Advances in Computer Science and Information Technology (ACSIT)
Print ISSN: 2393-9907; Online ISSN: 2393-9915; Volume 2, Number 4; April-June, 2015 pp. 342-346
© Krishi Sanskriti Publications
http://www.krishisanskriti.org/acsit.html

2D Line Clipping Algorithm Using Pixel Color Attribute

Soumya Chakravarty[1] and Arijit Sil[2]

[1,2]Meghnad Saha Institute of Technology Madurdaha, Beside Urbana Complex Uchhepota, Kolkata 700 150
E-mail: [1]soumya.chakravarty@yahoo.com, [2]arijitsil2005@gmail.com

Abstract—*This paper presents an algorithm for clipping a line segment in a 2D plane against a rectangular clipping window. It initially uses two trivial tests for detecting the line segments that are completely within the boundaries of the clipping region or falls along the boundaries of the clipping rectangle, otherwise, changes a particular pixel property along the line segment and uses a routine that eventually returns either a single or two intersection points of the line segment with the clipping window by comparing the pixel property at the point of intersection. If the routine is unable to return any intersection point, it suggests that the line segment is completely outside the window.*

Keywords: *Computer Graphics; Line Clipping; Algorithm; Pixel; Color Attribute;*

1. INTRODUCTION

Removing a portion of an image outside of an area of interest is known as clipping. In computer graphics, line clipping is a basic and important operation which identifies those lines or portion of lines that are either inside or outside of a specified region of space. For example, extracting a part of a defined scene for viewing is inherently performed by line clipping. The region against which the object is to be clipped is called a clip window which is usually rectangular in shape with sides parallel to the coordinate axes. Line clipping is useful in Geographic Information System, VLSI circuits design, designing building architecture, to mention a few.

The early and classical algorithms [5] of line clipping are Cohen-Sutherland Line Clipping Algorithm [1] [6] [7], Cyrus-Beck Line Clipping Algorithm [2] [6] [7], Liang-Barsky Line Clipping Algorithm [3] [6] [7] and Nicholl-Lee-Nicholl Line Clipping Algorithm [4] [6] [7].

Cohen-Sutherland Line Clipping algorithm [1] is one of the oldest and most popular line-clipping procedures. The algorithm uses a rectangular window with a coding scheme to subdivide the two-dimensional space into nine regions which includes the clipping window. Each endpoint of a line segment is assigned a code of the sub-region in which it lies. To begin with, Cohen-Sutherland [1] algorithm quickly removes the lines that are completely outside the window and also detects lines that are completely inside the window. For lines that intersect the window boundaries, the algorithm repeatedly tests and finds point of intersection of the line segment with each of the window boundaries until the inside test is satisfied.

[8] forwards an improved algorithm based on Cohen–Sutherland which by the coordinate values of clipping window vertexes and the implicit equation of the line segment can rapidly judge which clipping window edge has real intersection point(s) with the line segment. In [9] the line-clipping algorithm QuickClip is proposed which is simpler and more concise than CS algorithm.

Later, M.Cyrus and J.Beck proposed Cyrus –Beck algorithm [2]. The algorithm treats a line in parametric form. 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. Besides, for the general case (the line segments will cross all the boundaries of the window), the algorithm first makes computations and find the parameters of the intersection points. Then, according to the signs of the denominators of the parameters and the values of the parameters, the algorithm determines which part of the line segment is inside the window.

Independently, You-dong Liang, Brian A.Barsky, offered a faster algorithm [3]. This algorithm is based on a parametric representation of the line segment. The algorithm is somewhat complicated and inefficient. To clip a line segment which is neither vertical nor horizontal, it will perform 16 comparisons, 7 subtractions, and 4 divisions.

Nicholl-Lee-Nicholl [4], in order to replace division by multiplication, uses a slightly modified version of the algorithm used by Cyrus-Beck and Liang-Barsky. So, for the algorithm, finding the intersection points are efficient, but finding the positions of the two endpoints of the line segment is more complicated than Cohen-Sutherland Line Clipping algorithm. Though the NLN algorithm is more efficient than the CS and LB algorithm but since there is a large number of cases to be considered; development, testing and validation of the algorithm are difficult.

