University of West Bohemia at Pilsen

Faculty of Applied Sciences

Department of Computer Science and Engineering

Dissertation thesis

Algorithms for Line Clipping and Their Complexity

by

Ing. *Duc Huy Bui*

E-mail: bui@kiv.zcu.cz

URL: http://iason.zcu.cz/~bui

Supervisor: Prof. Ing. Václav Skala, CSc. Pilsen, October 1999

Table of contents

Preface

Clipping is essential technique in computer graphics, and as such it has been studied very extensively in the past. Nowadays, the problem is very often considered as solved. In many aspects, I must admit that this is indeed true. Nevertheless, as the computational resources have developed dramatically in the last decade a lot of traditional approaches are now seen under very different circumstances. We are confronted with large amounts of available data that we would like not only to visualize, but also to make it as realistic and as fast as possible. Therefore, any improvement in solving the clipping problem is always welcome. The main contribution of this thesis is to summarize the work in the field of clipping over last two decades and present some results I have made during my postgraduate study.

There is a hope that presented approaches and modifications could also help in other fields of algorithm design.

Algorithms for Line Clipping and Their Complexity

Duc Huy Bui

University of West Bohemia

ABSTRACT

Clipping is one of the fundamental problems of the computational geometry with applications in various areas of computer graphics, visualization and CAD system. This work resumes the clipping problem solutions in E^2 and E^3. It also demonstrates some thoughts and connection between algorithm complexity, speed and influence of possible pre-processing to the final algorithm complexity. Also, how some precise formulations can lead to better and faster algorithms to decrease algorithm complexity, will be shown.

Keywords: Line Clipping, Convex Polyhedron, Computer Graphics, Algorithm Complexity, Geometric Algorithms, Algorithm Complexity Analysis.

1. Introduction

In the most general sense, clipping is the evaluation of the intersection between two geometrical entities. These geometrical entities may be lines, line segments, rectangles, polygons, polyhedrons, curves, surfaces and so on, or their assemblies. Clipping is a very important stage in the viewing pipeline of the computer graphics. It involves computing which part of a geometric primitive, such as a line segment or polygon is visible with respect to a clipping region. The simplest case in 2D graphics is if the clipping region is rectangular window and the primitive is a line or line segment. The major application is computing the part of a scene visible with respect to a window. Clipping is also often used to determine whether the cursor position is within a given tolerance of a geometric primitive, by testing the visibility of the primitive against a small rectangle that has the cursor position at its center.

There are three major approaches to the solution of the line clipping problem.

- The first one, known as the Cohen-Sutherland (CS) algorithm [Fol90a], uses a coding scheme of the line segment end-points to quickly reject line segments having both end-points outside a particular clipping boundary, and quickly accept line segments totally inside the clipping region. In all other cases the line segment is cut by intersecting it with one of the boundaries that it is known to cross, and the procedure is repeated with the curtailed line segment.

- The second approach, parametric clipping, was introduced by Cyrus and Beck [Cyr79a] and by Liang and Barsky [Lia84a], where the line is represented in parametric form. The values of the parameter corresponding to the points where the extended line segment intersects the clipping boundary are used to find the final clipped line segment.
- The third approach solves the problem by eliciting among all possible relationships between the line segment and the clip region, which particular one has happened. The efficiency of the algorithm depends on the method chosen to determine which position of line segment has occured. Intersections or parameter values are found only for those cases where the line segment actually intersects a boundary. This approach can be found in the work of Nicholl, Lee and Nicholl [Nic87a] and Sobkow, Pospisil and Yang [Sob87a] and in the optimal tree algorithm of Liang and Barsky [Lia92a].

Commonly, the clipping problem in E^2 and in E^3 can be classified as clipping by

- orthogonal window, cube or viewing pyramid
- convex polygon or polyhedron,
- non-convex polygon or polyhedron
- non-linear region or object

and as

- line segment clipping
- line clipping
- polygon or polyhedron clipping
- curve clipping

The rest of the thesis is organized as follows: Section 2 briefly introduces the algorithm complexity terms, which is usually used to assess algorithms. Seven following sections will subsequently describe the particular clipping algorithms. For each category, we discuss the most often used algorithm followed by the new proposed algorithm (if exists with experimental efficiency verification) that is more efficient.

Section 10 mentions principles of clipping in homogeneous co-ordinates. Section 11 summarises algorithm complexities. The conclusion and expected future work will be mentioned in section 12. The main references can be found in section 13. List of published or accepted papers for publication with reached results can be found in section 14. Published, accepted or submitted papers for publication are also enclosed in appendices.

2. Algorithm complexity

Before describing particular algorithms for line clipping, it is necessary to deal with algorithm complexity. Algorithm complexity is very often used to compare different algorithms in order to assess their properties. There are two major types of complexity: the space complexity and the time complexity. The space complexity usually indicates the memory required by the algorithm to solve the problem, whereas the time complexity shows the computation speed of the algorithm. Let us consider a problem P of size N and let $I_1, I_2,..., I_k$ denote all instances of problem P, and $T_1, T_2,..., T_k$ denote the times required for the instances $I_1, I_2,..., I_k$ using algorithm A. There are three following basic types of time complexity

- The **worst case** time complexity of algorithm A

$$T_w(N) = \max \{T_1, T_2,..., T_k\}$$

- The **best case** time complexity of algorithm A

$$T_b(N) = \min \{T_1, T_2,..., T_k\}$$

- The **average case** or **expected** time complexity of algorithm A

$$T_a(N) = \Sigma\, p_i T_i$$

where p_i is the probability of instance I_i occurring.

The time complexity without the specification implicitly denotes the worst case time complexity because most of the time we are interested in the worst case behavior of an algorithm.

Algorithm complexity is usually represented as a function of problem size using one of four following notations:

- The first one, called **big-O** notation, set the upper limit of algorithm performance. Function g: $\mathbf{N} \rightarrow \mathbf{R}$ is big-O of f: $\mathbf{N} \rightarrow \mathbf{R}$, denoted $g = O(f)$, if and only if there exists a natural number N_0 and a real constant $c > 0$ such that $g(N) \leq c*f(N)$ for all $N \geq N_0$. In other words, function g grows no faster than a function f for sufficiently large N.
- The second notation, **Omega** notation, set the lower limit of algorithm performance. Function g: $\mathbf{N} \rightarrow \mathbf{R}$ is Omega of f: $\mathbf{N} \rightarrow \mathbf{R}$, denoted $g = \Omega(f)$, if and only if there exists a natural number N_0 and a real constant $c > 0$ such that $g(N) \geq c*f(N)$ for all

$N \geq N_0$. In other words, function g grows no slower than a function f for sufficiently large N.

- The **Theta** notation is used for optimal algorithms. Function g: $\mathbf{N} \rightarrow \mathbf{R}$ is theta of f: $\mathbf{N} \rightarrow \mathbf{R}$, denoted $g = \theta(f)$, if and only if there exists a natural number N_0 and real constants $c_1 > 0$, $c_2 > 0$ such that $c_1 * f(N) \leq g(N) \leq c_2 * f(N)$ for all $N \geq N_0$. In other words, function g grows at the same rate as a function f for sufficiently large N.
- The fourth one, called **small-o** notation, is defined as follows: function g: $\mathbf{N} \rightarrow \mathbf{R}$ is small-o of f: $\mathbf{N} \rightarrow \mathbf{R}$, denoted $g = o(f)$, if and only if for all real constants $c > 0$ there is a natural number N_0 such that $g(N) \leq c * f(N)$ for all $N \geq N_0$.

In the following text, the particular algorithms for line clipping will be described in details. Currently, their complexity will be also shown and the big-O notation will be often used for the most of mentioned algorithms.

3. Clipping by a rectangular window

The most frequent case in 2D graphics is line or line segment clipping against a rectangular window. Let us assume that we have a line segment with end-points *$A(x_A, y_A)$* and *$B(x_B, y_B)$* and a clipping rectangle determined by two points *(x_{min}, y_{min})* and *(x_{max}, y_{max})* in E^2. The most famous and popular algorithm for clipping by rectangular window in E^2 is the Cohen-Sutherland (CS) algorithm for line segment clipping and Liang-Barsky (LB) algorithm for line clipping.

3.1. Cohen-Sutherland algorithm

The CS algorithm is based on coding of the end-points of the given line segment. According to the position against the clipping window, the end-point is encoded by the four-bit code, see Figure 3.1.

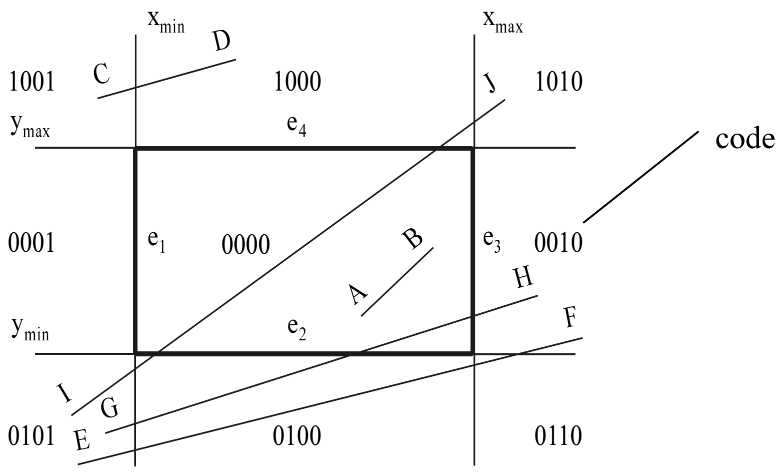


Figure 3.1: Region codes used in CS algorithm.

The least significant bit of the code is set to “1” if the point is to the left of the clipping window, i.e. $x < x_{min}$. The second bit indicates that the point is to the right of the clipping window, i.e. $x > x_{max}$. Similarly the third or fourth bit is set to “1” if the point is below or above the clipping window, respectively. Let c_A and c_B denote the codes of end-points *A* and *B*. It is obvious that if $c_A = 0$ and $c_B = 0$ then the given line segment is inside the clipping rectangle and is trivially accepted. In all the cases where at least one bit is set to “1” in both codes c_A and c_B, both end-points are outside the appropriate clipping boundary and the line segment is trivially rejected. In the other cases, the line segment intersects the appropriate clipping boundary and the intersection point replaces one end-point. The procedure is repeated with the curtailed line segment until it is rejected or accepted. The CS algorithm can be implemented by Algorithm 3.1.

```
global var $x_{min}$, $x_{max}$, $y_{min}$, $y_{max}$: real;  {clipping window size}
          {operators land and lor are bitwise and and or operators, respectively.}
{EXIT means leave the procedure}
procedure CS_Clip ( $x_A$, $y_A$, $x_B$, $y_B$: real);
var     x, y: real;
        c, $c_A$, $c_B$: integer;
procedure CODE (x, y: real; var c: integer); {implemented as a macro}
begin  c := 0;
        if $x < x_{min}$ then c := 1 else if $x > x_{max}$ then c := 2;
        if $y < y_{min}$ then c := c + 4 else if $y > y_{max}$ then c := c + 8;
end {CODE};

begin  CODE ($x_A$, $y_A$, $c_A$); CODE ($x_B$, $y_B$, $c_B$);
        if ($c_A$ land $c_B$) $\neq$ 0 then EXIT;  {the line segment is outside}
        if ($c_A$ lor $c_B$) = 0 then begin DRAW_LINE ($x_A$, $y_A$, $x_B$, $y_B$); EXIT  end;
                                {the line segment is inside the clipping rectangle}
        repeat if  $c_A \neq 0$ then c = $c_A$ else c = $c_B$;
                if (c land ‘0001’) $\neq$ 0 then                {divide line at the left edge}
                begin  y := $y_A + (x_{min} - x_A)*(y_B - y_A) / (x_B - x_A)$;
                        x := $x_{min}$
                end
                else    if (c land ‘0010’) $\neq$ 0 then          {divide line at the right edge}
                        begin  y := $y_A + (x_{max} - x_A)*(y_B - y_A) / (x_B - x_A)$;
                                x := $x_{max}$
                        end
                        else    if (c land ‘0100’) $\neq$ 0 then{divide line at the bottom edge}
                                begin  x := $x_A + (y_{min} - y_A)*(x_B - x_A) / (y_B - y_A)$;
                                        y := $y_{min}$
                                end
                                else    if (c land ‘1000’) $\neq$ 0 then    {at the top edge}
                                        begin  x := $x_A + (y_{max} - y_A)*(x_B - x_A) / (y_B - y_A)$;
                                                y := $y_{max}$
                                        end;
                if  c = $c_A$ then begin $x_A$ := x; $y_A$ := y; CODE ($x_A$, $y_A$, $c_A$) end
                                else begin $x_B$ := x; $y_B$ := y; CODE ($x_B$, $y_B$, $c_B$) end;
                if ($c_A$ land $c_B$) $\neq$ 0 then EXIT;
        until ($c_A$ lor $c_B$) = 0;
        DRAW_LINE ($x_A$, $y_A$, $x_B$, $y_B$);
end {CS_Clip};
```

Algorithm 3.1: Cohen-Sutherland algorithm.

It can be seen that the CS algorithm is very simple and robust. It enables the detection of all the cases where the line segment is completely inside the given rectangle and cases where the line segment has both end-points outside a particular clipping boundary very quickly. In Figure 3.1, the segments *AB* and *CD* are handled in a very simple way. However the line segments *EF* and *GH* are not distinguished at all and full intersection computations must be done. In the worst case, see line segment *IJ*, all intersection points with each boundary line, on which the rectangle edges lie, are computed whereas only two intersection points form the end-points of the clipped line segment.

3.2. LSSB algorithm for line segment clipping

Detailed analysis of the CS algorithm and a question if there is any meaning of the arithmetic sum of end-points' codes resulted in the new algorithm (LSSB). It is based on the CS algorithm but the arithmetic sum of end-points' codes and a new coding technique for the line segment direction are used in order to remove cases which the original CS algorithm is unable to distinguish. The LSSB algorithm for line segment clipping was developed, verified and published in ***Proceedings of the international conference SCCG'97***, see [Bui97a] and in ***The Visual Computer***, see [Bui98a].

Let us assume characteristic situations from Figure 3.2 and let c_A, c_B denote CS codes of line segment's end-points; $\alpha,\gamma,\eta,\omega$ areas denote corner areas and β,θ,ξ,δ denote side areas.

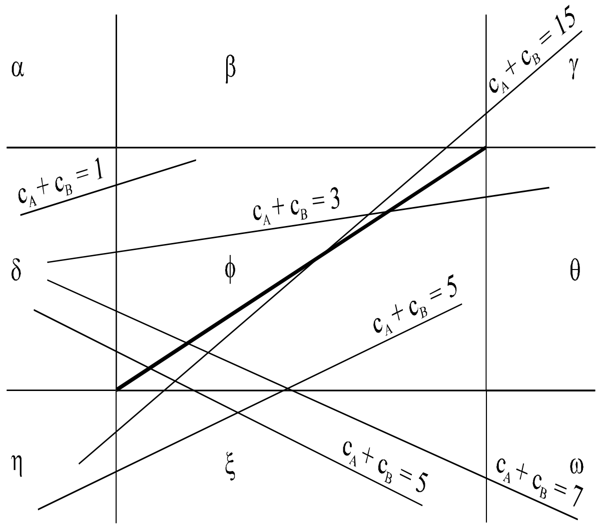


Figure 3.2: Arithmetic sum of end-points' codes enables to classify cases.

By testing arithmetic sum of end-points' codes we can distinguish the following situations:

- One end-point of the line segment is inside the clipping rectangle and the second one is in the side area (the cases $c_A + c_B \in \{1,2,4,8\}$). In these cases, only one intersection point with the known edge is computed.
- The end-points of the line segment are in the opposite side areas (the cases $c_A + c_B \in \{3,12\}$). In these cases, two intersection points are computed (the clipping edges are already determined).
- One end-point of the line segment is in the corner area of the clipping rectangle and the second one is in the side area (the cases $c_A + c_B \in \{7,11,13,14\}$). In these cases, one clipping edge (if exists) is already determined and an additional test will help to discover if the second one is opposite or neighboring to the first.
- The cases when $c_A + c_B \in \{5,6,9,10\}$. There are two possible situations:
  a) The end-points of the line segment are in the near-by side areas, i.e. (δ,β), (δ,ξ), (θ,β), (θ,ξ). Two clipping edges (if exist) are already determined.
  b) One end point of the line segment is inside of the clipping rectangle and the second one is in the corner area. The only one intersection point can lie on the horizontal or vertical edge of clipping rectangle and we need one test more to discover it.
- The end-points of the line segment are in the opposite corner areas (the cases $c_A + c_B = 15$). The comparison between directions of the given line and the clipping rectangle's diagonal decides which edges (horizontal or vertical) are used to compute the intersection points first.

Distinguishing all those cases enables avoidance of unnecessary calculation and gains considerable speed-up. Detailed description of the proposed LSSB algorithm is shown in Algorithm 3.2.

Table 3.1 shows the theoretical and experimental comparisons between new LSSB and CS algorithm for 21 different cases in Figure 3.3. All the cases can be derived from those presented by symmetry or rotation. The coefficient of efficiency ν was computed as:

$$\nu = \frac{T_{CS}}{T_{LSSB}}$$

where T_{CS}, T_{LSSB} are times consumed by the CS and LSSB algorithms, respectively.

```
procedure LSSB_Clip ( $x_A$, $y_A$, $x_B$, $y_B$: real);
{EXIT means leave the procedure}
var     $\Delta x$, $\Delta y$, $k$, $m$, $r$ : real;  $c_A$ , $c_B$ : integer;
        procedure  CODE (x,y: real; var c: integer); {implemented as a macro}
        begin  $c := 0$;
               if $x < x_{min}$ then $c := 1$ else if $x > x_{max}$ then $c := 2$;
               if $y < y_{min}$ then $c := c + 4$ else if $y > y_{max}$ then $c := c + 8$;
        end {CODE};

begin
   CODE ($x_A$, $y_A$, $c_A$); CODE ($x_B$, $y_B$, $c_B$);
   if ($c_A$ land $c_B$) <> 0 then EXIT; {the line segment is outside}
   if ($c_A$ lor $c_B$) = $0$ then {the line segment is inside the clipping rectangle}
       begin DRAW_LINE($x_A$, $y_A$, $x_B$, $y_B$); EXIT; end;
   $\Delta x := x_B - x_A$; $\Delta y := y_B - y_A$;
   case $c_A + c_B$ of {see Figure 3.2}
     1: if $c_A = 1$ then begin $x_A := x_{min}$; $y_A := (x_{min} - x_B) * \Delta y / \Delta x + y_B$ end
                     else begin $x_B := x_{min}$; $y_B := (x_{min} - x_A) * \Delta y / \Delta x + y_A$ end;
     3: begin $k := \Delta y / \Delta x$; $y_A := (x_{min} - x_A) * k + y_A$; $x_A := x_{min}$;
             $y_B := (x_{max} - x_B) * k + y_B$; $x_B := x_{max}$
        end;
     5: begin  $k := \Delta y / \Delta x$; $r := (x_{min} - x_A) * k + y_A$;
               if $r < y_{min}$ then
                      case $c_A$ of
                          0: begin  $x_B := x_B + (y_{min} - y_B)/k$; $y_B := y_{min}$ end;
                          5: begin  $x_A := x_A + (y_{min} - y_A)/k$; $y_A := y_{min}$ end;
                          else  EXIT {the line segment is outside}
                      end
               else  case $c_A$ of
                          0: begin $x_B := x_{min}$; $y_B := r$ end;
                          1: begin $x_B := x_B + (y_{min} - y_B)/k$; $y_B := y_{min}$; $x_A := x_{min}$; $y_A := r$ end;
                          4: begin $x_A := x_A + (y_{min} - y_A)/k$; $y_A := y_{min}$; $x_B = x_{min}$; $y_B := r$ end;
                          5: begin $x_A := x_{min}$; $y_A := r$ end
                      end
        end;
     7: case  $c_A$  of
          1: begin $k := \Delta y / \Delta x$; $y_A := (x_{min} - x_B) * k + y_B$;
               if $y_A < y_{min}$ then EXIT;{the line segment is outside}
               $x_A := x_{min}$; $y_B := (x_{max} - x_{min}) * k + y_A$;
               if $y_B < y_{min}$ then begin $x_B := (y_{min} - y_B)/k + x_{max}$; $y_B := y_{min}$ end else $x_B := x_{max}$
             end;
          {similarly for cases $c_A = 2, 5, 6$}
        end;
    15: case $c_A$ of
          5: if $\Delta y * (x_{max} - x_{min}) < \Delta x * (y_{max} - y_{min})$  then
             begin $k := \Delta y / \Delta x$; $y_A := (x_{min} - x_B) * k + y_B$;
                 if $y_A > y_{max}$ then EXIT; {the line segment is outside}
                 $y_B := (x_{max} - x_{min}) * k + y_A$;
                 if $y_B < y_{min}$ then EXIT; {the line segment is outside}
```

```
                if $y_A < y_{min}$ then begin $x_A := (y_{min} - y_A)/k + x_{min}$; $y_A := y_{min}$; $x_B := x_{max}$ end
                else begin $x_A := x_{min}$;
                        if $y_B > y_{max}$ then begin $x_B := (y_{max} - y_B)/k + x_{max}$; $y_B := y_{max}$ end
                                      else $x_B := x_{max}$
                    end;
            end else
            begin $m := \Delta x / \Delta y$; $x_A := (y_{min} - y_B) * m + x_B$;
                if $x_A > x_{max}$ then EXIT; {the line segment is outside}
                $x_B := (y_{max} - y_{min}) * m + x_A$;
                if $x_B < x_{min}$ then EXIT; {the line segment is outside}
                if $x_A < x_{min}$ then begin $y_A := (x_{min} - x_A)/m + y_{min}$; $x_A := x_{min}$; $y_B := y_{max}$ end
                else begin $y_A := y_{min}$;
                        if $x_B > x_{max}$ then begin $y_B := (x_{max} - x_B)/m + y_{max}$; $x_B := x_{max}$ end
                                      else $y_B := y_{max}$;
                    end;
            end
          {similarly for cases $c_A$ = 6, 9, 10}
      end;
    {cases 2, 4, 8 are similar as case 1, cases 6, 9, 10 are similar as case 5}
    {cases 11, 13, 14 are similar as case 7, case 12 is similar case 3}
  end {of case $c_A + c_B$};
  DRAW_LINE ($x_A, y_A, x_B, y_B$);
end {LSSB_Clip};
```

Algorithm 3.2: LSSB algorithm for line segment clipping.

It was theoretically proved that the LSSB algorithm is significantly faster than the CS algorithm in all **non-trivial** cases and experimental results proved that the expected speed up is in interval <**1.00 , 2.08**>, see [Bui97a], [Bui98a] for details.

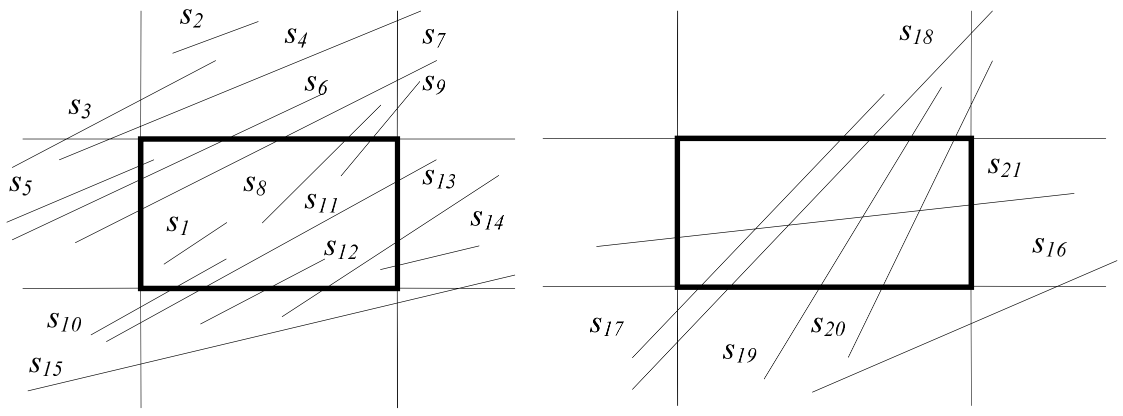


Figure 3.3: Generic cases for comparison between LSSB and CS Algorithm.

	Theoretical considerations													Exp. results		
	CS						LSSB						*v*	CS	LSSB	*v*
	=	<	±	x	/	t[s]	=	<	±	x	/	t[s]		t[s]	t[s]	
s_1	2	10	0	0	0	125,4	2	10	0	0	0	125,4	**1,00**	150,3	150,3	**1,00**
s_2	4	9	2	0	0	132,2	4	9	2	0	0	132,2	**1,00**	151,7	151,7	**1,00**
s_3	11	17	6	1	1	300,0	8	12	6	1	1	223,9	**1,34**	238,4	210,7	**1,13**
s_4	12	17	6	1	1	306,7	9	12	6	1	1	230,6	**1,33**	238,9	213,9	**1,12**
s_5	9	17	4	1	1	282,0	7	11	5	1	1	203,7	**1,38**	237,0	206,1	**1,15**
s_6	19	27	9	2	2	494,6	12	12	8	1	2	275,1	**1,80**	355,4	245,4	**1,45**
s_7	21	33	14	3	3	608,8	13	13	10	2	2	299,9	**2,03**	462,6	272,6	**1,70**
s_8	9	22	5	1	1	340,3	7	12	6	1	1	217,2	**1,57**	250,9	222,5	**1,13**
s_9	17	32	10	2	2	539,5	10	12	8	1	2	261,7	**2,06**	361,2	258,9	**1,39**
s_{10}	16	27	10	2	2	476,8	10	10	8	1	2	239,3	**1,99**	327,5	237,9	**1,38**
s_{11}	24	37	14	3	3	673,7	13	12	8	2	2	284,1	**2,37**	440,3	267,8	**1,64**
s_{12}	9	20	5	1	1	317,9	7	11	6	1	1	206,0	**1,54**	240,3	211,9	**1,13**
s_{13}	16	30	9	2	2	508,1	12	12	8	1	2	275,1	**1,85**	356,7	257,2	**1,39**
s_{14}	9	20	4	1	1	315,6	7	12	5	1	1	214,9	**1,47**	249,0	221,5	**1,12**
s_{15}	19	26	10	2	2	485,7	9	11	6	1	1	219,4	**2,21**	327,5	210,0	**1,56**
s_{16}	11	19	5	1	1	320,1	8	12	6	1	1	223,9	**1,43**	240,3	221,5	**1,08**
s_{17}	24	39	15	3	3	698,4	12	12	9	2	1	259,9	**2,69**	442,9	231,9	**1,91**
s_{18}	31	49	20	4	4	890,9	13	15	11	4	1	309,4	**2,88**	554,8	266,8	**2,08**
s_{19}	16	32	10	2	2	532,8	11	10	9	2	1	230,8	**2,31**	358,8	223,5	**1,61**
s_{20}	24	42	15	3	3	732,0	12	13	9	2	1	271,1	**2,70**	465,9	245,4	**1,90**
s_{21}	15	28	8	2	2	476,7	11	10	7	2	1	226,2	**2,11**	353,5	222,5	**1,59**

Table 3.1: Comparison between LSSB and CS Algorithm.

3.3. Liang-Barsky algorithm

In many applications it is necessary to clip lines instead of line segments. The well-known algorithm for the line clipping is the Liang-Barsky algorithm (LB) [Fol90]. It can be shown that CS algorithm is faster than the LB algorithm for line segment clipping. However, for line clipping, the CS algorithm cannot actually be used because the line has not any end-point. The LB is based on clipping of the given line by each rectangle's boundary. The given line which goes through the points $A(x_A, y_A)$ and $B(x_B, y_B)$ is parametrically represented as follows: $x(t) = x_A + \Delta x * t,$

$$y(t) = y_A + \Delta y * t,$$

where: $\Delta x = x_B - x_A,$, $\Delta y = y_B - y_A,$ $t \in (-\infty, +\infty)$

It is necessary to find the interval of parameter value *t*, for which $x_{min} \le x(t) \le x_{max}$ and $y_{min} \le y(t) \le y_{max}$, i.e. $x_{min} - x_A \le \Delta x * t \le x_{max} - x_A$ and $y_{min} - y_A \le \Delta x * t \le y_{max} - y_A$.

