An Efficient Algorithm for Line Clipping in Computer Graphics Programming

S. R. Kodituwakku¹, K. R. Wijeweera¹, M. A. P. Chamikara²

¹Department of Statistics & Computer Science, University of Peradeniya, Sri Lanka
²Postgraduate Institute of Science, University of Peradeniya, Sri Lanka

(*Corresponding author’s email: salukak@pdn.ac.lk)
Received: 08 February 2012 / Accepted after revision: 13 January 2013

ABSTRACT

Most of the line clipping algorithms are based on Cohen-Sutherland and Liang-Barsky algorithms. These algorithms involve a lot of calculations. This paper proposes a new line clipping algorithm for 2D space which is more efficient than the existing algorithms. The possible extended algorithm for 3D space is also presented. The algorithm proposed for the 2D space is compared against traditional line clipping algorithms. The proposed algorithm was tested for a large number of random line segments and the results showed that it performs better than the Cohen-Sutherland and Liang-Barsky algorithms.

INTRODUCTION

Line clipping is a basic and an important operation in computer graphics. There are many applications that involve line clipping. For example, line clipping is needed to extract a part of a given scene for viewing. Generally lines are clipped by using a region that includes the part of the given scene. It is known as the clipping window and it is a rectangle or a general polygon (Huang, 2010).

The traditional line clipping algorithms include Cohen-Sutherland line clipping algorithm (Hearn and Baker, 1998), Liang-Barsky line clipping algorithm (Huang, 2010), Cyrus-Beck line clipping algorithm (Cyrus and Beck, 1978) and Nicholl-Lee-Nicholl line clipping algorithm (Nicholl et al., 1987). The Cohen-Sutherland and the Liang-Barsky algorithms can be extended to three-dimensional clipping (Huang, 2010). The Nicholl-Lee-Nicholl algorithm performs fewer comparisons and divisions. Therefore, it is faster than others (Huang, 2010). The major disadvantage of this algorithm is that it can only be applied to two-dimensional clipping (Huang, 2010). On the other hand, the Liang-Barsky and the Cohen-Sutherland methods are easily extended to three-dimensional scenes (Huang, 2010).

The Cohen-Sutherland line clipping algorithm is one of the earliest and most widely used line clipping methods (Huang, 2010). In this algorithm, a rectangular clipping window along with a coding scheme is used to divide the space into regions. Then, each end point of the line segment is assigned a region code according to the region which has been occupied by that point. Then the “AND” and “OR” operations are performed over the region codes of the end points to decide whether the line segment is inside the clipping window or outside the clipping window. This algorithm is very faster for simple situations such as line segment is completely inside or outside of the clipping window. When the line segment cannot be classified as completely inside or outside, the algorithm needs to be repeated several times to convert it into a simple situation. So, if the line segment is intersecting more than two boundaries of the clipping window, lots of unnecessary computations are involved (Huang, 2010).

Cyrus and Beck have proposed another algorithm that deals with the parametric form of the line. In order to clip a line segment which is neither vertical nor horizon-
tal and lies entirely within the window, 12 additions, 16 subtractions, 20 multiplications and 4 divisions are required (Huang, 2010). Besides, for the general case (the line segments crossing all the boundaries of the window), the algorithm first makes computations and finds the parameters of the intersection points. According to the signs of the denominators of the parameters, then it determines which parts of the line segment is outside of the window. This algorithm mainly suffers from the above mentioned limitations.

Nicholl-Lee-Nicholl line clipping algorithm makes four rays which pass an endpoint of the line segment and four vertices of the window, and creates three regions by the four rays. Then, the algorithm determines which region that the line segment lies in, and finds the intersections or rejects the line segment. Before finding the intersection points of the line segment and the window, the algorithm first determines the position of the first endpoint of the line segment for the nine possible regions relative to the clipping window. If the point is not in one of the three especial regions, the algorithm has to transform the point to one of the three especial regions. To find the region in which the other endpoint of the line segment lies, it compares the slope of the line segment to the slopes of the four rays. Because of that, for the algorithm, finding the intersection points are efficient, but finding the positions of the two endpoints of the line segment are more complicated than Cohen-Sutherland line clipping algorithm (Huang, 2010).

