The Optimal Tree Algorithm for Line Clipping

You-Dong Liang† and Brian A. Barsky

Computer Science Division
University of California
Berkeley, California 94720
U.S.A.

Abstract

This paper develops a new algorithm for line clipping based on the concept of the optimal tree. A careful analysis results in an algorithm that classifies a given line segment in such a way that at most one procedure is invoked to clip it; furthermore, there are five such procedures that cover all cases. The result is an algorithm that is provably optimal and according to experimental tests outperforms previous algorithms. For both the two-dimensional and three-dimensional cases, and on both the Sun 3/160 and the DECStation 5000/200, the new algorithm performed uniformly faster than all the other "standard" algorithms for each of four different sizes of data space. Only the two-dimensional case is described in detail. Although in the three-dimensional case this algorithm is significantly faster than the other known algorithms, the code is huge and more complex than the new two-dimensional algorithm, and there are more special cases that need to be handled.

0. Introduction and Previous Work

Line clipping is a basic and important step in computer graphics that involves the removal of that portion of line segment that lies outside a region called the *clip window* or *visible region* or *clipping region.* This will be explained in detail in Section I.

An early algorithm for line clipping is due to Ivan Sutherland and Danny Cohen.[8, 14, 22] This algorithm uses a coding scheme to subdivide space. Then, each endpoint of a line segment is assigned the code of the sub-region in which it lies. Several properties of these codes are used to facilitate the determination of visibility of the line segment. When the use of the codes is insufficient to determine visibility, then the line segment is decomposed for further examination.

Later, the authors[10, 11] employed a parametric representation of the line segment to be clipped, used minimum and maximum calculations to determine the parametric values corresponding to the endpoints of the visible line segment, and developed several new definitions for trivial reject to speed up the approach. A similar approach was independently adopted by Cyrus and Beck[5] although that work did not develop the trivial reject

---

† Permanent address: Department of Mathematics, Zhejiang University, Hangzhou, Zhejiang People's Republic of China

cases that are so important for efficiency.

At SIGGRAPH'87, Nicholl, Lee, and Nicholl[15,16] presented a very efficient two-dimensional line clipping algorithm that is based on a case-by-case approach and which improved upon the efficiency of existing line clipping algorithms. However, this algorithm is only applicable in two dimensions. Their algorithm computes intersection points of the line with the clip boundaries only when the new point is definitely one of the endpoints of the clipped line. This approach subdivides the plane based on the location of the "first" endpoint of the line. The algorithm uses a small number of generic cases such that all others can be transformed into these by reflections or rotations The final clipped line, if any, must then be transformed back. In their more recent paper, Nicholl and Nicholl[16] showed how their program could be transformed to reduce the number of procedure calls and assignments, but the extent to which this reduction could be practically accomplished, and its effects on execution speed remain unknown. The program is already very long and if each call were replaced by in-line code, the program would be massive in size.

These three algorithms are well-known in computer graphics, and are covered in detail in.[8] However, there are lesser known variants of these three algorithms for line clipping and there are other algorithms as well.

Improvements to the Sutherland/Cohen algorithm have been developed in;[1,2,7,18] they have concentrated on speeding up the algorithm by minimizing the work involved in recomputing line segment endpoint encodings.

In,[12,13] the original Liang/Barsky parametric line clipping algorithm is improved by reducing the number of divisions in favor of more comparisons. Dorn[6] provides a different improvement by showing that the Liang/Barsky algorithm can be performed entirely in integer arithmetic.

Sobkow, Pospilsil, and Yang[21] developed an algorithm which is similar to the Nicholl/Lee/Nicholl algorithm in substance (if not in form) in the sense that they enumerate all possible relationships between a line and the clipping region, and only compute intersection points where these are required as part of the output.

In,[4] Brewer and Barsky analyze the projection transformation and thus show how clipping algorithms can be modified so that three-dimensional clipping is performed after the perspective division. Clipping after projection is simple to implement and allows three-dimensional graphics to exploit the benefits of two-dimensional clipping, including hardware clipping and complex boundaries, including curved edges. Recently, Slater and Barsky[20] introduced the idea of tracing the path of the line segment through the spatial subdivision induced by the extended clipping region boundaries, only computing intersections when the line segment enters the central cell (the clipping region). This approach avoids the large number of tests needed in, for example, the algorithms of Sobkow, Pospilsil, and Yang, or of Nicholl, Lee, and Nicholl, and this can be extended for polygon clipping. Finally, the complexity of the general clipping problem, of line segments against polygons, has been given attention in the computational geometry literature (where it is known as the Line-Polygon Classification problem, LPC), and is discussed, for example, in.[17,19,23]

In this paper, we present a new algorithm for line clipping that is completely different from our previous parametric line clipping algorithm. It is based on the concept of

the *optimal tree*. A careful analysis results in an algorithm that classifies a given line segment in such a way that at most one procedure is invoked to clip it; furthermore, there are five such procedures that cover all cases. The result is an algorithm that is provably optimal and according to experimental tests outperforms previous algorithms. For both the two-dimensional and three-dimensional cases, and on both the Sun 3/160 and the DECStation 5000/200, the new algorithm performs uniformly faster than all the other "standard" algorithms for each of four different sizes of data space. Only the two-dimensional case will be described in detail. Although in the three-dimensional case this algorithm is significantly faster than the other known algorithms, the code is huge and more complex than the new two-dimensional algorithm, and there are more special cases that need to be handled.

I. Set-up

Given a visible region, extend the boundary lines to be of infinite extent. The left, right, bottom, and top boundary lines are denoted by $x=x_L$, $x=x_R$, $y=y_B$, and $y=y_T$, respectively. These boundary lines of infinite extent subdivide the plane into a set of areas, which we call *partitions*, one of which is the visible region. There are nine such partitions using a rectangular clipping window. For convenience, we denote these partitions, or areas, by A_{ij}, i ∈ {L, C, R} and j ∈ {B, M, T}. This arrangement is illustrated in Figure 1.

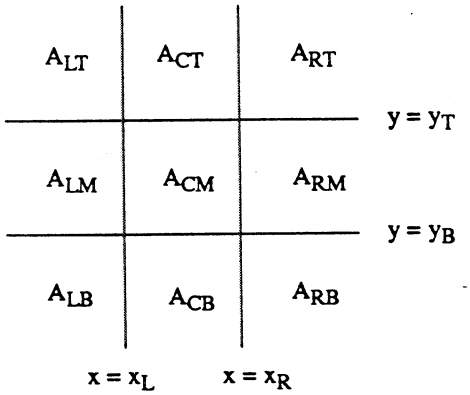


Figure 1: The boundary lines subdivide the plane into nine partitions.

The line segment to be clipped is denoted $\mathbf{P_0}\,\mathbf{P_1}$, where $\mathbf{P_0}$ and $\mathbf{P_1}$ are its endpoints with coordinates (x_0,y_0) and (x_1,y_1), respectively. We classify the line segment according to which partition each endpoint lies in. For example, a line segment might start in the bottom left partition and terminate in the top right partition; that is $\mathbf{P_0}$ lies in A_{BL} and $\mathbf{P_1}$ is in A_{RT}. We call each such classification a *partition-pair* and denote it by (A_{ij},A_{kl}). Since we distinguish explicitly between which partition $\mathbf{P_0}$ is in and which partition $\mathbf{P_1}$ is in, the total number of possible partition-pairs is simply 9×9=81.

Extending the line segment $\mathbf{P_0}\,\mathbf{P_1}$ to be a line of infinite extent, this intersects the left, right, bottom, and top boundary lines at $y=y_L^I$, $y=y_R^I$, $x=x_B^I$ and $x=x_T^I$, respectively.

II. Definition of Line Clipping

Recalling that the line segment to be clipped has endpoints denoted as $\mathbf{P_0}(x_0,y_0)$ and $\mathbf{P_1}(x_1,y_1)$, let the line segment that results from the clipping operation have endpoints denoted as $\mathbf{P_0'}(x_0',y_0')$ and $\mathbf{P_1'}(x_1',y_1')$. The line clipping operation can be regarded

as a mapping, which we denote by LCM, from $\mathbf{P}_0\,\mathbf{P}_1$ to $\mathbf{P}_0'\,\mathbf{P}_1'$. This can be written in several ways:

$$\mathbf{P}_0\,\mathbf{P}_1 \xrightarrow{LCM} \mathbf{P}_0'\,\mathbf{P}_1'$$

or

$$\mathbf{P}_0'\,\mathbf{P}_1' = LCM(\mathbf{P}_0, \mathbf{P}_1)$$

or

$$(x_0, y_0, x_1, y_1) \xrightarrow{LCM} (x_0', y_0', x_1', y_1')$$

The clipped endpoints could be expressed as follows:

$$x_0' = \begin{cases} x_L' & x_0 \le x_1 \\ x_R' & \text{otherwise} \end{cases} \quad x_1' = \begin{cases} x_R' & x_0 \le x_1 \\ x_L' & \text{otherwise} \end{cases}$$

$$y_0' = \begin{cases} y_B' & y_0 \le y_1 \\ y_T' & \text{otherwise} \end{cases} \quad y_1' = \begin{cases} y_T' & y_0 \le y_1 \\ y_B' & \text{otherwise} \end{cases}$$

where

$$x_L' = \max(\min(x_0, x_1), x_L, \min(x_B^I, x_T^I))$$

$$x_R' = \min(\max(x_0, x_1), x_R, \max(x_B^I, x_T^I))$$

$$y_B' = \max(\min(y_0, y_1), y_B, \min(y_L^I, y_R^I))$$

$$y_T' = \min(\max(y_0, y_1), y_T, \max(y_L^I, y_R^I))$$

In the case of a horizontal line, $\min(x_B^I, x_T^I)$ and $\max(x_B^I, x_T^I)$ are taken to be $-\infty$ and $+\infty$, respectively. Similarly, in the case of a vertical line, $\min(y_L^I, y_R^I)$ and $\max(y_L^I, y_R^I)$ are taken to be $-\infty$ and $+\infty$, respectively.

The line segment is partially visible if $x_L' \le x_R'$ or $y_B' \le y_T'$; otherwise, it is invisible. If the line is partially visible, then the visible segment is $\mathbf{P}_0'\,\mathbf{P}_1'$.

Although the above formula provides a theoretical solution for the line clipping problem, it is not a practical computation because it requires two multiplications, three divisions, four additions, four subtractions, and twelve comparisons for each line segment to be clipped.

III. Line Clipping Tree

One of the underlying ideas of our approach is to determine a careful ordering of comparison tests to be performed to solve the line clipping problem. This is represented by a *line clipping tree*. The nodes of the line clipping tree are comparison tests. There are two types of such comparison tests.

The first type compares an endpoint of the line segment to be clipped to a boundary line. These comparisons are:

$$x_0 > x_R \quad x_0 < x_L \quad y_0 < y_B \quad y_0 > y_T$$
$$x_1 > x_R \quad x_1 < x_L \quad y_1 < x_B \quad y_1 > y_T$$

We call these the *primary comparisons.*

The second type compares a point of intersection between the extended line and the extended boundary lines to a boundary line. These comparisons are:

$$x_B^I > x_R \quad x_T^I < x_L \quad y_L^I < x_B \quad y_L^I > x_T$$
$$x_T^I > x_R \quad x_T^I < x_L \quad y_R^I < x_B \quad y_R^I > x_T$$

We call these the *secondary comparisons.*

We will use the idea of *convex regions* which contain one or more of the partitions A_{ij}. Examples of such convex regions are shown in Figure 2. As we did for the partitions A_{ij}, we specify a pair of convex regions, each of which defines where one of the endpoints of the line segments to be clipped can be located. Such a pair of convex regions is called a *region-pair*. For the line segment to be clipped $\mathbf{P_0}\ \mathbf{P_1}$, the convex regions for the endpoints are denoted by C_0 and C_1, and the region-pair is denoted by (C_0, C_1).

In the case where both C_0 and C_1 are on the invisible side (the side that does not contain the window) of the same boundary line, then (C_0, C_1) is a *trivial reject region-pair*; otherwise, (C_0, C_1) is a *non-reject region-pair*. An example of a trivial reject region-pair is shown in Figure 3(a). In the case where both C_0 and C_1 are the visible region, then (C_0, C_1) is a *visible region-pair*, as shown in Figure 3(b). Finally, if (C_0, C_1) contains neither a trivial reject region-pair nor a visible region-pair, then (C_0, C_1) is called a *key region-pair*. An example of a key region-pair is shown in Figure 3(c).

IV. Subcases of Line Segments for a Given Partition-Pair

As mentioned earlier, we classify the line segment to be clipped according to which partition, A_{ij}, each endpoint lies in, called a partition-pair. Each partition-pair is further decomposed into subcases according to which side of the window the line enters or exits, and which corner of the window is nearest to the line missing the window.

For example, consider the partition-pair (A_{RB}, A_{LT}). For this partition-pair, there are six subcases, two of which are reject subcases. Figures 4(a) and 4(b) show the two reject subcases and four non-reject subcases, respectively.

The line segments to be clipped are not uniformly distributed among these subcases. To see this, consider a line segment of the subcase $\mathbf{E_0}\ \mathbf{E_1}$ in Figure 4(b) that intersects the window on the left boundary at a given point. The range of geometric possibilities for

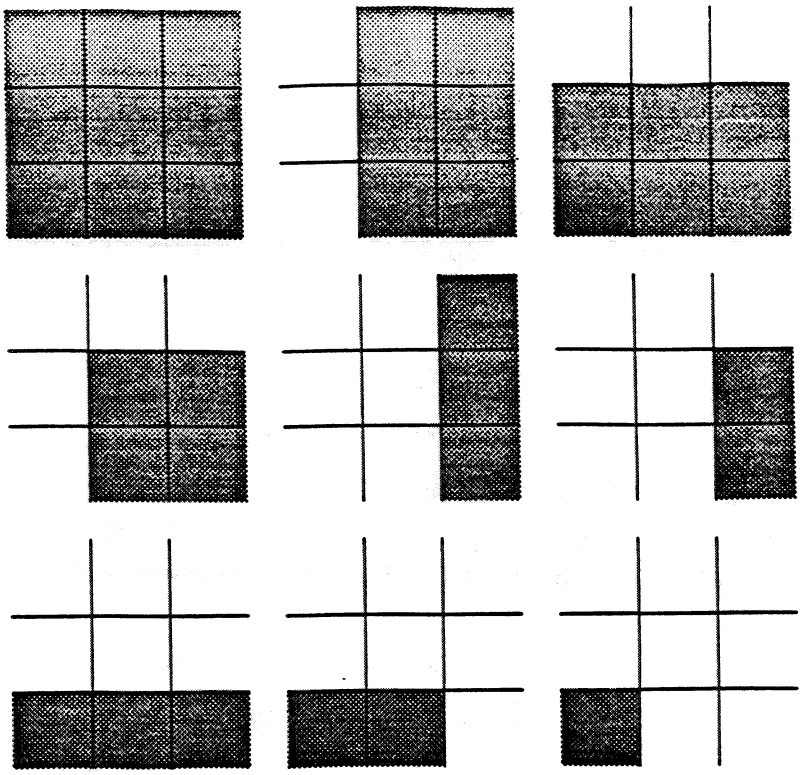


Figure 2: Examples of convex regions.

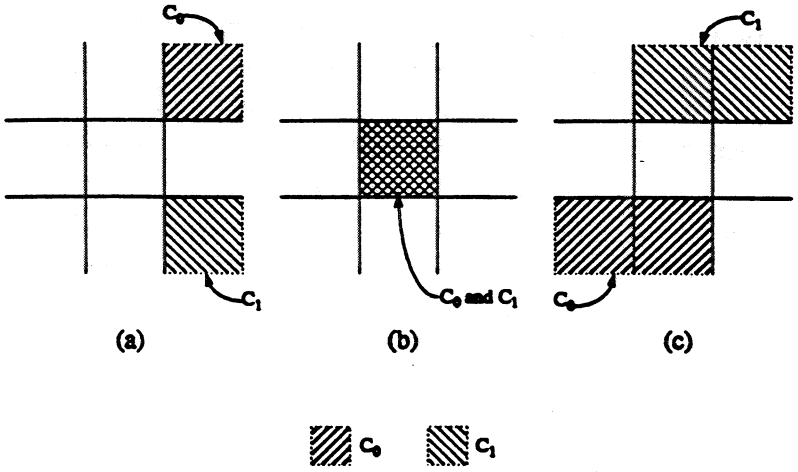


Figure 3: Region-pairs.
(a) Trivial reject region-pair.
(b) Visible region-pair.
(c) Key region-pair.

this subcase is shown in Figure 5. It is intuitively clear that of all the possible line segments that pass through the point **I**, the proportion that will be of subcase E_0 E_1 would be far less than the 1/6 that would occur if the subcases were uniformly distributed. Using this type of analysis, we can estimate the relative proportions of the occurrences of each of the subcases, for this partition-pair. These proportions will be used as weights in calculating the computational requirements.

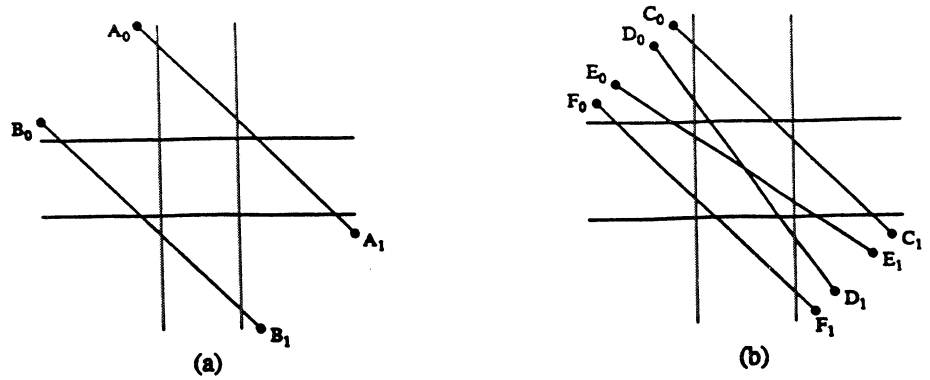


Figure 4: The partition-pair (A_{RB}, A_{LT}) has six subcases.
(a) The two reject subcases.
(b) The four non-reject subcases.

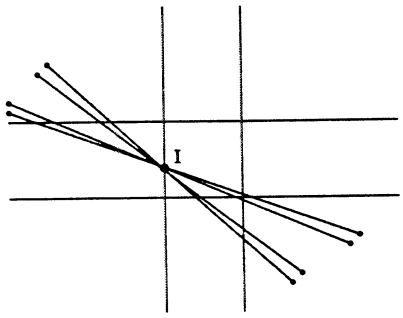


Figure 5: Geometric possibilities for line segments of the subcase $\mathbf{E_0}$ $\mathbf{E_1}$ in Figure 4(b) that intersect the window on the left boundary at a given point.

Now, for a given partition-pair (A_{ij}, A_{kl}) we consider all its subcases. The relative proportions of the occurrences of each subcase are used as weights in computing the average computational requirements for the partition-pair.

The computational requirements are, of course, dependent upon the ordering of the comparison tests. As mentioned in Section III, a specific ordering is represented by a line clipping tree whose nodes are comparison tests. Clearly, there are many possible line clipping trees for any partition-pair.

For a given partition-pair (A_{ij}, A_{kl}), consider an arbitrary line clipping tree which we will denote by $T(A_{ij}, A_{kl})$. We can calculate the computational requirements of $T(A_{ij}, A_{kl})$ by considering the ordering as specified by the tree and taking a weighted average of the computational requirements of each subcase where the weights are the above-mentioned relative proportions of the occurrences of each subcase for the given partition-pair. In particular, we consider three separate weighted averages: (i) the number of additions and subtractions, (ii) the number of multiplications and divisions, and (iii) the number of comparisons.