Without loss of generality we can consider only the common inequality

$$p * t \le q$$

It is obvious that this inequality has the following solution:

- $t \ge q/p$ if $p < 0$
- $t \le q/p$ if $p > 0$
- $\forall t$ if $p = 0$ and $q > 0$
- *not* $\exists t$ if $p = 0$ and $q < 0$

```
global var $x_{min}$, $x_{max}$, $y_{min}$, $y_{max}$: real;        {clipping window size; given values}
procedure LB_Clip ( $x_A$, $y_A$, $x_B$, $y_B$: real);
var     $t_1$, $t_2$, $\Delta x$, $\Delta y$ : real;

        function ClipT (p, q : real; var $t_1$, $t_2$ : real):boolean;
        var     r : real;
        begin  ClipT := true;
               if p < 0 then
                      begin  r := q / p;
                             if r > $t_2$ then ClipT := false
                                    else if r > $t_1$ then $t_1$:= r
                      end
               else   if p > 0 then
                             begin  r := q / p;
                                    if r < $t_1$ then ClipT := false
                                           else if r < $t_2$ then $t_2$:= r
                             end
                      else if q < 0 then ClipT := false
        end {ClipT};

begin
        $t_1$:= $-\infty$; $t_2$:= $+\infty$;  $\Delta x$ := $x_B$ - $x_A$;
        if ClipT($-\Delta x$, $x_A$ - $x_{min}$, $t_1$, $t_2$) then
               if ClipT($\Delta x$, $x_{max}$ - $x_A$, $t_1$, $t_2$) then
                      begin
                             $\Delta y$ := $y_B$ - $y_A$;
                             if ClipT($-\Delta y$, $y_A$ - $y_{min}$, $t_1$, $t_2$) then
                                    if ClipT($\Delta y$, $y_{max}$ - $y_A$, $t_1$, $t_2$) then
                                           begin  $x_B$ := $x_A$ + $\Delta x$ * $t_2$;
                                                  $y_B$ := $y_A$ + $\Delta y$ * $t_2$;
                                                  $x_A$ := $x_A$ + $\Delta x$ * $t_1$;
                                                  $y_A$ := $y_A$ + $\Delta y$ * $t_1$;
                                                  DRAW_LINE($x_A$, $y_A$, $x_B$, $y_B$)
                                           end
                      end
end { of LB_Clip };
```

Algorithm 3.3: Liang-Barsky algorithm.

At the beginning the parameter t is limited by interval $(-\infty, +\infty)$ and then this interval is subsequently curtailed by all the intersection points with each boundary line of the clipping rectangle, see Algorithm 3.3. It can be seen that an additional trivial rejection test (function *ClipT*) is used to avoid calculation of all four parameter values for lines that do not intersect the clipping rectangle.

3.4. LSB algorithm for line clipping

The new LSB algorithm was suggested to clip lines against a rectangular window, see [Bui97a], [Bui98a]. As mentioned above, the CS algorithm is faster than the LB algorithm for line segment clipping but for line clipping the CS algorithm cannot actually be used. Detailed analysis of the LB algorithm was made and understanding of inefficient parts led to a new algorithm for line clipping denoted LSB algorithm. The proposed LSB algorithm for line clipping is based on a new coding technique for line slope. The comparison between slope of the given line and clipping rectangle's diagonal decides which edges (horizontal or vertical) should be used first to compute the intersection points between the line and the clipping window, see Algorithm 3.4. This comparison is used to avoid the calculation of the intersection points that do not lie on the boundary edges of the clipping rectangle.

The theoretical comparison between new LSB and LB algorithm was made and it is presented in the first part of Table 3.2. The second part of Table 3.2 contains experimental results obtained for Pentium-75MHz/32MB RAM. Comparisons were done for 10 generic different cases, see Figure 3.4. All the cases can be derived from those presented by symmetry or rotation. The coefficient of efficiency ν in Table 3.2 was computed as:

$$\nu = \frac{T_{LB}}{T_{LSB}}$$

where T_{LB}, T_{LSB} are times consumed by the LB and LSB algorithms.

Table 3.2 shows that the LSB algorithm is significantly faster than the LB algorithm theoretically as well as experimentally and the speed-up can be expected in <**1.2 , 3.0**>, see [Bui97a], [Bui98a] for details.

```
procedure LSB_Clip ( $x_A$, $y_A$, $x_B$, $y_B$: real);
{EXIT means leave the procedure}
var     $\Delta x$, $\Delta y$, $k$, $m$: real;

begin
    $\Delta x := x_B - x_A$;
    if $\Delta x = 0$ then begin
                        if $(x_A < x_{min})$ or $(x_A > x_{max})$ then EXIT; {the line is outside}
                        $y_A := y_{min}$; $y_B := y_{max}$; DRAW_LINE($x_A$, $y_A$, $x_B$, $y_B$); EXIT
                    end;
    $\Delta y := y_B - y_A$;
    if $\Delta y = 0$ then begin
                        if $(y_A < y_{min})$ or $(y_A > y_{max})$ then EXIT; {the line is outside}
                        $x_A := x_{min}$; $x_B := x_{max}$; DRAW_LINE($x_A$, $y_A$, $x_B$, $y_B$); EXIT
                    end;
    if $\Delta x > 0$ then
      if $\Delta y > 0$ then
        if $(\Delta y*(x_{max} - x_{min}) < \Delta x*(y_{max} - y_{min}))$ then
        begin $k := \Delta y / \Delta x$; $y_A := (x_{min} - x_B)* k + y_B$;
           if $y_A > y_{max}$ then EXIT; {the line is outside the clipping rectangle}
           $y_B := (x_{max} - x_{min})* k + y_A$;
           if $y_B < y_{min}$ then EXIT; {the line is outside the clipping rectangle}
           if $y_A < y_{min}$ then begin $x_A := (y_{min} - y_A)/k + x_{min}$; $y_A := y_{min}$; $x_B := x_{max}$ end
                     else begin $x_A := x_{min}$;
                                  if $y_B > y_{max}$ then begin $x_B := (y_{max} - y_B)/k + x_{max}$;
                                                          $y_B := y_{max}$
                                                      end
                                              else $x_B := x_{max}$
                          end
        end
        else
        begin $m := \Delta x / \Delta y$; $x_A := (y_{min} - y_B)* m + x_B$;
           if $x_A > x_{max}$ then EXIT; {the line is outside of the clipping rectangle}
           $x_B := (y_{max} - y_{min})* m + x_A$;
           if $x_B < x_{min}$ then EXIT; {the line is outside of the clipping rectangle}
           if $x_A < x_{min}$ then begin $y_A := (x_{min}-x_A)/m + y_{min}$; $x_A := x_{min}$; $y_B := y_{max}$ end
                     else begin $y_A := y_{min}$;
                                  if $x_B > x_{max}$ then begin $y_B := (x_{max} - x_B)/m + y_{max}$;
                                                          $x_B := x_{max}$
                                                      end
                                              else $y_B := y_{max}$
                            end
         end;
    {similarly for the other cases}
    DRAW_LINE($x_A$, $y_A$, $x_B$, $y_B$);
end; {LSB_Clip}
```

Algorithm 3.2: LSB algorithm for line clipping.

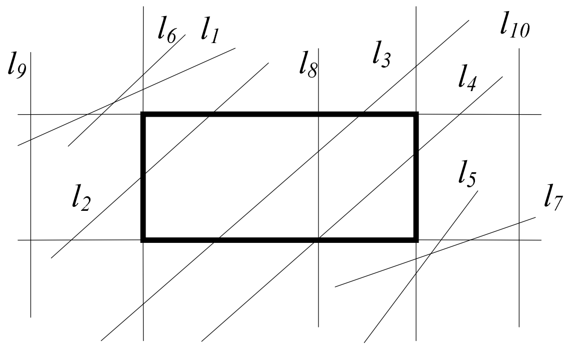


Figure 3.4: Generic lines for comparison between LSB and LB Algorithm.

	Theoretical considerations													Exp. results		
	L B						LSB						v	LB	LSB	v
	=	<	±	x	/	t[s]	=	<	±	x	/	t[s]		t[s]	t[s]	
l_1	10	13	6	0	4	305,6	5	6	8	4	1	148,1	**2,06**	303,5	191,7	**1,58**
l_2	15	14	10	4	4	368,7	8	7	10	4	2	203,8	**1,81**	349,2	237,0	**1,47**
l_3	16	14	10	4	4	375,4	7	8	8	4	1	183,9	**2,04**	351,1	216,3	**1,62**
l_4	15	14	10	4	4	368,7	8	8	10	4	2	215,0	**1,71**	349,2	248,1	**1,41**
l_5	9	9	5	0	3	232,0	4	5	6	3	1	123,3	**1,88**	241,3	167,6	**1,44**
l_6	10	13	6	0	4	305,6	4	5	6	3	1	123,3	**2,48**	303,4	167,1	**1,82**
l_7	9	9	5	0	3	232,0	5	6	8	4	1	148,1	**1,57**	241,3	191,2	**1,26**
l_8	12	13	10	4	2	297,8	3	3	1	0	0	56,0	**5,32**	297,6	98,7	**3,01**
l_9	3	3	2	0	0	58,3	1	2	1	0	0	31,4	**1,86**	101,7	85,7	**1,19**
l_{10}	3	6	3	0	0	94,2	1	3	1	0	0	42,6	**2,21**	134,4	96,8	**1,39**

Table 3.2: Comparison between LSB and LB Algorithm.

3.5. Algorithms using the separation function (SF)

3.5.1. The SF algorithm for line clipping

Further study of line clipping problem against a rectangular window led to another algorithm for line clipping that is more efficient, see [Bui99b]. The new separation function algorithm (denoted SF algorithm) is based on a new coding technique for vertices of the given clipping rectangle. It is obviously that the given line p divides the whole plane into two half-planes, see Figure 3.5, defined by a separation function. If the separation function is defined as:

$$F(x,y) = a*x + b*y + c$$

where $a = \Delta y = y_B - y_A$, $b = -\Delta x = x_A - x_B$, $c = x_B*y_A - x_A*y_B$.

then the sign of the separation function value $F(V_i)$ ($i \in [1,4]$) in the i-th vertex of the rectangular window determines the half-plane in which the vertex lies. Using the value of the separation function in all vertices we can distinguish 7 fundamental cases, see Figure 3.6.

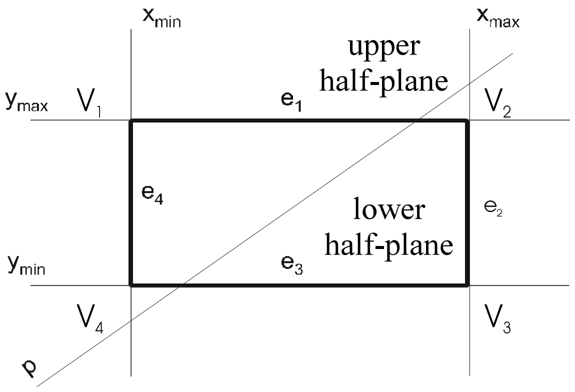


Figure 3.5: Line p divides the plane into two half-planes.

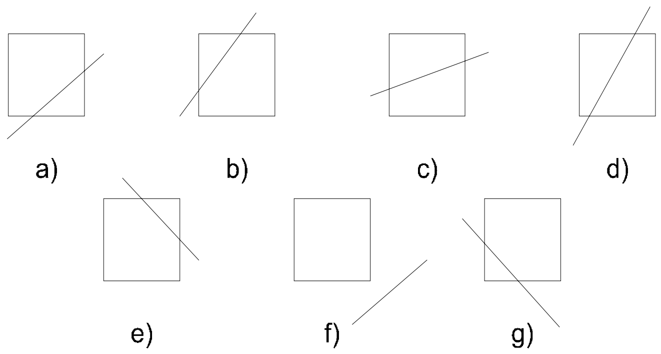


Figure 3.6: Fundamental mutual positions between the line and clipping window.

This observation led naturally to the new SF algorithm. The basic steps can be defined as:

- calculate the coefficients a, b, c of separation function F for the given line p,
- use the separation function F to characterise the location of vertices of the given clipping rectangle,
- determine the appropriate case,
- compute the intersection points with appropriate edges.

It can be seen that only the intersection points required for the output are computed. Now, we will describe the classification process more in detail.

Let us denote c_1, c_2, c_3, c_4 values of the separation function in the clipping rectangle's vertices V_1, V_2, V_3, V_4, respectively, see Figure 3.5.

There are two major cases to be distinguished:

- the vertices V_1 and V_3 lie on the different sides of line *p*, see Figure 3.6.a-d,
- the vertices V_1 and V_3 lie on the same side of line *p*, see Figure 3.6.e-f.

a) The vertices V_1 and V_3 are in the different sides of the line, i.e. $c_1 * c_3 < 0$. In this case, the sign of expression (c_2*c_4) determines whether V_2 and V_4 lie on the same sides of line *p* (Figure 3.6.a-b) or not (Figure 3.6.c-d).

If the vertices V_2 and V_4 lie on the same side of line *p*, the additional test $c_1*c_2>0$ determines on which edges the intersection points lie. If $c_1*c_2 > 0$ then the intersection points lie on the edges e_2 and e_3, see Figure 3.6.a. Otherwise, the intersection points lie on the edges e_1 and e_4, see Figure 3.6.b.

In the case that V_2 and V_4 lie on the different sides of line *p*, if $c_1*c_2 > 0$ then the intersection points lie on the edges e_2 and e_4, see Figure 3.6.c. Otherwise, the intersection points lie on the edges e_1 and e_3, see Figure 3.6.d.

b) The vertices V_1 and V_3 lie on the same side of the line. In this case, if $c_1*c_2 < 0$, i.e. V_1 and V_2 lie on the different sides of the line, then the intersection points lie on the edges e_1 and e_2, see Figure 3.6.e. Otherwise, the additional test $c_1*c_4 > 0$ determines whether the whole line is outside of clipping rectangle, see Figure 3.6.f, or the intersection points lie on the edges e_3 and e_4, see Figure 3.6.g.

The complete SF algorithm can be implemented by the Algorithm 3.5.

```
procedure SF_Clip ( $x_A$, $y_A$, $x_B$, $y_B$: real);
{points A($x_A$, $y_A$,) and B($x_B$, $y_B$) determine the clipped line}
global var $x_{min}$, $x_{max}$, $y_{min}$, $y_{max}$: real;  {co-ordinates of clipping window corners}
var    t, $\Delta x$, $\Delta y$, c, $c_1$, $c_2$, $c_3$, $c_4$ : real;
begin  $\Delta x := x_B - x_A$;
       if  $\Delta x = 0$ then
              begin  if ($x_A < x_{min}$) or ($x_A > x_{max}$) then EXIT; {the line is outside}
                     $y_A := y_{min}$; $y_B := y_{max}$;
                     DRAW_LINE ($x_A$, $y_A$, $x_B$, $y_B$); EXIT {SF_Clip}
              end;
       $\Delta y := y_B - y_A$;
       if  $\Delta y = 0$ then
              begin  if ($y_A < y_{min}$) or ($y_A > y_{max}$) then EXIT; {the line is outside }
                     $x_A := x_{min}$; $x_B := x_{max}$;
                     DRAW_LINE ($x_A$, $y_A$, $x_B$, $y_B$); EXIT {SF_Clip}
              end;
```

```
    c := $x_B * y_A – x_A * y_B$;  $c_1 := \Delta y * x_{min} – \Delta x * y_{max} + c$;
    $c_2 := \Delta y * x_{max} – \Delta x * y_{max} + c$; $c_3 := \Delta y * x_{max} – \Delta x * y_{min} + c$;
    **if** ($c_1 * c_3 < 0$) **then**
        **begin** $c_4 := \Delta y * x_{min} – \Delta x * y_{min} + c$;
            **if** ($c_2 * c_4 > 0$) **then**
                **if** ($c_1 * c_2 > 0$) **then**                {case a}
                    **begin** $y_B := y_A + (x_{max} - x_A) * \Delta y / \Delta x$; $x_B := x_{max}$;
                        $x_A := x_A + (y_{min} – y_A) * \Delta x / \Delta y$; $y_A := y_{min}$
                    **end**
                **else**                                  {case b}
                    **begin** $x_B := x_A + (y_{max} – y_A) * \Delta x / \Delta y$; $y_B := y_{max}$;
                        $y_A := y_A + (x_{min} - x_A) * \Delta y / \Delta x$; $x_A := x_{min}$
                    **end**
            **else**                                      {($c_2 * c_4 < 0$)}
                **if** ($c_1 * c_2 > 0$) **then**                {case c}
                    **begin**  $t := \Delta y / \Delta x$;
                        $y_B := y_A + (x_{max} - x_A) * t$; $x_B := x_{max}$;
                        $y_A := y_A + (x_{min} - x_A) * t$; $x_A := x_{min}$
                    **end**
                **else**  **begin**  $t := \Delta x / \Delta y$;         {case d}
                        $x_B := x_A + (y_{max} – y_A) * t$; $y_B := y_{max}$;
                        $x_A := x_A + (y_{min} – y_A) * t$; $y_A := y_{min}$
                    **end**
        **end**
    **else**                                              {($c_1 * c_3 > 0$)}
        **begin** **if** ($c_1 * c_2 < 0$) **then**                 {case e}
                **begin** $y_B := y_A + (x_{max} - x_A) * \Delta y / \Delta x$; $x_B := x_{max}$;
                    $x_A := x_A + (y_{max} – y_A) * \Delta x / \Delta y$; $y_A := y_{max}$
                **end**
            **else**  **begin** $c_4 := \Delta y * x_{min} – \Delta x * y_{min} + c$;
                    **if** ($c_1 * c_4 > 0$) **then EXIT**;     {case f}
                    **else**                              {case g}
                        **begin** $x_B := x_A + (y_{min} – y_A) * \Delta x / \Delta y$; $y_B := y_{min}$;
                            $y_A := y_A + (x_{min} - x_A) * \Delta y / \Delta x$; $x_A := x_{min}$
                        **end**
                **end**
        **end**;
    **DRAW_LINE** ($x_A, y_A, x_B, y_B$)
**end** {SF_Clip};
```

Algorithm 3.5: SF algorithm.

3.5.2. Modified SF algorithm

It can be seen that some modifications can be done to improve the efficiency of SF algorithm.

a) The first modification is based on the observation that the co-ordinates of the intersection points can be calculated from the separation function value of the clipping window's vertices. It is very simply to derive the following expressions:

- $x = x_A + (y_{max} - y_A) * \Delta x/\Delta y = x_{min} - c_1/\Delta y$

  (the intersection point on the top boundary)
- $y = y_A + (x_{max} - x_A) * \Delta y/\Delta x = y_{max} + c_2/\Delta x$

  (the intersection point on the right boundary)
- $x = x_A + (y_{min} - y_A) * \Delta x/\Delta y = x_{min} - c_4/\Delta y$

  (the intersection point on bottom boundary)
- $y = y_A + (x_{min} - x_A) * \Delta y/\Delta x = y_{max} + c_1/\Delta x$

  (the intersection point on the left boundary)

It can be seen that we can save one addition and one multiplication for each intersection point by using these expressions.

b) The better results can be obtained while replacing the direct calculation of c_2, c_3, c_4 by using the pre-calculated values as follows:

- $c_2 = \Delta y * x_{max} - \Delta x * y_{max} + c = c_1 + \Delta y * w$
- $c_3 = \Delta y * x_{max} - \Delta x * y_{min} + c = c_2 + \Delta x * h$
- $c_4 = \Delta y * x_{min} - \Delta x * y_{min} + c = c_1 + \Delta x * h$

where: $w = x_{max} - x_{min}$ (the clipping window's width)

$h = y_{max} - y_{min}$ (the clipping window's height)

c) We can get further speed-up when applying the following replacements: instead of two statements ($c := x_B * y_A - x_A * y_B;\ c_1 := \Delta y * x_{min} - \Delta x * y_{max} + c$), we can use only one ($c_1 := \Delta y * (x_{min} - x_A) - \Delta x * (y_{max} - y_A)$) and instead of the condition ($c_1 * c_3 < 0$), we can use the condition ($c_1 * (c_2 + \Delta x * h) < 0$).

All of above mentioned modifications can be implemented by the MSF algorithm, see Algorithm 3.6.

Since the conditions $\Delta x = 0$, $\Delta y = 0$ occur practically with **zero probability** and the test of these conditions can be left out without lost of stability or correctness, the further speed-up can be reached by removing this test. The shortened algorithm will be reported as the **MSF-1** algorithm.

```
procedure MSF_Clip ( $x_A$, $y_A$, $x_B$, $y_B$: real);
{points A($x_A$, $y_A$,) and B($x_B$, $y_B$) determine the clipped line}
global var $x_{min}$, $x_{max}$, $y_{min}$, $y_{max}$, h, w: real;
{corners' co-ordinates and size of clipping window}
var     t, $\Delta x$, $\Delta y$, $c_1$, $c_2$, $c_3$, $c_4$ : real;

begin  $\Delta x := x_B - x_A$;
       /* {tests if line is vertical or horizontal}
       if  $\Delta x = 0$ then
              begin  if ($x_A < x_{min}$) or ($x_A > x_{max}$) then EXIT; {the line is outside}
                     $y_A := y_{min}$; $y_B := y_{max}$;
                     DRAW_LINE ($x_A$, $y_A$, $x_B$, $y_B$); EXIT {MSF_Clip}
              end;
       $\Delta y := y_B - y_A$;
       if  $\Delta y = 0$ then
              begin  if ($y_A < y_{min}$) or ($y_A > y_{max}$) then EXIT; {the line is outside}
                     $x_A := x_{min}$; $x_B := x_{max}$;
                     DRAW_LINE ($x_A$, $y_A$, $x_B$, $y_B$); EXIT {MSF_Clip}
              end;
       /*{end of the section to be removed for MSF-1 algorithm}
       $c_1 := \Delta y * (x_{min} - x_A) - \Delta x * (y_{max} - y_A)$;          $c_2 := c_1 + \Delta y * w$;
       if $c_1 * (c_2 + \Delta x * h) < 0$ then
              begin  $c_4 := c_1 + \Delta x * h$;
                     if ($c_2 * c_4 > 0$) then
                            if ($c_1 * c_2 > 0$) then                          {case a}
                                   begin  $y_B := y_{max} + c_2 / \Delta x$; $x_B := x_{max}$;
                                          $x_A := x_{min} - c_4 / \Delta y$; $y_A := y_{min}$
                                   end
                            else                                               {case b}
                                   begin  $x_B := x_{min} - c_1 / \Delta y$; $y_B := y_{max}$;
                                          $y_A := y_{max} + c_1 / \Delta x$; $x_A := x_{min}$
                                   end
                     else                                                      {($c_2 * c_4 < 0$)}
                            if ($c_1 * c_2 > 0$) then                          {case c}
                                   begin    t := 1.0 / $\Delta x$;
                                          $y_B := y_{max} + c_2 * t$;        $x_B := x_{max}$;
                                          $y_A := y_{max} + c_1 * t$;        $x_A := x_{min}$
                                   end
                            else   begin    t := 1.0 / $\Delta y$;           {case d}
                                          $x_B := x_{min} - c_1 * t$;        $y_B := y_{max}$;
                                          $x_A := x_{min} - c_4 * t$;        $y_A := y_{min}$
                                   end
              end
       else                                                                    {($c_1 * c_3 > 0$)}
              begin  if ($c_1 * c_2 < 0$) then                                 {case e}
                            begin  $y_B := y_{max} + c_2 / \Delta x$; $x_B := x_{max}$;
                                   $x_A := x_{min} - c_1 / \Delta y$; $y_A := y_{max}$
                            end
                     else   begin  $c_4 := c_1 + \Delta x * h$;
```

```
                        if ($c_1 * c_4 > 0$) then EXIT;      {case f}
                        else                                 {case g}
                            begin  $x_B := x_{min} - c_4 / \Delta y$;   $y_B := y_{min}$;
                                   $y_A := y_{max} + c_1 / \Delta x$; $x_A := x_{min}$
                            end
                    end
            end;
        DRAW_LINE ($x_A, y_A, x_B, y_B$)
end { of MSF_Clip };
```

Algorithm 3.6: MSF algorithm.

3.5.3. Experimental results

For experimental verification of the LB and LSB algorithms and the proposed SF, MSF and MSF-1 algorithms, all fundamental cases were tested and 8.10^6 different lines were randomly generated for each considered case, see Figure 3.7 and Figure 3.8.

Let us introduce the coefficients of efficiency v_{LSB}, v_{MSF}, v_{MSF-1} as:

$$v_{LSB} = \frac{T_{LB}}{T_{LSB}}, \; v_{MSF} = \frac{T_{LB}}{T_{MSF}}, \; v_{MSF-1} = \frac{T_{LB}}{T_{MSF-1}}$$

where T_{LB}, T_{LSB}, T_{MSF}, T_{MSF-1} are times consumed by the LB, LSB algorithm and the modifications MSF and MSF-1 algorithms of the SF algorithm (the MSF-1 algorithm is the MSF algorithm without testing of conditions $\Delta x = 0$, $\Delta y = 0$).

Table 3.3 shows experimental results obtained for Pentium II-350MHz/64MB RAM/512KB CACHE, similar results were also obtained for Pentium-75MHz/32MB RAM and Pentium PRO-200MHz/128MB RAM. This table shows that the LSB and MSF algorithms are significantly faster than the LB algorithm in all cases. It can be seen that the speed-up varies from 1.3 to 1.87 for all common cases. The common case is the case when the given line is neither horizontal nor vertical (cases p_1-p_7).

The MSF-1 algorithm is based on the fact that strictly horizontal or vertical lines are highly impossible in normal situations. The speed-up of the MSF-1 algorithm can be expected from 1.7 to 2.16 for all common cases, see Table 3.3.

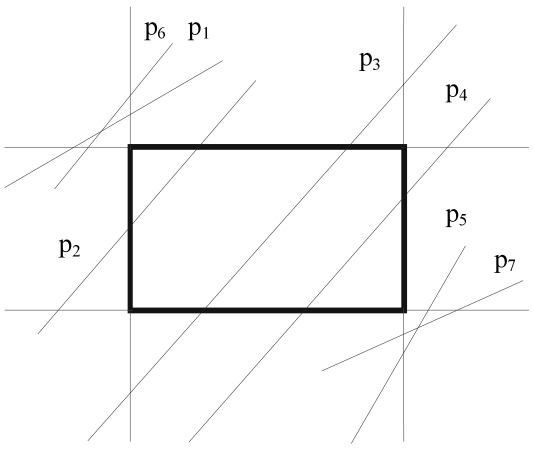


Figure 3.7: The common cases.

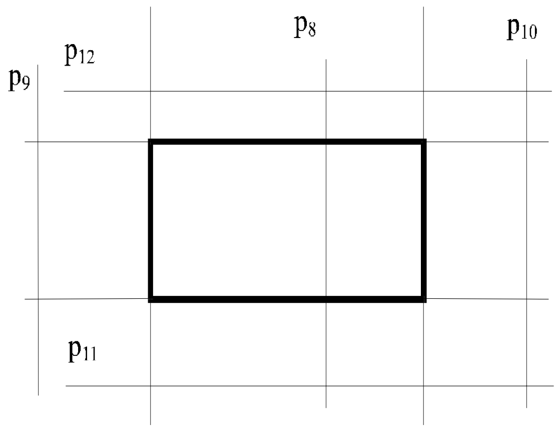


Figure 3.8: The special cases (horizontal or vertical).

	Pentium II 350MHz/64MB RAM		
case	v_{LSA}	v_{MSF}	v_{MSF-1}
p_1	1.63	1.87	**2.16**
p_2	1.55	1.59	1.70
p_3	1.65	1.74	1.87
p_4	1.30	1.53	1.68
p_5	1.42	1.49	1.72
p_6	1.81	1.87	**2.16**
p_7	1.37	1.49	1.72
p_8	**2.62**	**2.62**	1.52
p_9	1.16	1.16	0.75
p_{10}	1.26	1.26	0.94
p_{11}	2.19	2.20	1.55
p_{12}	2.17	2.19	1.75

Table 3.3: Comparison between LSB, MSF, MSF-1and LB algorithm.

3.6. Nicholl-Lee-Nicholl algorithm for line segment clipping

Nicholl, Lee and Nicholl have created a better 2D line segment clipper ([Nic87a], [Fol90a]) than the CS algorithm. Although the Nicholl-Lee-Nicholl (NLN) algorithm has great many cases, the basic idea is simple enough that understanding one case lets us generate all the others. Consider a line segment *AB* that is to be clipped. We first determine where point *A* lies. If we divide the plane into the same nine regions used in the CS algorithm, see Figure 3.1, then point *A* must lie in one of these regions. By determining the position of point *B* relative to the lines from *A* to each of corners, we can determine which edges of the clipping rectangle the line segment *AB* intersects.