Authors of [10] present an algorithm for line clipping based on the concept of optimal tree. In [11] the proposed algorithm does not use predicates like NLN algorithm, but calculates intersections speculatively. The approach in [12] does not require a division operation and uses homogeneous coordinates for input and output point representation. Proposed approach for [13] applies affine transformations to the line segments and the window, and changes the slopes of the line segments and the shape of the window.

In this paper a new approach for 2D line clipping against rectangular clipping window is proposed which does not require implicit area codes, redundant calculations and explicit calculation reusing. The algorithm considers different categories of lines for solution. Here, the region of line in a rectangle window is not considered for categorization rather relationship of line and the clipping rectangle is used for selecting categories. Initially, the algorithm performs a couple of trivial tests to detect the line segment which falls completely within the clipping window or falls along any of the four clipping edges. Otherwise, the color property of all the pixels along the line segment is set to a predefined value and a routine is called which uses the color attribute of each pixel on clipping boundaries to find the intersection points with the line segment. Resulting intersection points are joined to get the clipped line. If, no intersection points are available then the line is outside the clipping region. Since the lines are categorized dynamically, according to the two end points of the line, this process is able to trivially reject or accept a line even before doing any clipping calculation. This approach leads to a solution that is simpler, robust, and easy to implement.

2. METHODOLOGY

In this paper a new 2D Line Clipping Algorithm is proposed which has explored the use of color attribute of pixels in designing a clipping method for straight lines that reduces the computation overhead involved in case of the previous approaches.

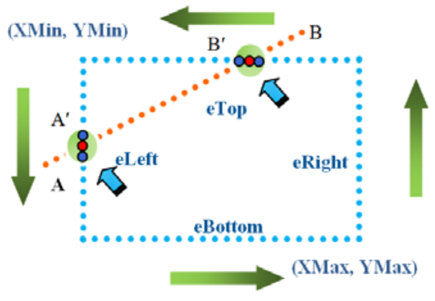

Fig. 1: Basic Diagram

The algorithm considers a rectangular clipping region with (XMin, YMin) and (XMax, YMax) being the two diagonal points on a 2D plane. A line segment AB having end point coordinates (X1, Y1) and (X2, Y2) is to be clipped against the clipping boundaries. To achieve this, the algorithm needs to find the intersection points of the line segment with all the clipping edges. At most there will be two intersection points. If no intersection point can be identified then the line segment falls either completely inside or outside of the clipping region.

In the first step the positions of the end point coordinates of the input line segment are checked with respect to the rectangular region boundaries using two inequality conditions. The first one checks whether the line segment directly falls on any of the edges of the clipping rectangle or not. If the line segment is found to be running along any clipping edge, it is discarded and the algorithm stops without producing any clipped line. In the second test if it is detected that both the end points are residing within the clipping rectangle then the line segment is completely inside the clipping area and no further processing is required and the algorithm stops after redrawing the original line segment as the resultant line.

Otherwise, in the next step the procedure checks further whether only one of the end points falls within the clipping region or not. If it finds such a point then it is assigned to a point object *pointInside*.

Then the original straight line is redrawn by setting the color property of all the pixels present along it to a predefined color value say LineColor (Red in Fig. 1) which is different from the color say ClippingWindowColor (Blue in Fig. 1) used to draw the clipping rectangle.

Next, starting from top left corner point (XMin, YMin) of the rectangle, the algorithm uses an iterative procedure to scan every pixel along all the four clipping edges (in the order eLeft, eBottom, eRight, eTop) to identify points having pixel color attribute value different from clipping edge color and matches with the current pixel color of the straight line. As and when encountered, those points are marked as the intersection points of the line segment with the clipping rectangle.

At the end of the iteration process if only one such point is identified then the line segment has intersected the clipping window only once and one or both the end points must be residing inside or outside the clipping window. The algorithm then checks the value of the object *pointInside*. If the value of the object is not null then it is joined with the intersection point to generate the clipped line. Otherwise, no clipped line is generated.