For a given partition-pair, we can find a line clipping tree that simultaneously minimizes each of these three weighted averages. Except for geometric symmetry, the tree is unique. It is said to be the *optimal tree* of the partition-pair (A_{ij}, A_{kl}) and will be denoted by $OT(A_{ij}, A_{kl})$.

To see this, we carefully construct the optimal tree for various types of partition-pairs. First, we divide the partitions into three categories, as in.[15] A partition is a *side* if it has a common *edge* with the window; these are A_{LM}, A_{RM}, A_{CB}, and A_{CT}, as was shown in Figure 1. A partition is a *corner* if it has only one corner point in common with the window; these are A_{LB}, A_{RB}, A_{LT}, and A_{RT}, as was also shown in Figure 1. A partition is the *window* if it is the visible region; this is A_{CM}, as was also shown in Figure 1.

Now, consider the possible types of partition-pairs. The partition-pair (A_{CM}, A_{CM}) is window-window and is a trivial accept. This leaves five distinct types of partition-pairs. These are: (1) window-side or side-window, (2) window-corner or corner-window, (3) side-side, (4) side-corner or corner-side, and (5) corner-corner. Note that the last two types contain the trivial reject case of line segments that are completely on the invisible side of a boundary line. This trivial reject case is excluded in the construction of the optimal tree for these types of partition-pairs. We turn now to the optimal tree for each of these five types of partition-pairs:

IV.1. Optimal Tree of Window-Side or Side-Window

The eight instances of this type are shown in Figure 6. In each case, the result of the clip is immediately determined with no comparisons.

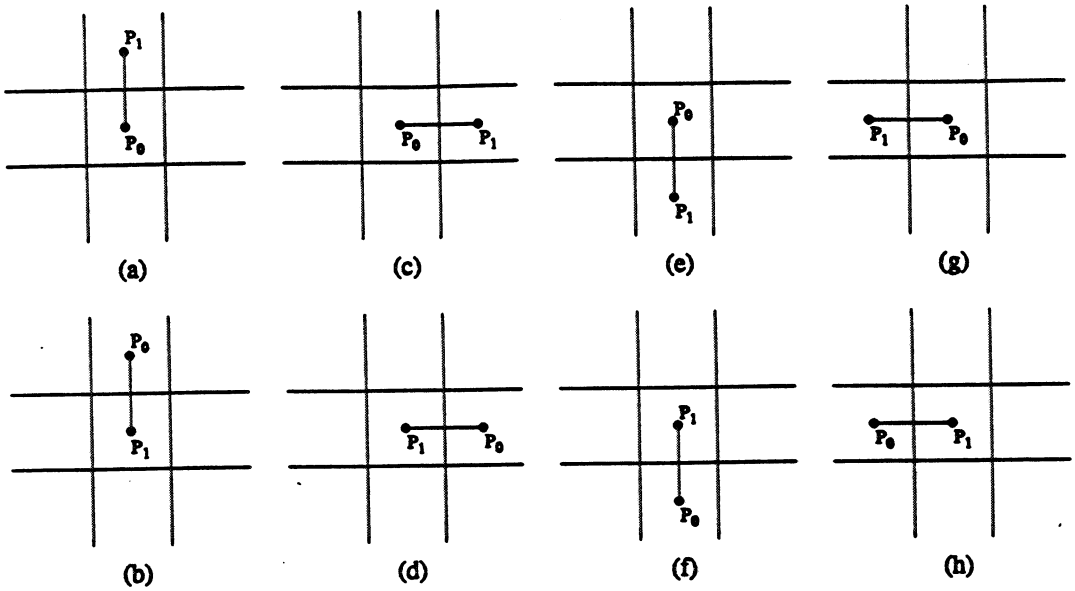


Figure 6: Eight instances of window-side or side-window type of partition-pair.

As a representative, consider the instance illustrated in Figure 6(a). The result of the clip is:†

$$y_1 = y_T$$
$$x_1 = x_T^I$$

† This requires four additions / subtractions.

IV.2. Optimal Tree of Window-Corner or Corner-Window

The eight instances of this type are shown in Figure 7.

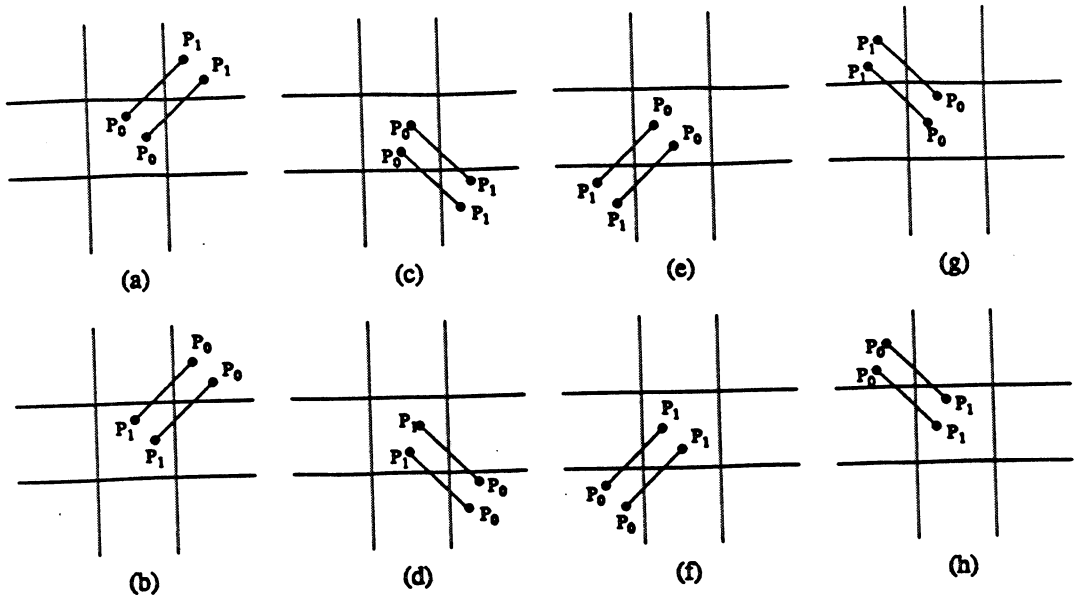


Figure 7: Eight instances of window-corner or corner-window type of partition-pair.

As a representative, consider the instance illustrated in Figure 7(a). The optimal tree for this is shown in Figure 8.

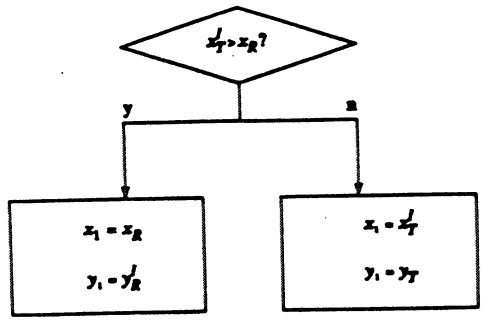


Figure 8: Optimal tree for Figure 7(a).

Note that we use the convention that the left hand branch is followed if the condition is satisfied and the right branch is followed otherwise.

IV.3. Optimal Tree of Side-Side

The twelve instances of this type are divided into two groups. Four instances involve "opposite sides" as shown in Figure 9 and eight instances involve "adjacent sides" as shown in Figure 10.

Consider first the "opposite sides" group shown in Figure 9. As a representative, consider the instance illustrated in Figure 9(a). The result of the clip is:

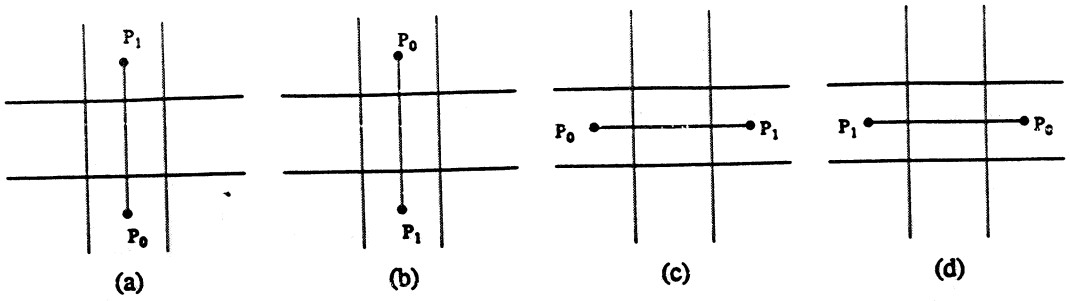


Figure 9: Four instances of opposite sides group of side-side type of partition-pair.

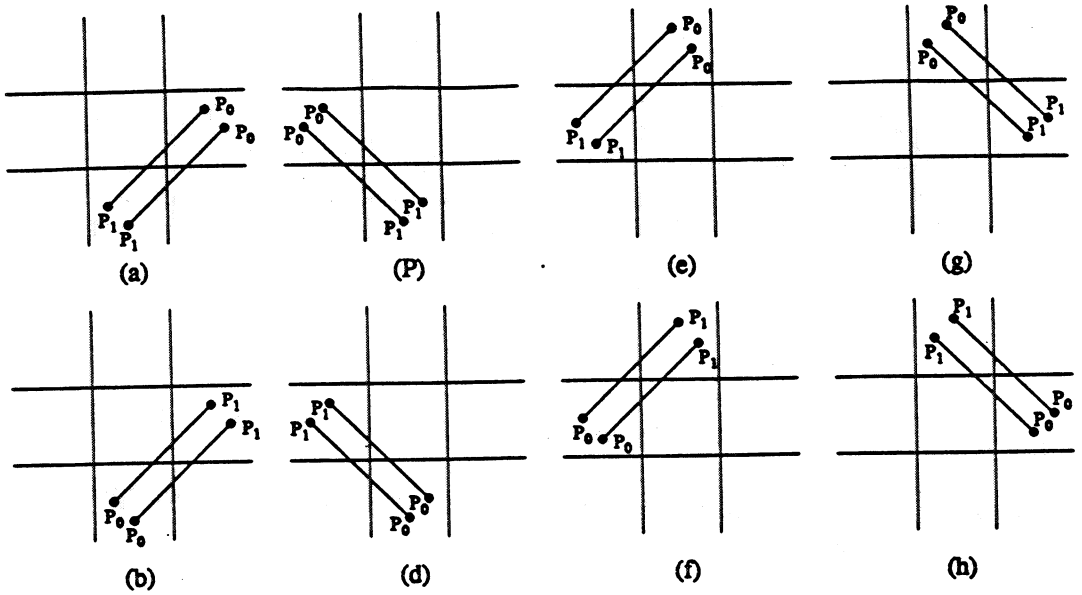


Figure 10: Eight instances of adjacent sides group of side-side type of partition-pair.

$$x_0 = x_B^I$$
$$y_0 = y_B$$
$$x_1 = x_T^I$$
$$y_1 = y_T$$

Now, consider the "adjacent sides" group shown in Figure 10. As a representative, consider the instance illustrated in Figure 10(a). The optimal tree for this is shown in Figure 11.

IV.4. Optimal Tree of Corner-Side or Side-Corner

The sixteen instances of this type are shown in Figure 12. As a representative, consider the instance illustrated in Figure 12(l). The optimal tree for this is shown in Figure 13.

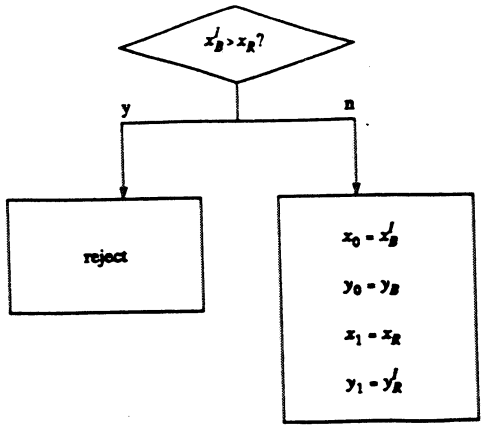


Figure 11: Optimal tree for Figure 10(a).

IV.5. Optimal Tree of Corner-Corner

The four instances of this type are shown in Figure 14.

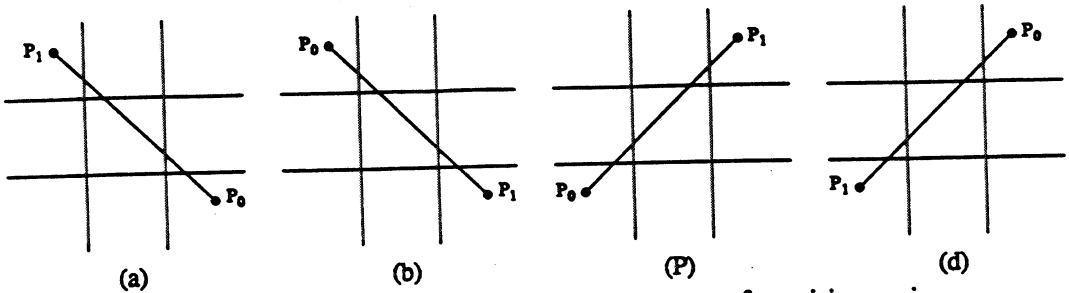


Figure 14: Four instances of corner-corner type of partition-pair.

As a representative, consider the instance illustrated in Figure 14(a). The optimal tree for this is:

V. Optimal Tree for the Region-Pair

Having considered the optimal tree for various type of partition-pairs, let us now investigate the optimal tree for a region-pair, as was discussed in Section III. Note that a region-pair can always be decomposed uniquely into the union of non-overlapping partition-pairs. That is,

$$(C_0, C_1) = \cup\, (A_{ij}, A_{kl})$$

where each (A_{ij}, A_{kl}) is said to be a *subpartition* of (C_0, C_1).

Unlike the optimal trees for the various cases of partition-pairs, it is not always possible to have an optimal tree for a region-pair. That is, it is not always possible to minimize simultaneously (i) the number of additions and subtractions, (ii) the number of multiplications and divisions, and (iii) the number of comparisons. The following two examples will illustrate this. However, if we exclude the number of comparisons, then it is possible to minimize simultaneously the arithmetic operations (that is, (i) and (ii)).

Example 1: Figure 16 shows the region-pair for this example. In this case, the convex region C_0 corresponds to the union of partitions A_{LB} and A_{CB} while the convex region C_1 corresponds to partition A_{CT}. Consider the line clipping trees shown in

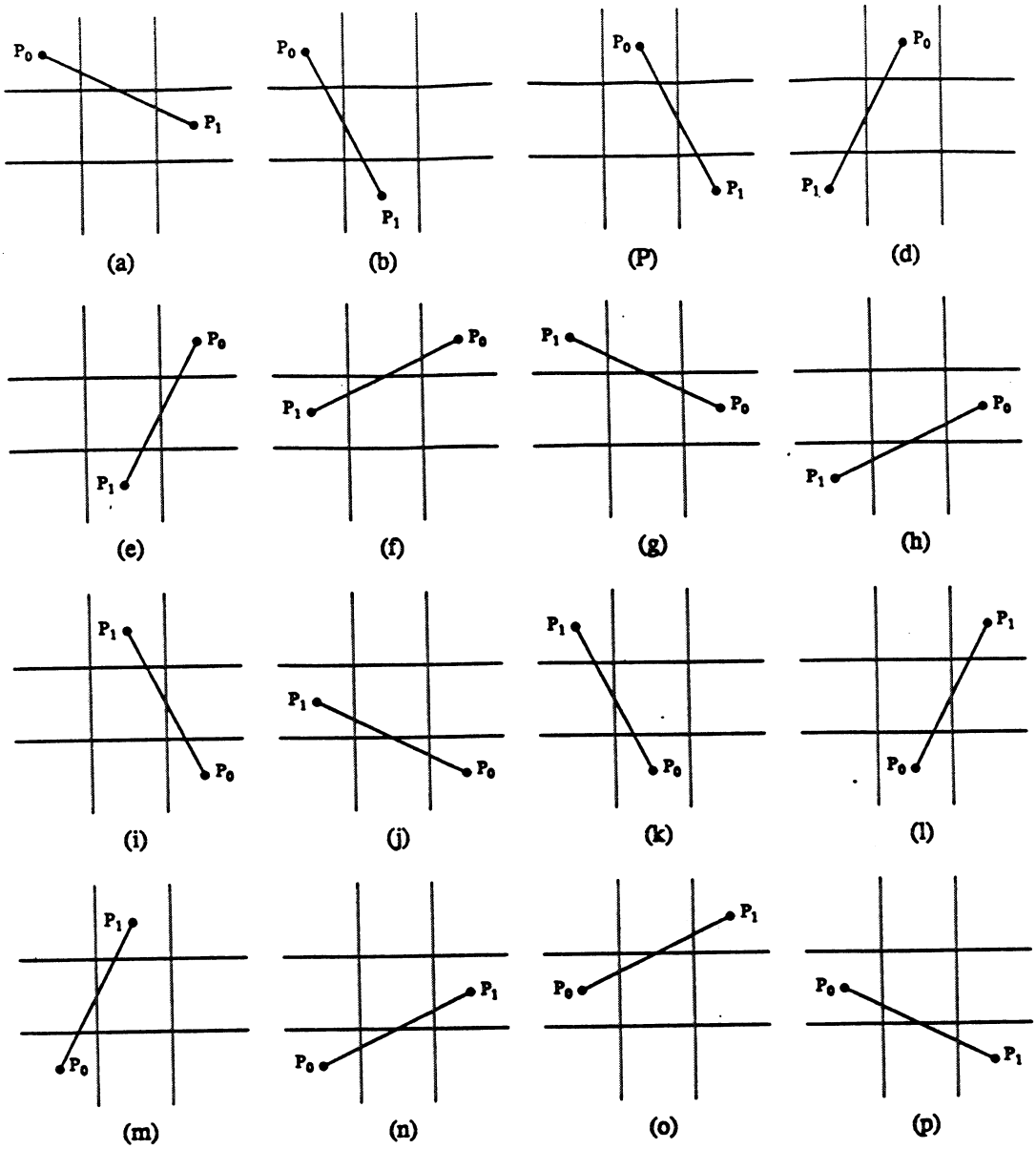


Figure 12: Sixteen instances of corner-side or side-corner type of partition-pair.

Figure 17. The tree shown in Figure 17(a) minimizes the number of arithmetic operations ((i) and (ii) above) while the tree shown in Figure 17(b) minimizes the number of comparisons.

Example 2: The region-pair for this example is shown in Figure 18. In this situation, the convex region C_0 corresponds to the union of partitions A_{LM} and A_{LT} while the convex region C_1 corresponds to the union of partitions A_{CB} and A_{RB}. Consider the line clipping trees shown in Figure 19. The tree shown in Figure 19(a) minimizes the number of arithmetic operations while the tree shown in Figure 19(b) minimizes the number of comparisons.