Suppose that point *A* lies in the lower-left corner region, as in Figure 3.9. If point *B* lies below y_{min} or to the left of x_{min}, then the line segment *AB* cannot intersect the clipping rectangle (this amounts to checking the Cohen-Sutherland outcodes). The same is true if point *B* lies to the left of the line from *A* to the upper-left corner or if point *B* lies to the right of the line from *A* to the lower-right corner. Many cases can be trivially rejected by these checks. We also check the position of point *B* relative to the line from *A* to the lower-left corner. We will discuss the case when *B* is above this line, as shown in Figure 3.9. If point *B* is below the top of the clipping rectangle, it is either inside the clipping rectangle or to the right of it. Hence the line segment *AB* intersects the clipping rectangle either at its left edge or at both the left and right edges. If *B* is above the top of the clipping rectangle, it may be to the left of the line from *A* to the upper-left corner and the line segment *AB* is rejected. If not, it may be to the right of the right edge of the clipping rectangle. This later case divides into the two cases: *B* is to the left of the line from *A* to the upper-right corner and to the right of it. The regions in Figure 3.9 are labeled by edges cut by a segment from *A* to any point in those regions; LT, for example, means the line from *A* to any point in this region intersects both the left and top edges of the clipping rectangle.

The remaining cases, when *A* is inside the clipping rectangle or in one of the side regions, are similar. Thus, it is worthwhile to recognize the symmetries of the various cases and to write a program to transform three general cases (*A* is inside the clipping rectangle, in the corner or in the side region) into nine different cases.

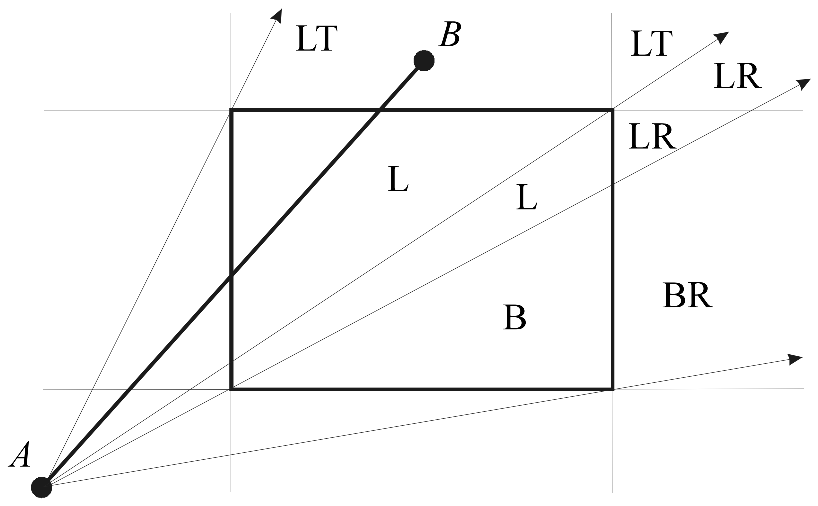


Figure 3.9: The regions determined by the lines from *A* to the corners.

In summary, for line clipping against a rectangular window, the CS algorithm is efficient when the outcode testing can be done cheaply (for example, by doing bitwise operations in assembly language) and trivial acceptance or rejection is applicable to the majority of line segments. For lines that cannot be trivially rejected by CS algorithm because they do not lie in an invisible half-plane, the rejection tests of LB algorithm are clearly preferable to the repeated clipping required by CS algorithm. The NLN algorithm is generally preferable to either CS or LB algorithm but does not generalize to 3D, as does CS and LB algorithms. The new LSSB algorithm was developed, verified and tested for clipping line segments and it claim the superiority over the CS algorithm. For clipping lines against a rectangular window, the new LSB and MSF algorithm claim the superiority over the LB algorithm for all cases. Experiments proved that the speed-up could be considered up to 1.6 times on the average for the LSB algorithm. The MSF and MSF-1 algorithms were also implemented as the modifications of the SF algorithm. The speed-up of MSF-1 algorithm can be considered up to 1.86 times on the average for all common cases.

4. Clipping by a convex polygon

Many algorithms for line or line segment clipping by convex polygon has been proposed, see references, nevertheless the Cyrus-Beck (CB) algorithm is the most often used because it is very simple to implement and numerically stable.

4.1. Cyrus-Beck algorithm

Let the convex clipping polygon be given by N points

$$\boldsymbol{x}_i = [x_i, y_i]^T \quad , i = 0, \dots ,N-1$$

where: points $\boldsymbol{x}_0$ and $\boldsymbol{x}_N$ are identical, x_i and y_i are co-ordinates of the vertex $\boldsymbol{x}_i$ (column notation is used), $\boldsymbol{n}_i$ is the normal vector of the edge $\boldsymbol{e}_i$ $(\boldsymbol{x}_i\ \boldsymbol{x}_{i+1})$ and points out of the convex polygon. The given line p is determined by two points:

$$\boldsymbol{x}_A = [x_A, y_A]^T \ , \qquad \boldsymbol{x}_B = [x_B, y_B]^T$$

The given line can be parametrically represented as follows:

$$\boldsymbol{x}(t) = \boldsymbol{x}_A + (\boldsymbol{x}_B - \boldsymbol{x}_A) * t,$$

or

$$x(t) = x_A + \Delta x * t,$$
$$y(t) = y_A + \Delta y * t,$$

where: $\Delta x = x_B - x_A, \quad \Delta y = y_B - y_A,$

$t \in (-\infty, +\infty)$ (for line clipping) or $t \in <0,1>$ (for line segment clipping)

Let us consider the vector $\boldsymbol{x}(t) - \boldsymbol{x}_i$ from vertex $\boldsymbol{x}_i$ to one point on line p. We must find the parameter value t at the intersection of line p with the edge $\boldsymbol{e}_i$, i.e. we have to solve the following equation:

$$\boldsymbol{n}_i^T\, [\boldsymbol{x}(t) - \boldsymbol{x}_i] = 0$$

First, substitute for $\boldsymbol{x}(t)$:

$$\boldsymbol{n}_i^T\, [\boldsymbol{x}_A + (\boldsymbol{x}_B - \boldsymbol{x}_A) * t - \boldsymbol{x}_i] = 0$$

Next, group terms and distribute the dot product:

$$\boldsymbol{n}_i^T\, [\boldsymbol{x}_A - \boldsymbol{x}_i] + \boldsymbol{n}_i^T\, [\boldsymbol{x}_B - \boldsymbol{x}_A] * t = 0$$

Let $\boldsymbol{s} = \boldsymbol{x}_B - \boldsymbol{x}_A$ be the vector from $\boldsymbol{x}_A$ to $\boldsymbol{x}_B$ then we can write:

$$t = \frac{\boldsymbol{n}_i^T\, [\,\boldsymbol{x}_i - \boldsymbol{x}_A\,]}{\boldsymbol{n}_i^T\, \boldsymbol{s}}$$

Note that this gives a valid value of t if and only if the denominator of the expression is nonzero. Therefore, the algorithm must check that

$\boldsymbol{n}_i \neq 0$ (i.e. the normal should not be $[0,\ 0]^T$)

$\boldsymbol{s} \neq 0$ (i.e. $\boldsymbol{x}_A \neq \boldsymbol{x}_B$)

$\boldsymbol{n}_i^T \boldsymbol{s} \neq 0$ (i.e. line p and the edge $\boldsymbol{e}_i$ are not parallel. If they were parallel then either the entire line must be rejected or the algorithm moves to the next vertex).

We calculate the parameter value t for all intersections between the given line p and each edge of the clipping polygon. The next step is to determine which (if any) of the values correspond to the internal intersections of the given line with edges of the clipping polygon, i.e. we need to determine whether the intersection lies on the clip boundary. The intersections are characterized as "potentially entering" (*PE*) or "potentially leaving" (*PL*) the clipping polygon, as follows: if moving from $\boldsymbol{x}_A$ to $\boldsymbol{x}_B$ causes us to cross a particular edge to enter the edge's inside half-plane, the intersection is *PE*. If it causes us to leave the edge's inside half-plane, it is *PL*. Notice that with this distinction, two interior intersection points of a line intersecting the clip polygon have opposing labels, see Figure 4.1. On another view, intersections can be classified as *PE* or *PL* on the basis of the non-orientated angle between $\boldsymbol{s}$ and $\boldsymbol{n}_i$: If this angle is less than 90^0, the intersection is *PL*. If it is greater than 90^0, it is *PE*. This information is contained in the sign of the dot product of $\boldsymbol{s}$ and $\boldsymbol{n}_i$. Notice that $\boldsymbol{n}_i^T \boldsymbol{s}$ is merely the denominator of expression for the parameter value t, which means that, in the process of calculating, the intersection can be trivially categorized.

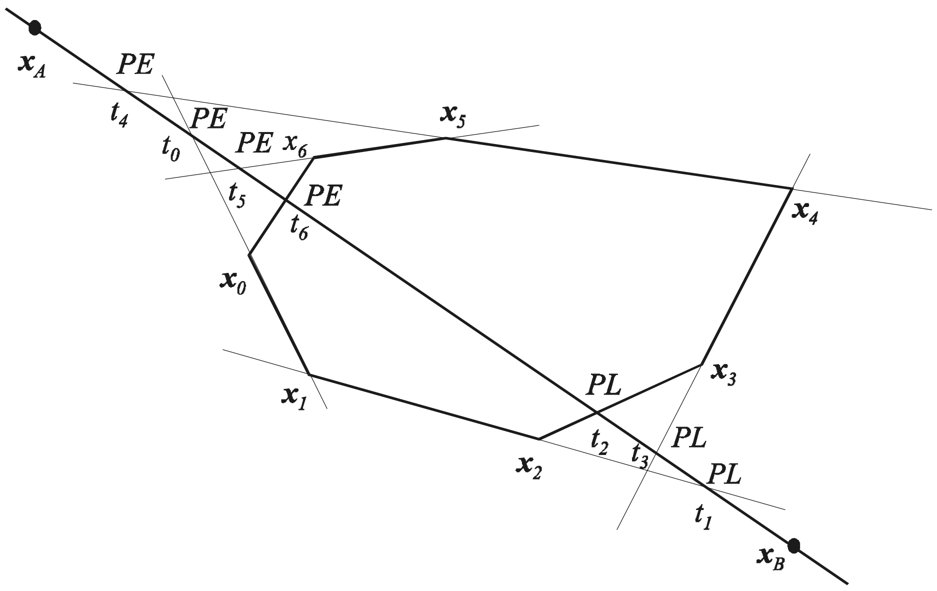


Figure 4.1: Line clipping by convex polygon.

With this categorization, we can do the final step in the process. We must choose a (*PE*, *PL*) pair that defines the clipped line. The portion of the given line *p* that is within the clipping polygon is bounded by the *PE* intersection with the largest *t* value called t_{min} and the *PL* intersection with the smallest *t* value called t_{max}. The range (t_{min}, t_{max}) then defines the clipped line segment. If $t_{min} > t_{max}$ then no portion of line *p* is within the clipping polygon and the entire line is rejected. The values of t_{min} and t_{max} are then used to calculate the corresponding *x* and *y* co-ordinates.

The CB algorithm can be implemented as Algorithm 4.1.

```
procedure CB_Clip ( $\mathbf{x}_A$, $\mathbf{x}_B$);
var   $t_{min}, t_{max}, t, k$ : real;
      $i$ : integer;
      {$\mathbf{n}_i$ is a normal vector of edge $\mathbf{e}_i$ $(\mathbf{x}_i\,\mathbf{x}_{i+1})$}
      {$\mathbf{n}_i$ points out of the convex clipping polygon}
      {all vectors $\mathbf{n}_i$ are precomputed}
begin
      $t_{min} := -\infty$; $t_{max} := +\infty$; {for line segment $t_{min} := 0$; $t_{max} := 1$;}
      $i := 0$;
      $\mathbf{s} := \mathbf{x}_B - \mathbf{x}_A$;
      while $i < N$ do {$N$ is the number of edges of the clipping polygon}
      begin
            $\mathbf{s}_i := \mathbf{x}_i - \mathbf{x}_A$; { $\mathbf{s}_i$ is the vector from $\mathbf{x}_A$ to $\mathbf{x}_i$ }
            $k := \mathbf{n}_i^T \mathbf{s}$;
            if $k <> 0$ then
            begin  $t := \mathbf{n}_i^T \mathbf{s}_i / k$;
                  if $k > 0$ then $t_{max} :=$ min $(t, t_{max})$
                        else $t_{min} :=$ max $(t, t_{min})$
            end
            else Special case solution;
            $i := i+1$;
      end;
      if $t_{min} > t_{max}$ then EXIT; {the given line is rejected}
      $\mathbf{x}_B := \mathbf{x}_A + \mathbf{s} * t_{max}$;
      $\mathbf{x}_A := \mathbf{x}_A + \mathbf{s} * t_{min}$;
      DRAW_LINE($\mathbf{x}_A$, $\mathbf{x}_B$)
end { CB_Clip };{EXIT means leave the procedure}
```

Algorithm 4.1: Cyrus-Beck algorithm.

The CB algorithm relies on a brute force as it computes intersections of all edges (or lines on which the edges lie) with the given line or with a line on which the given line segment lies, therefore it has *O(N)* complexity. It leads to ineffective algorithm, as ***N-2*** computed intersection points are lost because the only two can be valid.

In algorithm design there is a very old rule "***Make tests first and then compute***". There were several attempts to find an intersection detection method. One of them, known as an ECB (Efficient Cyrus-Beck) algorithm, used the separation function that is represented by the implicit function of the given line or by the cross product of two vectors and brought a significant increase of speed to the CB algorithm.

4.2. ECB algorithm

In the Figure 4.2, it is obvious that line p intersects the edge $\boldsymbol{x}_0\boldsymbol{x}_1$ of the given polygon **if and only if** the vector $\boldsymbol{s}$ lies between two vectors $\boldsymbol{s}_0$ and $\boldsymbol{s}_1$. Let ξ and η denote the z co-ordinate of the cross products as follows:

$$\xi = [\boldsymbol{s} \times \boldsymbol{s}_i]_z, \quad \eta = [\boldsymbol{s} \times \boldsymbol{s}_{i+1}]_z, \qquad i = 0, \ldots, N-1$$

where N is number of edges.

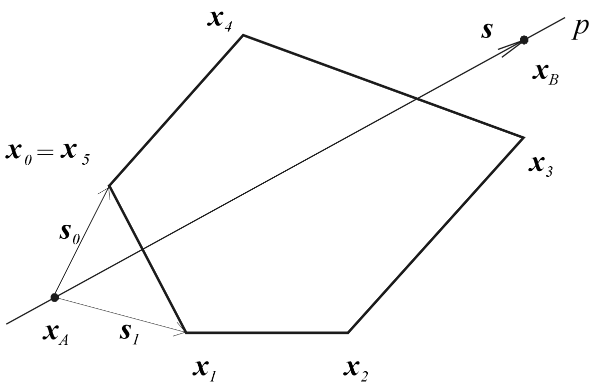


Figure 4.2: Line p intersect the edge $\boldsymbol{x}_0\boldsymbol{x}_1 \Leftrightarrow [\boldsymbol{s} \times \boldsymbol{s}_0]_z * [\boldsymbol{s} \times \boldsymbol{s}_1]_z < 0.$

It is possible to show that line p given by the end-point $\boldsymbol{x}_A$ and by the vector $\boldsymbol{s}$ intersects the edge $\boldsymbol{x}_i\boldsymbol{x}_{i+1}$ if and only if the following condition is valid:

$$(\xi > 0) \text{ xor } (\eta > 0), \text{ i.e. } \xi * \eta < 0$$

The above expressions are independent on polygon orientation. It means that edges needn't to be ordered. That is the main idea of the ECB algorithm, which is based on the presumption that a line can intersect a convex polygon at most in two points. The ECB algorithm implementation is shown in the Algorithm 4.2. It can be seen that the ECB algorithm only detects whether the given line intersects the clipping polygon's edge and then computes intersection points, while CB algorithm computes all possible intersection points with all edges. With cross product used as a separation function experiments proved the speed-up can be expected in <**1.37, 2.85**>, for details see [Ska93b]. It is necessary to point out that the detection function can be replaced by the implicit function of the given line with better speed-up.

```
procedure ECB_Clip ( $\boldsymbol{x}_A$,  $\boldsymbol{x}_B$);
{$N$ is the edge number of the clipping polygon}
var    $t_{min}$, $t_{max}$, $t_1$, $t_2$ : real;
       $i, j, k$ : integer;
begin
       $k := 0; i := N-1; j := 0;$
       $\mathbf{s} := \boldsymbol{x}_B - \boldsymbol{x}_A;$
       $\xi := [\mathbf{s}\ x\ (\boldsymbol{x}_{N-1} - \boldsymbol{x}_A)]_z;$          {$z$ co-ordinate of the cross product}
       while $(j < N)$ and $(k < 2)$ do
       begin
              $\mathbf{s}_j := \boldsymbol{x}_j - \boldsymbol{x}_A;$                  { $\mathbf{s}_j$ is the vector from $\boldsymbol{x}_A$ to $\boldsymbol{x}_j$ }
              $\eta = [\mathbf{s}\ x\ \mathbf{s}_j]_z;$                   {$z$ co-ordinate of the cross product}
              if $\xi * \eta < 0$ then              {intersection exists}
              begin                         {save edge index having intersection}
                     $k := k + 1;$
                     index $_k := i$
              end
              else if $\xi * \eta = 0$ then Special case solution
                            else Intersection point does not exist;
              $\xi := \eta;$
              $i := j;$
              $j := j+1$
       end;
       if $k = 0$ then EXIT;                  {the given line is rejected}
       { $k = 2$ Intersections exist – edges saved in index $_k$}
       $t_{min} := -\infty$; $t_{max} := +\infty$; {for line segment $t_{min} := 0$; $t_{max} := 1$;}
       $i :=$ index$_1$; $t_1 := det[\boldsymbol{x}_i - \boldsymbol{x}_A \mid \boldsymbol{x}_i - \boldsymbol{x}_{i+1}] / det[\mathbf{s} \mid \boldsymbol{x}_i - \boldsymbol{x}_{i+1}]$ ;
       $i :=$ index$_2$; $t_2 := det[\boldsymbol{x}_i - \boldsymbol{x}_A \mid \boldsymbol{x}_i - \boldsymbol{x}_{i+1}] / det[\mathbf{s} \mid \boldsymbol{x}_i - \boldsymbol{x}_{i+1}]$ ;
       {recompute end-point if changed}
       if $t_1 > t_2$ then { swap $t_1 \leftrightarrow t_2$ values ?}
              begin
                     if $t_1 < t_{min}$ or $t_2 > t_{max}$ then EXIT;{the line segment is rejected}
                     if $t_1 < t_{max}$ then $\boldsymbol{x}_B := \boldsymbol{x}_A + \mathbf{s} * t_1;$
                     if $t_2 > t_{min}$ then $\boldsymbol{x}_A := \boldsymbol{x}_A + \mathbf{s} * t_2$
              end
       else   begin
                     if $t_2 < t_{min}$ or $t_1 > t_{max}$ then EXIT;{the line segment is rejected}
                     if $t_2 < t_{max}$ then $\boldsymbol{x}_B := \boldsymbol{x}_A + \mathbf{s} * t_2;$
                     if $t_1 > t_{min}$ then $\boldsymbol{x}_A := \boldsymbol{x}_A + \mathbf{s} * t_1$
              end;
       DRAW_LINE($\boldsymbol{x}_A$, $\boldsymbol{x}_B$)
end {ECB_Clip};
```

Algorithm 4.2: ECB algorithm.

4.3. *O(log N)* algorithm

The ECB algorithm does not use the important property of the given clipping polygon: **known order** of vertices (polygon's vertices are given as ordered), thus it has the complexity *O(N)*. This remark led to the thought that such knowledge could be used to decrease the algorithm complexity.

It can be shown that the testing of whether a line intersects the convex polygon is the dual problem to the testing of whether a point is inside of the convex polygon if dual representation is used, see [Kol94a]. It is very well known that the problem testing if a point is inside of the convex polygon has optimal complexity of ***O(log N)***. This has naturally led to the question if there is a line clipping algorithm with *O(log N)* complexity. Let us assume the situation from Figure 4.3.

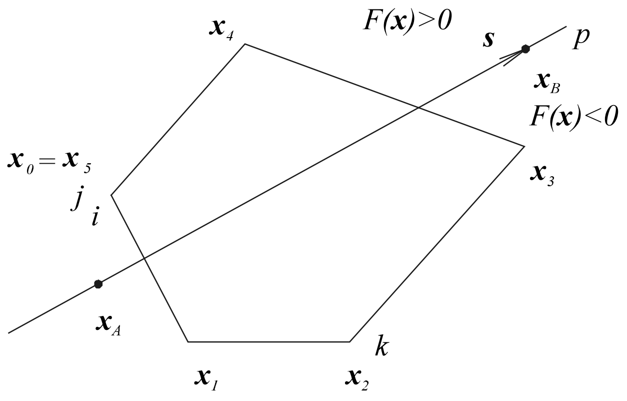


Figure 4.3: Line *p* divides the plane into two haft-plane with different sign of *F(**x**)*.

It can be seen that if we select an index *k* as follows:

$$k = \lfloor N/2 \rfloor$$

then we need only *log N* steps to find which edges are intersected on each chain of the edge segments, see Figure 4.3. Although some other cases are a little bit more complicated, it can be shown that the whole algorithm is of *O(log N)* complexity, see Algorithm 4.3. It is necessary to point out that for effective implementation values $F(\mathbf{x}_i)$ should be stored in separate variables because they are used several times.

This approach enabled to speed up line clipping significantly and it is faster than CB algorithm even for $N > 3$ (for $N = 100$ the speed up is over 10 times). See [Ska94a] for more details and analysis.

```
procedure CLIP _2D_log ($\mathbf{x}_A$, $\mathbf{x}_B$); {initiation for a clipping window $\mathbf{x}_N := \mathbf{x}_0$ }

    function macro $F(\mathbf{x})$: real;  {implemented as an in-line function }
        begin  $F := A * x + B * y + C$; end;

    function Solve ( $i$ , $j$ ): real;
    begin
        {finds two nearest vertices on the opposite sides of the given line $p$}
        while $(j - i) \geq 2$ do {$j \geq i$ always}
        begin  $k := (i + j)$ div $2$; {shift right }
            if $(F(\mathbf{x}_i) * F(\mathbf{x}_k) < 0)$ then $j := k$ else $i := k$;
        end { while };
        {gives the value $t$ for an point of line $p$ with the given segment $\mathbf{x}_i \mathbf{x}_j$ }
        Solve := Intersection ( $p$ , $\mathbf{x}_i$ , $\mathbf{x}_j$ );
    end { Solve };

begin
    {determine $A, B, C$ values for the $F(\mathbf{x})$ }
    $A := y_1 - y_2$ ; $B := x_2 - x_1$; $C := x_1 * y_2 - x_2 * y_1$;
    $i := 0$; $j := n$;
    while $(j - i) \geq 2$ do
    begin  $k := (i + j)$ div $2$;{shift to the right}
        if $(F(\mathbf{x}_i) * F(\mathbf{x}_k) < 0)$ then
        begin  $t_1$ := Solve ( $i$ , $k$ ) ; {finds an intersection on $\overline{\mathbf{x}_i\mathbf{x}_k}$ chain}
            $t_2$ := Solve ( $k$ , $j$ ); {finds an intersection on $\overline{\mathbf{x}_k\mathbf{x}_j}$ chain}
            /* {for line segment clipping include three following lines}
            if $t_1 > t_2$ then begin $t := t_2$; $t_2 := t_1$; $t_1 := t$  end;
            $t_1$:=max($0$, $t_1$);$t_2$:=min($1$, $t_2$);  {compute $<t_1 , t_2> \cap <0,1>$}
            if  $<t_1 , t_2> = \varnothing$ then EXIT; { exit procedure CLIP 2D log }
            */
            DRAW_LINE($\mathbf{x}(t_1)$, $\mathbf{x}(t_2)$)
        end { if };
        if $(F(\mathbf{x}_i) > 0)$ then
        begin
            if $F(\mathbf{x}_i) < F(\mathbf{x}_k)$ then
                if $F(\mathbf{x}_{i+1}) < F(\mathbf{x}_i)$  then $j := k$        {Delete $\overline{\mathbf{x}_k\mathbf{x}_j}$ chain}
                                         else $i := k$        {Delete $\overline{\mathbf{x}_i\mathbf{x}_k}$ chain}
            else
                if $F(\mathbf{x}_k) < F(\mathbf{x}_{k+1})$ then $j := k$         {Delete $\overline{\mathbf{x}_k\mathbf{x}_j}$ chain}
                                         else $i := k$        {Delete $\overline{\mathbf{x}_i\mathbf{x}_k}$ chain}
        end else
        begin {similarly for opposite line orientation}
        end
    end {while}
end {CLIP_2D_log}
```

Algorithm 4.3: *O(log N)* algorithm.

4.3.1. Modified *O(log N)* algorithm

It can be seen that the above mentioned *O(log N)* algorithm must indeed spend *log N* steps even for the case when the given line does not intersect the clipping convex polygon. The trying to eliminate these unnecessary steps resulted in the modified *O(log N)* algorithm, see [Bui99a]. Now, we are going to describe the modified *O(log N)* algorithm in more details. Again, let us suppose that we have a clipping convex polygon counter-clockwise oriented and line *p* is determined by two points:

$$\mathbf{x}_A = [x_A, y_A]^T , \qquad \mathbf{x}_B = [x_B, y_B]^T$$

The convex polygon is represented by *N* points

$$\mathbf{x}_i = [x_i, y_i]^T \quad , i = 0, \dots ,N-1$$

where: points $\mathbf{x}_0$ and $\mathbf{x}_N$ are identical (column notation is used), x_i and y_i are co-ordinates of the vertex $\mathbf{x}_i$.

The notation $\overline{\mathbf{x}_i\mathbf{x}_k}$ is used for a polyline from $\mathbf{x}_i$ to $\mathbf{x}_k$, i.e. it is a chain of line segments from $\mathbf{x}_i$ to $\mathbf{x}_k$.

Let us define the separation function $F(\mathbf{x})$ in the form

$$F(\mathbf{x}) = Ax + By + C$$

where $F(\mathbf{x}) = 0$ is an equation for the given line *p* and assume that the line has the orientation shown in Figure 4.4, $\mathbf{x}$ is defined as $\mathbf{x} = [x, y]^T$.

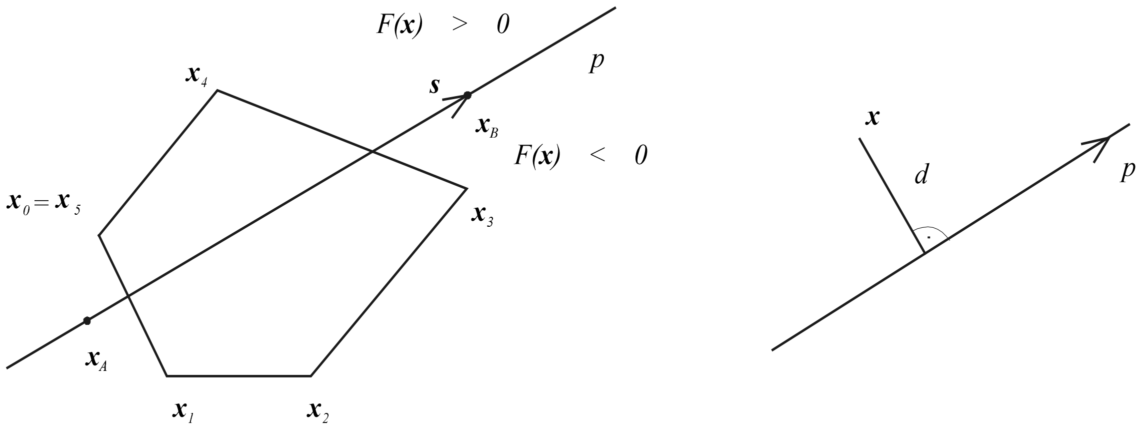


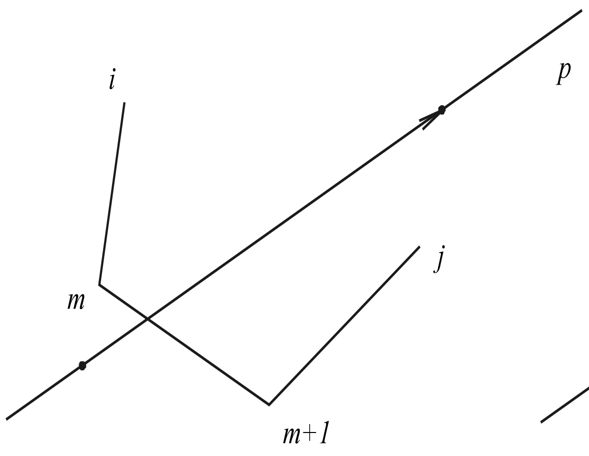
Figure 4.4: Separation function $F(\mathbf{x})$ of line *p* and oriented distance *d* of point $\mathbf{x}$ from *p*.