If exactly two intersection points are found then a line segment joining these two points represents the resulting clipped line.

If more than two intersection points are encountered during scanning a particular clipping edge - which occurs when a part of the line grazes the clipping edge in question (acute angel between the line and the edge is between zero to five degree), the algorithm picks up only one of the intersection points and considers it as the only point of intersection between that clipping edge and the straight line.

In case there are no point of intersection encountered after the iteration is completed it is implied that the straight line has never intersected the clipping edges and the line segment is completely outside the clipping window. So no output is generated.

In Fig. 1 the algorithm finds out that the input line AB intersects the clipping edges eTop and eLeft in points A′ and B′ respectively and thus the line segment A′B′, joining these two intersection points represents the resultant clipped line.

3. 2D LINE CLIPPING ALGORITHM

Input: A rectangular clipping window with diagonal points at (XMin, YMin) and (XMax, YMax) and a straight line with end point coordinates (X1, Y1) and (X2, Y2).

Output: No output clipped line if the input line is completely outside the clipping region or the resultant clipped line.

```
Begin
/* Following block determines whether the input straight line lies along any of the edges of the clipping rectangle or not */
If x1= xMin And x2 = xMax Or
 x1 = xMax And x2 = xMax Or
 y1 = yMin And y2 = yMin Or
 y1 = yMax And y2 = yMax

 Return
End If

/* Following block determines whether the straight line is completely inside the clipping rectangle or not */
If xMin < x1 And x1 < xMax
 And yMin < y1 And y1 < yMax
 And xMin < x2 And x2 < xMax
 And yMin < y2 And y2 < yMax then

 Return
Else
/* Finds any of the end points of the input straight line that resides inside the clipping rectangle and saves it in the point object 'pointInside' */

 If xMin < x1 And x1 < xMax
And yMin < y1 And y1 < yMax
 then
 pointInside ← p1(x1, y1)
End If
 If xMin < x2 And x2 < xMax
 And yMin < y2 And y2 < yMax
 then
 pointInside ← p2(x2, y2)
 End If
/*Following code block iteratively scans every pixel point on the clipping rectangle to get the pixel color using 'getPixelColor()' method. A point, where, the pixel color matches with the pixel color (i.e. Red) of the straight line, is treated as an intersection point. The very first intersection point is stored in the object 'intersectionPointOne'. For successive intersection points the object 'intersectionPointTwo' is repeatedly updated. */
 x← xMin
 y← yMin
 While y <= yMax do
 pixelColor ←getPixelColor(x, y)
If pixelColor = "Red" then
 If intersectionPointOne <> null then
 intersectionPointOne ← (x, y)
 Else
 intersectionPointTwo← (x, y)
 End If
 End If
 y ← y + 1
 End While
While x <= xMax do
 pixelColor ←getPixelColor(x, y)
If pixelColor = "Red" then
 If intersectionPointOne <> null then
 intersectionPointOne ← (x, y)
 Else
 intersectionPointTwo← (x, y)
 End If
 End If
 x ← x + 1
 End While
While y >= yMin do
 pixelColor ←getPixelColor(x, y)
If pixelColor = "Red" then
 If intersectionPointOne <> null then
 intersectionPointOne ← (x, y)
 Else
 intersectionPointTwo← (x, y)
 End If
 End If
 y ← y - 1
 End While
 While x >= xMax do
 pixelColor ←getPixelColor(x, y)
If pixelColor = "Red" then
 If intersectionPointOne <> null then
 intersectionPointOne ← (x, y)
 Else
 intersectionPointTwo← (x, y)
 End If
 End If
 x ← x - 1
 End While
/*The 'DrawLine()' method generates the clipped line depending on the values of the point objects. */
If pointInside <> null and
 intersectionPointOne <> null then
 DrawLine(pointInside, intersectionPointOne)
 Else
 If intersectionPointOne <> null and
 intersectionPointTwo <> null then
 DrawLine(intersectionPointOne,
 intersectionPointTwo)
 End If
 End If
 End If
End
```