Furthermore it is clear any region-pair where two regions contain those in Example 1 (or in Example 2) will exhibit the same difficulty; that is, there will be one tree that minimizes the arithmetic operations and another one that minimizes the number of

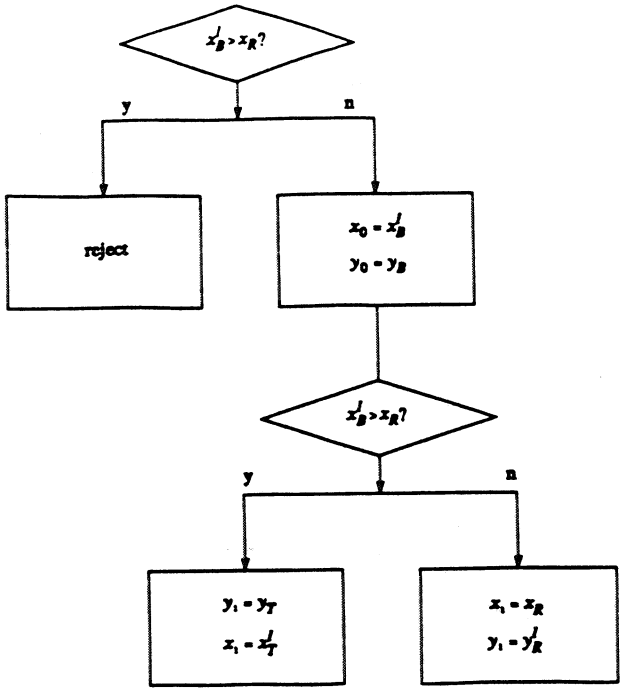


Figure 13: Optimal tree for Figure 12(l).

comparisons. However, any other region-pair whose two regions do not contain those in either example will not have this shortcoming. That is, there will be a tree that minimizes simultaneously (i) the number of additions and subtractions, (ii) the number of multiplications and divisions, and (iii) the number of comparisons.

In those cases where is is not possible to minimize simultaneously all these categories of operations ((i)-(iii)), then we will favor minimization of arithmetic operations ((i) and (ii)) over that of the comparisons ((iii)). Thus, we define the *shortest tree of a region-pair* as a tree that, first, absolutely minimizes the number of arithmetic operations ((i) and (iii)) and, further, if there is more than one such tree, as the particular tree among these that has the fewest number of comparisons.

This leads to the question regarding the existence, construction, and uniqueness of this shortest tree. To address these questions, we will first introduce *irreducible region-pairs* and find the shortest tree of the irreducible region-pairs and then transform the problem into minimizing the single quantity corresponding to the number of comparisons.

VI. Absolutely and Relatively Irreducible Optimal (*AIO* and *RIO*) Region-Pairs

A key idea in our approach is the judicious subdivision of region-pairs that contain a trivial reject region-pair of a visible region-pair (that is, as was defined at the end of Section III). Consider a non-key region-pair. We will call it an *absolutely irreducible optimal* (denoted by *AIO*) region-pair if the shortest tree of this region-pair contains no nodes that have primary comparisons. Furthermore, we will call a non-key region-pair a *relatively irreducible optimal* (denoted by *RIO*) region-pair if the root node of the shortest tree of this region-pair is a secondary comparison and the shortest tree contains some nodes that have primary comparisons. A non-key region-pair that is neither *AIO* nor *RIO*

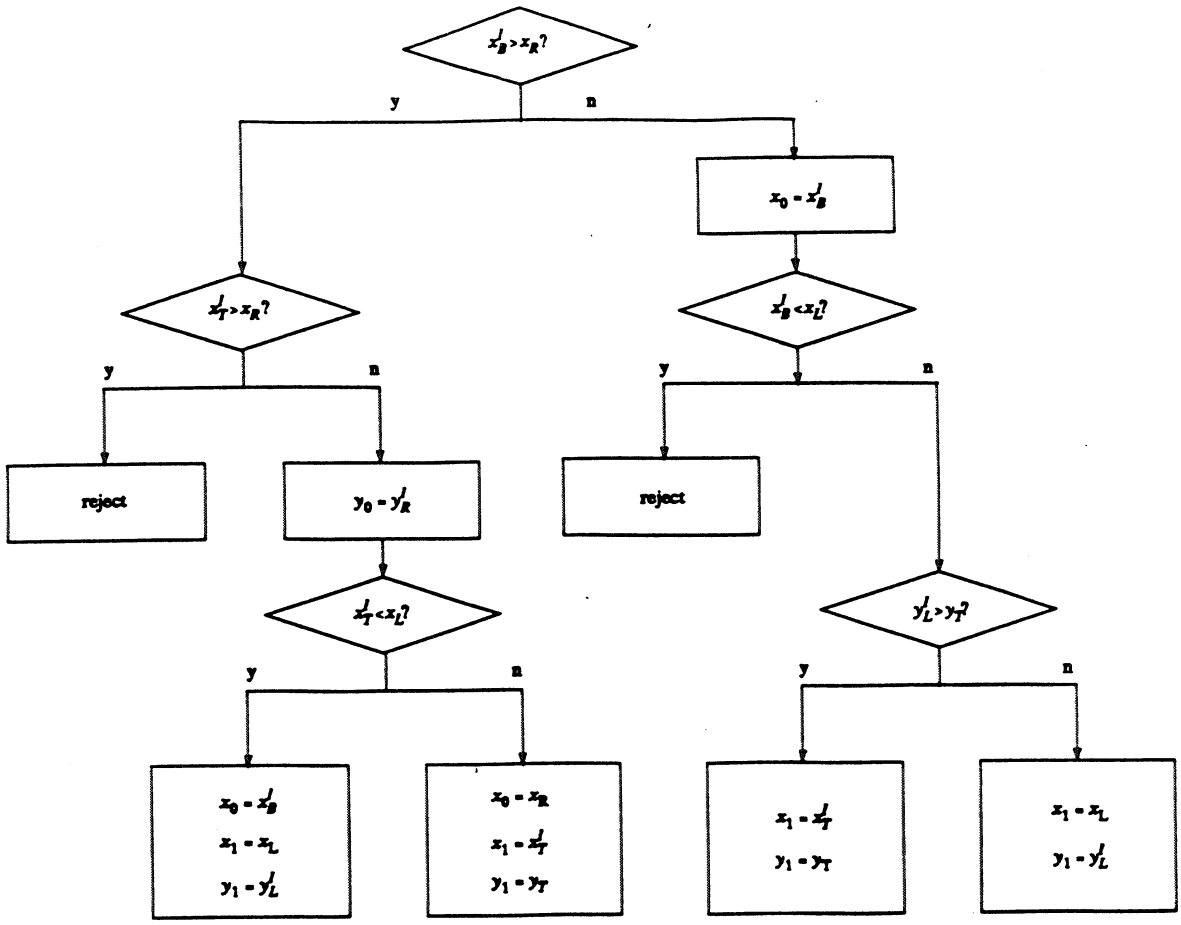


Figure 15: Optimal tree for Figure 14(a).

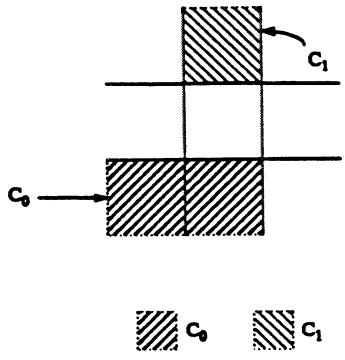


Figure 16: Region-pair for Example 1.

is a *reducible* region-pair.

To investigate the *AIO* and *RIO* region-pairs, some further distinctions are helpful. Consider a non-key region-pair. We will call this a *secondary comparison* region-pair if there exists a tree for this region-pair that has no nodes having primary comparisons.

Thus, a secondary comparison region-pair can have line clipping solved by using secondary comparisons exclusively. Note that an *AIO* region-pair must be such a

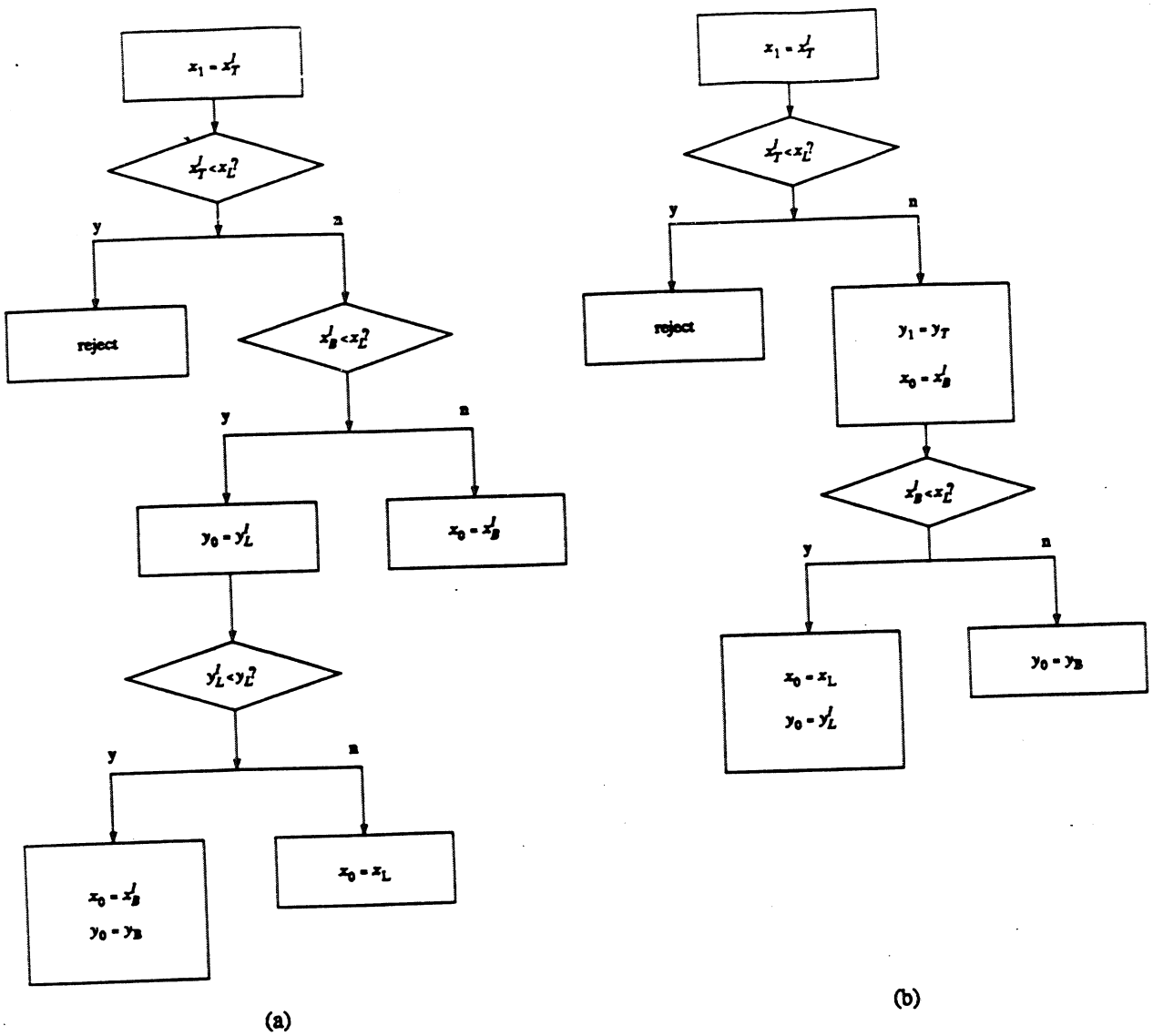


Figure 17: Two different line clipping trees for the region-pair in Figure 16.
(a) The tree minimizing the number of arithmetic operations.
(b) The tree minimizing the number of comparisons.

secondary comparison region-pair. Hence, we will now enumerate the secondary comparison region-pairs.

There are five types of secondary comparison region-pairs, two of which are *AIO*, two of which are *RIO*, and one of which is reducible. These five types of region-pairs will be denoted by *AIO*-1, *AIO*-2, *RIO*-1, *RIO*-2, and *R*, respectively. We will also name them to indicate their geometric arrangement, by considering the regions in the following ways: if a region is the window, if a region is not the window but contains it, if the two regions share only a common point, and if the two regions are located on opposite sides of the window.

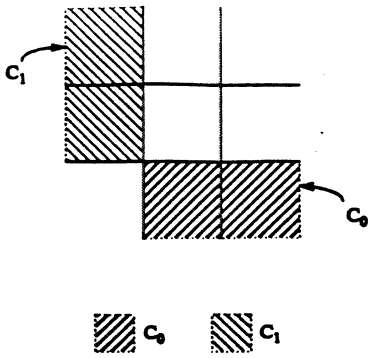


Figure 18: Region-pair for Example 2.

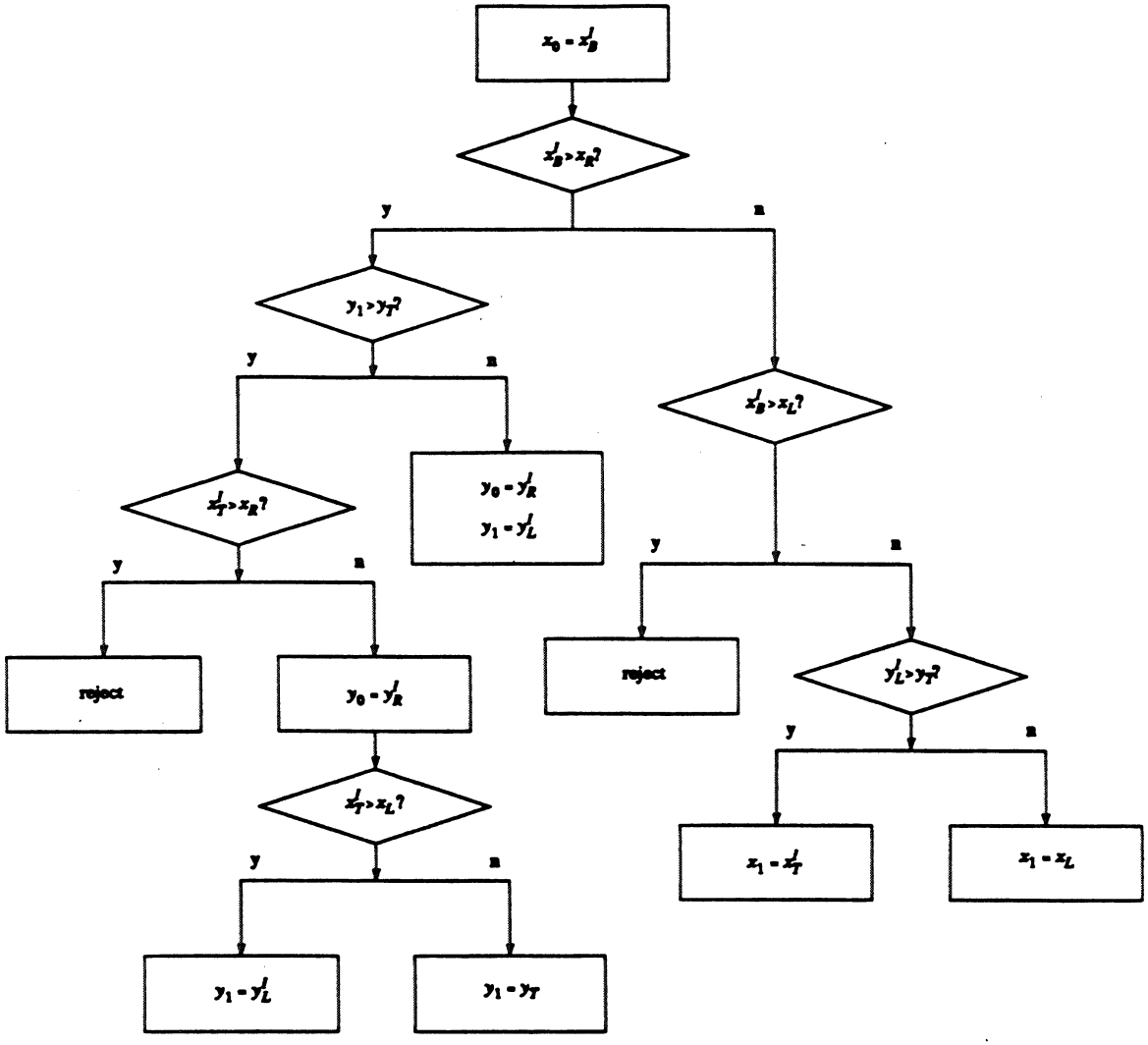


Figure 19: Two different line clipping trees for the region-pair in Figure 18. (a) The tree minimizing the number of arithmetic operations.

VI.1. *AIO*-1: *Opposite* Type Region-Pair

The *AIO*-1 type region-pair has one of its regions that is a strip of three partitions outside the window (this could be to the left, above, to the right, or below the window)

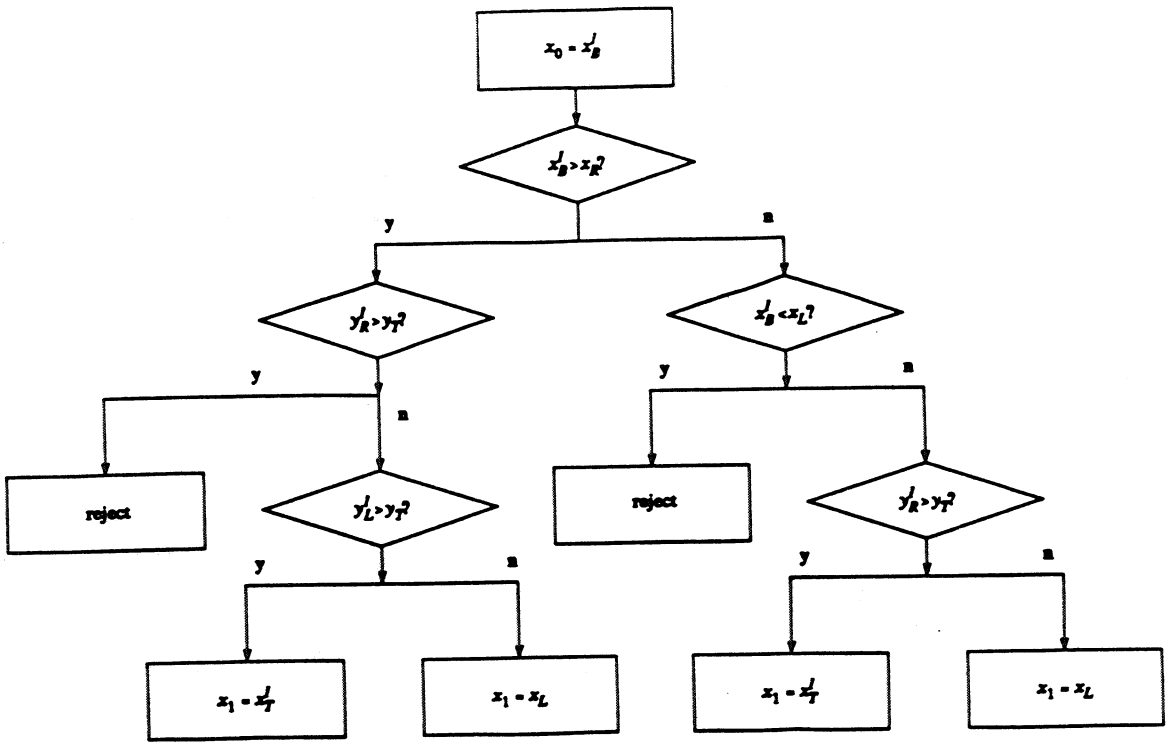


Figure 19: Two different line clipping trees for the region-pair in Figure 18. (b) The tree minimizing the number of comparisons.

where its other region is one partition on the opposite side of the window and central. Since the two regions are located on opposite sides of the window, we call this the *opposite* type region-pair. there are eight possibilities for the *AIO*-1 type region-pair and they are illustrated in Figure 20.