It can be seen in Figure 4.4, the oriented distance *d* of point $\mathbf{x}$ from line *p* can be determined as

$$d = \frac{Ax + By + C}{\sqrt{A^2 + B^2}}$$

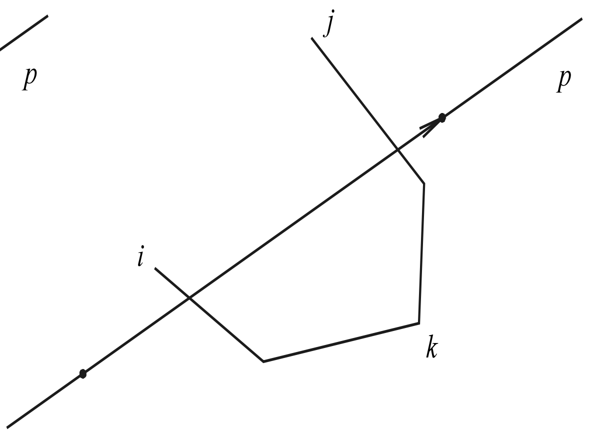
It means that the value of the function *F(**x**)* is actually proportional to the distance *d* of point ***x*** from the given line *p*. First of all, let us consider the chain $\overline{\mathbf{x}_i\mathbf{x}_j}$, where $0 \le i < j < N$. There are two following possible cases:

- In the first case, the points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are on the opposite sides of line *p*, i.e. $F(\boldsymbol{x}_i) * F(\boldsymbol{x}_j) < 0$, there must be just one intersection point with line *p* on the chains $\overline{\mathbf{x}_i\mathbf{x}_j}$ (because the given polygon is convex), i.e. there must exist an index *m* so that $F(\boldsymbol{x}_m) * F(\boldsymbol{x}_{m+1}) < 0 \quad i \le m < j$, see Figure 4.5. It is obvious that in this case the intersection point can be found in *O(lgM)* steps using binary search over vertices, where *M* is number of line segments in the chain $\overline{\mathbf{x}_i\mathbf{x}_j}$.
- Unfortunately, in the second case when points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are on the same side of line *p*, the situation is more complex to solve. Let us concentrate on point $\boldsymbol{x}_k$, where $k = (\, i + j\,)$ ***div*** 2. The condition $F(\boldsymbol{x}_i) * F(\boldsymbol{x}_k) < 0$ shows that point $\boldsymbol{x}_k$ is on one side of line *p*, whereas $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are on the opposite side. This also derives that there must be just one intersection point on the chains $\overline{\mathbf{x}_i\mathbf{x}_k}$ and $\overline{\mathbf{x}_k\mathbf{x}_j}$ for each chain, because the given polygon is convex. The intersection point on each chain can be again found in *O(lgM)* steps using binary search over vertices, where *M* is a number of line segments in the given chain, see Figure 4.6. The worse case will happen when all of three points $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ and $\boldsymbol{x}_k$ lie on the same side of line *p*. It is possible to distinguish all three fundamental sub-cases supposing the previously shown orientation of the separation function *F(**x**)*.



$\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are on the opposite sides of *p*

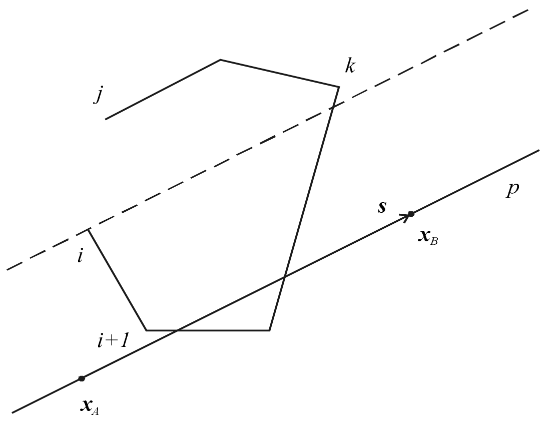
Figure 4.5



$\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are on the same side of *p*

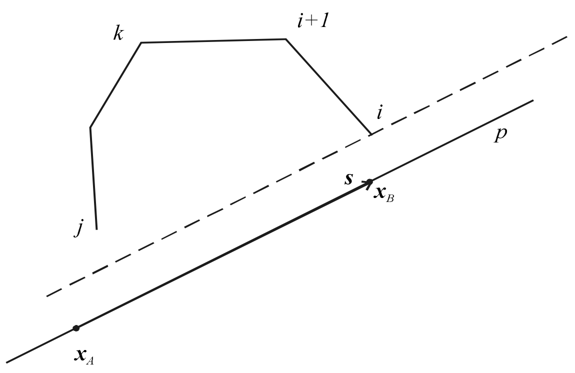
Figure 4.6

a) Point $\mathbf{x}_i$ is the closest point to line p, i.e. $F(\mathbf{x}_i) = \mathbf{min}\ \{F(\mathbf{x}_i), F(\mathbf{x}_j), F(\mathbf{x}_k)\}$. In this case, if $F(\mathbf{x}_{i+1}) < F(\mathbf{x}_i)$ then the chain $\overline{\mathbf{x}_i\mathbf{x}_k}$ can intersect line p, see Figure 4.7. This condition actually expresses that we are getting closer to line p, i.e. the oriented distance d is smaller, therefore, the chain $\overline{\mathbf{x}_k\mathbf{x}_j}$ can be removed by the assignment $j=k$. When this condition is not true, the whole chain $\overline{\mathbf{x}_i\mathbf{x}_j}$ is on one side of line p, i.e. there is no intersection point on this chain and the chain is rejected, see Figure 4.8.



$F(\mathbf{x}_{i+1}) < F(\mathbf{x}_i) \Rightarrow \overline{\mathbf{x}_k\mathbf{x}_j}$ is removed

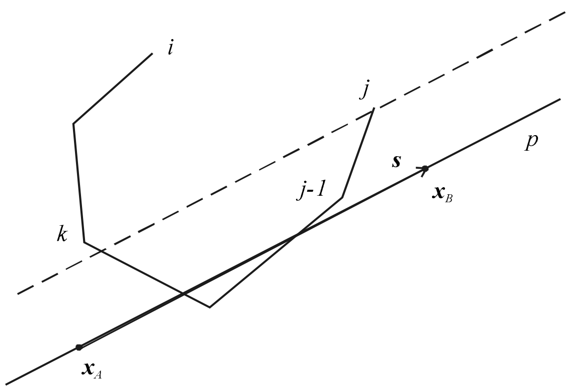
Figure 4.7



$F(\mathbf{x}_{i+1}) > F(\mathbf{x}_i) \Rightarrow \overline{\mathbf{x}_i\mathbf{x}_j}$ is rejected

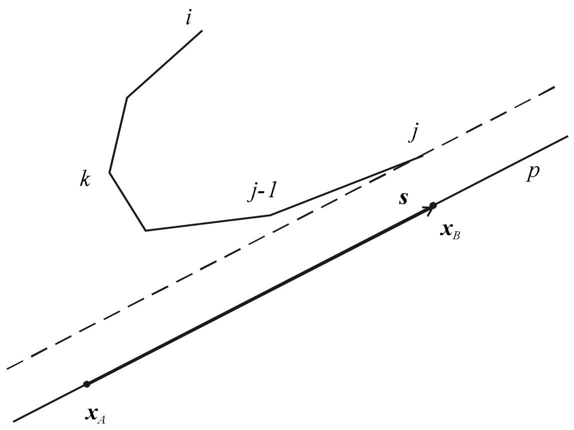
Figure 4.8

b) Similarly for the case when point $\mathbf{x}_j$ is the closest point to line p, i.e. $F(\mathbf{x}_j) = \mathbf{min}\ \{F(\mathbf{x}_i), F(\mathbf{x}_j), F(\mathbf{x}_k)\}$, see Figures 4.9-4.10, the intersection points can lie only the chain $\overline{\mathbf{x}_k\mathbf{x}_j}$ or do not exist at all. The condition $F(\mathbf{x}_{j-1}) < F(\mathbf{x}_j)$ decides that the chain $\overline{\mathbf{x}_i\mathbf{x}_k}$ can be removed and the index i must be changed to k,see Figure 4.9.



$F(\mathbf{x}_{j-1}) < F(\mathbf{x}_j) \Rightarrow \overline{\mathbf{x}_i\mathbf{x}_k}$ is removed

Figure 4.9



$F(\mathbf{x}_{j-1}) > F(\mathbf{x}_j) \Rightarrow \overline{\mathbf{x}_i\mathbf{x}_j}$ is rejected

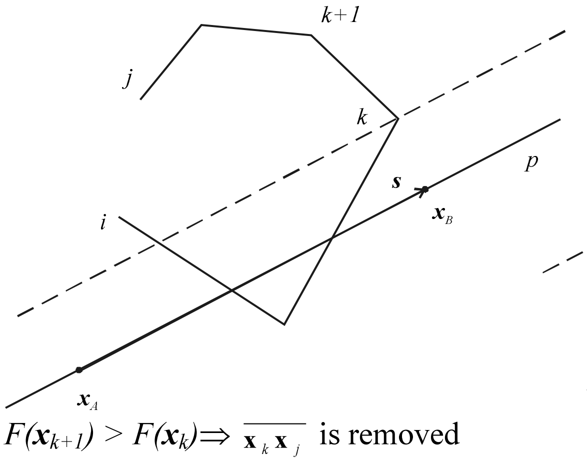
Figure 4.10

c) A little bit more complex situation is shown by Figure 4.11-4.12, where point $\boldsymbol{x}_k$ is the closest point to line p, i.e. $F(\mathbf{x}_k) = \mathbf{min}\ \{F(\mathbf{x}_i), F(\mathbf{x}_j), F(\mathbf{x}_k)\}$. In Figure 4.11 the chain $\overline{\mathbf{x}_k\mathbf{x}_j}$ can be removed, whereas in Figure 4.12 the chain $\overline{\mathbf{x}_i\mathbf{x}_k}$ can be removed. Therefore, index j or index i, respectively, must be changed to k. Theses cases can be distinguished by using criterion $F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$. Actually we must distinguish whether we are getting closer to the given line p or not.

It is very easy to derive the similar conditions for those cases when line p has an opposite orientation. Therefore, the modified *O(log N)* algorithm contains the following basic steps:

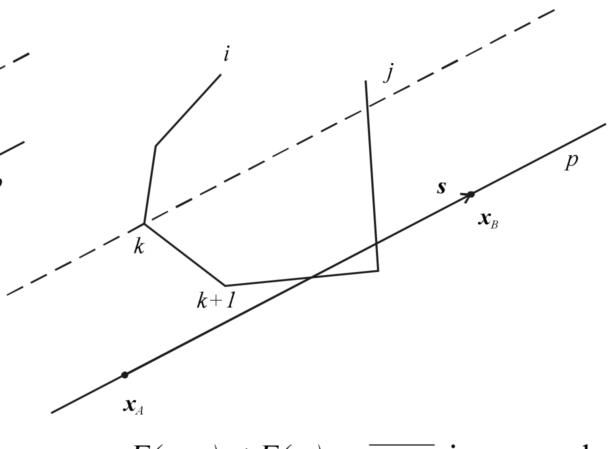
- The algorithm starts with $i = 0; j = n - 1$
- If points $\boldsymbol{x}_0$ and $\boldsymbol{x}_{n-1}$ are on the opposite sides of line p, i.e. $F(\boldsymbol{x}_0) * F(\boldsymbol{x}_{n-1}) < 0$ then one intersection point is on the edge $\overline{\mathbf{x}_{n-1}\mathbf{x}_0}$ and the second one is on the chains $\overline{\mathbf{x}_0\mathbf{x}_{n-1}}$ and can be found in *O(log N)* steps.
- If points $\boldsymbol{x}_0$ and $\boldsymbol{x}_{n-1}$ are on the same side of line p, the algorithm continues with the process to subsequently shorten the chain $\overline{\mathbf{x}_i\mathbf{x}_j}$. This process is repeated until the whole chain is rejected or $F(\mathbf{x}_i) * F(\mathbf{x}_k) < 0$. If this condition becomes true we will obtain two chains $\overline{\mathbf{x}_i\mathbf{x}_k}$ and $\overline{\mathbf{x}_k\mathbf{x}_j}$, which intersect line p and binary search over vertices can be used again as situation is similar situation in Figure 4.5.

Now it can be seen that all parts of the described algorithm are of complexity *O(logM)*, where M is a number of edges in the given chain because we have used the binary search over vertices of the clipping convex polygon for all steps. Therefore the algorithm has *O(logN)* complexity and it is described by Algorithm 4.4.



$F(\mathbf{x}_{k+1}) > F(\mathbf{x}_k) \Rightarrow \overline{\mathbf{x}_k\mathbf{x}_j}$ is removed

Figure 4.11



$F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k) \Rightarrow \overline{\mathbf{x}_i\mathbf{x}_k}$ is removed

Figure 4.12

```
procedure MOD_CLIP_2D_log ($\mathbf{x}_A$, $\mathbf{x}_B$);
{$N+1$ points $\mathbf{x}_i = [x_i, y_i]^T$ ($i = 0, \dots, N$, $\mathbf{x}_0 \equiv \mathbf{x}_N$) represent the convex polygon }
{line $p$ or line segment is determined by two points $\mathbf{x}_A = [x_A, y_A]^T$, $\mathbf{x}_B = [x_B, y_B]^T$}

        function macro $F(\mathbf{x})$: real;             { implemented as an in-line function }
        begin $F := A * x + B * y + C$; end;

        function INTERSECTION($p$, $\mathbf{x}_i$, $\mathbf{x}_j$ ): real;{implemented as an in-line function}
        begin INTERSECTION := $((x_j - x_i) * (y_i - y_A) - (y_j - y_i) * (x_i - x_A))$ /
                                   $((x_j - x_i) * (y_B - y_A) - (y_j - y_i) * (x_B - x_A))$;
        end {INTERSECTION};

        function SOLVE ( $i$, $j$, $i\_GT\_0$ ): real;
        {finds two nearest vertices on the opposite sides of the given line $p$}
        {$i\_GT\_0$ is a boolean parameter indicating whether $F(\mathbf{x}_i) > 0$}
        begin if $i\_GT\_0$ then    while $(j - i) \geq 2$ do {$j \geq i$ always}
                                    begin $k := (i + j)$ div $2$; {shift to the right}
                                          if $F(\mathbf{x}_k) < 0$ then $j := k$ else $i := k$
                                    end {while}
                      else          while $(j - i) \geq 2$ do {$j \geq i$ always}
                                    begin $k := (i + j)$ div $2$; {shift to the right}
                                          if $F(\mathbf{x}_k) < 0$ then $i := k$ else $j := k$
                                    end {while};
      {compute the value $t$ of an intersection point of line $p$ with the polygon edge $\mathbf{x}_i\,\mathbf{x}_j$}
                SOLVE := INTERSECTION ($p$, $\mathbf{x}_i$, $\mathbf{x}_j$);
        end { SOLVE };

begin {determine the $A, B, C$ values for the separation function $F(\mathbf{x})$}
        $A := y_A - y_B$; $B := x_B - x_A$; $C := x_A * y_B - x_B * y_A$;
        $i := 0$;         $j := N-1$;
        $Fc := F(\mathbf{x}_i)$ ; {proportional distance of the closer point}
        {for the polygon orientation shown in Figure 4.4}
        if $Fc > 0$ then
        begin {for the orientation of line $p$ shown in Figure 4.5}
              if $F(\mathbf{x}_{N-1}) < 0$ then
              begin {see Figure 4.5}
                    $t_1$ := SOLVE ( $0$, $N-1$, TRUE ) ; {find an intersection on $\overline{\mathbf{x}_0\mathbf{x}_{n-1}}$ }
                    $t_2$ := INTERSECTION ($p$, $\mathbf{x}_{N-1}$, $\mathbf{x}_0$); {intersection on $\overline{\mathbf{x}_{n-1}\mathbf{x}_0}$ edge}
                    /* {for line segment clipping include three following lines}
                    if $t_1 > t_2$ then begin $t := t_2$; $t_2 := t_1$; $t_1 := t$ end;
                    $t_1$:=max($0$, $t_1$);$t_2$:=min($1$, $t_2$); {compute $<t_1, t_2> \cap <0,1>$}
                    if $<t_1, t_2> = \emptyset$ then EXIT; {exit MOD_CLIP_2D_log}*/
                    DRAW_LINE($\mathbf{x}(t_1)$, $\mathbf{x}(t_2)$)
                    EXIT {exit procedure MOD_CLIP_2D_log}
              end {if};
              if $F(\mathbf{x}_{N-1}) < Fc$ then
              begin $Fc := F(\mathbf{x}_{N-1})$;
                    $i\_closer\_j$ := FALSE          {vertex $\mathbf{x}_j$ is closer than vertex $\mathbf{x}_i$}
              end else $i\_closer\_j$ := TRUE;       {vertex $\mathbf{x}_i$ is closer than vertex $\mathbf{x}_j$}
```

```
                while $(j - i) \geq 2$ do
                begin
                        $k := (i + j)$ div $2$; { shift to the right }
                        if $F(\mathbf{x}_k) < 0$   then
                        begin  { see Figure 4.6}
                                $t_1$ := SOLVE ($i,k,$TRUE); {find an intersection on $\overline{\mathbf{x}_i\mathbf{x}_k}$ }
                                $t_2$ := SOLVE ($k,j,$FALSE);{find an intersection on $\overline{\mathbf{x}_k\mathbf{x}_j}$ }
                                /* {for line segment clipping include 3 following lines}
                                if $t_1 > t_2$ then begin $t := t_2$; $t_2 := t_1$; $t_1 := t$  end;
                                $t_1$:=max($0$, $t_1$);$t_2$:=min($1$, $t_2$); {compute $<t_1,t_2> \cap <0,1>$}
                                if  $<t_1 , t_2> = \emptyset$ then EXIT;{exit MOD_CLIP_2D_log}*/
                                DRAW_LINE($\mathbf{x}(t_1)$, $\mathbf{x}(t_2)$)
                                EXIT {exit procedure MOD_CLIP_2D_log}
                        end {if};
                        if $F(\mathbf{x}_k) > Fc$  then { Figures 4.7-4.10}
                        begin
                                if $i\_closer\_j$ then { Figures 4.7-4.8}
                                        if $F(\mathbf{x}_{i+1}) < F(\mathbf{x}_i)$ then
                                                $j := k$ {remove chain ($k,j$)};        {Fig. 4.7}
                                        else EXIT                                        {Fig. 4.8}
                                else    if $F(\mathbf{x}_{j-1}) < F(\mathbf{x}_j)$ then
                                                $i := k$ { remove chain ($i,k$);        {Fig. 4.9}
                                        else EXIT                                        {Fig. 4.10}
                        end
                        else {Figures 4.11-4.12}
                        begin
                                if $F(\mathbf{x}_{k+1}) > F(\mathbf{x}_k)$ then
                                begin  $j := k$; { remove chain ($k,j$)};               {Fig. 4.11}
                                        $i\_closer\_j$ := FALSE {vertex $\mathbf{x}_j$ is closer than $\mathbf{x}_i$}
                                end
                                else
                                begin  $i := k$; { remove chain ( $i$ , $k$ );}         {Fig. 4.12}
                                        $i\_closer\_j$ := TRUE   {vertex $\mathbf{x}_i$  is closer than $\mathbf{x}_j$ }
                                end;
                                $Fc := F(\mathbf{x}_k)$ ;
                        end
                end { while }
        end
        else {$Fc < 0$}
        begin  { for an opposite orientation of the line situations are solved similarly }
        end
end {MOD_CLIP_2D_log}
```

Algorithm 4.4: Modified *O(log N)* algorithm.

4.3.2. Experimental results

The new modified *O(logN)* algorithm was verified experimentally on Pentium Pro, 200MHz, 128MB RAM, 512KB CACHE. The proposed algorithm has been tested against the Cyrus-Beck (CB) and the *O(logN)* algorithms on data sets (10^5) of line segments with end-points that have been randomly and uniformly generated inside a circle in order to eliminate an influence of rotation. Convex polygons were generated as *N*-sided convex polygons inscribed into a smaller circle.

To compare these algorithms, let us introduce coefficients of the efficiency v as

$$v_{CB} = \frac{T_{CB}}{T} \quad , \quad v_{LogN} = \frac{T_{LogN}}{T}$$

where: T_{CB}, T_{LogN}, T are execution times needed by the CB, *O(logN)* and the modified *O(logN)* algorithms, respectively.

The Table 4.1 and Table 4.2 present the obtained results. In these tables, the second row shows the number of polygon edges and the first column the percentage ***q*** of intersecting lines.

It can be seen that, see Table 4.1, the modified *O(logN)* algorithm is significantly faster then CB algorithm, specially for the high *N*. This is expectable because the algorithm runs with *O(logN)* complexity, whereas the complexity of CB algorithm is *O(N)*.

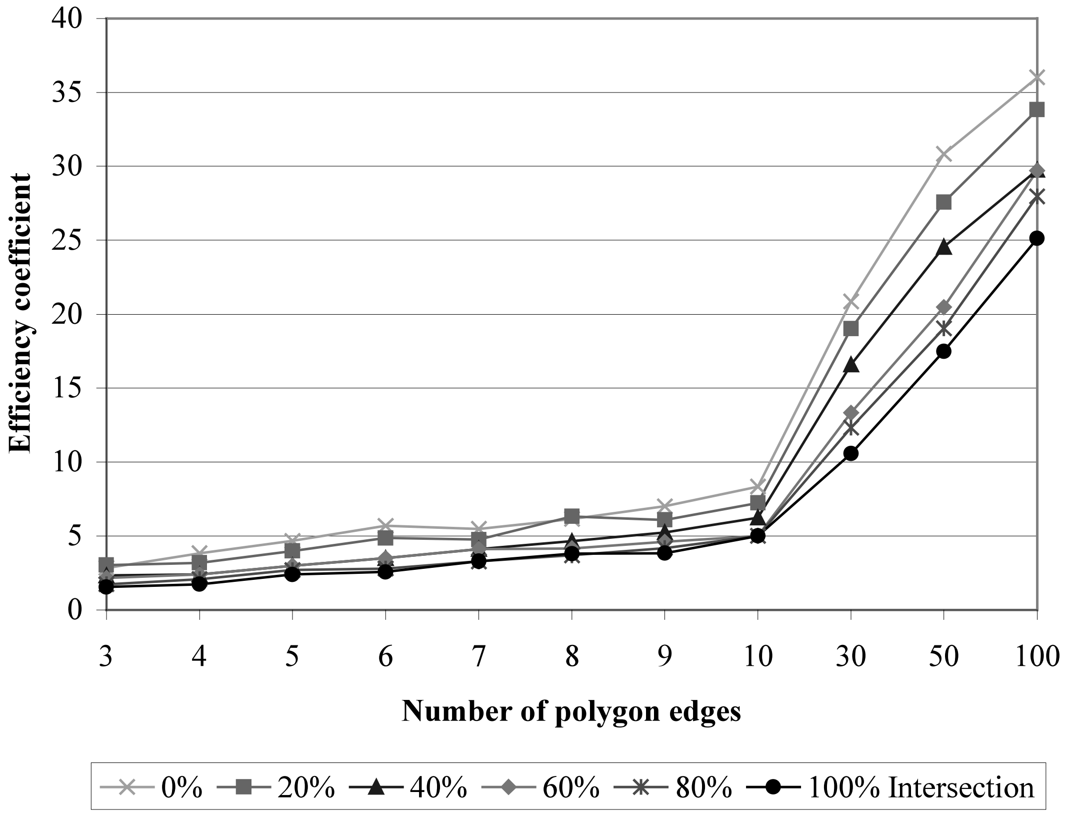
Table 4.2 shows that the modified *O(logN)* algorithm relatively improves the *O(logN)* algorithm significantly, especially for the cases when the given line does not intersect the clipping polygon. CB

v_{CB}	**N**										
q	3	4	5	6	7	8	9	10	30	50	100
0%	2.85	3.85	4.67	5.68	5.48	6.15	7.00	8.33	20.85	30.84	36.01
10%	2.93	3.18	3.97	4.82	5.48	5.37	6.97	8.33	18.48	27.67	33.42
20%	3.04	3.18	4.00	4.85	4.76	6.33	6.08	7.24	19.02	27.57	33.83
30%	2.33	3.15	4.00	4.18	4.76	4.61	5.23	7.03	16.26	24.58	31.51
40%	2.33	2.39	3.00	3.50	4.11	4.64	5.25	6.23	16.59	24.56	29.76
50%	2.33	2.36	3.47	3.61	4.14	4.08	4.60	5.61	14.87	24.67	31.45
60%	2.16	2.39	3.00	3.50	4.11	4.16	4.60	5.00	13.33	20.47	29.71
70%	1.97	2.36	2.69	3.08	3.62	4.16	4.82	5.00	12.30	20.47	27.96
80%	1.75	2.08	2.69	2.80	3.29	3.69	4.18	5.00	12.32	19.03	27.96
90%	1.91	1.91	2.40	2.57	3.02	3.80	3.83	4.98	11.28	18.76	26.56
100%	1.54	1.73	2.40	2.57	3.29	3.80	3.85	5.00	10.56	17.48	25.12

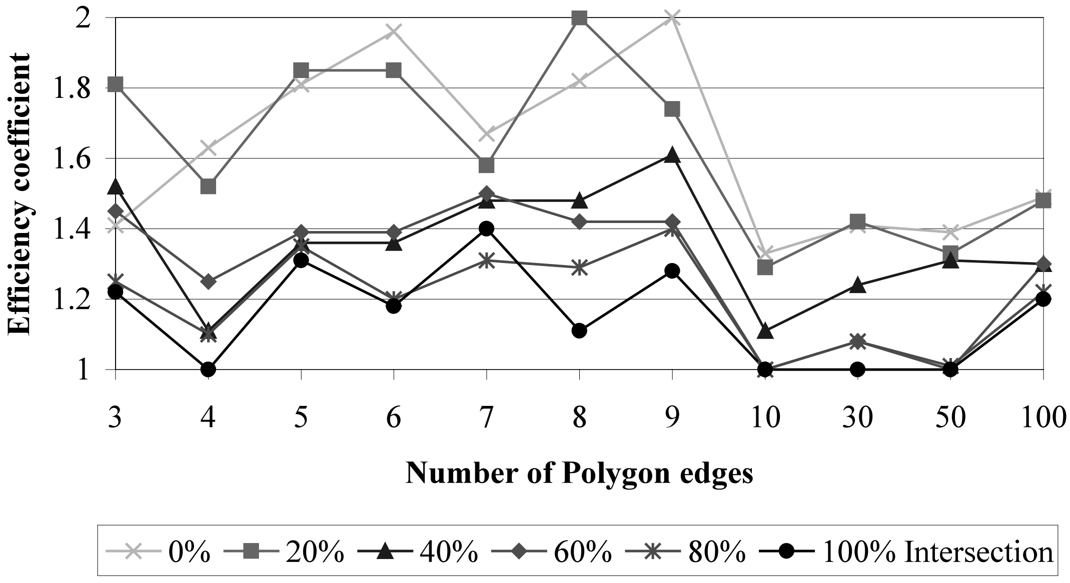
Table 4.1: Comparison between the CB algorithm and the modified *O(logN)* algorithm.

v_{LogN}	N										
q	3	4	5	6	7	8	9	10	30	50	100
0%	**1.41**	**1.63**	**1.81**	**1.96**	**1.67**	**1.82**	**2.00**	**1.33**	**1.41**	**1.39**	**1.49**
10%	1.57	1.48	1.67	1.67	1.85	1.58	2.00	1.30	1.25	1.35	1.40
20%	1.81	1.52	1.85	1.85	1.58	2.00	1.74	1.29	1.42	1.33	1.48
30%	1.48	1.48	1.82	1.58	1.74	1.48	1.61	1.26	1.22	1.31	1.31
40%	1.52	1.11	1.36	1.36	1.48	1.48	1.61	1.11	1.24	1.31	1.30
50%	1.67	1.25	1.74	1.50	1.50	1.42	1.42	1.12	1.11	1.29	1.44
60%	1.45	1.25	1.39	1.39	1.50	1.42	1.42	1.00	1.08	1.00	1.30
70%	1.54	1.36	1.35	1.32	1.42	1.44	1.45	1.00	1.00	1.00	1.27
80%	1.25	1.10	1.35	1.20	1.31	1.29	1.40	1.00	1.08	1.01	1.22
90%	1.40	1.09	1.29	1.20	1.20	1.11	1.28	1.00	1.00	1.07	1.22
100%	1.22	1.00	1.31	1.18	1.40	1.11	1.28	1.00	1.00	1.00	1.20

Table 4.2: Comparison between *O(logN)* and the modified *O(logN)* algorithm.



Graph 4.1: Comparison between CB and the modified *O(logN)* algorithm.



Graph 4.2: Comparison between the *O(logN)* and the modified *O(logN)* algorithm.

