

Table of Contents

Two-Dimensional Clipping Algorithms.....	1
Clipping Lines.....	1
Cohen-Sutherland's Line Clipping Algorithm.....	1
Calculating Intersections.....	2
The Algorithm.....	3
Liang and Barsky's Clipping Algorithm.....	4
The Algorithm.....	5
Polygon Clipping.....	6
Raster Scan Polygon Filling.....	7
Adjusting Segment Endpoints.....	8

Two-Dimensional Clipping Algorithms

The viewport is a region of the window in which graphics are displayed. The transformations from world coordinates, to camera coordinates, to window coordinates, and finally to viewport coordinates usually results in only a part of the scene being visible, and hence the need to clip lines to the viewport in 2D. Suppose a point (x, y) . It is inside the viewport if

$$\begin{aligned} X_{min} &\leq x \leq X_{max} \\ Y_{min} &\leq y \leq Y_{max} \end{aligned}$$

Clipping Lines

If both endpoints of a line lie inside the viewport, no clipping is necessary, as the line segment is completely visible. If only one endpoint is within the viewport, then we must clip the line at the intersection. If both endpoints are outside the viewport, then the line crosses two boundaries, or is completely outside the viewport.

Cohen-Sutherland's Line Clipping Algorithm

The viewing space is divided into nine encoded regions as shown below:

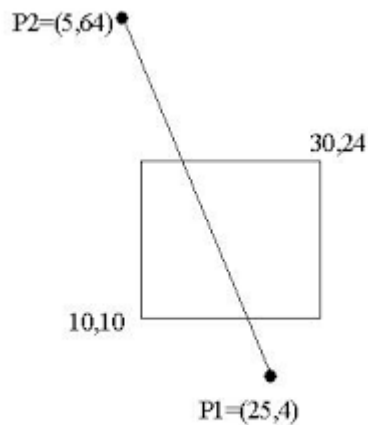
1001	1000	1010
0001	0000	0010
0101	0100	0110

For each endpoint of a line segment, we assign a 4-bit code following the rules below (the

code is formed in the following way: (bit 1, bit 2, bit 3, bit 4):

1. bit 1 is 1 if $x < X_{min}$
2. bit 2 is 1 if $x > X_{max}$
3. bit 3 is 1 if $y < Y_{min}$
4. bit 4 is 1 if $y > Y_{max}$

Hence, if (x_1, y_1) and (x_2, y_2) have code 0000 then both end points are inside. If both endpoints have a 1 in the same bit position, then the line segment is entirely outside. Performing a logical AND on both codes and obtaining something different from 0000 indicates that the line is outside. Otherwise, the line segment must be checked for intersections. Here is an example:



In this case, we must clip both endpoints:

1. p_1 code is (0100): Then this endpoint is outside and below. Compute the intersection p_1' with Y_{min} which is now the new endpoint for this segment.
2. p_2 code is (1001): This endpoint is outside and above. Compute intersection p_2' with Y_{max} which is the new endpoint.

Calculating Intersections

With a vertical boundary, x is either X_{min} or X_{max} . Hence:

$$y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1} \right) (X_{min} - x_1)$$

or

$$y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1} \right) (X_{max} - x_1)$$

For a horizontal boundary, y is either Y_{min} or Y_{max} . Hence:

$$x = x_1 + (Y_{min} - y_1) \left(\frac{x_2 - x_1}{y_2 - y_1} \right)$$

or

$$x = x_1 + (Y_{max} - y_1) \left(\frac{x_2 - x_1}{y_2 - y_1} \right)$$

The Algorithm

The inputs to the algorithm are the two endpoints of a line segment, and the attributes of the viewport. The output is the clipped line segment.

```

begin
if (  $p_1$  and  $p_2$  inside) then accept and return
else {
  while (code of  $p_1$  != 0000) do {
    if (  $p_1$  is outside)
    if (  $p_1$  is to the left) {
       $p_1$  = inters. of  $p_1p_2$  and  $x=X_{min}$  ;
      compute new code for  $p_1$  ;
    }
    else if (  $p_1$  is to the right) {
       $p_1$  = inters. of  $p_1p_2$  and  $x=X_{max}$  ;
      compute new code for  $p_1$  ;
    }
    else if (  $p_1$  is below) {
       $p_1$  = inters. of  $p_1p_2$  and  $y=Y_{min}$  ;
      compute new code for  $p_1$  ;
    }
    else if (  $p_1$  is above) {
       $p_1$  = inters. of  $p_1p_2$  and  $y=Y_{max}$  ;
      compute new code for  $p_1$  ;
    }
  }
}