VI.2. *AIO*-2: *Window* Type Region-Pair

The *AIO*-2 type region-pair has the window as one of its regions where its other region is a strip of three partitions outside the window. This is similar to the *AIO*-1 *opposite* type region-pair except that instead of the single partition region being outside the window, it actually is the window. Since one of the regions is the window, we call this the *window* type region-pair. Similar to the *AIO*-1 type region-pair, there are eight possibilities for the *AIO*-2 type region-pair and they are depicted in Figure 21.

VI.3. *RIO*-1: *Balanced-Pivotal* Type Region-Pair

The *RIO*-1 type region-pair has both of its region containing two partitions and sharing only a common point. Since the two regions are of equal size and the join at a single point, we dub this the *balanced-pivotal* type region-pair. There are eight arrangements for the *RIO*-1 type region-pair and they are shown in Figure 22.

VI.4. *RIO*-2: *Unbalanced-Pivotal* Type Region-Pair

The *RIO*-2 type region-pair is similar to the *RIO*-1 region-pair in that its regions share a single point; however, instead of both regions containing two partitions, the *RIO*-

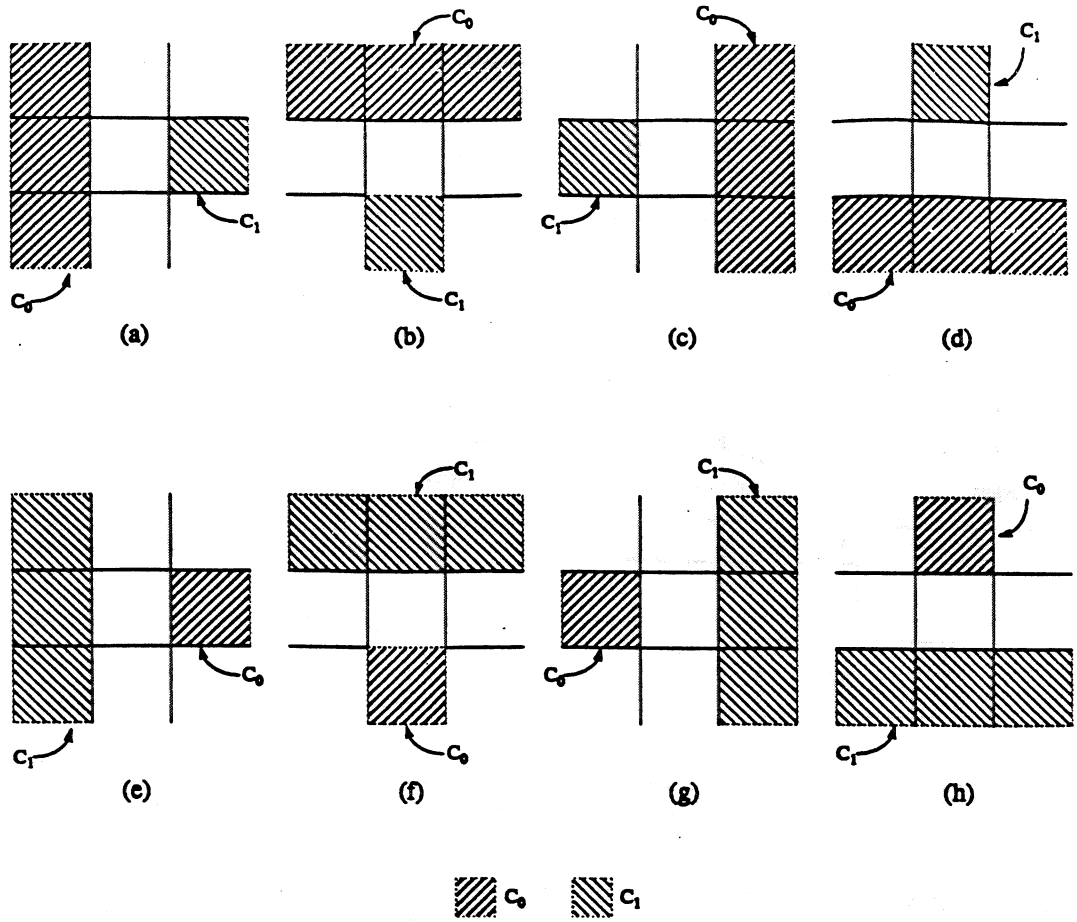


Figure 20: *AIO*-1: *Opposite* type region-pair.

2 region-pair has one of its regions being a single partition that is located outside of and adjacent to the window. Since the two regions join at a single point, but are not the same size, we refer to the *RIO*-2 region-pair as the *unbalanced-pivotal* type region-pair. There are sixteen possibilities for the *RIO*-2 type region-pair and they are given in Figure 23.

VI.5. *R*: *Strip* Type Region-Pair

The *R* region-pair has its regions together forming a strip of three partitions including the window. More specifically, one region has two partitions, one of which is the window and the other is a partition that shares a common edge with the window; the other region has one partition that shares the opposite edge of the window such that the two regions form a strip. Due to this configuration, we denote an *R* region-pair as a *strip* type. There are eight possibilities for a *strip* region-pair and they are shown in Figure 24.

VII. Framework of Program and Modules

For each of the five types of secondary comparison region-pairs, we will provide a simple clipping procedure. These five procedures cover all cases; any given line segment can be clipped by calling at most one procedure.

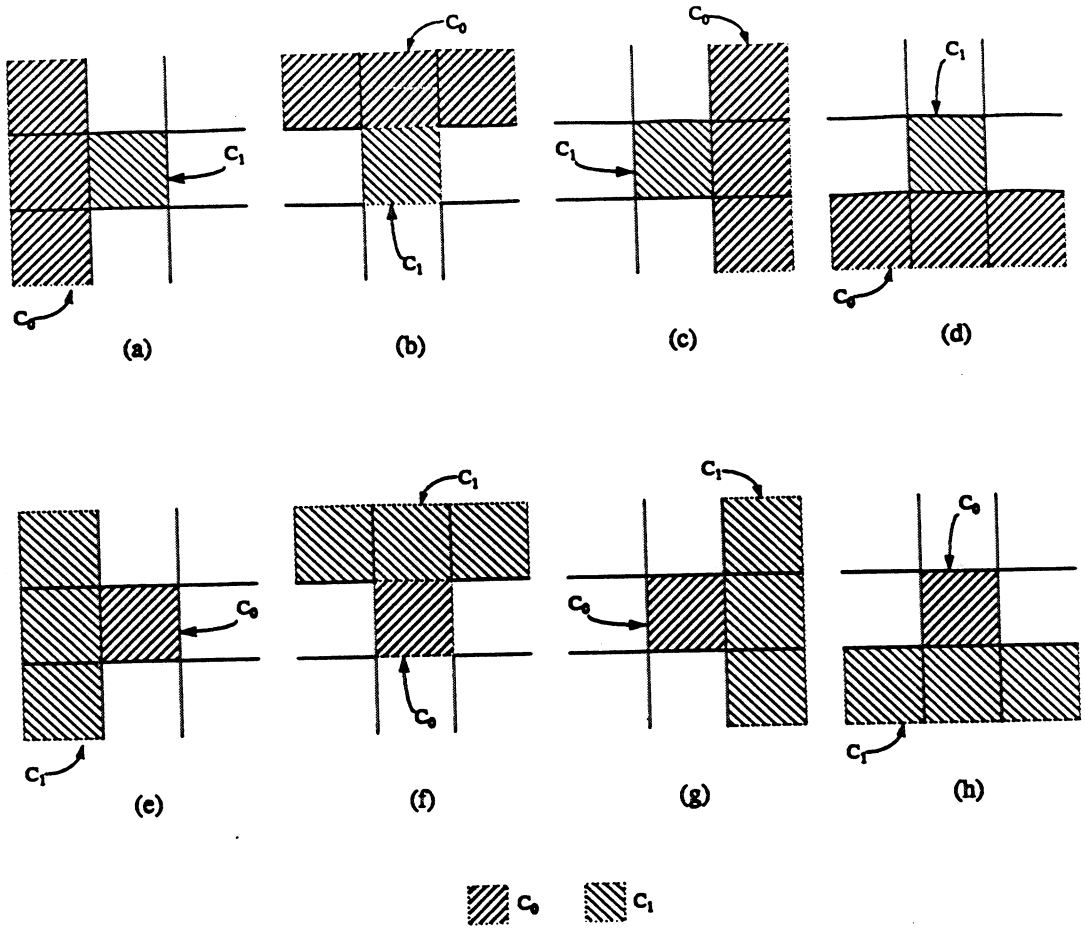


Figure 21: *AIO*-2: *Window* type region-pair.

The overall structure of the program is a careful analysis for the line segment to be clipped. The line segment is classified and then at most one procedure is invoked to clip it. The basic framework of the program is shown in Figure 25.

There are a total of 19 cases depicted in Modules A through E. In Module E, one case is a trivial accept, and two cases are intersection computations. Thus, there are a total of 16 instances of absolutely and relatively irreducible optimal region-pairs. These 16 cases are handled with five procedures, each of which can cover all geometric symmetries.

These 16 region-pairs comprise two instances of the *opposite* type, four of the *window* type, four of the *balanced-pivotal* type, four of the *unbalanced-pivotal* type, and two of the *strip* type region-pair. We turn now to the generic clipping procedures for each of the five types of region-pairs, and the appropriate calling sequence to accommodate all the 16 instances.

For efficiency, the procedures are developed so as to use multiplications instead of divisions. Also, the procedures for the relatively irreducible optimal region-pairs (that is, *RIO*-1, the *balanced-pivotal* type and *RIO*-2, the *unbalanced-pivotal* type are based on a parametric approach.

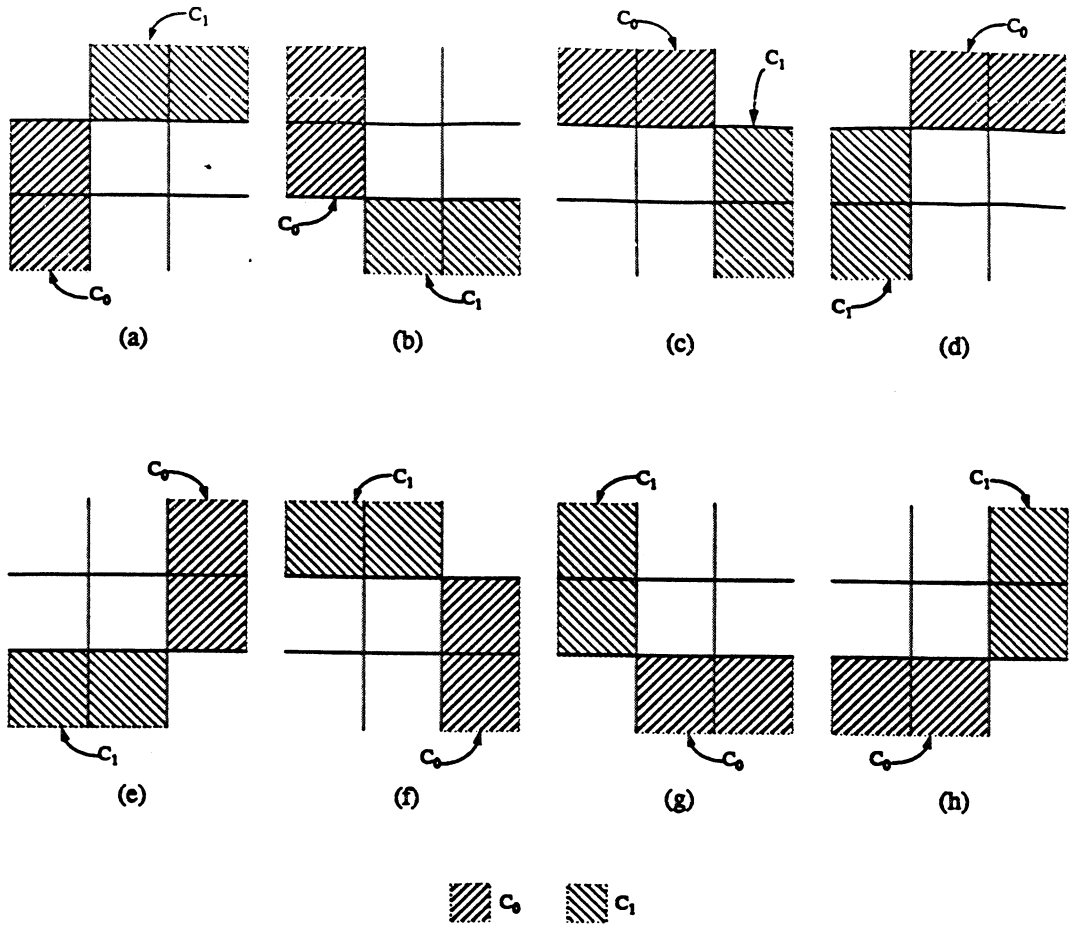


Figure 22: *RIO*-1: *Balanced-pivotal* type region-pair.

VIII. Clipping Procedures for the Five Types of Region-Pairs

VIII.1. *AIO*-1: *Opposite* Type Region-Pair Clipping Procedure

The *opposite* type region-pair occurs twice, once each in Module A and Module B. These instances have $\mathbf{P}_0$ in a horizontal strip of three partitions below and above the window, respectively, and $\mathbf{P}_1$ in a central partition that is above and below the window, respectively. For the generic clipping procedure, the boundary line adjacent to the strip of three partitions is denoted b1 and the boundary line alongside the centrally-located partition is denoted b2, and the procedure definition is **clip_opposite** (b1, b2).

Figure 31 gives the procedure and Figure 32 shows the instances of the *opposite* type region-pair with the associated procedure calling sequences.

VIII.2. *AIO*-2: *Window* Type Region-Pair Clipping Procedure

The window type region-pair occurs four times, once each in Module A and Module B and twice in Module E. In Module A, $\mathbf{P}_0$ is in the window and $\mathbf{P}_1$ is in a horizontal strip of three partitions below the window. The instance in Module B is similar except that $\mathbf{P}_1$ is in a horizontal strip that now is above the window. The two instances in Module E duplicate the instances of Module A and B except that it is $\mathbf{P}_1$ that is in the

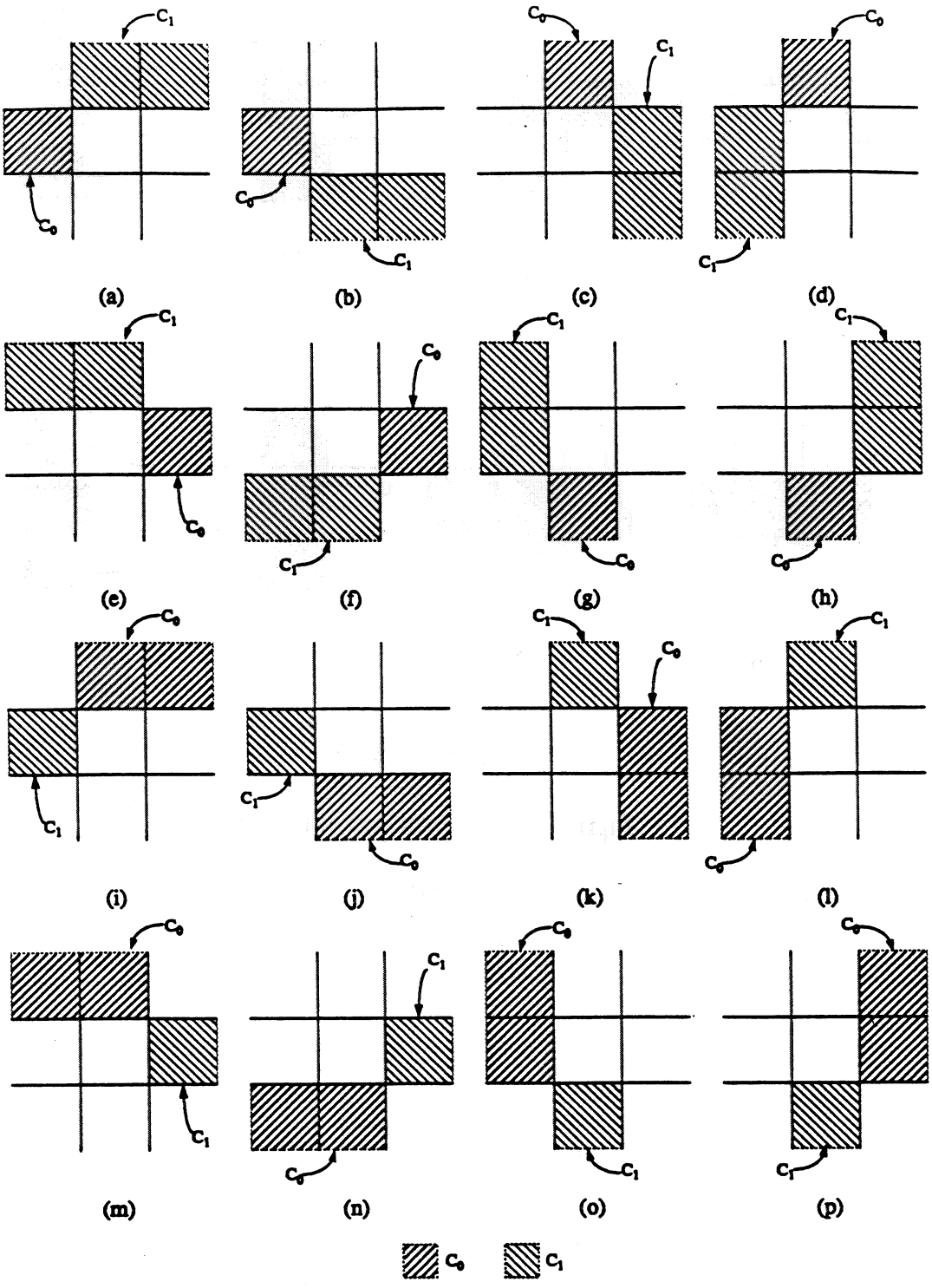


Figure 23: *RIO-2: Unbalanced-pivotal* type region-pair.

window and P_0 that is in the horizontal strip. For the generic clipping procedure, the point inside the window is denoted (x_window, y_window), the point in the strip is denoted (x_strip, y_strip), and the boundary line between them is denoted b1, and the procedure definition is **clip_window** (x_window, y_window, x_strip, y_strip, b1). Figure 33 provides the procedure and Figure 34 shows the instances of the *window* type region-

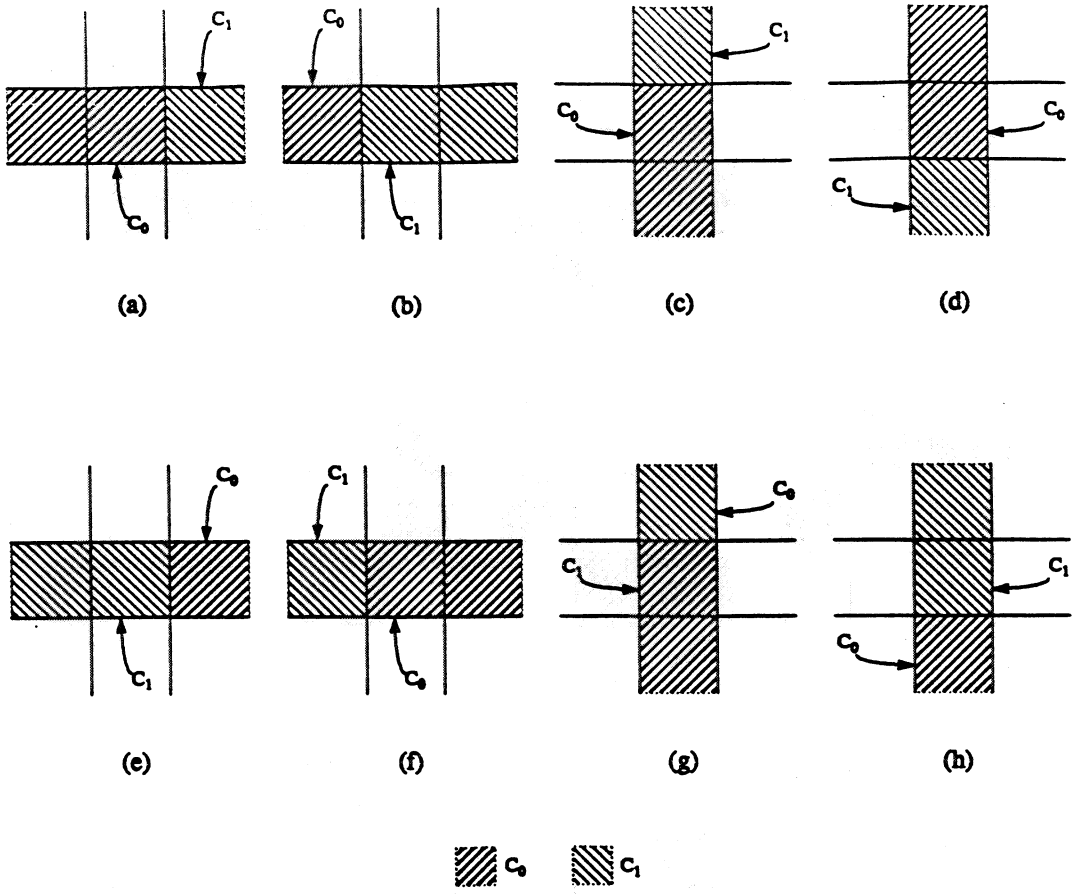


Figure 24: *R*: *Strip* type region-pair.

pair with the associated procedure calling sequences.