Graph 4.1 and Graph 4.2 give us the graphical presentation of obtained results.

As mentioned above, the test whether a line intersects the convex polygon is the dual problem to the test whether a point is inside of the convex polygon. Since an algorithm for testing of whether a point is inside of the convex polygon with *O(1)* expected complexity was developed in [Ska94b], it led to a question whether a line clipping algorithm with *O(1)* expected run-time complexity exists and what would be the complexity of pre-processing. Such algorithm was developed recently, see next paragraph.

4.4. *O(1)* algorithm

The *O(1)* algorithm is based on the dual space representation and on non-orthogonal space subdivision, see [Ska96b]. Therefore, it is necessary to describe these techniques before we will present the algorithm.

4.4.1. The semidual space representation

Any line $r \in E^2$ can be described by an equation

$$ax + by + c = 0$$

and rewritten as

$$y = kx + q \qquad \text{if} \qquad |k| \leq 1 \qquad b \neq 0$$

resp.

$$x = my + p \qquad \text{if} \qquad |m| < 1 \qquad a \neq 0$$

It means, that line $r \in E^2$ can be represented using an asymmetrical model of dual space representation as a point $D(r) = [k,q] \in D(E^2)$ or $D(r) = [m,p] \in D(E^2)$ respectively. This representation model has very interesting properties and usage that can be found in [Sto89a], [Kol94a], [Nie95a], [Zac95a].

The problem is that dual space representation for a convex polygon is **infinite**. Therefore we will split the dual space representation to two complementary dual spaces. Let us consider a modified rhomb box that contain the given polygon, see Figure 4.13.a). It can be seen that q, resp. p values are limited.

Now, the given line $r \in E^2$ can be represented as

$$y = kx + q \qquad \text{if} \qquad |k| \leq 1$$

and

$$x = my + p \qquad \text{if} \qquad |m| < 1$$

If both representations are used then k, resp. m values are limited. Then values $[k, q]$, resp. $[m, p]$ are from the limited area $<-1, 1> x <-h, h>$ in both space representations. We will denote those two limited spaces **semidual spaces**, see Figure 4.13.

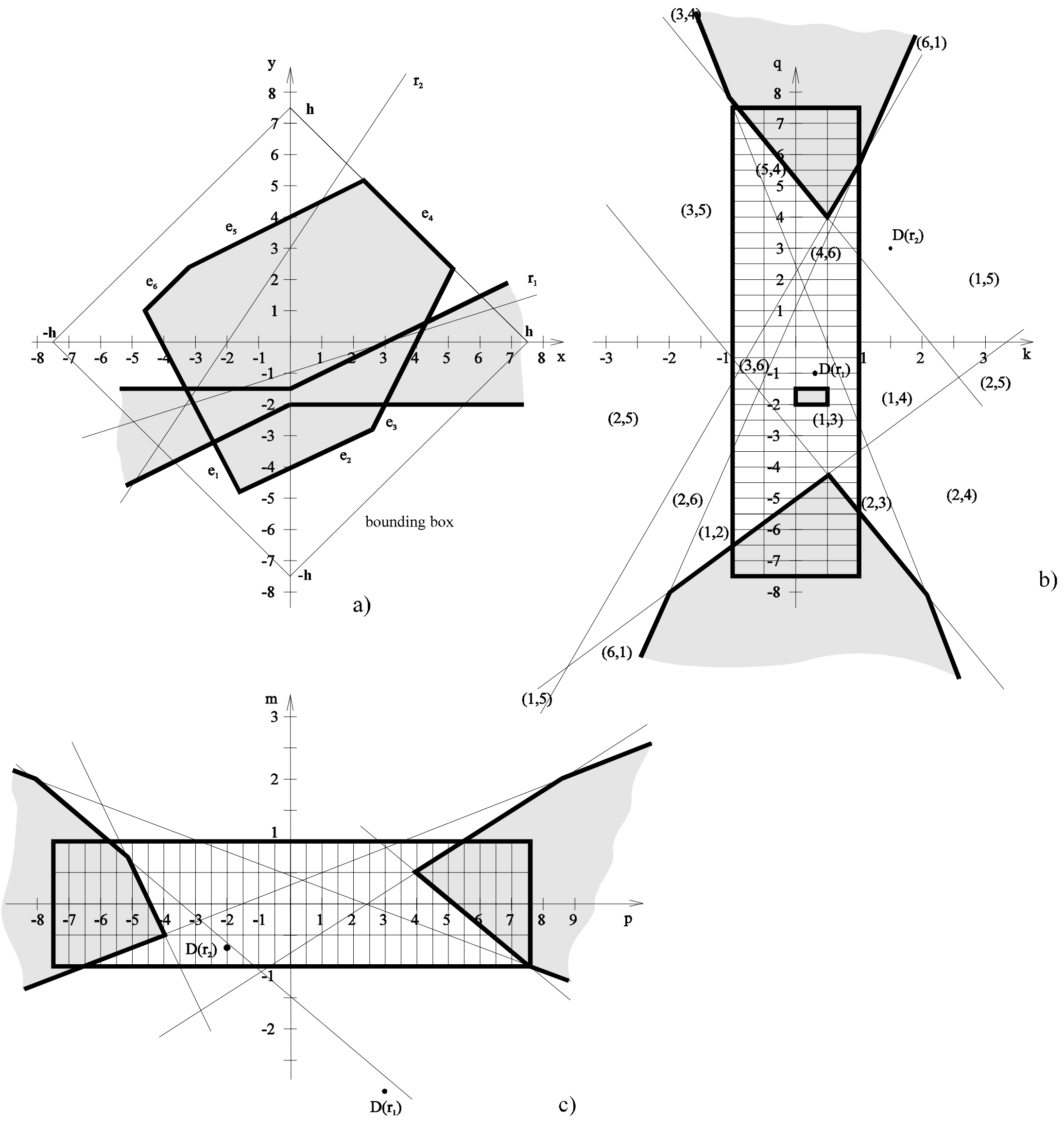


Figure 4.13: The semidual space representation.

4.4.2. Space subdivision

A space subdivision technique is used to detect the region in which a point *D(r)* lies. The semidual spaces for *(k, q)*, *(m, p)* respectively are subdivided into small rectangles. Each rectangle is a dual representation of a region "butterfly" in E^2, see Figure 4.13a).

For each rectangle it is possible to pre-compute a list of polygon edges that interfere in semidual space with the given rectangle. Such list is called the Active Edge List (AEL).

It is necessary to point out that the number of edges in AEL depends on the geometric shape of the given polygon and also on the fineness of the subdivision in *(k, q)*, *(m, p)* spaces respectively. If the rectangles are small enough (i.e., the subdivision is fine) then each list contains no more than two edges of the given polygon.

Experimental results show that subdivision in the direction q, p respectively, is more significant than subdivision in the direction k, respectively m, see [Ska96d] for details.

4.4.3. The *O(1)* line clipping algorithm

The algorithm for line clipping with an *O(1)* processing time complexity consists of the following steps:

- Pre-compute the AELs for the clipping polygon (this pre-computation is made once for all clipped lines),
- determine if the *(k, q)* or *(m, p)* semidual space will be used for the given line,
- compute values $[k, q]$, $[m, p]$ respectively of the given line,
- find a rectangle containing point $D(r)$,
- for all members of the AEL, test and compute the intersections with the given line if they exist.

Because all steps of the algorithm have an *O(1)* complexity, the algorithm on the whole also has an *O(1)* complexity. The detailed algorithm is described by Algorithm 4.5.

The COMPUTE function is based on the CB algorithm and is performed only for edges included in the AEL associated with the selected REGION(*i,j*).

It can be shown that computation of the AELs for all regions (pre-processing) is of $O(N*n_k*n_q)$ and $O(N*n_m*n_p)$ complexity, where:

- N is the number of edges of the given polygon,
- n_k, n_q are the number of subdivisions in the direction k and q, respectively ,
- n_m, n_p are the number of subdivisions in the direction m and p, respectively.

It was theoretically and experimentally proved that the algorithm has *O(1)* expected run-time complexity with pre-processing complexity $O(N^2)$, see [Ska96b], [Ska96d] for details.

```
procedure CLIP_2D_O1 ($\mathbf{x}_A$, $\mathbf{x}_B$);
begin
    $k_0 := n_q / (2*h)$;    $k_1 := n_k / 2$;   $k_3 := n_p / (2*h)$;    $k_4 := n_m / 2$;
    $t_0 := +\infty$;          $t_1 := -\infty$;      {initialisation - interval $< t_0, t_1 > = \emptyset$}
    $\Delta x := x_B - x_A$;      $\Delta y := y_B - y_A$;
    if $|\Delta x| \geq |\Delta y|$ then                    {$(k,q)$ semidual space}
    begin $k := \Delta y/\Delta x$;  $q := y_B - k*x_B$;
        $i :=$ int $((q + h) * k_0) + 1$;   $j :=$ int $((k + 1) * k_1) + 1$;
        test := false;
        {test all members of the AEL for region($i,j$);compute the appropriate value of $t$}
        test := COMPUTE (REGION($i,j$), $t_0$, $t_1$) {if intersections exist  then test = true}
    end
    else
    begin $m := \Delta x/\Delta y$; $p := x_B - m*y_B$;
        $i :=$ int $((p + h) * k_3) + 1$;   $j :=$ int $((m + 1) * k_4) + 1$;
        test := false;
        {test all members of the AEL for region($i,j$);compute the appropriate value of $t$}
        test := COMPUTE (REGION($i,j$), $t_0$, $t_1$) {if intersections exist then test = true}
    end;
    if line segment clipping then $< t_0,t_1> := < t_0,t_1> \cap <0,1>$;
    test := test and ($< t_0, t_1 > \neq \emptyset$);
    if test then  { an intersection exists }
    begin $x_0 := x_A + \Delta x * t_0$;   $y_0 := y_A + \Delta y * t_0$;
          $x_1 := x_A + \Delta x * t_1$;   $y_1 := y_A + \Delta y * t_1$
    end;
end {CLIP_2D_O1}
```

Algorithm 4.5: *O(1)* algorithm.

4.4.4. The *O(1)* line clipping algorithm using polar co-ordinate system

Another way to solve the line clipping problem against a convex polygon with *O(1)* run-time complexity is by using the polar co-ordinate system and the space subdivision in (ρ, φ) plane [Ska99a]. Let (ρ, φ) is polar co-ordinate of system origin's projection on the given line, see Figure 4.14. For simplicity, let us suppose that $\rho \neq 0$. It can be seen that (ρ, φ) unambiguously determines line *p* and so it can be used to represent line *p* in polar co-ordinate system. If we transform the convex polygon to the polar co-ordinate system representation we get an image in (ρ, φ) co-ordinates, see Figure 4.15, where each region represents a set of (ρ, φ) values of the clipped line that intersect the same edges, in our case edges e_1 and e_4.

It can be seen that if the space subdivision technique is used we can directly determine edges that are intersected by the given line *p* regardless to the number of edges of the given polygon for infinite subdivision in (ρ, φ) space. It means that if the subdivision is coarser we will have to test more than two edges, see Figure 4.15. In other words, an algorithm that has *O(1)* run-time complexity can be designed by using this principle, see Algorithm 4.6.

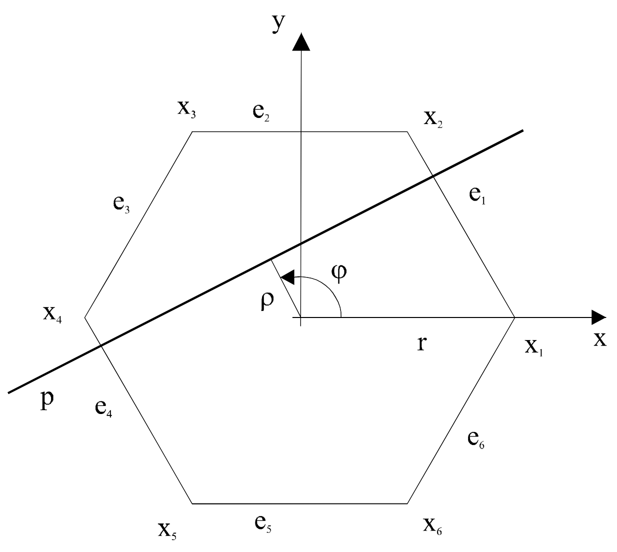


Figure 4.14: The polar co-ordinate representation

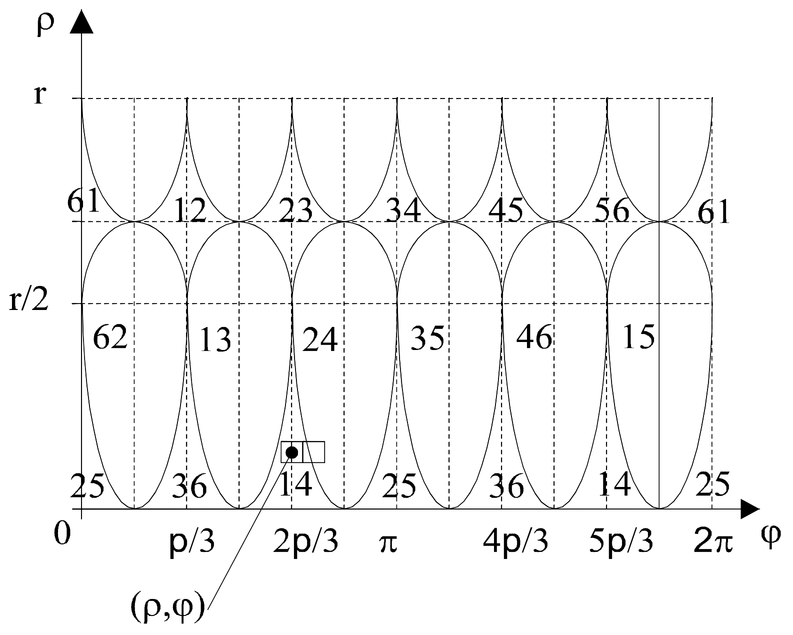


Figure 4.15: The *(ρ, φ)* space subdivision.

The COMPUTE function is again based on the CB algorithm and is performed only for edges included in the AEL associated with the selected REGION(*i,j*).

It can be shown that computation of the AELs for all regions (pre-processing) runs in $O(N^* n_\rho^* n_\varphi)$ time, where:

- N is the number of edges of the given polygon,
- n_ρ is the number of subdivisions in the direction ρ,
- n_φ is the number of subdivisions in the direction φ.

```
procedure POL_CLIP_2D_O1 ($\mathbf{x}_A$, $\mathbf{x}_B$);
begin
    $q_\rho := n_\rho / r;$      $q_\varphi := n_\varphi / (2*Pi);$
    $t_0 := +\infty;$            $t_1 := -\infty;$      {initialisation - interval $< t_0, t_1 > = \emptyset$}
    Compute $(\rho, \varphi)$ co-ordinates of line $p$;
    $i := \rho * q_\rho + 1;$    $j := \varphi * q_\varphi + 1;$
    {test all members of the AEL for region($i,j$);compute the appropriate value of $t$}
    COMPUTE (REGION($i,j$), $t_0$, $t_1$)
    if line segment clipping then $< t_0,t_1> := < t_0,t_1> \cap <0,1>;$
    if $< t_0, t_1 > \neq \emptyset$ then DRAW_LINE($\mathbf{x}(t_0)$, $\mathbf{x}(t_1)$) {an intersection exists}
end {POL_CLIP_2D_O1}
```

Algorithm 4.6: *O(1)* algorithm using polar co-ordinate.

5. Clipping by a non-convex polygon

In technical practice, many times we need clip lines or line segments by the non-convex polygon. Clipping by non-convex polygon is relatively more complex than clipping by convex polygon, because the line can intersect the polygon in more than two points. Therefore, this section is devoted to the clipping problem against non-convex polygon. Algorithm for line clipping against a non-convex polygon is based on the parametric representation of lines or line segments [Ska89a]. It can be seen that, there are some special cases that are necessary to be considered, see lines p_2, p_3 and p_4 in Figure 5.1.

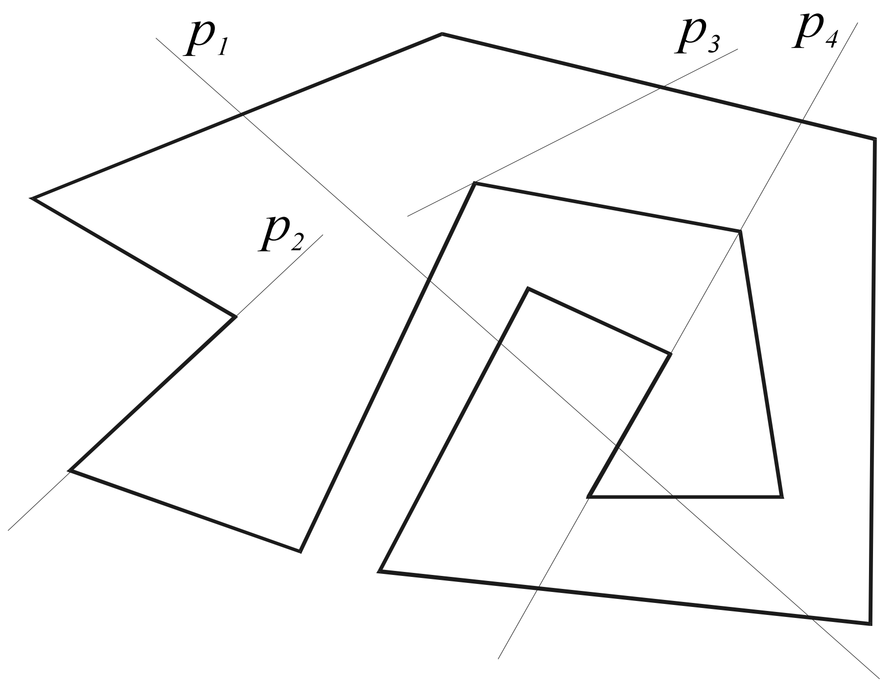


Figure 5.1: Line clipping against non-convex polygon.

Let us suppose that we have a non-convex polygon that is clockwise or counter-clockwise oriented and represented by $N+1$ points

$$\mathbf{x}_i = [x_i, y_i]^T \quad , i = 0, \dots ,N$$

where: points $\mathbf{x}_0$ and $\mathbf{x}_N$ are identical (x_i and y_i are co-ordinates of the vertex $\mathbf{x}_i$). Line p passes two points:

$$\mathbf{x}_A = [x_A, y_A]^T \ , \qquad \mathbf{x}_B = [x_B, y_B]^T$$

and is parametrically represented as

$$\mathbf{x}(t) = \mathbf{x}_A + (\mathbf{x}_B - \mathbf{x}_A) * t$$

where $t \in (-\infty, +\infty)$ (for line clipping) and $t \in <0,1>$ (for line segment clipping).

For simplicity let us assume that:

- The polygon should not have holes.
- The polygon vertices should not coincide.
- Two successive polygon edges should not lie on the same line.

but the algorithm can be modified to circumvent these constrains.

If we represent *i*-th polygon edge parametrically as follows:

$$\mathbf{x}(q) = \mathbf{x}_i + (\mathbf{x}_{i+1} - \mathbf{x}_i) * q \text{ where } q \in <0,1),\ i = 0, 1, \ldots, n-1$$

then the intersection point of line *p* with *i*-th edge can be found by solving the following linear equation system:

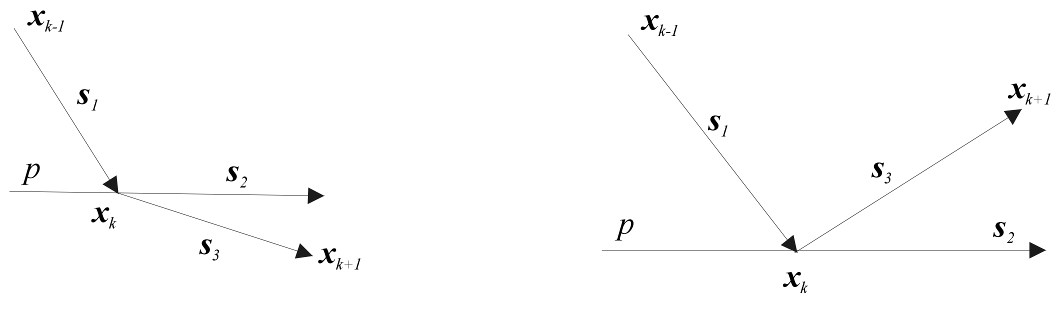
$$\mathbf{x}(t) = \mathbf{x}_A + (\mathbf{x}_B - \mathbf{x}_A) * t$$
$$\mathbf{x}(q) = \mathbf{x}_i + (\mathbf{x}_{i+1} - \mathbf{x}_i) * q$$

Co-ordinates of all intersection points of the given line with the non-convex polygon therefore will be determined by the parameter value *t*. But it is necessary to take into consideration the following special cases:

- Line *p* passes or touches a polygon vertex.
- A polygon edge lies on line *p*.

Let us define $\mathbf{s}_1 = \mathbf{x}_k - \mathbf{x}_{k-1}$, $\mathbf{s}_2 = \mathbf{x}_A - \mathbf{x}_B$ and $\mathbf{s}_3 = \mathbf{x}_{k+1} - \mathbf{x}_k$.

In the first case, there are only two possibilities, see Figure 5.2. In sub-case a) ($[\mathbf{s}_1 \times \mathbf{s}_2]_z . [\mathbf{s}_3 \times \mathbf{s}_2]_z > 0$) only one parameter value *t* is generated, whereas in sub-case b) ($[\mathbf{s}_1 \times \mathbf{s}_2]_z . [\mathbf{s}_3 \times \mathbf{s}_2]_z < 0$) double parameter value *t* is generated. In both sub-cases, the parameter value *t* corresponds to the location of given polygon vertex on line *p*.



a) line *p* passes the vertex $\mathbf{x}_k$ b) line *p* touches the polygon at $\mathbf{x}_k$

Figure 5.2: The vertex $\mathbf{x}_k$ lies on line *p*.

Quite different situation arises when a polygon edge lies on line *p*. In that case, there are four possible sub-cases that are illustrated in the Figure 5.3. It is impossible to directly decide how parameter value *t* should be generated and therefore a special attribute associated to the parametric value must be generated. The additional attribute depends on the sign of the *z* co-ordinate of the cross product result of the vectors $\mathbf{s}_1$ and $\mathbf{s}_2$ or $\mathbf{s}_3$ and $\mathbf{s}_2$, respectively. Therefore, the intersection point will be specified not only by the parameter value *t* but also by the additional attribute. For the case, when the polygon edge does not lie on the given line, it is unnecessary to determine the additional attribute and it will be set to empty (.). Otherwise, the attribute is + or – according to the sign of the *z* co-ordinate of the cross product $[\mathbf{s}_1 \, x \, \mathbf{s}_2]$ or $[\mathbf{s}_3 \, x \, \mathbf{s}_2]$, respectively.

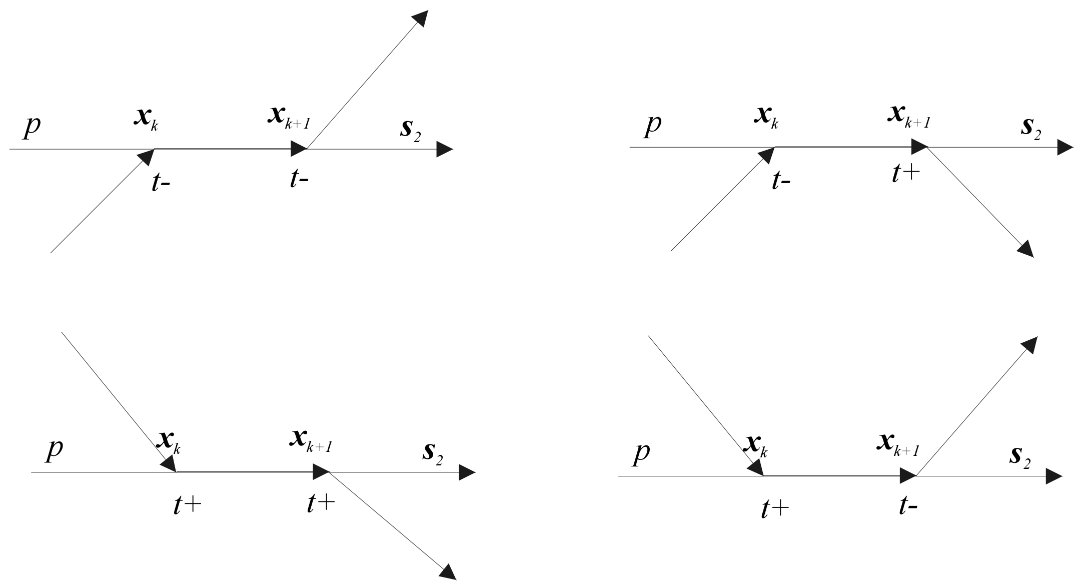
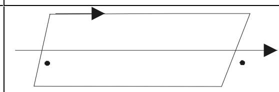
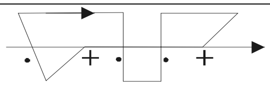
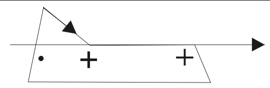
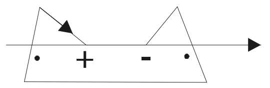
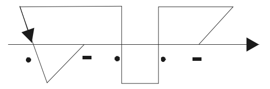
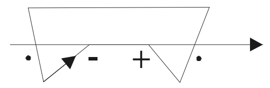
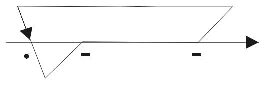
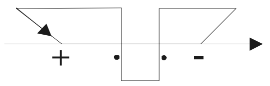
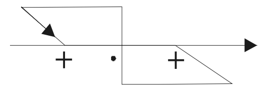
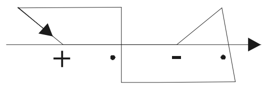
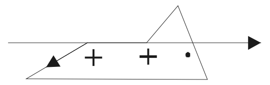
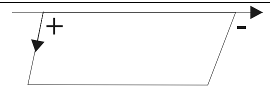
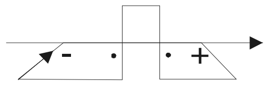
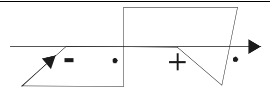
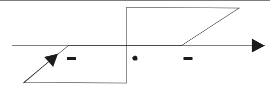
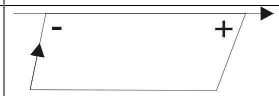
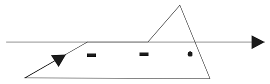


Figure 5.3: The edge $\mathbf{x}_k\mathbf{x}_{k+1}$ lies on line *p*.

After determining all intersection points with their additional attribute, the generated parametric values will be sorted together with their attributes. In next step, the sorted set of generated values will be reduced according to the rule in Table 5.1. The final result will be the set of parametric values that in pairs determine the segments of line *p* which are inside the given non-convex polygon. For line segment clipping it is necessary to make intersection of all pairs of obtained parameter values *t* with the interval $<0,1>$. Above described algorithm can be illustrated in Algorithm 5.1.

Attribute			**Situation**	**Action**
t_i	t_{i+1}	t_{i+2}		
.	.	*		save(t_i, t_{i+1}); $i= i+2$
.	+	.		save(t_i, t_{i+2}); $i= i+2$; change attribute of t_i to +
.	+	+		save(t_i, t_{i+2}); $i= i+3$
.	+	-		save(t_i, t_{i+2}); $i= i+2$; change attribute of t_i to .
.	-	.		save(t_i, t_{i+2}); $i= i+2$; change attribute of t_i to -
.	-	+		save(t_i, t_{i+2}); $i= i+2$; change attribute of t_i to .
.	-	-		save(t_i, t_{i+2}); $i= i+3$
+	.	.		save(t_i, t_{i+2}); $i= i+2$; change attribute of t_i to +
+	.	+		save(t_i, t_{i+2}); $i= i+3$
+	.	-		save(t_i, t_{i+2}); $i= i+2$; change attribute of t_i to .
+	+	*		save(t_i, t_{i+1}); $i= i+1$; change attribute of t_i to .
+	-	*		save(t_i, t_{i+1}); $i= i+2$
-	.	.		save(t_i, t_{i+2}); $i= i+2$; change attribute of t_i to -
-	.	+		save(t_i, t_{i+2}); $i= i+2$; change attribute of t_i to .
-	.	-		save(t_i, t_{i+2}); $i= i+3$
-	+	*		save(t_i, t_{i+1}); $i= i+2$
-	-	*		save(t_i, t_{i+1}); $i= i+1$; change attribute of t_i to .

* marks all cases, i.e. +, -, .