4. EXPERIMENTAL RESULTS AND DISCUSSIONS

Scenario A

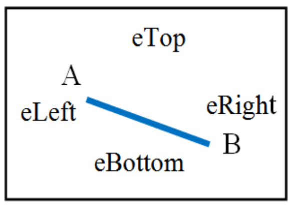

Figure 2(a)

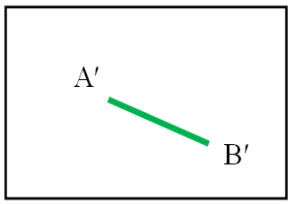

Figure 2(b)

The algorithm, as proposed in this paper has been realized in C# language on Windows platform. Random line segments in a wide range of geometrical distribution are considered for testing. Some special situations are discussed in this section as test cases to show how the algorithm has successfully resolved all possible input instances.

In this scenario [Fig. 2(a)] the input straight line AB falls completely within the clipping window. The algorithm figures this out using the inequality condition in the very first step. No further processing is required and the algorithm draws the original line as the clipped line A′B′ [Fig. 2(b)].

Scenario B

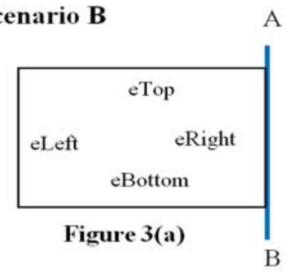

Figure 3(a)

Figure 3(b)

Here, the input line AB [Fig. 3(a)] falls on the clipping edge eRight. The algorithm detects it using the inequality condition in the very first step and rejects the input line as outside the clipping region, as a result, no output is generated [Fig. 3(b)].

Scenario C

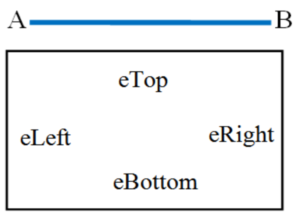

Figure 4(a) **Figure 4(b)**

In the third scenario [Fig. 4(a)] the original straight line AB is above the clipping edge eTop and no part of it enters the clipping rectangle. As a result after completion of the iteration process the algorithm fails to find any intersection point and thus identifies the input line to be completely outside the clipping region and does not generate any resultant clipped line [Fig. 4(b)].

Scenario D

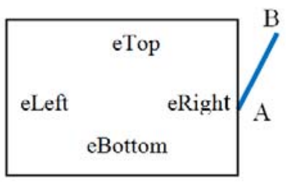

Figure 5(a)

Figure 5(b)

The current scenario [Fig. 5(a)] considers a straight line AB which originates from edge eRight and goes outward from the clipping window. After all the steps are completed the algorithm detects a single intersection point A of AB with the clipping edge eRight. The point object *pointInside* is now checked and found to be NULL as no end point of the input line resides inside the clipping region. Therefore, no output clipped line is drawn [Fig. 5(b)].

Scenario E

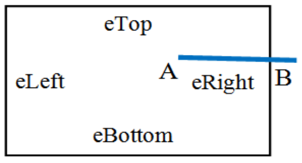

Figure 6(a)

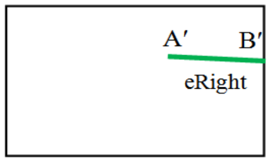

Figure 6(b)

Fig. 6(a) depicts a scenario where the input line AB originates from within the clipping rectangle and goes outside of it through edge eRight. The algorithm first detects that the endpoint A is inside the clipping region and puts a reference of the point in the object *pointInside*. After the iteration process is done a sole intersection point B′ with edge eRight is recorded. Next, reference of the point stored inside the object *pointInside* is checked (point A here) and a line segment joining point A with intersection point B′ is generated as the output A′B′ [Fig. 6(b)].