You-dong Liang, Brian A. Barsky and Mel Slater also introduced faster line clipping algorithm (Hearn and Baker, 1998; Chen and Lu 2006). This algorithm is based on a parametric representation of the line segment. It 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 (Huang, 2010).

Skala (1994) proposed a line clipping algorithm for convex polygon window. The algorithm uses binary search to find the intersections in the clipping window. The complexity is \( O(\log N) \). For the rectangle window, the algorithm does not have an advantage in comparison with the Cyrus-Beck algorithm (Huang, 2010).

According to Hearn and Baker (Hearn and Baker, 1998) any line clipping algorithm involves three steps:

i. Test a given line segment to check whether it lays completely inside the clipping window
ii. If not check whether it lies completely outside the clipping window
iii. Otherwise perform intersection calculations with one or more clipping boundaries.

These three steps lead the algorithm to lot of calculations as explained above. In this paper, a new line clipping algorithm is proposed without using the traditional three step procedure. Instead the line segment is directly passed through the algorithm without any classification. It was tested for a large number of line segments and the results proved that the algorithm is more efficient than Cohen-Sutherland and Liang-Barksy algorithms.

**METHODOLOGY**

This section presents the proposed line clipping algorithm and analyzes its performance. For line clipping, a rectangular clipping window is considered. Following conventions have been used to label the rectangular window.

![Figure 1: Clipping window](image)

The general equation of a line, \( y = m \times x + c \), is used, where \( m \) is the gradient and \( c \) is the \( y \)-intercept. End points of the line segment are \( A = (x[0], y[0]) \) and \( B = (x[1], y[1]) \).
Pseudo code of the proposed algorithm for 2D space

All the symbols used in the following pseudo code are shown in Figure 1. To increase the understandability of the pseudo code we have omitted following cases from it.

i. Line segment is just a point
ii. Line segment is parallel to principle axes

Above two cases have been addressed at the implementation stage (See Appendix). Then the abstract pseudo code is as follows.

```plaintext
// Calculating m and c
For i = 0 to i = 1
    If x[i] < minx
        x[i] = minx;
        y[i] = m * minx + c;
    ElseIf x[i] > maxx
        x[i] = maxx;
        y[i] = m * maxx + c;
    EndIf
    If y[i] < miny
        x[i] = ( miny – c ) / m;
        y[i] = miny;
    ElseIf y[i] > maxy
        x[i] = ( maxy – c ) / m;
        y[i] = maxy;
    EndIf
EndFor
// Initial line is completely outside
If ( x[0] - x[1] < 1 ) AND ( x[1] - x[0] < 1 )
    // Do nothing
Else
    // Save the line with end points (x[0], y[0]), (x[1], y[1])
EndIf
```

Analysis of the algorithm for 2D space

This subsection analyzes the proposed algorithm for some of the most important test cases.

Case 1: Line is completely inside

Therefore, the initial position of A is not changed.
Consider point B,

x[B] < minx → false
x[B] > maxx → false
y[B] < miny → false
y[B] > maxy → false

Therefore, the line with the end points A and B is drawn.

Case 2: Line is completely outside

Therefore, the initial position of B is not changed.

```plaintext
( x[A] - x[B] < 1 ) && ( x[B] - x[A] ) → false
```

Therefore, the line is ignored.