Table 5.1: Reduction rule for generated parameter values.

```
procedure NonConvex_Clip ($\mathbf{x}_A$, $\mathbf{x}_B$);
{N is the edge number of the clipping non-convex polygon}
var    i, k : integer;
begin
        $k := N - 1;$
        $i := 0;$
        $\mathbf{s}_1 := \mathbf{x}_k - \mathbf{x}_{k-1};$          $\mathbf{s}_2 := \mathbf{x}_B - \mathbf{x}_A;$
        while $(i < N)$ do
        begin
                $\mathbf{s}_3 := \mathbf{x}_i - \mathbf{x}_k;$            { $\mathbf{s}_3$ is the vector from $\mathbf{x}_k$ to $\mathbf{x}_i$ }
                Compute (t);
                if $\mathbf{x}_k$ lies on p then
                begin
                        if $[\mathbf{s}_3 \; x \; \mathbf{s}_2]_z = 0$ then          {the edge $\mathbf{x}_k\mathbf{x}_i$ lies on p}
                                Generate (t with sign($[\mathbf{s}_1 \; x \; \mathbf{s}_2]_z$) as attribute)
                        else    if $[\mathbf{s}_1 \; x \; \mathbf{s}_2]_z = 0$ then   {the edge $\mathbf{x}_{k-1}\mathbf{x}_k$ lies on p}
                                        Generate (t with sign($[\mathbf{s}_3 \; x \; \mathbf{s}_2]_z$) as attribute)
                                else    if $[\mathbf{s}_1 \; x \; \mathbf{s}_2]_z . [\mathbf{s}_3 \; x \; \mathbf{s}_2]_z < 0$ then  {$\mathbf{x}_k$ touches p}
                                                Generate (t, t with empty attribute)
                                        else                             {p passes $\mathbf{x}_k$}
                                                Generate (t with empty attribute)
                end
                else    if the intersection between line p and the edge $\mathbf{x}_k\mathbf{x}_i$ exists then
                                Generate (t with empty attribute);
                $\mathbf{s}_1 := \mathbf{s}_3;$
                $k := i;$
                $i := i+1$
        end;
        Sort (generated parameter values t);
        Reduce (sorted parameter values t);
        {determine sequent segments}
        $i := 1$ ;
        while $(i <$ Intersection number$)$ do
                begin
                        DRAW_LINE($\mathbf{x}(t_i)$, $\mathbf{x}(t_{i+1})$) ;{for line clipping}
                        {for line segment clipping
                        if max$(0, t_i) \leq$ min$(1, t_{i+1})$ then
                                DRAW_LINE($\mathbf{x}($max$(0, t_i))$, $\mathbf{x}($min$(1, t_{i+1}))$); }
                        $i := i+2$ ;
                end
end {NonConvex_Clip};
```

Algorithm 5.1: Algorithm for line clipping against non-convex polygon.

The presented algorithm 5.1 enables to clip a given line or line segment against a non-convex polygon. It is obviously that the principle of ECB algorithm, see section 4.2, can be applied to get more efficiency. Moreover, the *O(1)* algorithm can be also modified for clipping by non-convex polygon.

6. Clipping by a non-convex area

So far the presented algorithms have solved the line clipping by convex or non-convex polygon, i.e. by areas that are bounded by linear edges. But plenty of applications require clipping over areas that are formed by linear edges and arcs, see Figure 6.1. Provided a non-convex area is given by its vertices in the clockwise or counter-clockwise order and if the edge is not linear then information whether the right or left part of the circle is to be taken from the actual vertex, see Figure 6.1. It is also assumed that all vertices have different co-ordinates, that no vertex lies on an edge or arc and that two edges or arcs might have only a vertex as a common point.

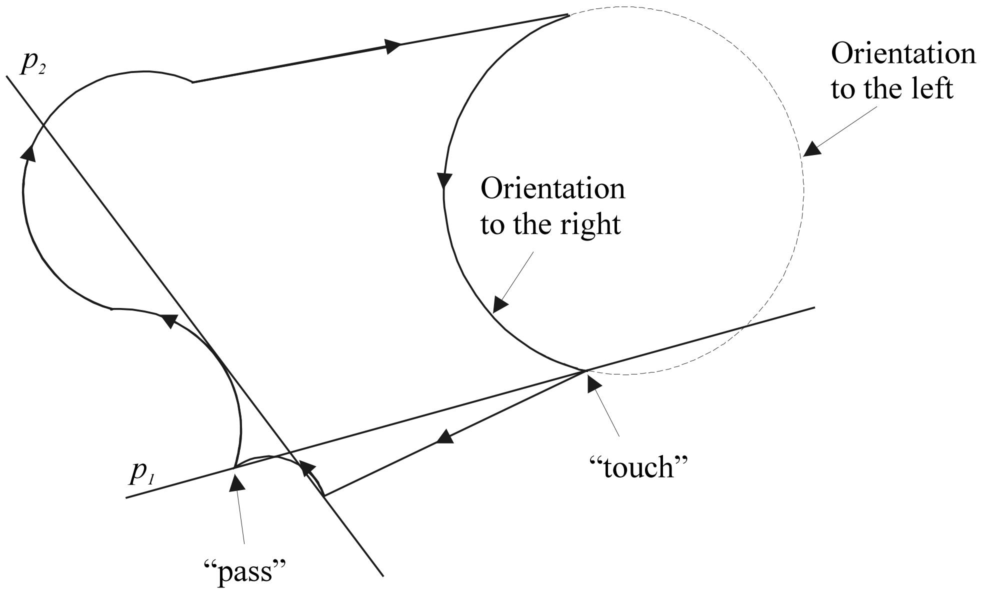


Figure 6.1: Clipping by non-convex area that is formed by linear edges and arcs.

Contrary to the linear edge case, line p can intersect the arc edge in two points. It partially increases the complexity of the given problem.

The given line is described by the parametric equation:

$$\boldsymbol{x}(t) = \boldsymbol{x}_A + (\boldsymbol{x}_B - \boldsymbol{x}_A) * t$$

The procedure for finding all intersection points is similar to the algorithm for line clipping against non-convex polygon, but now in case of arc edge it is necessary to solve the following equation system:

$$\boldsymbol{x}(t) = \boldsymbol{x}_A + (\boldsymbol{x}_B - \boldsymbol{x}_A) * t$$
$$(\boldsymbol{x} - \boldsymbol{x}_C)^T.(\boldsymbol{x} - \boldsymbol{x}_C) = r^2$$

where $\boldsymbol{x}_C = [x_C, y_C]^T$ is the centre of the given arc and r is the radius of this arc.

To solve this equation system with regard to variable t, a quadratic equation $at^2 + bt + c = 0$ will be obtained, where:

$$a = (\mathbf{x}_B - \mathbf{x}_A)^T.(\mathbf{x}_B - \mathbf{x}_A)$$
$$b = 2 * (\mathbf{x}_B - \mathbf{x}_A)^T.(\mathbf{x}_A - \mathbf{x}_C)$$
$$c = (\mathbf{x}_A - \mathbf{x}_C)^T.(\mathbf{x}_A - \mathbf{x}_C) - r^2$$

In the case that line p intersects or touches the given circle two solutions are obtained, not necessarily different, as:

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Now it is necessary to determine which part of the circle forms the boundary of the given area. Because the border is oriented it can be discerned whether the arc is on the right or on the left to the connection of $\mathbf{x}_k$ and $\mathbf{x}_{k+1}$ points. If line p is considered then it must be decided which intersection point ought to be taken. It is obvious that only the point that lies on the proper arc can be considered. It means that:

- if the left arc is considered then point $\mathbf{x}(t)$ will be taken into consideration if and only if $[\mathbf{s}_3 \, x \, \mathbf{s}]_z > 0$
- if the right arc is considered then point $\mathbf{x}(t)$ will be taken into consideration if and only if $[\mathbf{s}_3 \, x \, \mathbf{s}]_z < 0$

assuming that $\mathbf{x}(t) \neq \mathbf{x}_k$, $\mathbf{s}_3 = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{s} = \mathbf{x}(t) - \mathbf{x}_k$.

Of course some special situations must be solved again, e.g. when the given line passes or touches the vertex $\mathbf{x}_k$. In those cases the tangent vectors $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{s}_3$ are determined as:

- $\mathbf{s}_1 = [y_k - y_C, x_C - x_k]^T$ for the arc $\mathbf{x}_{k-1}\mathbf{x}_k$ with centre $\mathbf{x}_C$
  $\mathbf{s}_1 = [x_k - x_{k-1}, y_k - y_{k-1}]^T$ for the linear edge
- $\mathbf{s}_2 = [x_B - x_A, y_B - y_A]^T$
- $\mathbf{s}_3 = [y_k - y_C, x_C - x_k]^T$ for the arc $\mathbf{x}_k\mathbf{x}_{k+1}$ with centre $\mathbf{x}_C$
  $\mathbf{s}_3 = [x_k - x_{k-1}, y_k - y_{k-1}]^T$ for the linear edge

Using tangent vectors, we can decide whether the single or double parameter value should be generated and determine the corresponding attribute.

If the arc is oriented to the right then the sign of tangent vector must be changed in some situations.

The whole algorithm for line clipping by non-convex area is shown in Algorithm 6.1.

```
procedure Area_Clip ($\mathbf{x}_A$, $\mathbf{x}_B$);        {$N$ is the number of area's vertices}
        procedure Compute_Tangent($\mathbf{x}_1$, $\mathbf{x}_2$, $r$, $t$);
        begin  if $\mathbf{x}_1 \mathbf{x}_2$ is linear then  begin  $\mathbf{s} := \mathbf{x}_2 - \mathbf{x}_1$; $r := [\mathbf{s} \times \mathbf{s}_2]_z$ end
                else    {$\mathbf{x}_1 \mathbf{x}_2$ is the arc}
                begin  $\mathbf{s} := [y_k - y_C, x_C - x_k]^T$; $r := [\mathbf{s} \times \mathbf{s}_2]_z$    {$[x_C, y_C]^T$ is arc's centre}
                        if $r = 0$ then    if $t$ then $r := \mathbf{s}^T\mathbf{s}_2$ else $r := -\mathbf{s}^T\mathbf{s}_2$
                                else    if the arc is oriented to the right then $r := -r$
                end
        end

begin  $k := N - 1$;        $i := 0$;            $\mathbf{s}_2 := \mathbf{x}_B - \mathbf{x}_A$;
        while $(i < N)$ do
        begin  if $\mathbf{x}_k$ lies on $p$ then
                begin  Compute_Tangent ($\mathbf{x}_k$, $\mathbf{x}_i$, $b$, True);
                        Compute_Tangent ($\mathbf{x}_{k-1}$, $\mathbf{x}_k$, $a$, False);
                        if $\mathbf{x}_k \mathbf{x}_i$ is linear then
                        begin  Compute_Value ($t$);
                                if $a.b < 0$ then  {$p$ touches $\mathbf{x}_k$}
                                        Generate ($t$, $t$ with empty attribute)
                                else    if $a.b > 0$ then  {$p$ passes $\mathbf{x}_k$}
                                                Generate ($t$ with empty attribute)
                                        else    if $a = 0$ then{the edge $\mathbf{x}_{k-1}\mathbf{x}_k$ lies on $p$}
                                                        Generate ($t$ with attribute sign $b$)
                                                else    {the edge $\mathbf{x}_k\mathbf{x}_i$ lies on $p$}
                                                        Generate ($t$ with attribute sign $a$)
                        end
                        else    {$\mathbf{x}_k \mathbf{x}_i$ is the arc}
                        begin  Compute_Values ($t_1$, $t_2$);
                                if $a.b < 0$ then  {$p$ touches $\mathbf{x}_k$}
                                        Generate ($t_1$, $t_1$, $t_2$* with empty attribute)
                                else    if $a.b > 0$ then  {$p$ passes $\mathbf{x}_k$}
                                                Generate ($t_1$, $t_2$*  with empty attribute)
                                        else    {the edge $\mathbf{x}_{k-1}\mathbf{x}_k$ lies on $p$}
                                                Generate ($t_1$ with attribute sign $b$,
                                                        $t_2$* with empty attribute)
                        end
                end
                else
                begin  if $\mathbf{x}_k \mathbf{x}_i$ is linear then
                        begin  Compute_Value ($t$);
                                if an intersection point in inside of <$\mathbf{x}_k \mathbf{x}_i$) then
                                        Generate ($t$ with empty attribute)
                        end
                        else    {$\mathbf{x}_k \mathbf{x}_i$ is the arc}
                        begin  Compute_Values ($t_1$, $t_2$);
                                Generate ( $t_1$*, $t_2$* with empty attribute)
                        end
                end
                $k := i$; $i := i+1$
        end;
        **Sort** (generated parameter values $t$);
        **Reduce** (sorted parameter values $t$);
        {determine sequent segments}
        $i := 1$ ;
        **while** ($i <$ Intersection number) **do**
            **begin**
                DRAW_LINE($\mathbf{x}(t_i)$, $\mathbf{x}(t_{i+1})$) ;{for line clipping}
                {for line segment clipping
                **if** max($0, t_i$) $\leq$ min($1, t_{i+1}$) **then**
                    DRAW_LINE($\mathbf{x}$(max($0, t_i$)), $\mathbf{x}$(min($1, t_{i+1}$))); }
                $i := i+2$ ;
            **end**
**end** {Area_Clip};
{* means if the intersection point lies on the proper side of the arc $\mathbf{x}_k \mathbf{x}_i$ }
```

Algorithm 6.1: Algorithm for line clipping against non-convex area.

It is obvious that the presented algorithm for line clipping by non-convex area can be modified for the case when the area is formed by linear segments and quadratic arcs. In this case it is necessary to define conveniently the quadratic arcs. Generally all quadratic curves are described by the function $f(x,y)$ together with their tangent vectors as:

$$f(x,y) = ax^2 + by^2 + 2cxy + 2dx + 2ey + g = 0$$

and

$$\mathbf{s} = [f_y, -f_x]^T$$

If the given area consists of some holes it is necessary to apply the presented algorithm for all the given holes themselves and merge the obtained parameter values together appropriately, see [Ska89b] for more details.

7. Polygon Clipping

An algorithm that clips a polygon must deal with many different cases. One concave polygon can be clipped into two or more separate polygons. Generally, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clipping polygon, new edges must be added, and existing edges must be discarded, retained, or divided. Multiple polygons may result from clipping a single polygon. We need a systematic way to deal with all these cases. There are many algorithms for the polygon clipping (see references), but the Sutherland-Hodgman (SH) algorithm is the most often used.

Sutherland and Hodgman's polygon-clipping algorithm [Sut74a] uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle, successively clip a polygon against a clip rectangle. It is necessary to point out that SH algorithm is originally suggested for polygon clipping against a rectangular window but its principle can be used to clip polygons against a convex polygon.

Note the difference between this strategy for clipping a polygon and the Cohen-Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the outcode to see which edge is crossed, and clips only when necessary.

The algorithm works by moving around the polygon from $\boldsymbol{x}_n$ to $\boldsymbol{x}_1$ to $\boldsymbol{x}_n$ ($\boldsymbol{x}_0$ and $\boldsymbol{x}_n$ are identical) taking into account the relationship between successive vertices and the clip boundary. For each vertex pair, either zero, one, or two vertices are added to the output list of vertices.

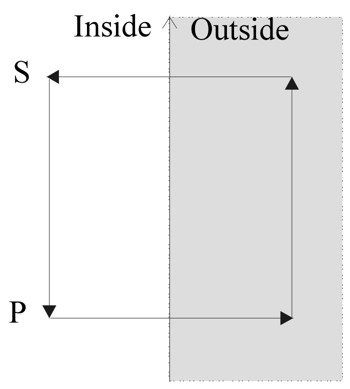
There are four possible test cases need to be examined, see Figure 7.1. For each case, we will assume the polygon edge to be clipped is from vertex *S* to *P*.

- Case a : Wholly inside visible region - save end-point *P*
- Case b : Exit visible region - save the intersection *I*
- Case c : Wholly outside visible region - save nothing
- Case d : Enter visible region - save intersection *I* and end-point *P*

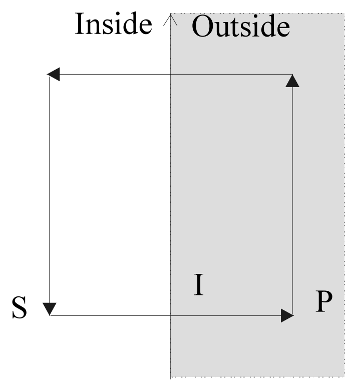
The SH algorithm can be implemented by Algorithm 7.1. In this implementation, it has a couple of assumptions:

- accepts an array in_v of polygon vertices and creates an array out_v
- procedure OUTPUT(out_v,v,out_length) places vertex v into out_v and updates the number of vertices into out_ length
- function INTERSECT(*S*,*P*,clip_boundary) returns the intersection of segment (*SP*) with clip_boundary.

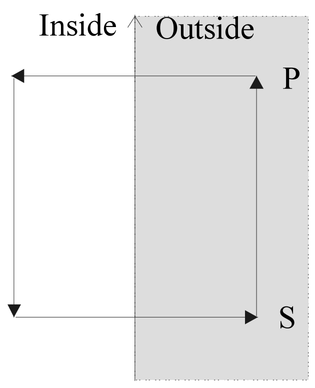
Function INSIDE(point,clip_boundary) returns true if the point is inside the clip boundary where inside is defined to be to the left of the clip boundary (if the polygon is counter-clockwise oriented). More specific, inside is to the left when looking from the first point of the clip boundary to the second. function INSIDE uses the cross product of two vectors formed by using the point and the clip boundary.



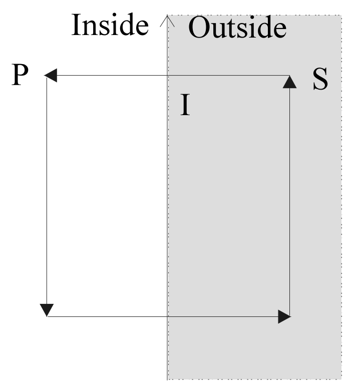
a) *SP* is inside.



b) *SP* exits the visible region.



c) *SP* is outside.



d) *SP* enters the visible region.

Figure 7.1: Four possible cases of relationship between polygon edge and clip boundary.

```
type    vertex=array[1..2] of real;
        boundary=array[1..2] of vertex;
        vertex_array[1..max] of vertex;

procedure clip_poly( in_v  : vertex_array;          {input vertex array}
                     var out_v : vertex_array;      {output vertex array}
                     in_length : integer;           {length of in_v}
                     var out_length : integer;      {length of out_v}
                     clip_boundary : boundary);     {clip boundary}
var     I,P,S : vertex;
        j : integer;
begin   out_length := 0;
        S := in_v[in_length];
        for j := 1 to in_length do
        begin
                P := in_v[j];
                if INSIDE(P,clip_boundary) then
                        if INSIDE(S,clip_boundary) then
                                OUTPUT(out_v,P,outlength)
                        else begin
                                I := INTERSECT(S,P,clip_boundary);
                                OUTPUT(out_v,I,out_length);
                                OUTPUT(out_v,P,out_length)
                             end
                else
                        if INSIDE(S,clip_boundary) then
                        begin
                                I := INTERSECT(S,P,clip_boundary);
                                OUTPUT(out_v,I,out_length)
                        end
                S := P;
        end
end {clip_poly};
```

Algorithm 7.1: Sutherland-Hodgman Algorithm.

Because clipping against one edge is independent of all others, it is possible to arrange the clipping stages in a pipeline. The input polygon is clipped against one edge and any points that are kept are passed on as input to the next stage of the pipeline. By this way, four polygons can be at different stages of the clipping process simultaneously. This is often implemented in hardware.

It can be seen that the SH algorithm runs in *O(M*N)* time, where *M*,*N* are number of vertices of each polygon. For clipping a non convex polygon against a convex polygon, Rappaport proposed an algorithm [Rap91a] that runs in *O(M*LogN)* time. Moreover, an algorithm with *O(M+N)* complexity for clipping a convex polygon against a convex polygon was published by Toussaint, see [Tou85a] for more details.

8. Clipping by a pyramid in E^3

Previously described algorithms enable the line or line segment clipping against a rectangular window, a (convex or non-convex) polygon or an area in E^2. It is very easy to extend Cohen-Sutherland algorithm and Liang-Barsky algorithm for the line clipping against a canonical parallel or perspective view volume (cube or pyramid, respectively) in E^3 [Fol90a]. In this section we will focus on the clipping algorithms against a pyramid, which represents the visible region of an observer in the perspective projection. Many algorithms for clipping lines or line segments in E^3 have been published, see [Cyr78a], [Lia83a], [Lia84a], [Fol90a], [Ska96a], [Ska97a], [Ska97c] for main references. For a long time the CS algorithm and its extensions to E^3 (see [Fol90a]) were the only line segment clipping algorithms found in most textbooks. The LB algorithm proposed for line clipping can be also used for the line segment clipping, but it is slower than the CS algorithm. It is possible to say that algorithms for a line clipping can be modified for a line segment clipping but those modifications are generally slower than algorithms originally developed for the line segments clipping.

Let us assume that we have a line or line segment with end-points $A(x_A, y_A, z_A)$ and $B(x_B, y_B, z_B)$ and a unitary clipping pyramid. The unitary clipping pyramid is defined at the set of all points (x, y, z) such that $-z \leq x \leq z$ and $-z \leq y \leq z$ ($z \geq 0$). The intersection of the pyramid and the given line or line segment is a continuous portion of the line or line segment (if not empty), and so it can be represented by two end-points. Therefore, we must determine whether the intersection is empty and if not compute the co-ordinates of its end-points. Before describing particular algorithms, it is necessary to mention some common definitions:

8.1. Definitions

The planes $x = -z$, $x = z$, $y = -z$ and $y = z$ are called the right, left, bottom and top boundaries of the unitary pyramid, respectively. We will say that:

- a point or a line segment is visible, if it lies entirely inside the given pyramid,
- a point, a line or line segment is invisible, if it lies entirely outside the given pyramid,
- a line or line segment is partially visible, if it lies partly inside the given pyramid and partly outside.

If a line or line segment is invisible, then no part of the line or line segment appears in the output, the line or line segment is said to be rejected by the clipping algorithm. The boundaries of the pyramid divide the Cartesian positive half-space ($z \geq 0$) into 9 regions. Regions that are bounded by only two boundaries are called the corner regions and regions which are bounded by three boundaries are called the edge regions, see Figure 8.1.

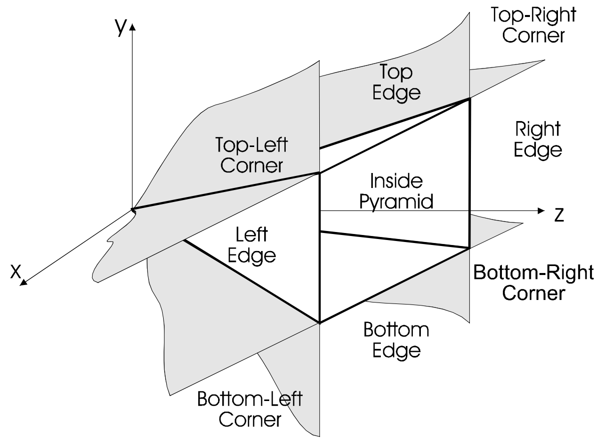


Figure 8.1: Subdivision of the positive haft-space into regions.

8.2. CS-3D algorithm for line segment clipping against a pyramid

The CS-3D algorithm is very well known algorithm. It is the extension of the CS algorithm for line segment clipping against a rectangular window in E^2. It is simple and robust. It enables to detect all the cases when the line segment is completely inside of the given pyramid and some cases when the line segment is outside of the given pyramid, see Figure 8.2. The CS-3D algorithm uses a coding system to distinguish several cases. This approach divides the problem into a manageable number of cases and leads to a shorter program. However the coding system can be the cause of inefficiency. Some comparisons must be performed to encode the location of the line segment. Further comparisons on the encoding are then needed before the appropriate calculation of an intersection is done. Moreover, in some cases, all intersection points of the line segment with each pyramid's boundary plane are computed, but only two intersection points are needed, see the line segment EF in Figure 8.2. The CS-3D algorithm can be implemented by Algorithm 8.1.

```
procedure CS_Clip_3D ( $x_A$, $y_A$, $z_A$, $x_B$, $y_B$, $z_B$: real);
var     x, y, z, t: real;
        c, $c_A$, $c_B$: integer; {operators land, resp. lor are bitwise and, resp. or operators }
procedure CODE (x, y, z: real; var c: integer); {implemented as a macro}
begin   c := 0;
        if x < -z then c := 1 else if x > z then c := 2;
        if y < -z then c := c + 4 else if y > z then c := c + 8
end { of CODE };
begin   CODE ($x_A$, $y_A$, $z_A$, $c_A$); CODE ($x_B$, $y_B$, $z_B$, $c_B$);
        if ($c_A$ land $c_B$) $\neq$ 0 then EXIT;        {the line segment is outside the pyramid}
        if ($c_A$ lor $c_B$) = 0 then                   {the line segment is inside of the pyramid}
                begin DRAW_LINE ($x_A$, $y_A$, $z_A$, $x_B$, $y_B$, $z_B$); EXIT end;
        repeat if  $c_A \neq 0$ then c = $c_A$ else c = $c_B$;
                if (c land '0001') $\neq$ 0 then
                begin  t := ($z_A + x_A$) / (($x_A - x_B$) - ($z_B$ - $z_A$));
                       z := $z_A$ + t*($z_B$ - $z_A$);   x := -z;  y := $y_A$ + t*($y_B$ - $y_A$)
                end
                else   if (c land '0010') $\neq$ 0 then
                       begin  t := ($z_A$ - $x_A$) / (($x_B - x_A$) - ($z_B$ - $z_A$));
                              z := $z_A$ + t*($z_B$ - $z_A$);   x := z;  y := $y_A$ + t*($y_B$ - $y_A$)
                       end
                       else   if (c land '0100') $\neq$ 0 then
                              begin  t := ($z_A + y_A$) / (($y_A - y_B$) - ($z_B$ - $z_A$));
                                     z := $z_A$ + t*($z_B$ - $z_A$);x := $x_A$ + t*($x_B$ - $x_A$);y := -z
                              end
                              else   if (c land '1000') $\neq$ 0 then
                                     begin  t := ($z_A$ - $y_A$) / (($y_B - y_A$) - ($z_B$ - $z_A$));
                                            z:= $z_A$+t*($z_B$ - $z_A$);x:= $x_A$ + t*($x_B$ - $x_A$);y:= z
                                     end;
                if  c = $c_A$  then begin $x_A$ := x; $y_A$ := y; $z_A$ := z; CODE ($x_A$, $y_A$, $z_A$, $c_A$) end
                              else begin $x_B$ := x; $y_B$ := y; $z_B$ := z; CODE ($x_B$, $y_B$, $z_B$, $c_B$) end;
                if ($c_A$ land $c_B$) $\neq$ 0 then EXIT
        until ($c_A$ lor $c_B$) = 0;
        DRAW_LINE ($x_A$, $y_A$, $z_A$, $x_B$, $y_B$, $z_B$)
end {CS_Clip_3D};
```

Algorithm 8.1: Cohen-Sutherland-3D algorithm.

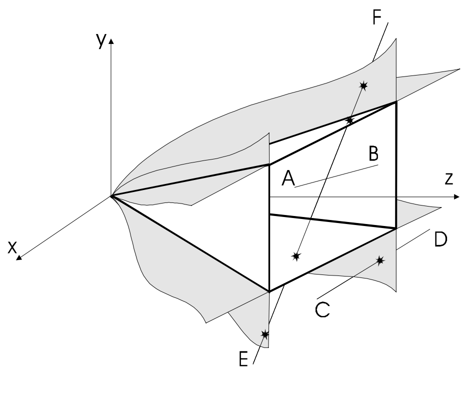


Figure 8.2: Line segment clipping against a pyramid.

8.3. Pyramidal clipping (PC) algorithm

In order to efficiently to clip line segments against a given pyramid in E^3, a new algorithm called Pyramidal clipping algorithm was developed, verified and tested. The Pyramidal clipping algorithm (PC) uses a similar approach as Nicholl-Lee-Nicholl algorithm [Nic87a] that was derived for E^2 case only. This algorithm is robust, reliable and convenient for all applications if line segment clipping in E^3 is to be used.

8.3.1. Brief description of the algorithm

Let us assume that two points $A(x_A, y_A, z_A)$ and $B(x_B, y_B, z_B)$ are given and we wish to compute the intersection of the line segment *AB* with the unitary clipping pyramid. For simplicity we assume that both points *A* and *B* are in the positive half-space. The end-point *A* therefore can lie inside of the pyramid, in an edge region or in a corner region. For each of these cases, we can divide the positive half-space into certain number of sub-regions, see Figures 8.3-8.5. These sub-regions are bounded by the pyramid's boundaries and planes determined by point *A* and one edge of pyramid. These planes will be denoted ρ_1, ρ_2, ρ_3, ρ_4, clockwise from the top-left edge, see Figures 8.3-8.5. All other cases can be obtained from one of these cases in Figures 8.3-8.5 by rotating the scene around the z axis.