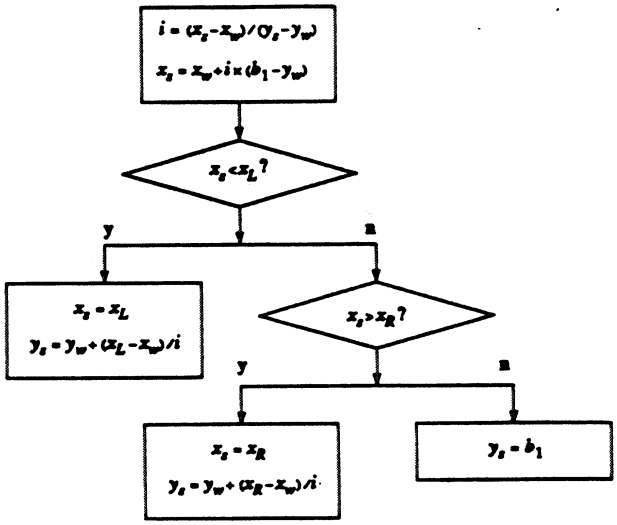


Figure 33: clip_window (x_window, y_window, x_strip, y_strip, b1).

```
if y0 < yB (* P0 below bottom *)
then if y1 < yB
    then reject (* P0 and P1 both below bottom *)
    else (* P0 below bottom and P1 above bottom
        in one of four cases of Module A shown in Figure 26 *)
        ...
else (* P0 above bottom *)
    if y0 > yT (* P0 above top *)
    then if y1 > yT
        then reject (* P0 and P1 both above top *)
        else  (* P0 above top and P1 below top in
            one of four cases of Module B shown in Figure 27  *)
            ...
    else  (* P0 above bottom and below top *)
        if x0 > xR
        then if x1 > xR
            then reject  (* P0 and P1 both right of right *)
            else  (* P0 is above bottom, below top, and right of right and P1
                is left of right in one of four cases of Module C shown in Figure 28  *)
                ...
        else (* P0 is above bottom, below top, and left of right *)
            if x0 < xL
            then if x1 < xL
                then reject  (* P0 and P1 both left of left *)
                else  (* P0 is above bottom, below top, left of left, and P1
                    is right of left in one of four cases of Module D shown in Figure 29 *)
                    ...
            else  (* P0 is above bottom, below top, left of right, and right of left,
                and thus P0 is inside window and P1 is in one of five cases
                of Module E shown in Figure 30 *)
                    ...
```

Figure 25: The basic framework of the program.

VIII.3. *RIO*-1: *Balanced-Pivotal* Type Region-Pair Clipping Procedure

The *balanced-pivotal* type region-pair occurs four times, twice each in Module A and B. These instances have two regions both containing two partitions and joining at a common point. In all four instances, the point P_0 is in the horizontally-oriented region and P_1 is in the vertically-oriented region. For the generic clipping procedure, the following notation is established: b1 denotes the horizontal boundary along the longer side of the region containing P_0, b2 denotes the vertical boundary slicing through this region, b3 denotes the vertical boundary along the longer side of the region containing P_1, b4 denotes the horizontal boundary slicing through this region, flag0 indicates if P_0 is on the invisible side of the vertical boundary b2, and flag1 indicates if P_1 is on the invisible side of the horizontal boundary b4, and the procedure definition is **clip_balanced_pivotal** (b1, b2, b3, b4, flag0, flag1). Figure 35 gives the procedure and Figure 36 depicts the instances of the *balanced-pivotal* type region-pair with the associated procedure calling sequences.

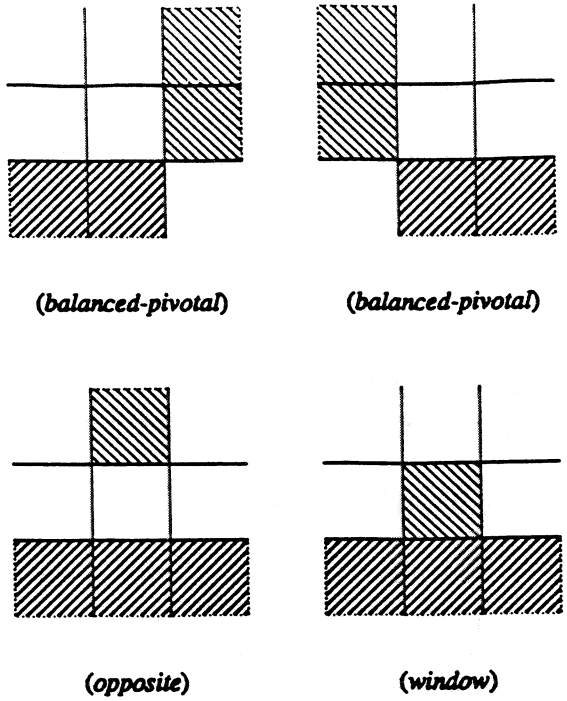


Figure 26: Module "A" comprises these four cases.

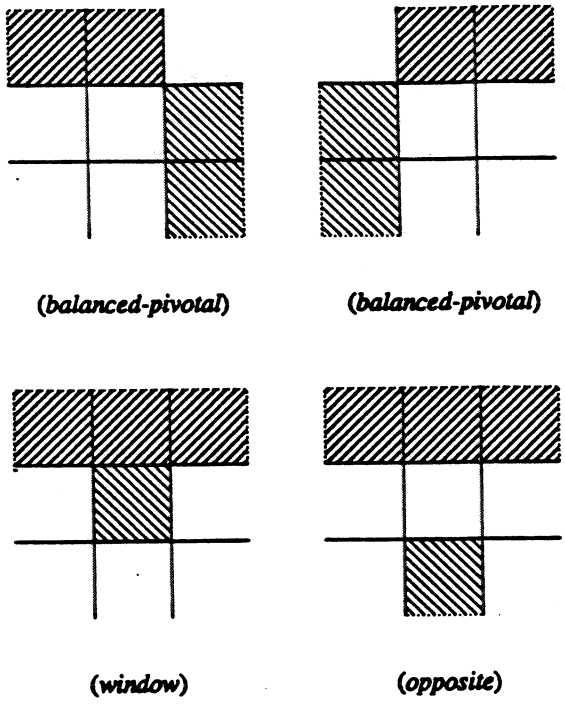


Figure 27: Module "B" comprises these four cases.

VIII.4. *RIO-2: Unbalanced-Pivotal* Type Region-Pair Clipping Procedure

The *unbalanced-pivotal* type region-pair occurs four times, twice each in Module C and D. These instances have a central partition joining a horizontally-oriented region containing the partitions. The point P_0 is in the central partition which is to the right of the window in Module C and to the left of the window in Module D. For the generic

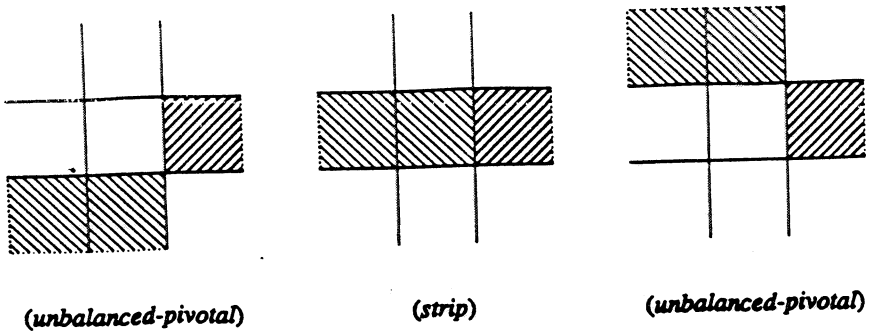


Figure 28: Module "C" comprises these three cases.

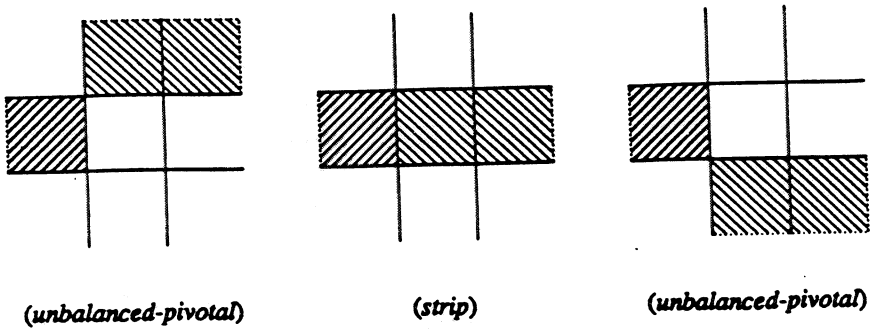


Figure 29: Module "D" comprises these three cases.

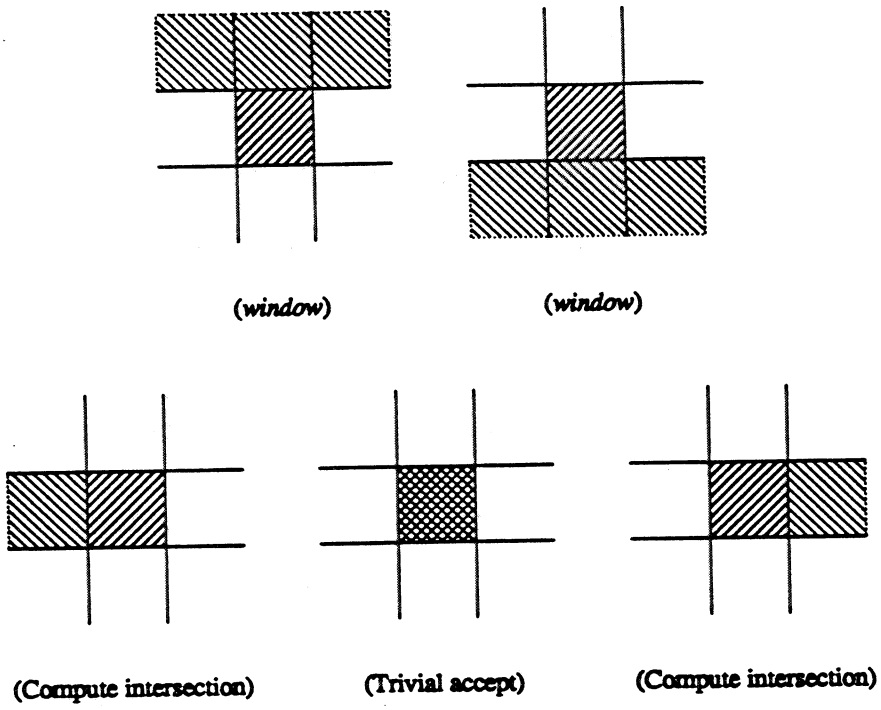


Figure 30: Module "E" comprises these five cases.

clipping procedure, the following notation is established: b2 denotes the vertical boundary line between the window and the central partition containing $\mathbf{P}_0$, b3 denotes the other vertical boundary line slicing through the horizontally-oriented region containing

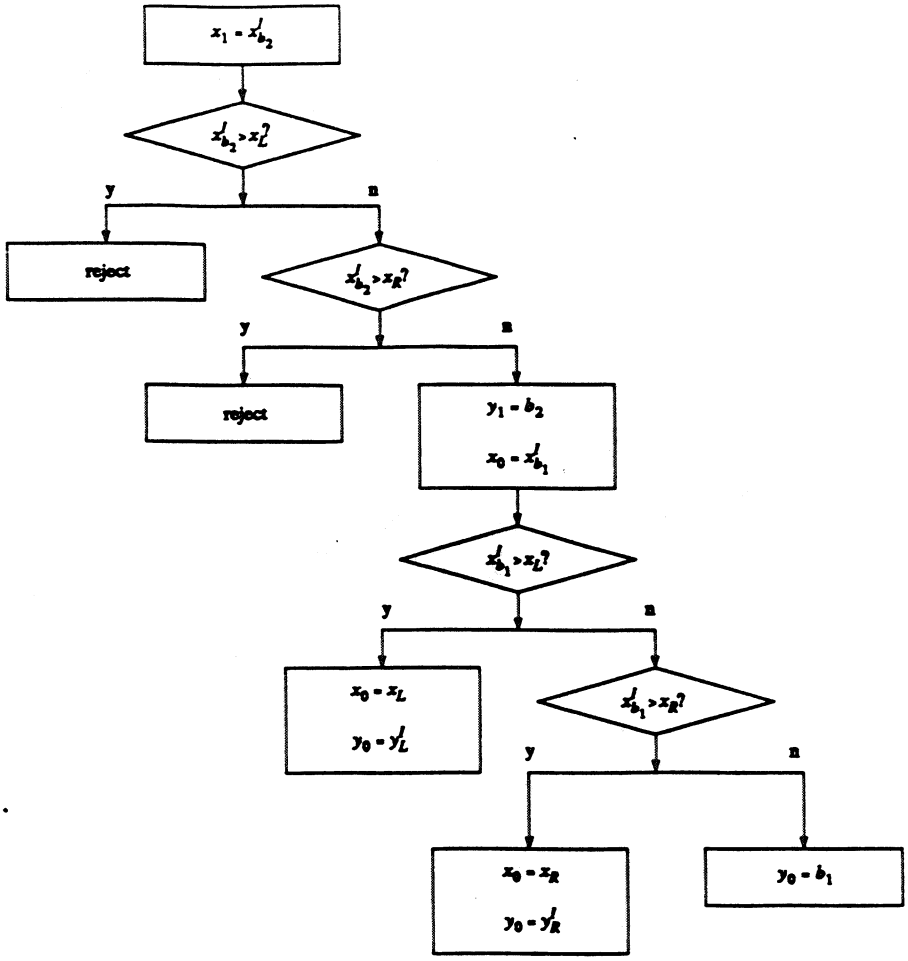
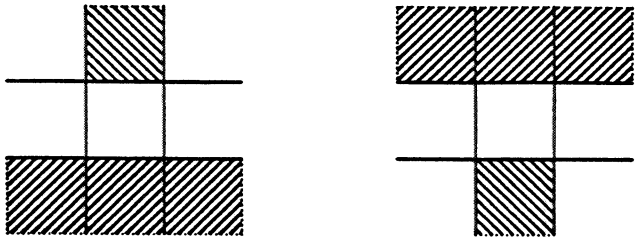


Figure 31: clip_opposite (b1, b2).



clip_opposite(yB, yT) **clip_opposite(yT, yB)**

Figure 32: clip_opposite procedure calling sequences.

two partitions, b4 denotes the horizontal boundary along the longer side of this region and flag indicates if $\mathbf{P}_1$ is on the invisible side of the vertical boundary b3, and the procedure definition is **clip_unbalanced_pivotal** (b2, b3, b4, flag). Figure 37 gives the procedure and Figure 38 illustrates the instances of the *unbalanced-pivotal* type region-pair with the associated procedure calling sequences.

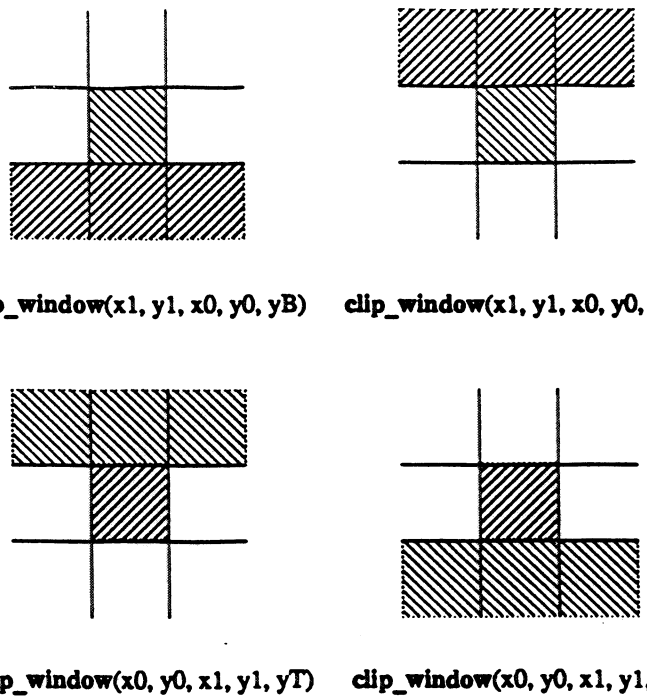


Figure 34: clip_window procedure calling sequences.

VIII.5. *R*: *Strip* Type Region-Pairs

The strip type region-pair occurs twice, once in Module C and once in Module D. These instances have two regions forming a horizontal strip of three partitions including the window. One region has two partitions, one of which is the window, while the other region is a single partition. In module C, the two-partition region contains the window and the partition to its left while the other region is the partition to the right of the window. In module D, the two-partition region contains the window and the partition to its right while the other region is the partition to the left of the window. For the generic clipping procedure, the following notation is established: b1 denotes the vertical boundary line between the two regions, b2 denotes the other vertical boundary line slicing through the two-partition region, and flag indicates if P_1 is outside the window in the two-partition region, and the procedure definition is **clip_strip** (b1, b2, flag). Figure 39 gives the procedure and Figure 40 depicts the instances of the *strip* type region-pair with the corresponding procedure calling sequences.

IX. Putting It Altogether

All the necessary details for the construction of the complete algorithm have now been covered. The structure is depicted schematically in Figure 41. This shows the Modules A through E as well as the appropriate procedure calling sequence for each case. The actual program coded in Pascal is provided in Appendix I. Finally, a fully annotated version of the program code, complete with symbolic depictions of the cases is provided in Appendix II.