Figure 2: Line is completely inside

Figure 3: Line is completely outside
Case 3: Line intersects the clipping window

Figure 4: Line is intersecting the boundaries

\[ x[A] \neq x[B] \Rightarrow \text{true} \]
\[ y[A] \neq y[B] \Rightarrow \text{true} \]

Consider point \( A \),
\[ x[A] < \text{minx} \Rightarrow \text{true} \]
Therefore, \( A \rightarrow A' \)
\[ y[A'] < \text{miny} \Rightarrow \text{true} \]
Therefore, \( A' \rightarrow A'' \)

Consider point \( B \),
\[ x[B] < \text{minx} \Rightarrow \text{false} \]
\[ x[B] > \text{maxx} \Rightarrow \text{true} \]
Therefore, \( B \rightarrow B' \)
\[ y[B'] < \text{miny} \Rightarrow \text{false} \]
\[ y[B'] > \text{maxy} \Rightarrow \text{true} \]
Therefore, \( B' \rightarrow B'' \)

Therefore, the line with the end points \( A'' \) and \( B'' \) is drawn.

Case 4: Line partially inside the clipping window

Figure 5: Line is partially inside the clipping window

\[ x[A] \neq x[B] \Rightarrow \text{true} \]
\[ y[A] \neq y[B] \Rightarrow \text{true} \]

Consider point \( A \),
\[ x[A] < \text{minx} \Rightarrow \text{false} \]
\[ x[A] > \text{maxx} \Rightarrow \text{false} \]
\[ y[A] < \text{miny} \Rightarrow \text{false} \]
\[ y[A] > \text{maxy} \Rightarrow \text{false} \]
\[ y[A] < \text{miny} \Rightarrow \text{true} \]

Therefore, the initial position of \( A \) is not changed.

Consider point \( B \),
\[ x[B] < \text{minx} \Rightarrow \text{false} \]
\[ x[B] > \text{maxx} \Rightarrow \text{true} \]
Therefore, \( B \rightarrow B' \)
\[ x[B'] < \text{minx} \Rightarrow \text{true} \]
\[ x[B'] > \text{maxx} \Rightarrow \text{true} \]
Therefore, \( B' \rightarrow B'' \)

Therefore, the line with the end points \( A \) and \( B'' \) is drawn.

Pseudo code of the proposed algorithm for 3D space

In this case, an additional coordinate, \( Z \) coordinate, is involved. So the end points look like below.

\[ (x[1], y[1], z[1]) \]
\[ (x[0], y[0], z[0]) \]

Equation of the line,
\[ \frac{x - x[0]}{l} = \frac{y - y[0]}{m} = \frac{z - z[0]}{n} \]
\( l, m, n \) are constants

Substitute, \( (x[1], y[1], z[1]) \)
\[ \frac{x[1] - x[0]}{l} = \frac{y[1] - y[0]}{m} = \frac{z[1] - z[0]}{n} \]

Take,
\[ x[1] - x[0] \]
\[ y[1] - y[0] \]
\[ z[1] - z[0] \]

Take,
\[ \frac{y[1] - y[0]}{m} = \frac{z[1] - z[0]}{n} \]
\[ a = \frac{m}{l} \]
\[ b = \frac{n}{l} \]

Point of intersection with \( x = p \) plane,
\[ \left( p, \frac{p - x[0]}{a} + y[0], \frac{p - x[0]}{a} + z[0] \right) \]

Point of intersection with \( y = q \) plane,
\[ \left( a \ast (q - y[0]) + x[0], q - y[0] + z[0] \right) \]

Point of intersection with \( z = r \) plane,
\[ \left( a \ast b \ast (r - z[0]) + x[0], b \ast (r - z[0]) + y[0], r \right) \]

Pseudo code of the algorithm developed for the 2D space can be extended as follows. To increase the understandability of the pseudo code we have omitted following cases from it.

i. Line segment is just a point

\[ x[A] != x[B] \Rightarrow \text{true} \]
\[ y[A] != y[B] \Rightarrow \text{true} \]

Consider point \( A \),
\[ x[A] < \text{minx} \Rightarrow \text{false} \]
\[ x[A] > \text{maxx} \Rightarrow \text{false} \]
\[ y[A] < \text{miny} \Rightarrow \text{false} \]
\[ y[A] > \text{maxy} \Rightarrow \text{false} \]
\[ y[A] < \text{miny} \Rightarrow \text{true} \]
ii. Line segment is parallel to principle planes

Above two cases can be addressed at the implementation stage.