With the above definitions the PC algorithm can be described by the following basic steps:

- characterize the location of point *A* among the 9 regions,
- characterize the location of point *B* among the appropriate sub-regions,
- compute the intersection points according to above characterization.

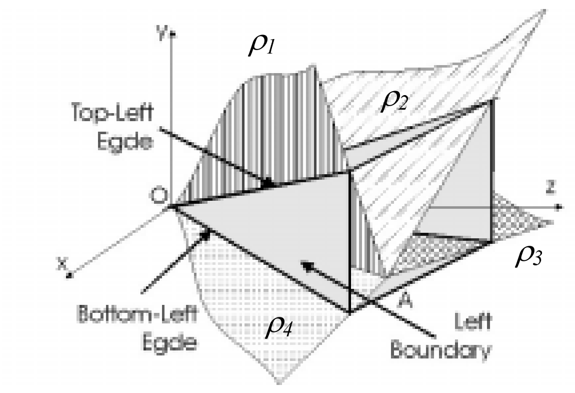


Figure 8.3: Sub-regions for the case when point *A* lies inside the pyramid.

The main advantage of this approach is that we can determine which pyramid boundaries are intersected and therefore avoid unnecessary computation of invalid intersection points.

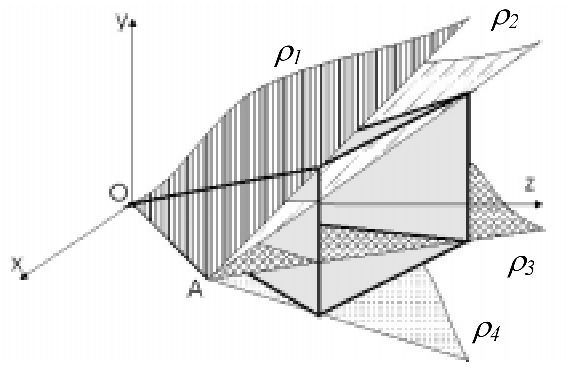


Figure 8.4: Sub-regions for the case when point *A* is in left edge region.

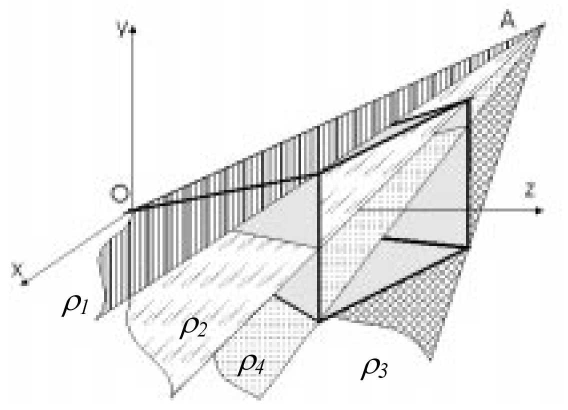


Figure 8.5: Sub-regions for the case when point *A* is in top-right corner region.

8.3.2. Details of the PC algorithm

The proposed PC algorithm will be explained in more detail by the top-down approach. At the beginning we must determine whether the end-point *A* of the given line segment is beyond the right boundary, beyond the left boundary or between those two boundaries. The main procedure, in Pascal-like code is:

```
procedure PC_Clip( $x_A$, $y_A$, $z_A$, $x_B$, $y_B$, $z_B$: real);
var     $\Delta x$, $\Delta y$, $\Delta z$ ,  k, m, r : real
begin
        if $x_A < -z_A$ then
                case_I                          {point A is beyond right boundary}
         else    if $x_A <= z_A$ then
                        case_II                 {point A is between 2 boundaries}
                else
                        case_III                {point A is beyond left boundary}
end {PC_Clip};
```

We will use the **EXIT** instruction in the following parts of algorithm to denote the end of the procedure and to avoid "**else if**" sequences unnecessary for algorithm explanation.

I. Point A is beyond the right boundary.

If point *B* is also beyond the right boundary, it is not necessary to further characterize point *A*, because the line segment is invisible. Therefore, we must check whether point *B* is beyond the right boundary before proceeding on. After that, we must test whether point *A* lies either in the corner region or in the edge region. This section of algorithm can be implemented as follows:

```
begin {case_I}
        if $x_B < -z_B$ then EXIT; {Line segment is rejected}
        if $y_A > z_A$ then
                case_I_1            {point A is in the top-right corner region}
        else if $y_A >= -z_A$ then
                case_I_2            {point A is in the right edge region}
        else
                case_I_3            {point A is in the bottom-right corner region}
end {case_I};
```

I.1. Point A is in the top-right corner region and point B is not beyond the right boundary.

If point *B* is above the top boundary, the line segment is invisible and no further computation is needed. Therefore, we need to check this condition first, and then characterize point *B* so that we can distinguish between the case when point *B* is beyond the left boundary and the case when point *B* is inside of the pyramid or in the bottom edge region, see Figure 8.5. The following pseudo-code shows how it can be implemented:

```
begin {case_I_1}
        if $y_B > z_B$ then EXIT; {Line segment is rejected}
        if $x_B > z_B$ then        case_I_1_a
                {Point B is in the left edge or in the bottom-left corner region}
                else               case_I_1_b
                {Point B is inside of the pyramid or in the bottom edge region}
end {case_I_1};
```

I.1.a) Point A is in the top-right corner region and point B is in the left edge region or in the bottom-left corner region.

If point *B* is above the plane ρ_1, the line segment is rejected, see Figure 8.5. Therefore, we must check this condition first, and then distinguish the case when point *B* is in the left edge region and the case when point *B* is in the bottom-left corner region. In the case of the left edge region, one intersection point lies on the pyramid's left boundary. The location of point *B* against the plane ρ_2 will specify that the second intersection point lies on the top or on the right boundary. By this way only the appropriate intersection point is computed. In the case of bottom-left corner region, the comparison of point *B* against the plane ρ_3 is performed first to eliminate the case when the line segment is rejected. After that, we compare the position of point *B* against the plane ρ_4 to determine location of the first intersection point (on the bottom or on the left boundary). At the end, a similar comparison with the plane ρ_2 is performed to determine the second intersection point. An implementation can be as follows:

```
begin {case I_1_a}
        $\Delta x := x_B - x_A$;  $\Delta y := y_B - y_A$;  $\Delta z := z_B - z_A$;
        if $((x_A - z_A)*(\Delta z - \Delta y) > (y_A - z_A)*(\Delta z - \Delta x))$ then EXIT; {Line segment is rejected}
        {first intersection point computation}
        if $y_B > -z_B$ then $t_l := (x_A - z_A)/(\Delta z - \Delta x)$        {B is in the left edge region}
        else {B is in the bottom left corner region}
                begin
                        if $((x_A + z_A)*(\Delta z + \Delta y) > (y_A + z_A)*(\Delta z + \Delta x))$ then EXIT;
                        if $((z_A - x_A)*(\Delta y + \Delta z) > (z_A + y_A)*(\Delta z - \Delta x))$ then
                                $t_l := (x_A - z_A)/(\Delta z - \Delta x)$  {intersection with left boundary}
                        else $t_l := -(y_A + z_A)/(\Delta z + \Delta y)$    {intersection with bottom boundary}
                end;
        {second intersection point computation}
        if $((x_A + z_A)*(\Delta z - \Delta y) > (z_A - y_A)*(\Delta z + \Delta x))$ then
                $t_2 := (y_A - z_A)/(\Delta z - \Delta y)$                  {intersection with top boundary}
        else    $t_2 := -(x_A + z_A)/(\Delta z + \Delta x)$                 {intersection with right boundary}
end {case I_1_a};
```

I.1.b) Point A is in the top-right corner region and point B is inside of the pyramid or in the bottom edge region.

In the case when point *B* is inside of the pyramid, the position of point *B* against the plane ρ_2 specifies whether the intersection point lies either on the top or on the right boundary. In the case when point *B* is in the bottom edge region, the comparison of point *B* against the plane ρ_3 must be performed first to eliminate the situation when the line segment is rejected. If the line segment *AB* is not rejected by the clipping algorithm then one intersection point lies on the bottom boundary. The other intersection point lies on the top or on the right boundary according to the position of point *B* against the plane ρ_2. This section can be implemented as follows:

```
begin {case I_1_b}
        $\Delta x := x_B - x_A$;  $\Delta y := y_B - y_A$;  $\Delta z := z_B - z_A$;
        {first intersection point computation}
        if $y_B < -z_B$ then          {B in bottom edge region}
                begin
                        if (($x_A+z_A$)*($\Delta z+\Delta y$)>($y_A+z_A$)*($\Delta z+\Delta x$)) then EXIT;
                        $t_1$ := -($y_A+z_A$)/($\Delta z+\Delta y$)
                end
        else $t_1$ := 1;              {B is inside the pyramid}
        {second intersection point computation}
        if (($x_A+z_A$)*($\Delta z-\Delta y$) > ($z_A-y_A$)*($\Delta z+\Delta x$)) then
                $t_2$ := ($y_A-z_A$)/($\Delta z-\Delta y$)          {intersection with top boundary}
        else    $t_2$ := -($x_A+z_A$)/($\Delta z+\Delta x$)         {intersection with right boundary}
end {case_I_1_b};
```

I.2. Point A is in right edge region and the point B is not beyond the right boundary.

We need to distinguish the cases, when point *B* is bellow the bottom boundary (point *B* is in bottom-left corner region or in the bottom edge region), or above the top boundary (point *B* is in top-left corner region or in the top edge region) or between top and bottom boundaries. An implementation can be as follows:

```
begin {case_I_2}
        if $y_B < -z_B$ then case_I_2_a
                {point B is in the bottom-left corner or in the bottom edge region}
        else  if $y_B <= z_B$ then case I_2_b
                        {point B is in the left edge region or inside of the pyramid}
              else    case_I_2_c
                        {point B is in the top-left corner or in the top edge region}
end {case_I_2}
```

I.2.a) Point A is in the right edge region and point B is in the bottom-left corner or in the bottom edge region.

The location of point *B* against the plane ρ_3 helps us to eliminate the case when the line segment is rejected. If point *B* is in the bottom edge region then the intersection points are on the right and the bottom boundaries. If point *B* is in the bottom-left corner region then one intersection point lies on the right boundary and the second intersection point's location (either on the left or on the bottom boundary) is determined by the location of point *B* against the plane ρ_4.

```
begin {case_I_2_a}
    if ((x_A+z_A)*(Δz+Δy) > (y_A+z_A)*(Δz+Δx)) then EXIT;
    {first intersection point computation}
    if x_B > z_B then {point B in the bottom-left corner }
        if ((z_A-x_A)*(Δz+Δy) > (z_A+y_A)*(Δz-Δx)) then
            t_1:=(x_A–z_A)/(Δz-Δx)          {intersection with left boundary}
        else   t_1:= -(y_A+z_A)/(Δz+Δy)     {intersection with bottom boundary}
    else {point B in bottom edge region}
        t_1:= -(y_A+z_A)/(Δz+Δy);           {intersection with bottom boundary}
    {second intersection point computation}
    t_2 := -(x_A+z_A)/(Δz+Δx)
end {case_I_2_a};
```

I.2.b) Point A is in the right edge region and point B is inside of the pyramid or in the left edge region.

In this case, one intersection point is on the right boundary and the second one (if point *B* is in the left edge region) is on the left boundary, see following pseudo-code:

```
begin {case_I_2_b}
    {first intersection point computation}
    t_1 := -(x_A+z_A)/(Δz+Δx);
    {second intersection point computation}
    if x_B > z_B then t_2 := (x_A–z_A)/(Δz-Δx)      {point B in left edge region}
              else   t_2:= 1
end {case_I_2_b};
```

I.2.c) Point A is in the right edge region and point B is in the top-left corner or in the top edge region: similar to case_I_2_a.

I.3. Point A is in the bottom-right corner region and point B is not beyond the right boundary.

This case is similar to case_I_1.

II. Point A is between the left and the right boundaries.

In this case, we need to characterize the location of point *A* to specify that, if point *A* lies inside of the pyramid or in an edge region. The following pseudo-code shows how it can be done:

```
begin {case_II}
    if $y_A > z_A$ then case_II_1              {point A is in the top edge region}
    else    if $y_A < -z_A$ then case_II_2     {point A is in the bottom edge region}
            else case_II_3                      {point A is inside the pyramid}
end {case_II};
```

We need to consider only the case when point *A* is inside the pyramid (case_II_3). The cases, when point *A* is in the top (case_II_1) or bottom edge region (case_II_2), are similar as the case when point *A* is in the right edge region (case_I_2).

II.3. Point A is inside the pyramid.

If point *B* lies in an edge region then the boundary, on which the intersection point lies, is determined, see the Figure 8.3. And the appropriate intersection point is computed. If point *B* lies in a corner region then a comparison of point *B* with an appropriate plane ρ_i is necessary before the appropriate intersection point is computed. An implementation can be illustrated as follows:

```
begin {case_II_3}
    if $x_B < -z_B$ then                  {case_II_3_a-c}
        if $y_B > z_B$ then case_II_3_a {point B is in the top-right corner region}
        else    if $y_B >= -z_B$ then               case_II_3_b
                                      {point B is in the right edge region}
                else                                case_II_3_c
                                      {point B is in the bottom-right corner region}
    else    if $x_B > z_B$ then           {case_II_3_d-f}
                if $y_B > z_B$ then                 case_II_3_d
                                      {point B is in the top-left corner region}
```

```
                    else    if $y_B < -z_B$ then      case_II_3_e
                                    {point *B* is in the bottom-left corner region}
                            else                      case_II_3_f
                                    {point *B* is in the left edge  region}
            else                    {case_II_3_g-i}
                    if $y_B > z_B$ then               case_II_3_g
                                    {point B is in the top edge region}
                    else    if $y_B < -z_B$ then      case_II_3_h
                                    {point B is in the bottom edge region}
                            else                      case_II_3_i
                            {*B* is inside pyramid, whole line segment is visible}
end {case_II_3};
```

Now, we need to consider only the two cases when point *B* is in top-right corner region, i.e. case II.3.a or in the right edge region, i.e. case II.3.b. It can be seen that the other cases are similar.

II.3.a) Point A is inside of the pyramid and point B is in top-right corner region.

The comparison of point *B* with the plane ρ_2 specifies which boundary (top or right) to be used to compute the intersection point. An implementation can be as follows:

```
begin {case_II_3_a}
        if $((x_A+z_A)*(\Delta z-\Delta y) > (z_A-y_A)*(\Delta z+\Delta x))$ then
                $t_1 := (y_A-z_A) / (\Delta z-\Delta y)$          {intersection with top boundary}
        else $t_1 := -(x_A+z_A)/( \Delta z+\Delta x)$;            {intersection with right boundary}
        $t_2 := 0$;
end {case_II_3_a};
```

The cases case_II_3_c, case_II_3_d and case_II_3_e are similar to case_II_3_a.

II.3.b) Point A is inside of the pyramid and point B is in right edge region.

computed. The following pseudo-code shows how it can be implemented:

```
begin {case_II_3_b}
        $t_1 := -(x_A+z_A)/( \Delta z+\Delta x)$;
        $t_2 := 0$
end {case_II_3_b};
```

The cases case_II_3_f, case_II_3_g and case_II_3_h can be solved similarly.

III. Point A is beyond the left boundary.

This case can be solved similarly to the case when point *A* is beyond the right boundary (case_I).

Finally, we can easily compute the end-points of the output line segment from the parameter value *t* as follow:

$$x := t*\Delta x + x_A;$$

$$y := t*\Delta y + y_A;$$

$$z := t*\Delta z + z_A$$

It can be seen that all possible cases were solved and the complete algorithm can be got by substitution all procedures by appropriate codes.

8.3.3. Comparison between CS and PC algorithms

To be able to compare the CS-3D algorithm with the proposed PC algorithm and evaluate the efficiency of the PC algorithm, we introduce a coefficient of efficiency ν as:

$$\nu = \frac{T_{CS-3D}}{T_{PC}}$$

where: $T_{CS\text{-}3D}$, T_{PC} denote the time consumed by the CS-3D algorithm and by the proposed PC algorithm, respectively.

For experimental verification, 80.10^6 different line segments were randomly generated for each of the considered cases, see Figure 8.6-7. The tests were performed on the HP Vectra XA Pentium-Pro, 200MHz, 128MB RAM, 256 KB CACHE. The obtained results are presented in Table 8.1, which shows that the PC algorithm is significantly faster in all considered **non-trivial** cases (when the line segment is not entirely inside of the pyramid). In the case that the line segment is inside of the clipping pyramid the efficiency of the PC algorithm is as good as the efficiency of the CS-3D algorithm. It can be seen that the speed-up varies from 1.17 to 2.07 approximately for all other cases, see Table 8.1.

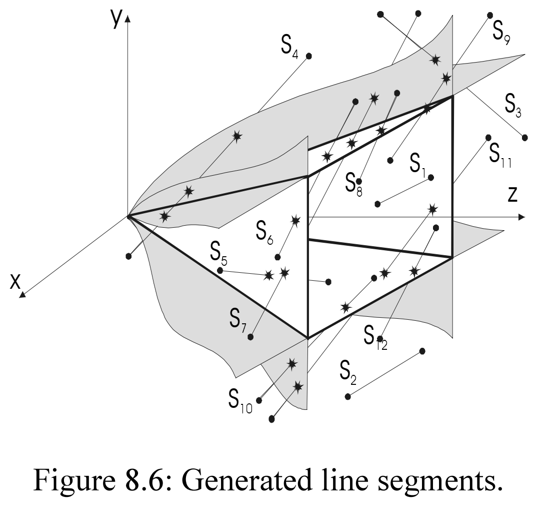


Figure 8.6: Generated line segments.

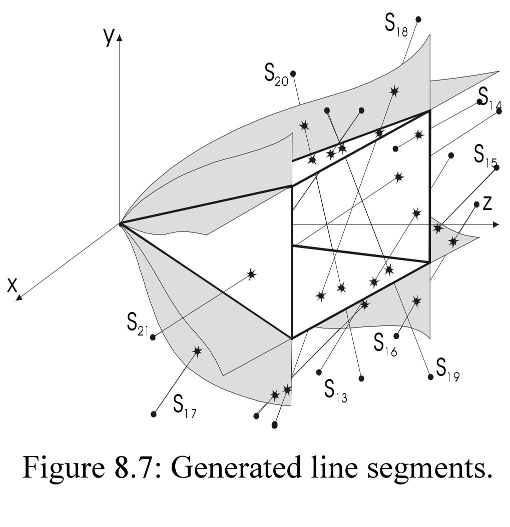


Figure 8.7: Generated line segments.

case	CS-3D	PC	ν
s1	137.86	137.58	**1.00**
s2	136.81	113.30	**1.21**
s3	227.86	158.57	**1.44**
s4	228.24	158.63	**1.44**
s5	227.03	179.95	**1.26**
s6	315.22	238.02	**1.32**
s7	409.78	252.36	**1.62**
s8	225.71	181.26	**1.25**
s9	319.56	196.54	**1.63**
s10	309.89	194.18	**1.60**
s11	402.91	252.31	**1.60**
s12	227.31	192.14	**1.18**
s13	319.89	248.63	**1.29**
s14	230.60	196.87	**1.17**
s15	309.18	149.34	**2.07**
s16	226.15	163.24	**1.39**
s17	398.46	248.35	**1.60**
s18	493.02	280.77	**1.76**
s19	315.77	240.60	**1.31**
s20	409.95	262.64	**1.56**
s21	319.56	240.00	**1.33**

Table 8.1: Comparison between CS-3D and PC algorithm.

8.4. Liang-Barsky algorithm for line clipping against a pyramid

As has been mentioned, in many applications it is necessary to clip lines instead of line segments. Therefore, LB algorithm was also extended to 3D for line clipping against a pyramid, see [Fol91]. The LB-3D algorithm is based on clipping of the given line by each boundary of the pyramid. The given line which passes points $A(x_A, y_A, z_A)$ and $B(x_B, y_B, z_B)$ is parametrically represented as follows:

$$x(t) = x_A + \Delta x * t,$$
$$y(t) = y_A + \Delta y * t,$$
$$z(t) = z_A + \Delta z * t,$$

where $\Delta x = x_B - x_A, \quad \Delta y = y_B - y_A, \quad \Delta z = z_B - z_A, \quad t \in (-\infty, +\infty)$

At the beginning of computation the parameter t is unlimited and then this interval is subsequently curtailed by all the intersection points with each boundary plane of the clipping pyramid, see Algorithm 8.2.

It can be seen that an additional trivial rejection test (function CLIPt) is used to avoid calculation of some parametric values for lines that do not intersect the clipping pyramid.

A weakness of the LB-3D algorithm is the need to compute the parameter t of intersection points that are not part of the result. For example, see line p in Figure 8.8, all four parametric values, representing intersection points with each boundary of pyramid, are computed, but only two are valid.

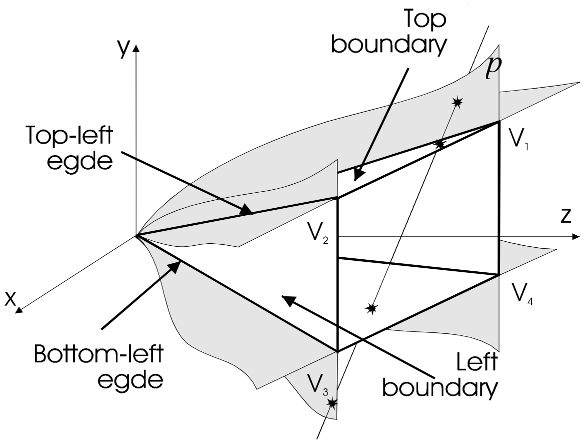


Figure 8.8: Line clipping against a viewing pyramid.

```
procedure LB-3D_Clip($x_A$, $y_A$, $z_A$, $x_B$, $y_B$, $z_B$: real);
          {two points A($x_A$,$y_A$,$z_A$), B($x_B$,$y_B$,$z_B$) determine the clipped line}

var       $t_1$, $t_2$, $\Delta x$, $\Delta y$, $\Delta z$: real;

          function CLIPt( p, q : real; var $t_1$, $t_2$ : real):boolean;
          var      r : real;
          begin  CLIPt:= true;
                   if $p < 0$ then
                            begin  $r := q / p$;
                                     if $r > t_2$ then CLIPt:= false
                                     else if $r > t_1$ then $t_1 := r$
                            end
                   else     if $p > 0$ then
                                     begin  $r := q / p$;
                                              if $r < t_1$ then CLIPt:= false
                                              else if $r < t_2$ then $t_2 := r$
                                     end
                            else     if $q < 0$ then CLIPt:= false
          end { of CLIPt };

begin
          $t_1 := -\infty$;                    $t_2 := +\infty$;
          $\Delta x := x_B - x_A$;  $\Delta z := z_B - z_A$;
          if CLIPt( $-\Delta x-\Delta z$, $x_A + z_A$, $t_1$, $t_2$) then
                   if CLIPt($\Delta x-\Delta z$, $z_A - x_A$, $t_1$, $t_2$) then
                            begin
                                     $\Delta y := y_B - y_A$;
                                     if CLIPt( $-\Delta y-\Delta z$, $y_A+z_A$, $t_1$, $t_2$)then
                                              if CLIPt($\Delta y-\Delta z$, $z_A- y_A$, $t_1$, $t_2$)then
                                                       begin  $x_B := x_A + \Delta x * t_2$;
                                                                $y_B := y_A + \Delta y * t_2$;
                                                                $z_B := z_A + \Delta z * t_2$;
                                                                $x_A := x_A + \Delta x * t_1$;
                                                                $y_A := y_A + \Delta y * t_1$;
                                                                $z_A := z_A + \Delta z * t_1$;
                                                                DRAW_LINE($x_A$,$y_A$,$z_A$, $x_B$,$y_B$,$z_B$)
                                                       end
                            end
end {of LB-3D_Clip };
```

Algorithm 8.1: The LB-3D algorithm.

8.5. The SF-3D Algorithm

Some considerations how to improve the efficiency of the LB-3D algorithm resulted into a new algorithm for line clipping, which is denoted as SF-3D algorithm. The separation function algorithm (SF-3D) for the line clipping in E^3 is an extension of SF algorithm for line clipping in E^2. It is based on a coding technique for vertices of the given clipping pyramid. Let us denote ρ as the plane, which passes the origin and line p. Plane ρ divides the whole space into two regions.

Let us define the separation function as:

$$f(x,y,z) = a*x + b*y + c*z + d$$

where $a = y_A*z_B - y_B*z_A$, $b = z_A*x_B - z_B*x_A$,

$c = x_A*y_B - x_B*y_A$, $d = 0$ (plane ρ passes the origin).

The sign of the separation function evaluated at a vertex determines the region in which the vertex lies. Using the value of the separation function for all vertices we can distinguish several possible cases, see Figure 8.9-8.19.

This analysis led to the SF-3D algorithm for E^3 case as a straightforward extension of the SF algorithm for E^2. The basic steps can be defined as:

- calculate the coefficients a, b, c of separation function,
- using the separation function to characterize the location of vertices of the given clipping pyramid,
- determine the appropriate case,
- compute the intersection points with appropriate boundaries.

It can be seen that only the intersection points required for the output are computed.

We will describe now the classification process more in detail. Let us denote c_1, c_2, c_3, c_4 as values of the separation function of the clipping pyramid's vertices V_1, V_2, V_3, V_4, see Figure 8.8.

There are two major cases to be distinguished:

- the vertices V_1 and V_3 lie on the different sides of plane ρ, i.e. $c_1 * c_3 < 0$, see Figures 8.9-8.12,
- the vertices V_1 and V_3 lie on the same side of plane ρ, see Figures 8.13-8.19.

a) **The vertices V_1, V_3 are in the different sides of plane** ρ. In this case, the sign of expression (c_1*c_2) determine whether V_1, V_2 lie on the different sides of plane ρ (the first two cases) or not (the two following cases). If V_1, V_2 lie on the different sides of plane ρ $(c_1*c_2<0)$, then one intersection point lies on the top boundary of clipping pyramid (Figures 8.9-8.10). Otherwise, it lies on the left boundary of clipping pyramid, see Figures 8.11-8.12.

To determine the location of the second intersection point, we use the condition $(c_1*c_4 < 0)$. If $(c_1*c_4 < 0)$ then V_1, V_4 lie on the different sides of plane ρ and the second intersection point lies on the right boundary of clipping pyramid, see Figures 8.10-8.12. Otherwise, V_1 and V_4 lie on the same side of plane ρ and the second intersection point lies on the bottom boundary of clipping pyramid, see Figures 8.9-8.11.

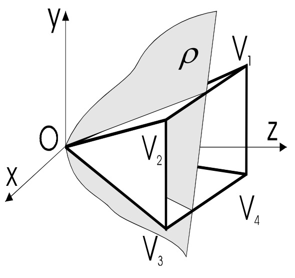


Figure 8.9

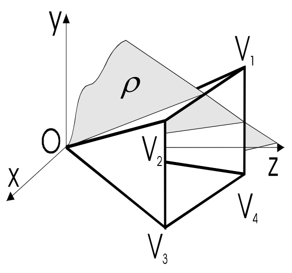


Figure 8.10

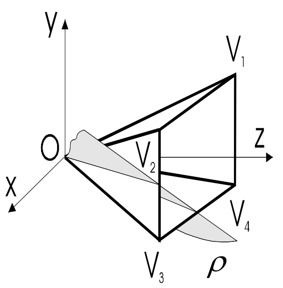


Figure 8.11

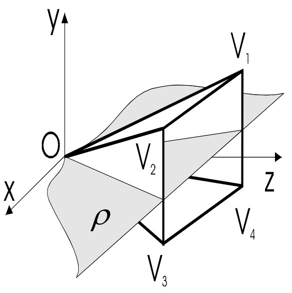


Figure 8.12

b) **The vertices V_1, V_3 lie on the same side of plane** ρ. In this case, if $(c_1*c_2 < 0)$, i.e. V_1, V_2 lie on the different sides of plane ρ, then the intersection points lie on the top and the left boundaries, see Figure 8.13. Otherwise, the additional test $(c_1*c_4 < 0)$ determines that the intersection points lie on the bottom and the right boundaries, see Figure 8.14. In the case when *$(c_1*c_4 < 0)$* is false, we need to test if the vertex V_1 lies on the plane ρ $(c_1 = 0)$. When the vertex V_1 does not lie on the plane ρ, the whole line is outside of clipping pyramid. For the case when the vertex V_1 lies on the plane ρ we need to distinguish two following sub-cases:

- The vertex V_2 also lies on the plane ρ $(c_2 = 0)$, the condition $(c_3 = 0)$ determines that the origin *O* lies on line *p*. For this case, one additional test must be done to distinguish between the case when the whole line is invisible (Figure 8.15) and the case when only half of the line is visible (Figure 8.16). If the origin does not lie on line p $(c_3 \neq 0)$, the whole line is outside of clipping pyramid.
- When the vertex V_2 does not lie on the plane ρ $(c_2 \neq 0)$, if $(c_2*c_4 >= 0)$ the whole line is outside of clipping pyramid, see Figure 8.19. Otherwise, one intersection point lies on top-right edge of the clipping pyramid and the additional test $(c_2*c_3 < 0)$ determines if the second intersection point lies on the left boundary (Figure 8.17) or the bottom boundary of the clipping pyramid (Figure 8.18).

