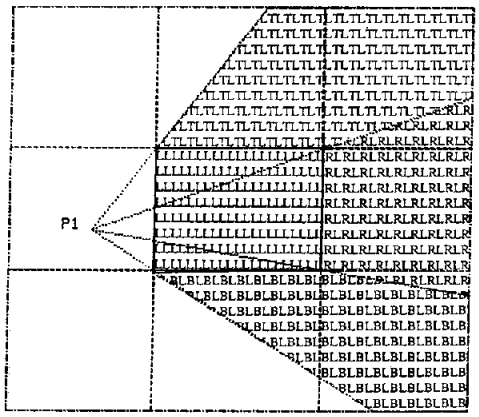

Fig. 4 Subdivision of the plane when P1 is in an edge region

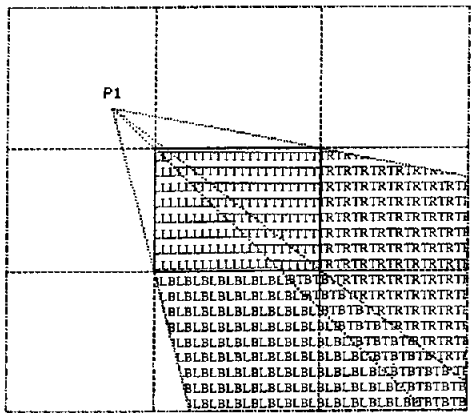

Fig. 5 Subdivision of the plane when P1 is in a corner region

CS	LB	NLN
25.5	20	8
1.5	2	1
4.5	6	4
1.5	2	2
1.5	4	1

CS	LB	NLN
21	20	7.5
1	2	1
3	6	3
1	2	1
1	4	1

COMPARISONS	CS	LB	NLN	P1
+	10	20	8	
-	0	0	0	
*	0	6	0	
/	0	0	0	
	0	4	0	

FIG.6 Analysis of the three algorithms when P1 is in the window

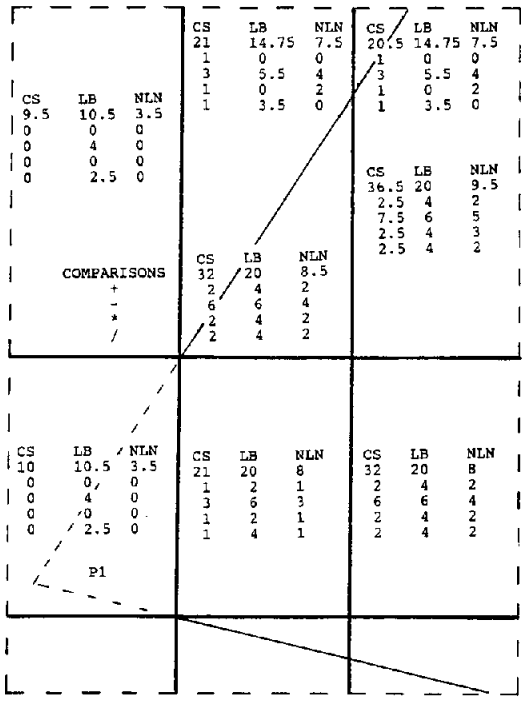

FIG.7 Analysis of the three algorithms when P1 is in an edge region

"Further subdivision is needed.
Average shown is for the largest
unbounded subregion."

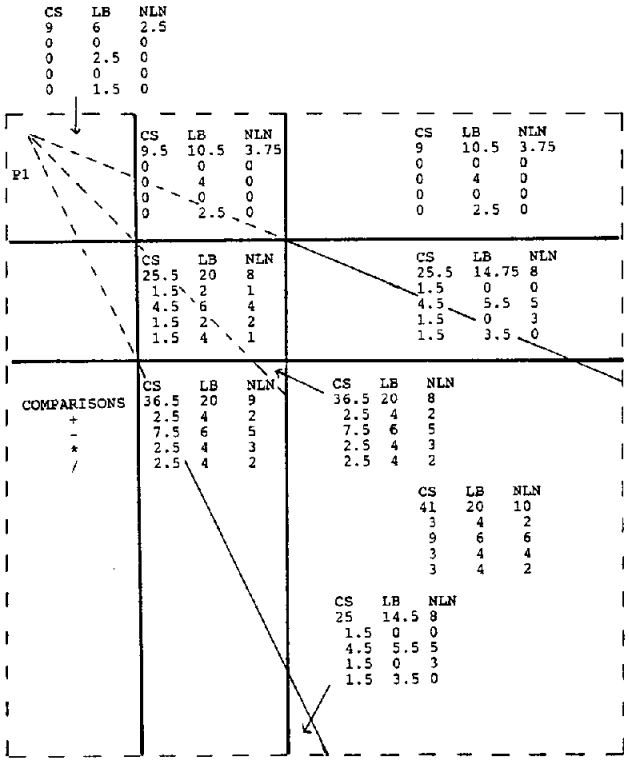

FIG.8 Analysis of the three algorithms when P1 is in a corner region