```

```

}
repeat these steps for  $p_2$ 
}
output  $p_1$  and  $p_2$ 
end

```

Midpoint Division Method

This technique consists of locating intersections without computing them with a binary search. First, compute the midpoint of the segment, and divide the segment into two as per the midpoint. For these two segments, test if they are totally inside or totally outside of the viewport and either accept or reject the segments accordingly. Repeat for each segment that cannot be either accepted or rejected.

Liang and Barsky's Clipping Algorithm

This algorithm uses the parametric form of line equations, written as:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} u = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} u$$

where $u \in [0,1]$. Using this particular form of equations then we can write that when the following inequalities are satisfied

$$X_{min} \leq x_1 + \Delta x u \leq X_{max}$$

$$Y_{min} \leq y_1 + \Delta y u \leq Y_{max}$$

then the line segment lies completely within the window. These inequalities can be written as

$$p_k u \leq q_k$$

for $k=1,2,3,4$.

$k=1$	$p_1 = -\Delta x$	$q_1 = x_1 - X_{min}$	Left boundary
$k=2$	$p_2 = \Delta x$	$q_2 = X_{max} - x_1$	Right boundary
$k=3$	$p_3 = -\Delta y$	$q_3 = y_1 - Y_{min}$	Bottom boundary
$k=4$	$p_4 = \Delta y$	$q_4 = Y_{max} - y_1$	Top boundary

If $p_k=0$ then the line segment is parallel to the k^{th} boundary:

- $p_1 = -\Delta x = 0$ and the line is vertical and parallel to the left and right boundaries
- if $q_1 \leq 0$ or $q_2 \leq 0$ then the line segment is completely outside the window.

Hence, if $(p_1=0) \wedge (q_1 \leq 0 \vee q_2 \leq 0)$ the line segment is completely outside. The same rules apply for p_3 and we can write that if $(p_3=0) \wedge (q_3 \leq 0 \vee q_4 \leq 0)$ then the segment is also outside. In general, if $q_k \geq 0$ then the line is inside the k^{th} boundary.

In cases when the line segment is not parallel to any of the view port's borders, then if $p_k < 0$ the infinite extension of the line segment proceeds from outside to inside the infinite extension of the k^{th} window boundary. The opposite situation occurs when $p_k > 0$.

Consequently, when $p_k \neq 0$, we compute the value of u that yields the intersection of the extended line segment with the extended k^{th} boundary. The value of u is given by equating the initial inequalities:

$$p_k u = q_k$$

and thus, the intersection of the extended line segment with the k^{th} boundary is simply given by:

$$u = \frac{q_k}{p_k}$$

The Algorithm

Computing u_1 :

for all $p_k < 0$ {

$$r_k = \frac{q_k}{p_k}$$

$$u_1 = \max\{0, r_k\}$$

}

Computing u_2 :

for all $p_k > 0$ {

$$r_k = \frac{q_k}{p_k}$$

$$u_2 = \min\{1, r_k\}$$

}

The main body of the algorithm is:

begin

 compute p_k and q_k , for $k=1,2,3,4$

 if $[(p_1=0) \wedge (q_1 < 0 \vee q_2 < 0)] \vee [(p_3=0) \wedge (q_3 < 0 \vee q_4 < 0)]$ then reject the segment

 else

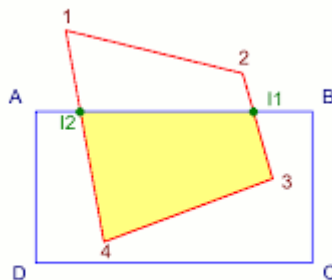
```

compute  $u_1$ 
compute  $u_2$ 
if  $u_1 > u_2$  then reject the segment
else
    clipped line segment is  $(x_1 + \Delta x u_1, y_1 + \Delta y u_1)$  ,  $(x_1 + \Delta x u_2, y_1 + \Delta y u_2)$ 
end

```

Polygon Clipping

Polygons need to be clipped when objects to render are made with them. Polygons are represented as ordered sets of vertices. An intuitive approach to clip polygons is to trace the vertices that are completely inside the viewport, and get rid of those that are completely outside. For all other vertices, and using their parametric representation, we can determine if they are tracked from the outside toward the inside of the area of the viewport, and conversely. Below is a simple example where the intersections of the vertices are all within the same boundary. All that is needed after clipping is to create a vertex joining the two intersections, and the polygon is clipped.