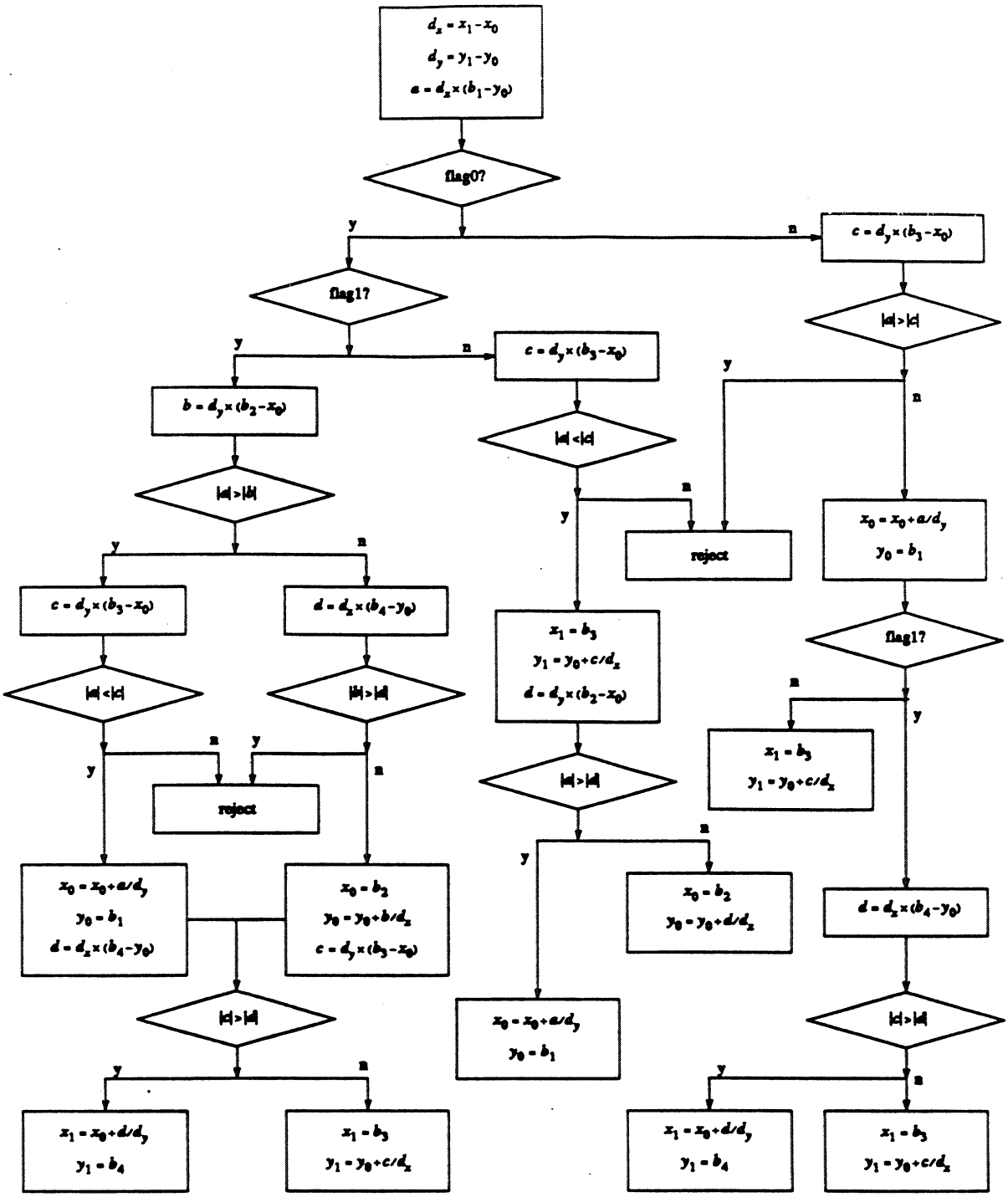


Figure 35: clip_balanced_pivotal (b1, b2, b3, b4, flag0, flag1).

X. Extending to Three Dimensions

X.1. Explanation of Set-up

For three-dimensional clipping, line segments are clipped against a *viewing pyramid.* In the coordinate system where the center of projection is at the origin, this

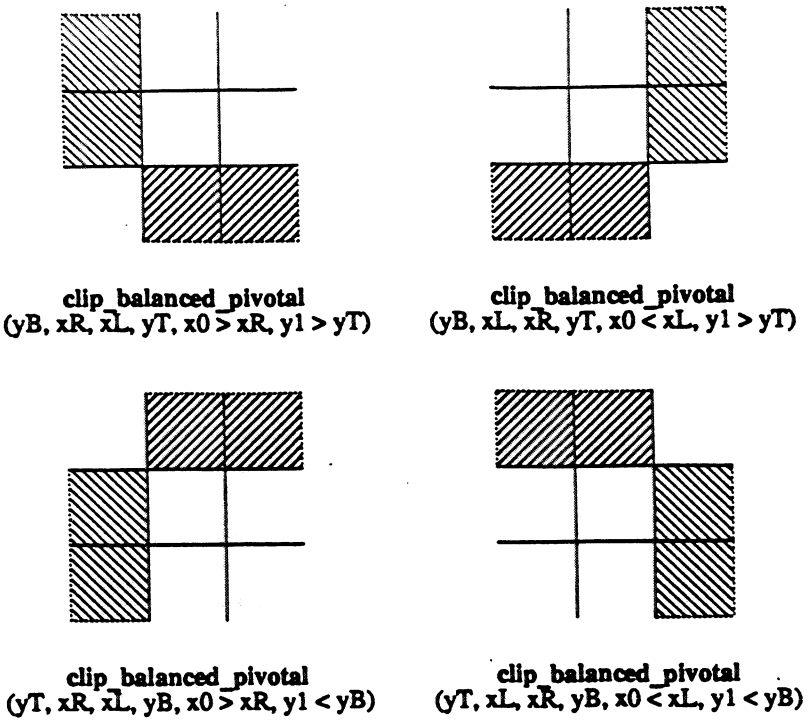


Figure 36: clip_balanced_pivotal procedure calling sequences.

volume is described by the following two pairs of inequalities:

$$-z_e \le cot(\frac{\Theta_x}{2})\, x_e \le z_e$$

$$-z_e \le cot(\frac{\Theta_y}{2})\, y_e \le z_e$$

where Θ_x and Θ_y are the angles of view with respect to the x- and y- axes, respectively. Note that either pair of these inequalities implies $-z_e \le z_e$ which implies $z_e \ge 0$.

From these inequalities, a more natural coordinate system suggests itself for clipping. This transformation is given by

$$x = cot(\frac{\Theta_x}{2})\, x_e$$

$$y = cot(\frac{\Theta_y}{2})\, y_e$$

$$z = z_e$$

Geometrically, this transformation scales the viewing pyramid so that it becomes a right pyramid. This volume then corresponds to the following inequalities:

$$-z \le x \le z$$

$$-z \le y \le z$$

and again $z \ge 0$ is implied.

This can be extended to cover the case of a finite viewing volume. Here, the viewing pyramid is truncated by hither and yon clipping planes to form a frustum of vision.

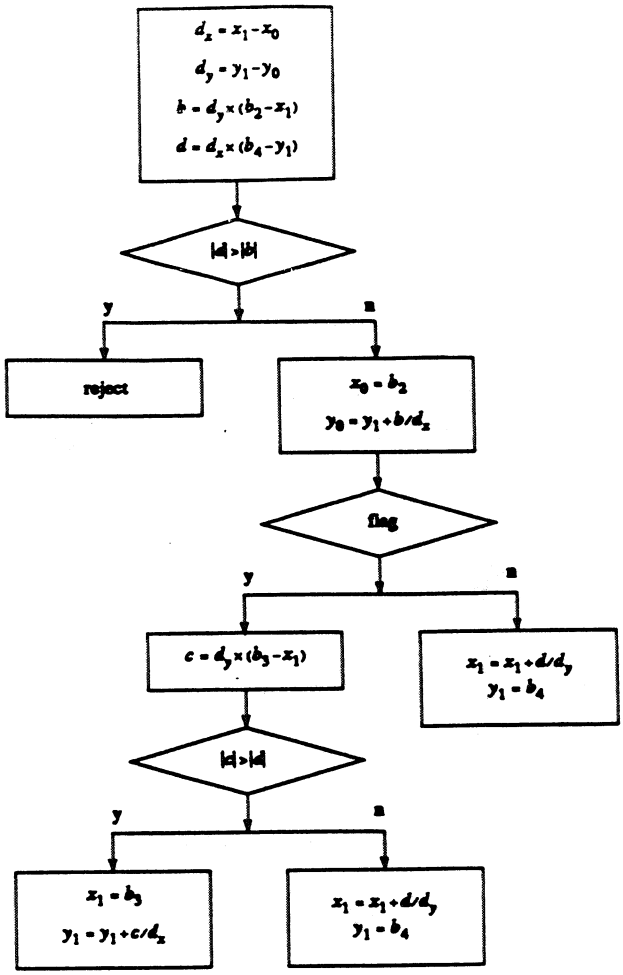


Figure 37: clip_unbalanced_pivotal (b2, b3, b4, flag).

Note that the location of these planes is completely independent of the position of the picture plane. This requires the addition of constraints for z. If the hither and yon clipping planes are $z=h$ and $z=f$, respectively, then these constraints are:

$$h \leq z \leq f$$

X.2. Optimal Tree Algorithm for Three Dimensions

One question that might arise is whether the above insights and method for successfully clipping a line segment in two dimensions extend to the problem of clipping a line segment in three dimensions. The answer is a qualified yes. Rather than attempt to describe the algorithm in detail, a brief overview will now be provided instead.

The same general approach as was used in the two-dimensional case is also used in the three-dimensional version. The 27 regions that are formed by slicing the viewing frustum with hither and yon clipping planes are partitioned into six modules. Each module was designed and optimized to clip a line segment whose endpoints are in particular sets of regions. By parametrizing the modules, the algorithm can exploit the symmetry of the clipping problem; for example, by switching P_0 and P_1 or the hither and yon clipping planes, the six modules are generalized to clip any line segment with endpoints

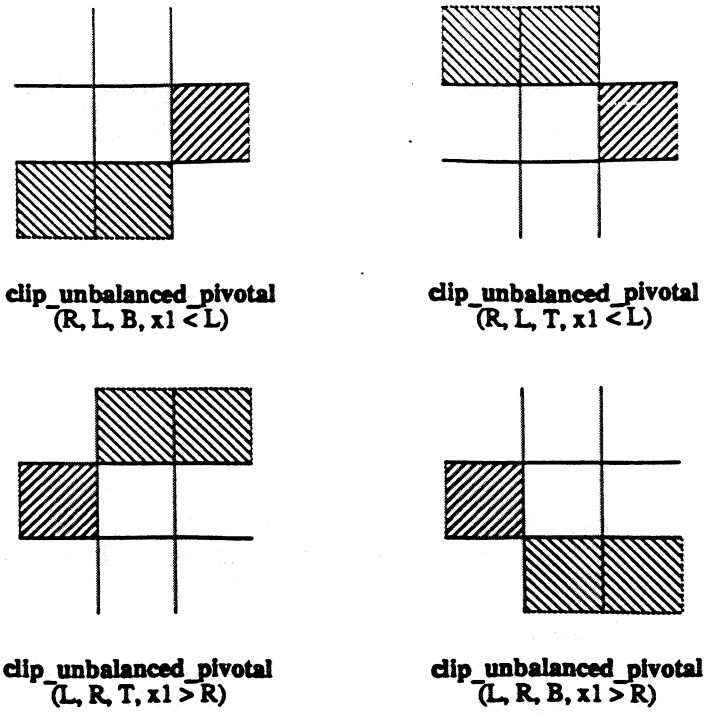


Figure 38: clip_unbalanced_pivotal procedure calling sequences.

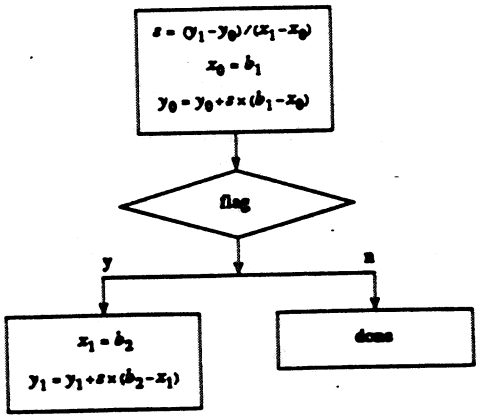


Figure 39: clip_strip (b1, b2, flag).

in any of the regions. The heart of the three-dimensional algorithm is the module that handles the case where both endpoints are between the hither and yon clipping planes. This notable module is simply a slightly modified version of the two-dimensional algorithm.

Again, as with the two-dimensional algorithm, an optimized decision tree is used to map line segments into modules. Each module then makes full use of comparisons, delaying arithmetic operations until such computations are necessary, either to more fully classify endpoints or to clip endpoints to a boundary. The comparisons are based on boundary intersection values, and once the endpoints of a line segment are completely determined, then the same intersection values are then used to calculate the clipped line

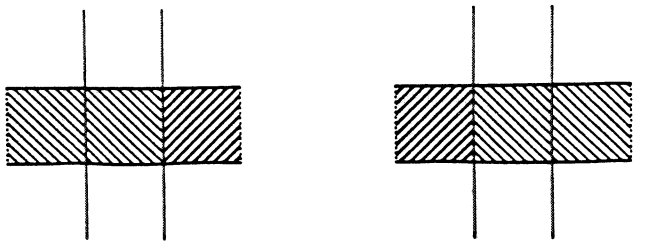


Figure 40: clip_strip procedure calling sequences.

segment.

The three-dimensional version of the new optimal tree algorithm was implemented and its performance was tested. The tables of performance tests below affirm the success of the three-dimensional algorithm; however, the length and complexity of the algorithm mitigate this success. Although fast, the code is also quite lengthy. (Our implementation exceeds 16 pages). In addition to the length, the extensive and not necessarily obvious uses of parametrization and symmetry conspire to cause the algorithm to be difficult to follow and understand. Nonetheless, it is worth mentioning that despite the length of the algorithm, our experience found that due to the modularity, debugging the code was no more difficult than for the two-dimensional algorithm.

XI. Analysis and Performance

XI.1. Performance Tests of the Two-dimensional Algorithms

The two-dimensional optimal tree line clipping algorithm was compared to the traditional Sutherland/Cohen algorithm, to both the original and improved versions of the Liang/Barsky parametric algorithm, as well as to the Nicholl/Lee/Nicholl algorithm, and to the Slater/Barsky subdivision algorithm. All algorithms were coded in Pascal. Pascal code for the two-dimensional case of the Sutherland/Cohen algorithm was copied verbatim from pages 66-67 of.[14] For the two-dimensional case of the Liang/Barsky parametric algorithm, Pascal code for the original version is given in[11] and the improved version is provided in.[13] The comparison with the Nicholl/Lee/Nicholl algorithm used Pascal code supplied by the authors of that algorithm. The appendix of[20] contains Pascal code for the Slater/Barsky subdivision algorithm. Pascal code for the optimal tree algorithm is shown in Appendix I.

The tests were performed both on a Sun 3/160 with a floating point coprocessor under Sun UNIX 4.2 and on a DECStation 5000/200 (MIPS architecture) under Ultrix 4.1.

The algorithms were executed on four different sets of data, each containing a thousand line segments whose endpoints were randomly generated according to uniform distributions. The four sets of data correspond to different sizes of data space; specifically, the four sets contain line segments having endpoints randomly generated from uniform distributions in a square whose base is of size 3000, 5000, 7000, and 9000, respectively. For each data set, the clipping window is a square whose base is of size 1000 and which is located in the middle of the data space. Hence, the four data sets

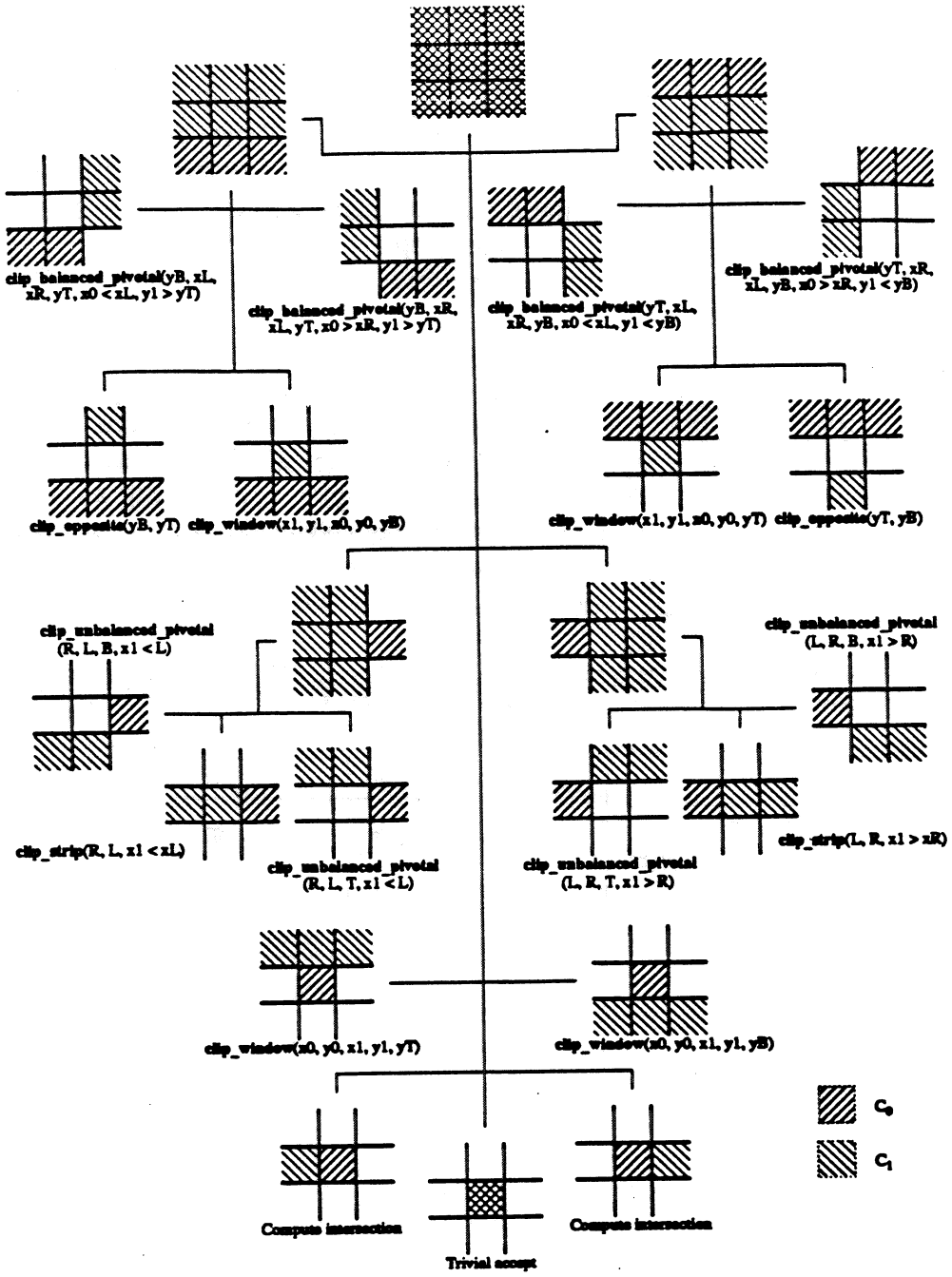


Figure 41: Schematic depiction of the algorithm.

correspond to decreasing ratios of clipping regions to data space.

Each line segment was clipped a thousand times to reduce the effects of random variation. Thus, for each of the four different sizes of data space, a million clipping operations were performed. The results presented here are given in seconds per million clips. Equivalently, this could be interpreted as the mean time, measured in microseconds, to clip a line segment whose endpoints are uniformly distributed as described above. Finally, these results are averaged over the four different sizes of data

space.