// Calculate a and b
For i = 0 to i = 1
If x[i] < minx
y[i] = ( minx - x[0] ) / a + y[0];
z[i] = ( minx - x[0] ) / ( a * b ) + z[0];
x[i] = minx;
ElseIf x[i] > maxx
y[i] = ( maxx - x[0] ) / a + y[0];
z[i] = ( maxx - x[0] ) / ( a * b ) + z[0];
x[i] = maxx;
EndIf
If y[i] < miny
x[i] = a * ( miny - y[0] ) + x[0];
z[i] = ( miny - y[0] ) / b + z[0];
y[i] = miny;
ElseIf y[i] > maxy
x[i] = a * ( maxy - y[0] ) + x[0];
z[i] = ( maxy - y[0] ) / b + z[0];
y[i] = maxy;
EndIf
If z[i] < minz
x[i] = a * b * ( minz - z[0] ) + x[0];
y[i] = b * ( minz - z[0] ) + y[0];
z[i] = minz;
ElseIf z[i] > maxz
x[i] = a * b * ( maxz - z[0] ) + x[0];
y[i] = b * ( maxz - z[0] ) + y[0];
z[i] = maxz;
EndIf
EndFor

// Initial line is completely outside
If (x[0] - x[1] < 1) AND (x[1] - x[0] < 1)
// Do nothing
Else
// Save the line with end points (x[0], y[0], z[0]), (x[1], y[1], z[1])
EndIf

RESULTS AND DISCUSSION

The proposed algorithm for the 2D space was tested for all the possible test cases. The test results indicated that it performs well in all possible situations. In order to validate the algorithm, it was compared against the Cohen-Sutherland and Liang-Barsky algorithms. The following hardware and software configurations were used for testing.

Computer: Intel(R) Pentium(R) Dual CPU; E2180 @ 2.00 GHz; 2.00 GHz, 0.98 GB RAM
IDE Details: Turbo C++; Version 3.0; Copyright(c) 1990, 1992 by Borland International, Inc.

Method: A window with values minx = miny = 100 and maxx = maxy = 300 was used as the clipping window. Random points were generated in the range 0-399 by using the randomize() function. These random points were considered as end points to generate random lines. Number of clock cycles taken by each algorithm to clip 10^8 random lines were counted using the clock() function (Huang, 2010). The results are shown in Table 1.

Table 1: Number of clock cycles comparison of the proposed algorithm versus traditional algorithms

<table>
<thead>
<tr>
<th>Step</th>
<th>Cohen-Sutherland</th>
<th>Liang-Barsky</th>
<th>Proposed algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2596</td>
<td>2452</td>
<td>2296</td>
</tr>
<tr>
<td>2</td>
<td>2593</td>
<td>2452</td>
<td>2295</td>
</tr>
<tr>
<td>3</td>
<td>2594</td>
<td>2452</td>
<td>2295</td>
</tr>
<tr>
<td>4</td>
<td>2595</td>
<td>2452</td>
<td>2296</td>
</tr>
<tr>
<td>5</td>
<td>2596</td>
<td>2451</td>
<td>2296</td>
</tr>
<tr>
<td>6</td>
<td>2593</td>
<td>2451</td>
<td>2294</td>
</tr>
<tr>
<td>7</td>
<td>2593</td>
<td>2452</td>
<td>2295</td>
</tr>
<tr>
<td>8</td>
<td>2592</td>
<td>2451</td>
<td>2295</td>
</tr>
<tr>
<td>9</td>
<td>2593</td>
<td>2452</td>
<td>2296</td>
</tr>
<tr>
<td>10</td>
<td>2594</td>
<td>2451</td>
<td>2296</td>
</tr>
</tbody>
</table>

The results shown in Table 1 prove that the proposed algorithm is faster than both Cohen-Sutherland and Liang-Barsky algorithms. Average ratios can be calculated using following equation.