Scenario F

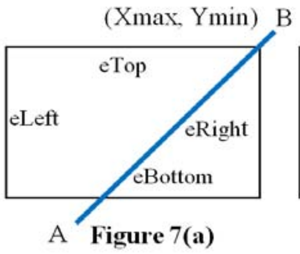

Figure 7(a)

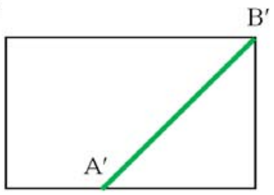

Figure 7(b)

In this case [Fig. 7(a)] a straight line AB which goes through the corner point (Xmax**,** Ymin) of the clipping rectangle is taken as input. During the iteration process the algorithm rightly finds out two intersection points. The first one A′ falls on edge eBottom and the second one B′ is with edge eTop (and not with edge eRight). The line segment A′B′ connecting these two points is our desired output [Fig. 7(b)].

Scenario G

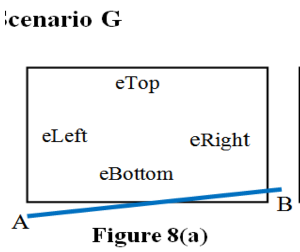

Figure 8(a)

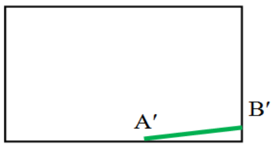

Figure 8(b)

In this last scenario [Fig. 8(a)] a straight line AB (having acute angel with edge eBottom <5 degree) moves towards the clipping window and grazes the edge eBottom producing multiple intersections before entering the clipping region. During the iteration process the algorithm accurately picks up only a single intersection point. Thus the output line A′B′ is generated taking the single intersection A′ with the edge eBottom and B′ with the edge eRight [Fig. 8(b)].

5. CONCLUSION

In this paper the basic pixel property (color) has been used to find the co-ordinates of the point/points of intersection of the input straight line (to be clipped) and the clipping window. Instead of using region sub division this algorithm solely focuses on the clipping edges. In the existing methods to clip a straight line, slope of the line is needed to be calculated and direction of the line has been considered. In other occasions the infinite extent of the straight line as well as the clipping edges has been considered to find all the intersection points that the line makes with the clipping edges within the display area. In the current approach these computational overheads are simply avoided and no information about the straight line is ever required to calculate the intersection point/points with the clipping edges. Further this algorithm can be rightly modified to clip a straight line against convex or concave polygonal window.

REFERENCES

[1] D. Hearn and M. P. Baker, "Computer Graphics," C Version, 2nd Edition, Prentice Hall, Inc., Upper Saddle River, 1998, p. 226.

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

[3] D. Hearn and M. P. Baker, "Computer Graphics," C Version, 2nd Edition, Prentice Hall, Inc., Upper Saddle River, 1998, p. 230.

[4] D. Hearn and M. P. Baker, "Computer Graphics," C Version, 2nd Edition, Prentice Hall, Inc., Upper Saddle River, 1998, p. 233.

[5] Duvanenko, V. J., Gyurcsik, R. S., Robbins, W. E. Simple and efficient 2D and 3D span clipping algorithms, Computer & Graphics.

[6] Rogers, D. F. Procedural elements for computer graphics, McGraw-Hill, New York.

[7] Schaum's Outlines "Computer Graphics".

[8] Baoqing Jiang and Jingjing Han, Improvement in the Cohen-Sutherland Line Segment Clipping Algorithm, IEEE International Conference on Granular Computing (GrC) 2013.

[9] Frank Devai, An analysis technique and an algorithm for line clipping, Computational Science and Its Applications - ICCSA 2006

[10] Y. D. Liang and B. A. Barsky, "The Optimal Tree Algorithm for Line Clipping," Technical Paper Distributed at Eurographics'92 Conference, Cambridge, 1992, pp. 1-38.

[11] Frank Devai, A Speculative Approach to Clipping Line Segments, Computational Science and Its Applications - ICCSA 2006

[12] Vaclav Skala, A new approach to line and line segment clipping in homogeneous coordinates, Visual Comput (2005) 21: 905–914

[13] Wenjun Huang, The Line Clipping Algorithm Basing on Affine Transformation, Intelligent Information Management, 2010, 2, 380-385