Table 1 reports the performance test results for the two-dimensional algorithms executed on the Sun 3/160. The column headings "3000", "5000", "7000", and "9000" are meant to indicate the size of the square in the uniform distribution used to generate the endpoints of the line segments, and "Average" shows the average over the four different sizes of data space. The entries in the columns under the rubric "Time" provide the time in seconds to accomplish a million clipping operations, and the corresponding entries in the columns with the rubric "Imp." indicate the percentage improvement over the Sutherland/Cohen algorithm.

From this, it can be seen that the Sutherland/Cohen algorithm used an average of 383 seconds, the original version of the Liang/Barsky parametric algorithm required an average of 305 seconds (an 20% improvement), the improved version of the Liang/Barsky parametric algorithm used an average of 268 seconds (a 30% improvement), the Nicholl/Lee/Nicholl algorithm was slightly faster, using an average of 264 seconds (a 31% improvement), the Slater/Barsky subdivision algorithm used an average of 161 seconds (a 58% improvement), and finally the new optimal tree algorithm used an average of 128 seconds, which constituted a 67% improvement relative to the Sutherland/Cohen algorithm.

Comparison of the performance tests of the two-dimensional algorithms executed on a Sun 3/160										
	3000		5000		7000		9000		Average	
Algorithm	Time	Imp.	Time	Imp.	Time	Imp.	Time	Imp.	Time	Imp.
Sutherland/Cohen[14]	426	...	378	...	361	...	341	...	376	...
Original Liang/Barsky parametric[11]	358	16%	309	18%	287	21%	277	19%	307	18%
Improved Liang/Barsky parametric[13]	336	21%	272	28%	245	32%	232	32%	271	28%
Nicholl/Lee/Nicholl[15]	294	31%	274	28%	256	29%	248	27%	268	29%
Slater/Barsky[20]	199	53%	163	57%	146	60%	139	59%	162	57%
New Liang/Barsky optimal tree	151	64%	131	66%	119	67%	114	66%	129	66%

Table 1.

The same timing experiments were performed on a DECStation 5000/200 (MIPS architecture) and the results are shown in Table 2. In these timing tests, the Sutherland/Cohen algorithm used an average of 9.14 seconds, the original version of the Liang/Barsky parametric algorithm required an average of 9.81 seconds (a 7% deterioration), the improved version of the Liang/Barsky parametric algorithm used an average of 8.4 seconds (an 8% improvement), the Nicholl/Lee/Nicholl algorithm required an average of 10.48 seconds (a 15% deterioration), the Slater/Barsky subdivision algorithm used an average of 5.51 seconds (a 40% improvement), and finally the new optimal tree algorithm used an average of 4.53 seconds, which constituted a 50% improvement relative to the Sutherland/Cohen algorithm.

The performance times for the MIPS architecture of the DECStation 5000/200 are much quicker than for the 68000-based CISC architecture of the Sun 3/160. The absolute values of the figures are not particularly important, since these are obviously machine dependent. For the DECStation 5000/200, the percentage improvements are less, and

Comparison of the performance tests of the two-dimensional algorithms executed on a DECStation 5000/200 (MIPS architecture)

Algorithm	3000		5000		7000		9000		Average	
	Time	Imp.	Time	Imp.	Time	Imp.	Time	Imp.	Time	Imp.
Sutherland/Cohen[14]	*10.17	...	9.22	...	8.71	...	8.46	...	9.14	...
Original Liang/Barsky parametric[11]	10.94	-8%	9.85	-7%	9.35	-7%	9.10	-8%	9.81	-7%
Improved Liang/Barsky parametric[13]	10.27	-1%	8.45	8%	7.63	12%	7.26	14%	8.40	8%
Nicholl/Lee/Nicholl[15]	11.23	-10%	10.55	-14%	10.17	-17%	9.96	-18%	10.48	-15%
Slater/Barsky[20]	6.25	39%	5.54	40%	5.20	40%	5.05	40%	5.51	40%
New Liang/Barsky optimal tree	4.98	51%	4.54	51%	4.34	50%	4.26	50%	4.53	50%

Table 2.

some of the more recent algorithms actually performed worse than Sutherland-Cohen. The relative change in performance of the algorithms may be due in part to the disparity in the number of procedure call parameters in the various algorithms. Some simple benchmarking experiments showed that procedure call parameters were relatively more expensive on the MIPS architecture (although in absolute terms many times faster) than on the 68000-based CISC architecture. This does indicate that operations counts by themselves are not sufficient to compare the performance of these algorithms and we are currently contemplating alternatives.

XI.2. Performance Tests of the Three-dimensional Algorithms

The tests for the three-dimensional algorithms were performed in the same manner as the tests for the two-dimensional algorithms. The same machines, a Sun 3/160 with a floating point coprocessor under Sun UNIX 4.2 and a DECStation 5000/200 under Ultrix 4.1, were used. Similarly, the algorithms were run on four sets of a thousand line segments whose endpoints were uniformly distributed in a cube whose edge was of size 3000, 5000, 7000, and 9000, respectively, and each line segment was clipped a thousand times. For each data set, the viewing pyramid was fixed and the hither and yon clipping planes were set at 500 and 1000, respectively. Since the four data sets correspond to increasing size of data space, this means that the viewing pyramid occupies a relatively smaller portion of each successive data space. Again, the results presented here are the times in seconds to accomplish a million clipping operations, or equivalently, the mean time per clip, in microseconds, together with the percentage improvement over the Sutherland/Cohen algorithm.

Four algorithms, all implemented in Pascal, were compared. The first one, used as a basis for comparison, was the traditional Sutherland/Cohen algorithm. The Pascal code for the three-dimensional case of this algorithm is based on that provided on page 345 of[14] with the correction explained in[11] and with the addition of hither/yon clipping. The second and third algorithms are the original and improved Barsky/Liang parametric algorithms, respectively. For the three-dimensional case of the Liang/Barsky parametric algorithm, Pascal code for the original version is given in[11] and the improved version is provided in.[13] The fourth algorithm is the three-dimensional version of the optimal tree algorithm presented in this paper.

When the timing tests were performed on the Sun 3/160, the Sutherland/Cohen algorithm used an average of 449 seconds, the original version of the Liang/Barsky parametric algorithm required an average of 306 seconds (a 32% improvement), the improved version of the Liang/Barsky parametric algorithm used an average of 264 seconds (a 41% improvement), and the new optimal tree algorithm used an average of 146 seconds, which represents a 68% improvement over to the Sutherland/Cohen algorithm. These performance test results are summarized in Table 3.

Table 4 reports the performance test results for the three-dimensional algorithms executed on the DECStation 5000/200. In these timing tests, the Sutherland/Cohen algorithm used an average of 8.5 seconds, the original version of the Liang/Barsky parametric algorithm required an average of 8.8 seconds (a 4% deterioration), the improved version of the Liang/Barsky parametric algorithm used an average of 6.5 seconds (a 23% improvement), and the new optimal tree algorithm used an average of 4.0 seconds, which constituted a 53% improvement relative to the Sutherland/Cohen algorithm.

Comparison of the performance tests of the three-dimensional algorithms executed on a Sun 3/160

Algorithm	3000		5000		7000		9000		Average	
	Time	Imp.	Time	Imp.	Time	Imp.	Time	Imp.	Time	Imp.
Sutherland/Cohen[14]	556	...	451	...	390	...	399	...	449	...
Original Liang/Barsky parametric[11]	338	39%	311	31%	295	24%	281	30%	306	32%
Improved Liang/Barsky parametric[13]	321	42%	265	41%	237	39%	233	42%	264	41%
New Liang/Barsky optimal tree	193	65%	146	68%	126	68%	117	71%	146	68%

Table 3.

Comparison of the performance tests of the three-dimensional algorithms executed on a DECStation 5000/200 (MIPS architecture)

Algorithm	3000		5000		7000		9000		Average	
	Time	Imp.	Time	Imp.	Time	Imp.	Time	Imp.	Time	Imp.
Sutherland/Cohen[14]	9.91	...	8.52	...	7.83	...	7.55	...	8.46	...
Original Liang/Barsky parametric[11]	9.99	-1%	8.85	-4%	8.26	-5%	8.06	-7%	8.79	-4%
Improved Liang/Barsky parametric[13]	8.55	14%	6.58	23%	5.71	27%	5.36	29%	6.55	23%
New Liang/Barsky optimal tree	4.89	51%	4.03	53%	3.63	54%	3.43	55%	4.00	53%

Table 4.

XI.3. Results of Performance Tests

For both the two-dimensional and three-dimensional cases, and on both the Sun 3/160 and the DECStation 5000/200, the new algorithm performed uniformly faster than all the other "standard" algorithms for each of the four different sizes of data space.

XII. Conclusion

A new algorithm for line clipping has been developed based on the concept of the optimal tree. A careful analysis results in an algorithm that classifies a given line segment in such a way that at most one procedure is invoked to clip it; furthermore, there are

five such procedures that cover all cases. The result is an algorithm that is provably optimal and according to experimental tests outperforms previous algorithms. For both the two-dimensional and three-dimensional cases, and on both the Sun 3/160 and the DECStation 5000/200, the new algorithm performed uniformly faster than all the other "standard" algorithms for each of four different sizes of data space. Only the two-dimensional case was described in detail. Although in the three-dimensional case this algorithm is significantly faster than the other known algorithms, the code is huge and more complex than the new two-dimensional algorithm, and there are more special cases that need to be handled.

Acknowledgements

The authors are grateful to Geoff Voelker as well as the other members of the Berkeley Clipping Group (Loretta Willis, Scott Drellishak, Cecilia Han, Francis W. Yun, Simon Hui, Alex Ho) for their help in many aspects of this paper. This work was supported in part by a National Science Foundation Presidential Young Investigator Award (number CCR-8451997), and in part by the National Science Foundation under grant number CCR-9015874. The DECStation 5000 workstations were supplied by a grant from Digital Equipment Corporation.

References

[1] R. D. Andreev, "Algorithm for Clipping Arbitrary Polygons," *Computer Graphics Forum*, Vol. 8, No. 3, 1989, pp. 183-91.

[2] James F. Blinn, "A Trip Down the Graphics Pipeline -- Line Clipping," *IEEE Computer Graphics and Applications*, Vol. 11, No. 1, January, 1991, pp. 98-105.

[3] James F. Blinn and Martin E. Newell, "Clipping Using Homogeneous Coordinates," pp. 245-251 in *SIGGRAPH '78 Conference Proceedings*, ACM, August, 1978.

[4] Eric A. Brewer and Brian A. Barsky, "Clipping After Projection: An Improved Perspective Pipeline." Submitted for publication.

[5] Mike Cyrus and Jay Beck, "Generalized Two- and Three-Dimensional Clipping," *Computers and Graphics*, Vol. 3, No. 1, 1978, pp. 23-28.

[6] Michael Dorr, "A New Approach to Parametric Line Clipping," *Computers and Graphics*, Vol. 14, No. 3/4, 1990, pp. 449-464.

[7] V. J. Duvanenko, W. E. Robbins, and R. S. Gyurcsik, "Improving Line Segment Clipping," *Dr. Dobb's Journal of Software Tools*, Vol. 15, No. 7, July, 1990, pp. 36,38,40,42,44-5,98,100.

[8] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Company, 1990. Second Edition.

[9] You-Dong Liang and Brian A. Barsky, "An Analysis and Algorithm for Polygon Clipping," *Communications of the ACM*, Vol. 26, No. 11, November, 1983, pp. 868-877.Corrigendum in *Communications of the ACM*, Vol. 27, No. 4, April 1984, p. 383.

[10] You-Dong Liang and Brian A. Barsky, "Introducing A New Technique for Line Clipping," pp. 548-559 in *Proceedings of the International Conference on Engineering and Computer Graphics*, Beijing, 27 August - 1 September 1984. Also in *Journal of Zhejiang University* Special Issue on Computational Geometry, 1984, pp.1-12.

[11] You-Dong Liang and Brian A. Barsky, "A New Concept and Method for Line Clipping," *ACM Transactions on Graphics*, Vol. 3, No. 1, January, 1984, pp. 1-22.

[12] You-Dong Liang and Brian A. Barsky, "An Improved Parametric Line Clipping Algorithm," pp. 405-424 in *Algorithms and Parallel VLSI Architectures*, ed. Deprettere, Ed F. and van der Veen, Alle-Jan, Elsevier Science Publishers, Amsterdam, 1991. Volume B. Conference held 10-16 June 1990 in Pont-à-Mousson, France.

[13] You-Dong Liang, Brian A. Barsky, and Mel Slater, *Some Improvements to a Parametric Line Clipping Algorithm*, Technical Report No. UCB/CSD 92/688, Computer Science Division, Electrical Engineering and Computer Sciences Department, University of California, Berkeley, California, USA, May, 1992.

[14] William M. Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, 1979. Second edition.

[15] Tina M. Nicholl, D. T. Lee, and Robin A. Nicholl, "An Efficient New Algorithm for 2-D Line Clipping: Its Development and Analysis," pp. 253-262 in *SIGGRAPH '87 Conference Proceedings*, ACM, Anaheim, July 27-31, 1987.

[16] Tina M. Nicholl and Robin A. Nicholl, "Performing Geometric Transformations by Program Transformation," *ACM Transactions on Graphics*, Vol. 9, No. 1, January, 1990, pp. 28-40.

[17] Ari Rappaport, "An Efficient Algorithm for Line and Polygon Clipping," *The Visual Computer*, Vol. 7, No. 1, 1991, pp. 19-28.

[18] K. K. Shi, J. A. Edwards, and D. C. Cooper, "An Efficient Line Clipping Algorithm," *Computers and Graphics*, Vol. 91, No. 2, 1990, pp. 297-301.

[19] Vaclav Skala, "Algorithms for 2D Line Clipping," pp. 355-366 in *Proceedings of Eurographics '89*, ed. Hansmann, W., Hopgood, F. Robert A.; and Strasser, Wolfgang, North-Holland, Amsterdam, Hamburg, 1989.

[20] Mel Slater and Brian A. Barsky, "2D Line and Polygon Clipping Based on Space Subdivision." Submitted for publication.

[21] M. S. Sobkow, P. Pospilsil, and Y.-H. Yang, "A Fast Two-dimensional Line Clipping Algorithm via Line Encoding," *Computers and Graphics*, Vol. 11, No. 4, 1987, pp. 459-467.

[22] Robert F. Sproull and Ivan E. Sutherland, "A Clipping Divider," pp. 765-775 in *Proceedings of the Fall Joint Computer Conference, Vol. 33*, AFIPS, Thompson Books, Washington, D.C., 1968.

[23] Robert B. Tilove, "Line/Polygon Classification: A Study of the Complexity of Geometrical Classification," *IEEE Computer Graphics and Applications*, Vol. 1, No. 2, April, 1981, pp. 75-86.

Appendix I: Actual Pascal code of the Two-dimensional Algorithm

```
#define setP0(expr1, expr2) \
begin \
  x0 := (expr1);  \
  y0 := (expr2) \
end

#define setP1(expr1, expr2) \
begin \
  x1 := (expr1);  \
  y1 := (expr2) \
end

program optimal_tree_clip(input, output);
var x0, y0, x1, y1, rx0, ry0, rx1, ry1, top, bottom, right, left : real;
    reject : boolean;
    clipNum, j : integer;
(*
 * Clip the line segment represented by (x0, y0), (x1, y1).
 * The clipping window is a rectangle with sides 'left', 'top',
 * 'right', and 'bottom'.  'reject' is true if the line segment
 * does not intersect the clipping window.
 *)
  procedure subclip(var x0, y0, x1, y1 : real; var reject : boolean);

  procedure clip_strip(b1, b2 : real; flag : boolean);
  var slope : real;
  begin
    slope := (y1 - y0) / (x1 - x0);           (* slope of P0-P1 line *)
    y0 := y0 + slope * (b1 - x0);             (* clip y0 to b1 *)
    x0 := b1;                                 (* set x0 to b1 *)
    if flag then begin                        (* clip P1 to b2 if outside b2 *)
      y1 := y1 + slope * (b2 - x1);
      x1 := b2;
    end
  end;

  procedure clip_window(var x_window, y_window, x_strip, y_strip : real; b1 : real);
  var invslope : real;
  begin
    invslope := (x_strip - x_window) / (y_strip - y_window);(*inverse slope of P0P1 line*)
    x_strip := x_window + invslope * (b1 - y_window); (* clip x_strip to b1 *)
    if x_strip < left then begin
      x_strip := left;
      y_strip := y_window + (left - x_window) / invslope)
    end
    else if x_strip > right then begin
      x_strip := right;
      y_strip := y_window + (right - x_window) / invslope)
    end
    else y_strip := b1
  end;

  procedure clip_unbalanced_pivotal(b2, b3, b4 : real; flag : boolean);
  var d, b, c, dx, dy : real;
  begin
    dx := x1 - x0;
    dy := y1 - y0;
    d := dx * (b4 - y1);
    b := dy * (b2 - x1);
    if abs(d) > abs(b) then reject := true
    else begin
      setP0(b2, y1 + b/dx);     (* clip P0 to b2 *)
      if flag then begin
        c := (b3 - x1) * dy;
```

```
      if abs(c) > abs(d) then setP1(b3, y1 + c/dx)
      else setP1(x1 + d/dy, b4)
    end
    else setP1(x1 + d/dy, b4)
  end
end;

procedure clip_balanced_pivotal(b1, b2, b3, b4 : real; flag0, flag1 : boolean);
var dx, dy, a, b, c, d, x0val, y0val : real;
begin
  dx := x1 - x0;
  dy := y1 - y0;
  x0val := x0;  (* x0 and y0 must be saved because their values are  *)
  y0val := y0;  (* used in calculations after they have been clipped *)
  a := dx * (b1 - y0);
  if flag0 then begin       (* P0 outside both b1 and b2 *)
    if flag1 then begin     (* P1 outside both b3 and b4 *)
      b := dy * (b2 - x0);
      if abs(a) > abs(b) then begin
        c := dy * (b3 - x0);
        if abs(a) > abs(c) then reject := true
        else begin
          d := (b4 - y0) * dx;  (* to test P1 against b3 and b4 *)
          setP0(x0 + a/dy, b1); (* clip P0 to b1 *)
        end
      end
      else begin
        d := (b4 - y0) * dx;
        if abs(b) > abs(d) then reject := true
        else begin
          c := dy * (b3 - x0);  (* to test P1 against b3 and b4 *)
          setP0(b2, y0 + b/dx)  (* clip P0 to b2 *)
        end;
      end;
      if not reject then
        if abs(c) > abs(d) then setP1(x0val + d/dy, b4)
        else setP1(b3, y0val + c/dx);
    end
    else begin    (* P0 outside b1 and b2, P1 outside b3 but inside b4 *)
      c := (b3 - x0) * dy;
      if abs(a) > abs(c) then reject := true
      else begin
        setP1(b3, y0 + c/dx);    (* clip P1 to b3 *)
        d := (b2 - x0val) * dy;
        if abs(a) > abs(d) then setP0(x0 + a/dy, b1)
        else setP0(b2, y0 + d/dx)
      end
    end
  end
  else begin  (* P0 outside b1 but inside b2 *)
    c := (b3 - x0) * dy;
    if abs(a) > abs(c) then reject := true
    else begin
      setP0(x0 + a/dy, b1);
      if flag1 then begin  (* P1 outside b3 and b4 *)
        d := (b4 - y0val) * dx;
        if abs(c) > abs(d) then setP1(x0val + d/dy, b4)
        else setP1(b3, y0val + c/dx)
      end
      else setP1(b3, y0val + c/dx) (* we know to clip P1 to b3 *)
    end
  end
end;
```

```
procedure clip_opposite(b1, b2 : real);
var invslope : real;
begin
  invslope := (x1 - x0)/(y1 - y0);
  x1 := x0 + invslope * (b2 - y0);
  if (x1 < left) or (x1 > right) then reject := true
  else begin
    y1 := b2;
    x0 := x0 + invslope * (b1 - y0);
    if x0 < left then setP0(left, y1 + (left - x1) / invslope)
    else if x0 > right then setP0(right, y1 + (right - x1) / invslope)
         else y0 := b1;
  end
end;

begin  (* subclip *)
  if y0 < bottom then
    if y1 < bottom then reject := true
    else if x1 < left then
      if x0 < left then reject := true
      else clip_balanced_pivotal(bottom, right, left, top, x0>right, y1>top)
    else if x1 > right then
      if x0 > right then reject := true
      else clip_balanced_pivotal(bottom,left,right,top,x0<left,y1>top)
    else if y1 > top then clip_opposite(bottom,top)
    else clip_window(x1, y1, x0, y0, bottom)
  else if y0 > top then
    if y1 > top then reject := true
    else if x1 < left then
      if x0 < left then reject := true
      else clip_balanced_pivotal(top,right,left,bottom,x0 > right,y1 < bottom)
    else if x1 > right then
      if x0 > right then reject := true
      else clip_balanced_pivotal(top,left,right,bottom,x0<left,y1<bottom)
    else if y1 < bottom then clip_opposite(top,bottom)
    else clip_window(x1,y1,x0,y0,top)
  else if x0 > right then
    if x1 > right then reject := true
    else if y1 > top then clip_unbalanced_pivotal(right,left,top,x1<left)
    else if y1 < bottom then clip_unbalanced_pivotal(right,left,bottom,x1<left)
    else clip_strip(right,left,x1<left)
  else if x0 < left then
    if x1 < left then reject := true
    else if y1 > top then clip_unbalanced_pivotal(left,right,top,x1>right)
    else if y1 < bottom then clip_unbalanced_pivotal(left,right,bottom,x1>right)
    else clip_strip(left,right,x1>right)
  else if y1 < bottom then clip_window(x0,y0,x1,y1,bottom)
  else if y1 > top then clip_window(x0,y0,x1,y1,top)
  else if x1 > right then begin
    y1 := y0 + (right - x0) * (y1 - y0)/(x1 - x0);
    x1 := right;
  end
  else if x1 < left then begin
    y1 := y0 + (left - x0) * (y1 - y0)/(x1 - x0);
    x1 := left;
  end
end;  (* subclip *)

begin  (* optimal_tree_clip *)
   left := 0.0;  right := 1000.0;
   bottom := 0.0;  top := 1000.0;
   clipNum := 1;
   reject := false;
   while not eof do begin
```

```
          readln(rx0, ry0, rx1, ry1);
          for j := 1 to clips do begin
                x0 := rx0;  y0 := ry0;
                x1 := rx1;  y1 := ry1;
                subclip(x0, y0, x1, y1, reject);
#ifdef printresults
                if reject then reject := false
                else begin
                    write(clipNum:5, '.', x0:11:3, y0:11:3);
                    writeln(x1:11:3, y1:11:3);
                end;
#endif
          end;
          clipNum := clipNum + 1
     end
end.
```