\[
\text{Average Ratio (Cohen-Sutherland: Proposed)} = \frac{\sum \text{Clock cycles for traditional algorithm}}{\sum \text{Clock cycles for proposed algorithm}}
\]

Average Ratio (Cohen-Sutherland: Proposed) = 25939/22960=1.1297

Average Ratio (Liang-Barsky: Proposed) = 24516/22960=1.0677

Therefore, the proposed algorithm is 1.13 times faster than the Cohen-Sutherland algorithm and 1.07 times faster than the Liang-Barsky algorithm.
CONCLUSION

A new algorithm for line clipping in 2D space was introduced. Additionally possibility of expanding it for 3D space was discussed. Algorithm proposed for the 2D space was tested and compared against two well-known algorithms. According to the test results, it is faster than the traditional algorithms. Therefore, the proposed algorithm can be successfully used in applications where line clipping involved since its performance is better. Even the line segment is completely outside the proposed algorithm needs to calculate some of the intersection points. That is a disadvantage of the proposed method compared to the traditional two algorithms that were tested above. But in all the other cases the performance of the proposed algorithm is better.

REFERENCES


APPENDIX

Implementation of the proposed algorithm for 2D space

The proposed algorithm for the 2D space was developed in C++ programming language. Here all the test cases have been considered. The source code is as follows.

```cpp
void clipMY(double x[], double y[], double minx, double miny, double maxx, double maxy) {
    // gradient and y-intercept of the line
    double m, c;
    int i;

    // non vertical lines
    if(x[0] != x[1]) {
        // non vertical and non horizontal lines
        if(y[0] != y[1]) {
            // calculate the gradient
            m = (y[0] - y[1]) / (x[0] - x[1]);
            // calculate the y-intercept
            c = (x[0]*y[1] - x[1]*y[0]) / (x[0] - x[1]);

            for(i = 0; i < 2; i++) {
                if(x[i] < minx) {
                    x[i] = minx;
                    y[i] = m*minx + c;
                } else if(x[i] > maxx) {
                    x[i] = maxx;
                    y[i] = m*maxx + c;
                }
            }
        }
    } // initial line is completely outside
    if((x[0] - x[1] < 1) && (x[1] - x[0] < 1)) {
        // do nothing
    } else {
        // draw the clipped line
    }
```
setcolor(15);
    line(x[0],y[0],x[1],y[1]);
}
}
// horizontal lines
else
{
    // initial line is completely outside
    if((y[0]<=miny) || (y[0]>=maxy))
    {
        // do nothing
    }
    else
    {
        for(i=0;i<2;i++)
        {
            if(x[i]<minx)
            {
                x[i]=minx;
            }
            else if(x[i]>maxx)
            {
                x[i]=maxx;
            }
        }
        // initial line is completely outside
        if((x[0]-x[1]<1) && (x[1]-x[0]<1))
        {
            // do nothing
        }
        // draw the clipped line
        else
        {
            setcolor(15);
            line(x[0],y[0],x[1],y[1]);
        }
    }
}
// vertical lines
else
{
    // initial line is just a point
    if(y[0]==y[1])
    {
        // initial point is outside
        if((y[0]<=miny) || (y[0]>=maxy))
        {
            // do nothing
        }
        // initial point is outside
        else if((x[0]<=minx) || (x[0]>=maxx))
        {
            // do nothing
        }
        // initial point is inside
        else
        {
            putpixel(x[0],y[0],15);
        }
    }
    // initial line is completely outside
    else if((x[0]<=minx) || (x[0]>=maxx))
    {
        // do nothing
    }
    else
    {
        for(i=0;i<2;i++)
        {
            if(y[i]<miny)
            {
                y[i]=miny;
            }
            else if(y[i]>maxy)
            {
                y[i]=maxy;
            }
        }
        // initial line is completely outside
        if((y[0]-y[1]<1) && (y[1]-y[0]<1))
        {
            // do nothing
        }
        // draw the clipped line
        else
        {
            setcolor(15);
            line(x[0],y[0],x[1],y[1]);
        }
    }
}