However, this simple method will not work when the intersections are not all with the same boundary. When this is the case, the polygon must be adjusted to include the part of the viewport that finds itself inside it.

In order to perform this type of polygon clipping, we must first know how to compute intersections when the segments are given in parametric form. Consider:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} u$$

and let's find the intersection between $\vec{p}_1 = (x_1, y_1), (x_2, y_2)$ and $\vec{p}_2 = (x_3, y_3), (x_4, y_4)$:

$$\begin{aligned} \vec{p}_1 &= \vec{x}_1 + \Delta \vec{x}_1 u \\ \vec{p}_2 &= \vec{x}_2 + \Delta \vec{x}_2 v \end{aligned}$$

and find (u, v) values such that $\vec{p}_1 = \vec{p}_2$. We can write this equality as:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} u + \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_4 - x_3 \\ y_4 - y_3 \end{pmatrix} v + \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}$$

and rearrange it in the following way:

$$\begin{pmatrix} x_2 - x_1 & x_3 - x_4 \\ y_2 - y_1 & y_3 - y_4 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x_3 - x_1 \\ y_3 - y_1 \end{pmatrix}$$

This is then simply a 2 by 2 linear system of equations that need to be solved. If the lines are not parallel, it has a solution for $(u, v)^T$ which yields the intersection point.

Raster Scan Polygon Filling

Rendering polygons imply filling them with color, texture, etc. Here is a simple scan-line algorithm to do just this, with the following steps:

1. Determine the number of scan lines. The first scan line starts at the minimum y value of all edge points while the last scan line is at the maximum y value of all edge points.
2. Create an edge list containing all the edges from the polygon, trace the horizontal edges and remove them from the list.
3. For each scan line, do:
 - create a list of active edges from the edge list. These are the edges that make an intersection with the scan line.
 - Find all intersections between the scan line and the edges from the list of active edges
 - Sort the edges by intersection, in increasing order of their x coordinates, and form a sorted intersection list.
 - Proceeding one intersection at a time, and forming an intersection pair with the next intersection, determine if the two intersections correspond to edge endpoints. If so, then test if the the two intersections are unequal. If this is the case, make the left intersection equal to the right intersection in the pair. If the intersections in the pair are equal (we have two edges that meet at the same coordinates with their endpoints) and if one segment goes up while the other goes down, (starting from the intersection point), remove the duplicate intersection from the intersection list.
 - Now that the intersection list is processed, start tracing from the fist intersection to the next, then stop tracing until the following intersection, then resume tracing until the next intersection, and so on.

To implement this algorithm efficiently, it is important to observe that each scan line has a

y value one pixel lower than the previous scan line and thus:

- $y_{i+1} = y_i - 1$
- slope of segment is $m = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$
- which implies $x_{i+1} = x_i - \frac{1}{m}$

This is how successive intersections are obtained for each scan line in an efficient manner. Other elements must be taken into consideration for a fast implementation:

- Find the bounding box of the polygon to determine the number of scan lines and their length.
- Maintain a list of segments that are crossed by the current scan line. That is, create a list of segments sorted in increasing order of their largest y and, if two segments have the same maximum y coordinates, sort them in decreasing order of their smallest y values.
- With such a sorted list, the segments crossed by the current scan line form a contiguous sub-list within the sorted list.

Adjusting Segment Endpoints

Instead of dealing with the number of intersections where the segments meet on the scan lines, we could also simply perform minor adjustments to the points of the polygons to ensure that each time an intersection is encountered, we either start or stop turning on pixels. On the following page are displayed all the cases (to the left) and how they should be dealt with (to the right).

While this technique will slightly modify the appearance of the polygon at low resolutions, it is virtually undetectable with current resolutions and speeds up the scan-line algorithm.

Illustration 1: Cases in need of endpoint adjustments