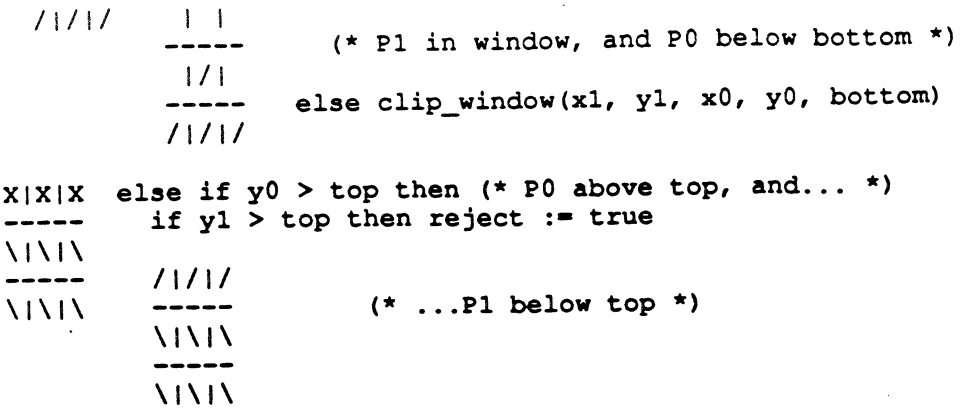
Appendix II: Fully Annotated Version of the Program Code for the Two-dimensional Algorithm

```
#define setP0(expr1, expr2) \
begin \
  x0 := (expr1);  \
  y0 := (expr2) \
end

#define setP1(expr1, expr2) \
begin \
  x1 := (expr1);  \
  y1 := (expr2) \
end

program optimal_tree_clip(input, output);
var x0, y0, x1, y1, rx0, ry0, rx1, ry1, top, bottom, right, left : real;
    reject : boolean;
    clipNum, j : integer;

(*
 * Clip the line segment represented by (x0, y0), (x1, y1).
 * The clipping window is a rectangle with sides 'left', 'top',
 * 'right', and 'bottom'.  'reject' is true if the line segment
 * does not intersect the clipping window.
 *)
  procedure subclip(var x0, y0, x1, y1 : real; var reject : boolean);


(*                                                                    | |
 * P0 is outside of b1, in the middle.  If 'flag' is true, then P1   -----
 * is outside of b2, in the middle, otherwise P1 is in the window.   0|1|1
 * b1 is nominally 'left' and b2 'right', but works in the opposite  -----
 * case.  Would work for any rotation if (x0, y0) and (x1, y1) were   | |
 * parametrized, but would not be used by the algorithm.             b1 b2
 *)
  procedure clip_strip(b1, b2 : real; flag : boolean);
  var slope : real;
  begin
    slope := (y1 - y0) / (x1 - x0);          (* slope of P0-P1 line *)
    y0 := y0 + slope * (b1 - x0);            (* clip y0 to b1 *)
    x0 := b1;                                (* set x0 to b1 *)
    if flag then begin                       (* clip P1 to b2 if outside b2 *)
      y1 := y1 + slope * (b2 - x1);
      x1 := b2;
    end
  end;

(*                                                                  | |
 * P0 is in the clip window.  P1 is outside b1 in any of the       -----
 * three regions.  b1 is nominally 'bottom', but works when         |0|
 * b1 is top.  Would work for any boundary if 'left' and 'right'   ----- b1
 * were parametrized, but would not be used by the algorithm.      1|1|1
 *)
  procedure clip_window(var x_window, y_window, x_strip, y_strip : real; b1 : real);
  var invslope : real;
  begin
    invslope := (x_strip - x_window) / (y_strip - y_window);(*inverse slope of P0P1 li
    x_strip := x_window + invslope * (b1 - y_window); (* clip x_strip to b1 *)
(* if P1 is in either of the corner regions outside b1, clip y_strip to the
 * side boundaries, otherwise clip y_strip to b1 *)
    if x_strip < left then begin
      x_strip := left;
      y_strip := y_window + (left - x_window) / invslope)
    end
    else if x_strip > right then begin
      x_strip := right;
```

```
      y_strip := y_window + (right - x_window) / invslope)
    end
    else y_strip := b1
  end;

(*                                                                    1|1|
 * P0 is outside b2, in the middle.  P1 is above b4, and             ----- b4
 * inside b2.  b2 is nominally 'right', b3 'left', and b4 'top',      | |0
 * but works for any combination in which b2 is either 'left'        -----
 * or 'right'.                                                        | |
 *                                                                   b3 b2
 *)
  procedure clip_unbalanced_pivotal(b2, b3, b4 : real; flag : boolean);
  var d, b, c, dx, dy : real;
  begin
    dx := x1 - x0;
    dy := y1 - y0;
    d := dx * (b4 - y1);
    b := dy * (b2 - x1);
(* if the slope of the P0-P1 line is greater than the slope of the line
 * from P1 to the intersection of b2 and b4, then the P0-P1 line does not
 * intersect the clip window *)
    if abs(d) > abs(b) then reject := true
    else begin
      setP0(b2, y1 + b/dx);     (* clip P0 to b2 *)
      if flag then begin
        c := (b3 - x1) * dy;
(* if the slope of the P0-P1 line is smaller than the slope of the line
 * from P1 to the intersection of b3 and b4, then clip P1 to b3, otherwise
 * clip P1 to b4 *)
        if abs(c) > abs(d) then setP1(b3, y1 + c/dx)
        else setP1(x1 + d/dy, b4)
      end
      else setP1(x1 + d/dy, b4)
    end
  end;

(*                                                                    1| |
 * P0 is outside b1, but inside b3, and P1 is outside b3 and         ----- b4
 * inside b1.  'flag0' is true if P0 is outside both b1 and b2,      1| |
 * and 'flag1' is true if P1 is outside both b3 and b4.              ----- b1
 * b1 is nominally bottom, b2 right, b3 left, and b4 top.             |0|0
 *                                                                   b3 b2
 *)
  procedure clip_balanced_pivotal(b1, b2, b3, b4 : real; flag0, flag1 : boolean);
  var dx, dy, a, b, c, d, x0val, y0val : real;
  begin
    dx := x1 - x0;
    dy := y1 - y0;
    x0val := x0;  (* x0 and y0 must be saved because their values are  *)
    y0val := y0;  (* used in calculations after they have been clipped *)
    a := dx * (b1 - y0);
    if flag0 then begin       (* P0 outside both b1 and b2 *)
      if flag1 then begin     (* P1 outside both b3 and b4 *)
        b := dy * (b2 - x0);
(* if the slope of the P0-P1 line is smaller than the slope of the line
 * from P0 to the intersection of b1 and b2 then... *)
        if abs(a) > abs(b) then begin
          c := dy * (b3 - x0);
(* if the P0-P1 slope is smaller than the slope of the line from P0 to the
 * intersection of b1 and b3, then the P0-P1 line does not intersect
 * the clip window;  otherwise, clip P0 to b1 and set up test to see if
 * P1 should be clipped to b3 or to b4 *)
          if abs(a) > abs(c) then reject := true
```

```
          else begin
            d := (b4 - y0) * dx;  (* to test P1 against b3 and b4 *)
            setP0(x0 + a/dy, b1); (* clip P0 to b1 *)
          end
        end
(* if the slope of the P0-P1 line is greater than the slope of the line
 * from P0 to the intersection of b1 and b2 then... *)
        else begin
          d := (b4 - y0) * dx;
(* if the P0-P1 slope is greater than the slope of the line from P0 to the
 * intersection of b2 and b4, then the P0-P1 line does not intersect
 * the clip window;  otherwise, clip P0 to b2 and set up test to see if
 * P1 should be clipped to b3 or to b4 *)
          if abs(b) > abs(d) then reject := true
          else begin
            c := dy * (b3 - x0);  (* to test P1 against b3 and b4 *)
            setP0(b2, y0 + b/dx)  (* clip P0 to b2 *)
          end
        end;
        if not reject then
(* if the P0-P1 slope is greater than the slope of the line from P0 to the
 * intersection of b3 and b4, then clip P1 to b4, otherwise clip P1 to b3 *)
          if abs(c) > abs(d) then setP1(x0val + d/dy, b4)
          else setP1(b3, y0val + c/dx);
      end
      else begin   (* P0 outside b1 and b2, P1 outside b3 but inside b4 *)
        c := (b3 - x0) * dy;
(* if the P0-P1 slope is smaller than the slope of the line from P0 to the
 * intersection of b1 and b3, then the P0-P1 line does not intersect
 * the clip window;  otherwise, clip P1 to b3 *)
        if abs(a) > abs(c) then reject := true
        else begin
          setP1(b3, y0 + c/dx);   (* clip P1 to b3 *)
          d := (b2 - x0val) * dy;
(* if the P0-P1 slope is smaller than the slope of the line from P0 to the
 * intersection of b1 and b2, then clip P0 to b1, otherwise clip P0 to b2 *)
          if abs(a) > abs(d) then setP0(x0 + a/dy, b1)
          else setP0(b2, y0 + d/dx)
        end
      end
    end
    else begin  (* P0 outside b1 but inside b2 *)
      c := (b3 - x0) * dy;
(* if the P0-P1 slope is smaller than the slope of the line from P0 to the
 * intersection of b1 and b3, then the P0-P1 line does not intersect
 * the clip window; otherwise, clip P0 to b1 and determine where to clip P1 *)
      if abs(a) > abs(c) then reject := true
      else begin
        setP0(x0 + a/dy, b1);
        if flag1 then begin  (* P1 outside b3 and b4 *)
          d := (b4 - y0val) * dx;
(* if the P0-P1 slope is larger than the slope of the line from P0 to the
 * intersection of b3 and b4 then clip P1 to b4, otherwise clip P1 to b3 *)
          if abs(c) > abs(d) then setP1(x0val + d/dy, b4)
          else setP1(b3, y0val + c/dx)
        end
        else setP1(b3, y0val + c/dx) (* we know to clip P1 to b3 *)
      end
    end
  end;

(*                                                              |1|
 * P0 is outside b1, and P1 is outside b2 in the middle.       -----
 * b1 is nominally 'bottom' and b2 'top', but also serves       | |
```

```
 * in reverse.                                                    -----
 *                                                               0|0|0
 *)
  procedure clip_opposite(b1, b2 : real);
  var invslope : real;
  begin
    invslope := (x1 - x0)/(y1 - y0);
(* clip x1 to b2;  if it is less than left or greater than right then
 * the line P0-P1 does not intersect the clip window *)
    x1 := x0 + invslope * (b2 - y0);
    if (x1 < left) or (x1 > right) then reject := true
    else begin
      y1 := b2;
(* clip x0 to b1;  if it is less than left, then clip P0 to left; if
 * it is greater than right, then clip P0 to right;  otherwise, just
 * set y0 to b1 *)
      x0 := x0 + invslope * (b1 - y0);
      if x0 < left then setP0(left, y1 + (left - x1) / invslope)
      else if x0 > right then setP0(right, y1 + (right - x1) / invslope)
           else y0 := b1;
    end
  end;

  begin  (* subclip *)  X|X|X     (* P0 in region      --  '/'
                        -----        P1 in region      --  '\'
                        X|X|X        Both in region    --  'X' *)
                        -----
                        X|X|X

 \|\|\  if y0 < bottom then
 -----    if y1 < bottom then reject := true
 \|\|\
 -----    \|\|\
 X|X|X    -----
          \|\|\          (* P0 below bottom, and P1 above bottom *)
          -----
          /|/|/

   \| |   else if x1 < left then   (* P1 above bottom, left of left, and... *)
   -----    if x0 < left then reject := true
   \| |
   -----
   /|/|/   \| |
           -----      (* ...P0 below bottom, right of left. *)
           \| |
           -----   else clip_balanced_pivotal(bottom, right, left, top,
            |/|/                          x0 > right, y1 > top)

    | |\  else if x1 > right then (* P1 above bottom, right of right, and... *)
   -----    if x0 > right then reject := true
    | |\
   -----
   /|/|/    | |\
           -----      (* ...P0 below bottom, left of right *)
            | |\
           -----   else clip_balanced_pivotal(bottom, left, right, top,
           /|/|                          x0 < left, y1 > top)

       (* P1 in the middle, above top, and P0 below bottom *)

    |\|   else if y1 > top then clip_opposite(bottom,top)
   -----
    | |
   -----
```



```
                (* P1 in window, and P0 below bottom *)

                else clip_window(x1, y1, x0, y0, bottom)

else if y0 > top then (* P0 above top, and... *)
  if y1 > top then reject := true

                (* ...P1 below top *)

else if x1 < left then   (* P1 below top, left of left, and... *)
  if x0 < left then reject := true

                (* ...P0 above top, right of left *)

else clip_balanced_pivotal(top, right, left, bottom,
                           x0 > right, y1 < bottom)

else if x1 > right then (* P1 below top, right of left, and... *)
  if x0 > right then reject := true

                (* ...P0 above top, left of right *)

else clip_balanced_pivotal(top, left, right, bottom,
                           x0 < left, y1 < bottom)

      (* P1 in middle, below bottom, and P0 above top *)

else if y1 < bottom then clip_opposite(top, bottom)

                (* ...P1 in window, and P0 above top *)

                else clip_window(x1, y1, x0, y0, top)

else if x0 > right then  (* P0 in middle, right of right, and... *)
  if x1 > right then reject := true

                (* ...P1 left of right *)

   (* P0 in middle, right of right, and P1 above top, left of right *)

else if y1 > top then clip_unbalanced_pivotal(right, left, top, x1 < left)

   (* P0 in middle, right of right, and P1 below bottom, left of right *)
```

```
   | |
  -----
   | |/  else if y1 < bottom then clip_unbalanced_pivotal(right, left, bottom, x1 < left)
  -----
  \|\|

   (* P0 in middle, right of right, and P1 in middle, left of right *)
   | |
  -----
  \|\|/  else clip_strip(right, left, x1 < left)
  -----
   | |

\|\|\  else if x0 < left then   (* P0 in middle, left of left, and... *)
-----    if x1 < left then reject := true
X|\|\
-----     |\|\
\|\|\    -----        (* P1 right of left *)
         /|\|\
         -----
          |\|\

   (* P0 in middle, left of left, and P1 above top, right of left *)
   |\|\
  -----
  /| |   else if y1 > top then clip_unbalanced_pivotal(left, right, top, x1 > right)
  -----
   | |

   (* P0 in middle, left of left, and P1 below bottom, right of left *)
   | |
  -----
  /| |   else if y1 < bottom then clip_unbalanced_pivotal(left,right,bottom,x1>right)
  -----
   |\|\

   (* P0 in middle, left of left, and P1 in middle, right of left *)
   | |
  -----
  /|\|\  else clip_strip(left, right, x1 > right)
  -----
   | |

 (* P0 in window, and P1 below bottom *)
 | |
-----
 |/|   else if y1 < bottom then clip_window(x0, y0, x1, y1, bottom)
-----
\|\|\

 (* P0 in window, and P1 above top *)
\|\|\
-----
 |/|   else if y1 > top then clip_window(x0, y0, x1, y1, top)
-----
 | |

 (* P0 in window, and P1 in middle, right of right *)
 | |   else if x1 > right then begin
-----    y1 := y0 + (right - x0) * (y1 - y0)/(x1 - x0);
 |/|\    x1 := right;
-----  end
 | |
```

```
 (* P0 in window, and P1 in middle, left of left *)
 | |   else if x1 < left then begin
-----    y1 := y0 + (left - x0) * (y1 - y0)/(x1 - x0);
\|/|     x1 := left;
-----  end
 | |
                | |
               -----
                |X|      (* P0 and P1 in window, do nothing *)
               -----
                | |

 end;  (* subclip *)

 begin  (* optimal_tree_clip *)
    left := 0.0;  right := 1000.0;
    bottom := 0.0;  top := 1000.0;
    clipNum := 1;
    reject := false;
    while not eof do begin
         readln(rx0, ry0, rx1, ry1);
         for j := 1 to clips do begin
               x0 := rx0;  y0 := ry0;
               x1 := rx1;  y1 := ry1;
               subclip(x0, y0, x1, y2, reject);
#ifdef printresults
               if reject then reject := false
               else begin
                   write(clipNum:5, '.', x0:11:3, y0:11:3);
                   writeln(x1:11:3, y1:11:3);
               end;
#endif
         end;
         clipNum := clipNum + 1
     end
end.
```