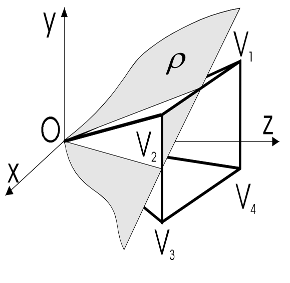


Figure 8.13

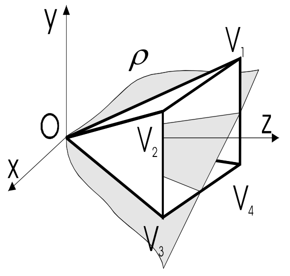


Figure 8.14

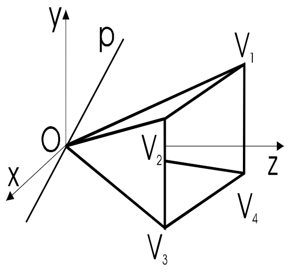


Figure 8.15

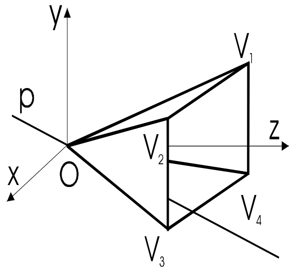
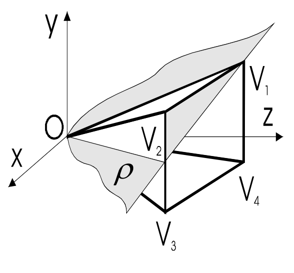


Figure 8.16

The complete SF-3D algorithm can be implemented by the Algorithm 8.3.

It can be seen that, before calculating the intersection point between line *p* and one boundary of the pyramid, we always need to test whether line *p* is parallel to that boundary, see function **CalcT** in Algorithm 8.3. In the parallel case, the intersection point lies in the infinite, see Figure 8.20. However, the parallelism of line *p* with some pyramid's boundaries is highly impossible in normal situations. Moreover, the conditions with $c_i = 0$ occur practically with **zero probability**, therefore, the test of these conditions (part {**Special cases**} in Algorithm 8.3) has no influence to the algorithm efficiency.



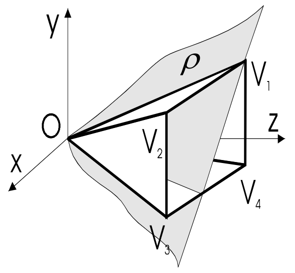


Figure 8.17 Figure 8.18

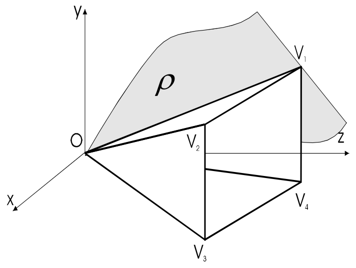


Figure 8.19

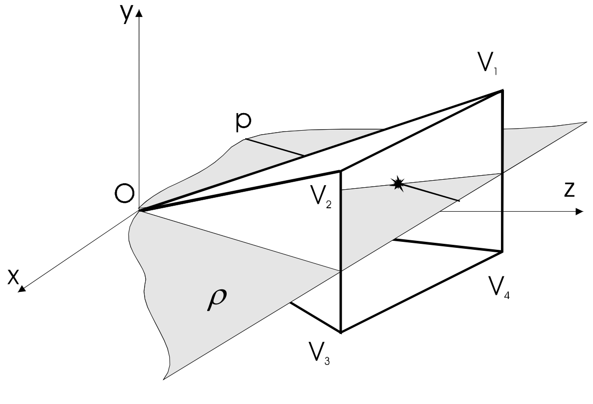


Figure 8.20: *p* is parallel to left boundary.

```
procedure SF-3D_Clip ( $x_A$, $y_A$, $z_A$, $x_B$, $y_B$, $z_B$: real; var $t_1$, $t_2$ : real);
{input : two points A($x_A$, $y_A$, $z_A$), B($x_B$, $y_B$, $z_B$) determine the clipped line}
{output : $t_1$, $t_2$ : parameter values of intersection points}

var     $\Delta x$, $\Delta y$, $\Delta z$, a, b, c, $c_1$, $c_2$, $c_3$, $c_4$, r, s: real;
        function CalcT ($a_1$, $s_1$, $a_2$, $s_2$ : real) : real;           {implemented as macro}
        begin    if ($s_1 = s_2$)  then  if ($\Delta z > 0$) then CalcT:= $+\infty$ else CalcT:= $-\infty$
                                   else      CalcT := $(a_1 - a_2) / (s_2 - s_1)$
        end;
begin   $t_1$:= $+\infty$; $t_2$:= $-\infty$;        {empty interval < $t_1$, $t_2$ >}
        $\Delta x := x_B - x_A$;  $\Delta y := y_B - y_A$;  $\Delta z := z_B - z_A$;
        $a := y_A * z_B - y_B * z_A$;  $b := z_A * x_B - z_B * x_A$;  $c := x_A * y_B - x_B * y_A$;
        $c_1$:= -a + b + c;        $c_2$:= a + b + c;        $c_3$:= a - b + c;        $c_4$:= -a - b + c;
        if ($c_1 * c_3 < 0$) then
             begin   if ($c_1 * c_2 < 0$)  then $t_1$ := CalcT($y_A$, $\Delta y$, $z_A$, $\Delta z$)    {Figure 9-10}
                                          else $t_1$ := CalcT($x_A$, $\Delta x$, $z_A$, $\Delta z$);    {Figure11-12}
                     if ($c_1 * c_4 < 0$)  then $t_2$ := CalcT(-$x_A$, -$\Delta x$, $z_A$, $\Delta z$) {Figure 10-12}
                                          else $t_2$ := CalcT(-$y_A$, -$\Delta y$, $z_A$, $\Delta z$)  {Figure 9-11}
             end
        else    if ($c_1 * c_2 < 0$) then                                        {Figure 13}
                     begin  $t_1$:= CalcT($y_A$,$\Delta y$,$z_A$,$\Delta z$);  $t_2$:= CalcT($x_A$,$\Delta x$,$z_A$,$\Delta z$) end
                else if ($c_1 * c_4 < 0$) then                                   {Figure 14}
                     begin $t_1$:= CalcT(-$y_A$,-$\Delta y$,$z_A$,$\Delta z$);  $t_2$:= CalcT(-$x_A$,-$\Delta x$,$z_A$,$\Delta z$) end
                   else begin if ($c_1$ <> 0) then EXIT{SF-3D_Clip};{Special cases}
                              if ($c_2$ = 0) then
                                 begin
                                    if ($c_3$ <> 0) then EXIT{SF-3D_Clip};
                                    if $\Delta z < 0$ then
                                         begin $\Delta x$:= -$\Delta x$; $\Delta y$:= -$\Delta y$; $\Delta z$:= -$\Delta z$ end;
                                    if ($\Delta x>\Delta z$) or ($\Delta x<-\Delta z$) or ($\Delta y>\Delta z$) or ($\Delta y<-\Delta z$)
                                         then EXIT {SF-3D_Clip}     {Figure 15}
                                         else begin                 {Figure 16}
                                                $t_1$:= CalcT(0,0,$z_A$,$\Delta z$);
                                                $t_2$:= CalcT(0,$\Delta z$,0,$\Delta z$)
                                              end
                                 end
                              else begin
{Figure 19}                         if ($c_2 * c_4 >= 0$) then EXIT{SF-3D_Clip};
{Figure 17}                         if ($c_2 * c_3 < 0$)  then $t_1$ := CalcT($x_A$, $\Delta x$, $z_A$, $\Delta z$);
{Figure 18}                                               else $t_1$ := CalcT(-$y_A$, -$\Delta y$, $z_A$, $\Delta z$);
                                    $t_2$ := CalcT(-$x_A$, -$\Delta x$, $z_A$, $\Delta z$)
                                 end
                        end;
        r := $z_A + \Delta z * t_1$;        s := $z_A + \Delta z * t_2$;
        if (r < 0) then if (s < 0)  then EXIT else if ($\Delta z > 0$) then $t_1$:= $+\infty$ else $t_1$:= $-\infty$
                   else if (s < 0)  then if ($\Delta z > 0$) then $t_2$:= $+\infty$ else $t_2$:= $-\infty$
end {of SF-3D_Clip};
```

Algorithm 8.3: SF-3D algorithm

The result of the SF-3D algorithm can be:

- line *p* is totally outside of the clipping pyramid and $t_1=+\infty$; $t_2=-\infty$, i.e. $< t_1, t_2>$ is empty
- a line segment (a part of line *p* that is inside of the clipping pyramid) is generated and its end-points can be determined as:

$$x_B := x_A + \Delta x * t_2;$$
$$y_B := y_A + \Delta y * t_2;$$
$$z_B := z_A + \Delta z * t_2;$$
$$x_A := x_A + \Delta x * t_1;$$
$$y_A := y_A + \Delta y * t_1;$$
$$z_A := z_A + \Delta z * t_1;$$

- a half-line (a part of line *p* that is inside of the clipping pyramid) is generated. In this case, the only one value of t_1, resp. t_2, is limited, the second one is unlimited and the only one valid end-point can be determined as:

  $x_A := x_A + \Delta x * t_1;$ if $t_1 \neq \pm\infty$

  $y_A := y_A + \Delta y * t_1;$

  $z_A := z_A + \Delta z * t_1;$

  resp.

  $x_B := x_A + \Delta x * t_2;$ if $t_2 \neq \pm\infty$

  $y_B := y_A + \Delta y * t_2;$

  $z_B := z_A + \Delta z * t_2;$

  Now the half-line can be represented as:

  $x(t) := x_A + \Delta x * t;$ if $t_1 \neq \pm\infty$

  $y(t) := y_A + \Delta y * t;$

  $z(t) := z_A + \Delta z * t;$

  where:

$$t \in \begin{cases} <0,\infty) & \text{if } t_2 = +\infty \\ (-\infty,0> & \text{if } t_2 = -\infty \end{cases}$$

  resp.

  $x(t) := x_B + \Delta x * t;$ if $t_2 \neq \pm\infty$

  $y(t) := y_B + \Delta y * t;$

  $z(t) := z_B + \Delta z * t;$

where:

$$t \in \begin{cases} <0,\infty) & \text{if } t_1 = +\infty \\ (-\infty, 0> & \text{if } t_1 = -\infty \end{cases}$$

8.5.1. Comparison between LB-3D and SF-3D algorithms

The new proposed SF-3D algorithm was verified experimentally on Pentium II, 350MHz, 64MB RAM, 512KB CACHE for evaluation of the SF-3D algorithm against the LB-3D algorithm. Some typical cases were tested and 80.10^6 different lines were randomly generated for each considered case, see Figure 8.21.

Let us introduce the coefficients of efficiency ν as:

$$\nu = \frac{T_{LB-3D}}{T_{SF-3D}}$$

Where T_{LB-3D}, T_{SF-3D} are times consumed by the LB-3D and SF-3D algorithms.

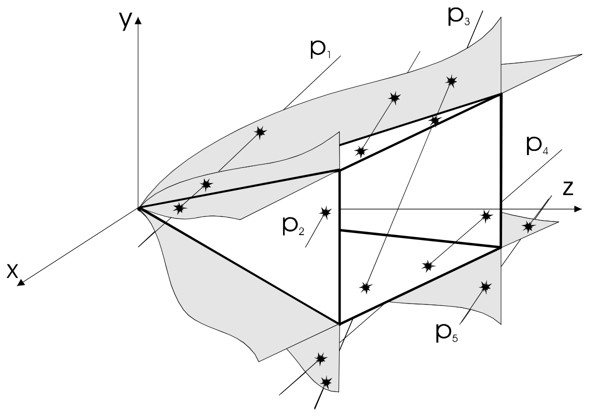


Figure 8.21: Generated lines for comparison between LB-3D and SF-3D algorithm.

The experimental results are presented in Table 8.2. This table shows that the SF-3D algorithm is significantly faster than the LB-3D algorithm in all considered cases. It can be seen that the speed-up varies from 1.29 to 1.90 approximately.

case	T_{LB-3D} **[s]**	T_{SF-3D}**[s]**	ν
p_1	118.846	62.473	**1.90**
p_2	137.747	101.648	**1.36**
p_3	137.143	106.044	**1.29**
p_4	134.121	101.923	**1.32**
p_5	96.593	62.527	**1.55**

Table 8.2: Comparison between LB-3D and SF-3D algorithm.

9. Clipping by a convex polyhedron in E^3

In technical practice, many applications need to clip lines or line segments not only against the viewing pyramid but also against the volumetric objects. In computer graphics, the volumetric objects are usually represented as polyhedrons. In this section, we are going to describe algorithms for line clipping against a polyhedron in more detail.

9.1. CB-3D algorithm

Algorithms for line clipping in E^3 are mostly based on the Cyrus-Beck (CB-3D) algorithm and its modifications, see [Cyr78a]. Algorithm 9.1 illustrates a shortened version of the CB-3D algorithm.

```
procedure CB_Clip_3D ( $\mathbf{x}_A$ , $\mathbf{x}_B$ );
        { $\mathbf{n}_i$ is a normal vector of the $i$-th facet }
        { $\mathbf{n}_i$ must point out of the convex polyhedron }
        { !! all normal vectors $\mathbf{n}_i$ are precomputed !! }
begin
        $t_{min} := 0;$   $t_{max} := 1;$   $\mathbf{s} := \mathbf{x}_B - \mathbf{x}_A;$
        { for the line clipping $t_{min} := -\infty;$ $t_{max} := +\infty;$ }
        for $i := 1$ to $N$ do        { $N$ is a number of facets }
        begin
                $\xi := \mathbf{s}^T \mathbf{n}_i;$   $\mathbf{s}_i := \mathbf{x}_i - \mathbf{x}_A;$
                if $\xi <> 0$ then
                begin
                        $t := \mathbf{s}_i^T \mathbf{n}_i / \xi;$
                        if $\xi > 0$ then $t_{max} :=$ min $(t, t_{max})$
                                  else $t_{min} :=$ max $(t, t_{min})$
                end
                else Special case solution;{ line is parallel to a facet }
                $i := i + 1$
        end;
        if $t_{min} > t_{max}$ then EXIT; { !! $< t_{min}, t_{max} > = \emptyset$ }
        { recompute end-points of the line segment if changed }
        { for lines points $\mathbf{x}_A$ , $\mathbf{x}_B$ must be always recomputed }
        if $t_{max} < 1$ then $\mathbf{x}_B := \mathbf{x}_A + \mathbf{s} * t_{max};$
        if $t_{min} > 0$ then $\mathbf{x}_A := \mathbf{x}_A + \mathbf{s} * t_{min};$
        DRAW_LINE ( $\mathbf{x}_A$ , $\mathbf{x}_B$ );
end { CB_Clip_3D };
```

Algorithm 9.1: CB-3D algorithm for line clipping by a convex polyhedron.

The main advantages of the CB-3D algorithm are its numerical stability and simplicity. The algorithm is based on the direct intersection computation of a line with a plane in E^3.

It is obvious that the algorithm runs with *O(N)* time complexity and so with increasing number of facets of the given polyhedron the efficiency of the CB-3D algorithm decreases because many invalid intersection points are computed.

The main disadvantage of the CB-3D algorithm is a direct line intersection computation for all planes which form the boundary of the given convex polyhedron. It means that ***N - 2* intersection computations are wasted** if N is a number of facets of the given convex polyhedron. That is substantial because the average number of facets of the given polyhedron is very high (a number of facets might easily reach for a sphere approximation value 10^4). Therefore, it is necessary to find an effective method for selection of facets that might be intersected by the given line. Such attempts have been done at our laboratory that resulted in the following algorithms.

9.2. Algorithm based on two planes use

Let the convex polyhedron P is defined by triangular facets (generally it is not necessary). We need to find an **effective test** whether line p intersects a facet of the given polyhedron P, see Figure 9.1, and then compute intersection points, see [Ska96a].

It can be seen that any line p in E^3 can be defined as an intersection of two non-parallel planes ρ_1 and ρ_2, see Figure 9.2. It means that if line p intersects the given facet then planes ρ_1 and ρ_2 intersect the given facet, too, but **not vice versa**, see Figure 9.3. Line p' that is defined as the intersection of ρ_3 and ρ_4 planes does not intersect the facet.

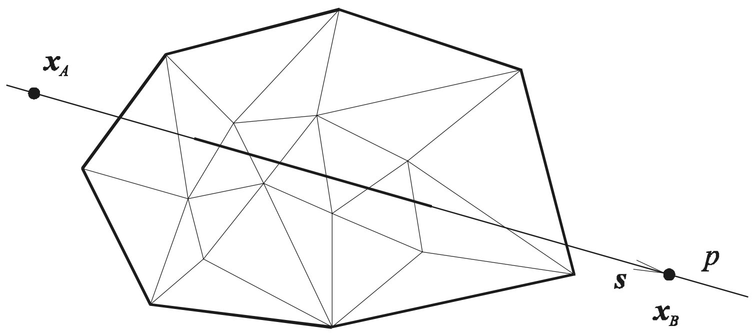
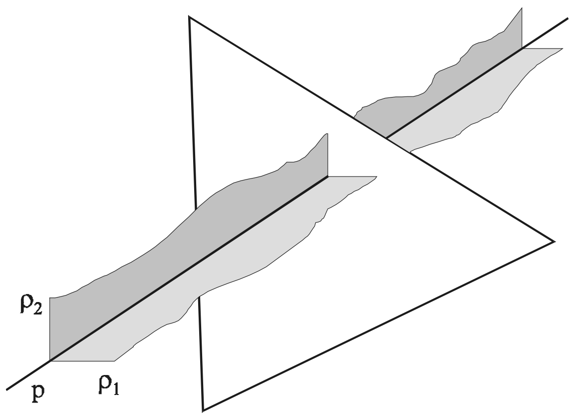
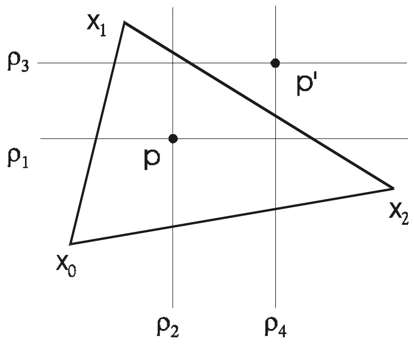


Figure 9.1: Line clipping by convex polyhedron in E^3.



Usage of two planes for line definition

Figure 9.2



Possible kinds of intersection

Figure 9.3

It is possible to test all facets of the given polyhedron against ρ_1 and ρ_2 planes. If both planes ρ_1 and ρ_2 intersect the given facet then compute detailed intersection test. The intersection of the given plane ρ_i and the facet (triangle) exists **if and only if** exist two vertices $\boldsymbol{x}_A$ and $\boldsymbol{x}_B$ of the triangle so that

$$sign(F_i(\mathbf{x}_A)) \neq sign(F_i(\mathbf{x}_B))$$

where $F_i(\mathbf{x}) = a_i\, x + b_i\, y + c_i\, z + d_i$ is an equation for the i-th plane ρ_i, i=1,2.

The substantial advantage is that ρ_1 and ρ_2 planes can be taken parallel with any co-ordinate axes. Those planes are usually called „diagonal“. Using these planes we save one addition and two multiplications for each separation function evaluation.

It can be seen that the separation function $F_1(\mathbf{x})$, $F_2(\mathbf{x})$ resp. are computed more times than needed because **each vertex is shared** by many triangles. Therefore it is convenient if the values $sign(F_1({}^i\mathbf{x}_k))$ are precomputed (${}^i\boldsymbol{x}_k$ is the k-th vertex of the i-th facet) and stored in a separate vector, see Algorithm 9.2. This modification significantly improves the efficiency of the algorithm. It is possible to select planes ρ_1 and ρ_2 as two „diagonal “ planes in order to avoid singular cases, see [Ska96a] for detail description. The algorithm can be easily modified for non-convex polyhedron and expected speed-up is up to **3,17**. Nevertheless, this algorithm is still of $O(N)$ complexity.

```
procedure CLIP_3D_MOD ( $\boldsymbol{x}_A$ , $\boldsymbol{x}_B$ );
begin
    $t_{min} := 0;$   $t_{max} := 1;$ $\boldsymbol{s} := \boldsymbol{x}_B - \boldsymbol{x}_A;$
    { for the line clipping $t_{min} := -\infty;$ $t_{max} := +\infty;$ }
    { $\rho_1 : F_1(\boldsymbol{x}) = a_1 x + c_1 z + d_1 = 0$     $\rho_2 : F_2(\boldsymbol{x}) = b_2 y + c_2 z + d_2 = 0$ }
    for $k := 1$ to $N_v$ do   { $N_v$ number of vertices }
        $Q_k$ := sign$(F_1(\boldsymbol{x}_k));$ { $Q$ is a vector of integer or char types }
    for $i := 1$ to $N$ do     { $N$ number of facets }
    begin
        { ${}^i\boldsymbol{x}_k$ means a $k$-th vertex of the $i$-th triangle }
        { INDEX($i,k$) gives the index of $k$-th vertex of the $i$-th triangle,}
        { i.e. ${}^i\boldsymbol{x}_k = \boldsymbol{x}_{\text{INDEX}(i,k)}$ }
        if $Q_{\text{INDEX}(i,0)} = Q_{\text{INDEX}(i,1)}$ then
            if $Q_{\text{INDEX}(i,0)} = Q_{\text{INDEX}(i,2)}$ then goto 1;
                        {do nothing $\rho_1$ does not intersect the $i$-th triangle}
            if sign$(F_2(\boldsymbol{x}_{\text{INDEX}(i,0)}))$ = sign$(F_2(\boldsymbol{x}_{\text{INDEX}(i,1)}))$ then
                if sign$(F_2(\boldsymbol{x}_{\text{INDEX}(i,0)}))$ = sign$(F_2(\boldsymbol{x}_{\text{INDEX}(i,1)}))$ then goto 1;
        { both planes $\rho_1$, $\rho_2$ intersect the $i$-th triangle }
        {detailed test finds a value $t_{min}$ or $t_{max}$ }
        {using a single step of the CB algorithm}
        { $\boldsymbol{n}_i$ must point out of the given convex polyhedron }
        $\xi := \boldsymbol{s}^T \boldsymbol{n}_i;$
        if $\xi <> 0$ then
        begin $\boldsymbol{s}_i := \boldsymbol{x}_i - \boldsymbol{x}_A;$
            $t := \boldsymbol{s}_i^T \boldsymbol{n}_i / \xi;$
            if $\xi > 0$ then $t_{max}$ := min $(t, t_{max})$
                    else $t_{min}$ := max $(t, t_{min})$
        end
        else Special case solution;{ line is parallel to a facet }
1:      $i := i + 1$
    end;
    { if $t_{min} > t_{max}$ then no intersection point exists }
    if $t_{min} \le t_{max}$ then DRAW_LINE ($\boldsymbol{x}(t_{min}), \boldsymbol{x}(t_{max})$)
end { of CLIP_3D_MOD };
```

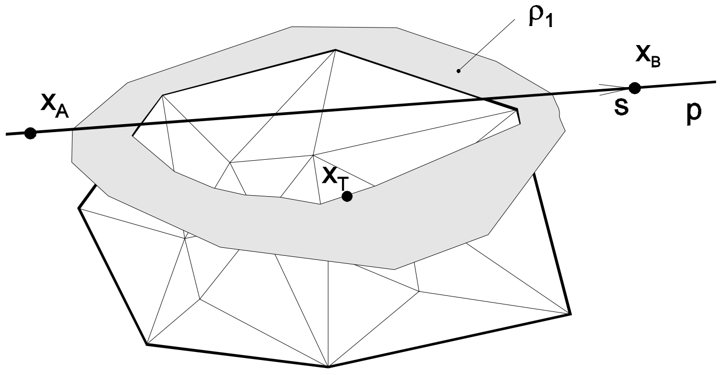
Algorithm 9.2: An efficient algorithm for line clipping by convex polyhedron.

9.3. $O(\sqrt{N})$ algorithm

Finding a facet candidate for the intersection point is a quite complex task and without knowing the "order" of facets, an algorithm will be generally of *O(N)* expected complexity. Let us select any triangle (facet) τ_k with two following properties:

- the given line does not lie on τ_k
- the given line does not intersect τ_k

Any point inside of the triangle τ_k, e.g., a centre of the facet $\boldsymbol{x}_T$ $(\boldsymbol{x}_T = \sum_1^3 \boldsymbol{x}_i/3)$ and the given line unambiguously define the plane ρ_1, on which the line lies, see Figure 9.4. It can be seen that more efficient strategy for testing triangles can be developed if we know facets that are intersected by this plane. If non-convex polyhedron is considered then two or more separate “rings of triangles“ are necessary to solve but only one will be detected and solved. Therefore, this algorithm is restricted to convex polyhedra.



Definition of the plane ρ_1

Figure 9.4

In many applications, data structures (modified winged edge) that define a convex polyhedron contain information about neighbours of the given triangular facet, see Figure 9.5. It is obvious that we can easily detect which edge of the given triangle τ_k is intersected by the selected plane ρ_1. In the next step we can take its neighbour, which has a common edge with the triangle τ_k, etc. Only the facets reached by this neighbour construction have to be taken into consideration, i.e. have to be tested against ρ_2.

Using "knowledge of order" we will have to test significantly less triangles than *N*. However, in the worst case we will have to test all *N* facets. It means that the algorithm is of *O(N)* complexity in the worst case, see [Ska97a] for details.

Let us consider a surface of a convex polyhedron, e.g. a sphere approximation. Number of intersected facets by a plane ρ_1 can be estimated as of $\sqrt{N}$ expected in average. This algorithm is briefly described in Algorithm 9.3. It is necessary to point out that there are some small obstacles in solving singular cases, especially if the plane ρ_1 intersects the triangle in vertex, etc. The presented approach can be easily modified for non-triangular facets if **get_next** procedure is modified properly. The expected speed-up for polyhedron with 500 facets is about **4,5-4,9** against CB-3D algorithm.

```
procedure SQRT_CLIP ( $\mathbf{x}_A$ , $\mathbf{x}_B$ );
begin
    $\mathbf{s} := \mathbf{x}_B - \mathbf{x}_A$;
    $t_{min} := 0$;    $t_{max} := 1$;        {for the line clipping $t_{min} := -\infty$; $t_{max} := +\infty$; }
    {finds a convenient facet $\tau_k$ and plane $\rho_1$ through $\mathbf{x}_A$ , $\mathbf{x}_B$ , $\mathbf{x}_T$}
    COMPUTE ( $\mathbf{x}_A$, $\mathbf{x}_B$, $k$, $\rho_1$ );
    {computes coefficients of the orthogonal plane to $\rho_2$}
    COMPUTE_ORTHOGONAL ($\mathbf{x}_A$, $\mathbf{x}_B$, $\rho_2$ );
    $k_0 := k$; {save index of first facet}
    $i := -1$; {set index of previous facet}
    repeat
        if TEST ($\tau_k$, $\rho_2$) then          {test the facet $\tau_k$ against the plane $\rho_2$}
        begin {a single step of the CB algorithm}
            $\xi := \mathbf{s}^T \mathbf{n}_k$; {$\mathbf{n}_k$ must point out of the given convex polyhedron}
            if $\xi <> 0$ then
            begin $\mathbf{s}_k := \mathbf{x}_k - \mathbf{x}_A$;
                  $t := \mathbf{s}_k^T \mathbf{n}_k / \xi$;
                  if $\xi > 0$ then $t_{max} :=$ min $( t , t_{max})$
                         else $t_{min} :=$ max $( t , t_{min})$
            end
            else Special case solution {line is parallel to a facet}
        end;
        {get next facet to $\tau_k$ intersected by plane $\rho_1$}
        {different from previous facet $\tau_i$ }
        $j :=$ get_next $( k , i)$;
        $i := k$;        $k := j$;
    until $k = k_0$;
    {if $t_{min} > t_{max}$ then no intersection point exists}
    if $t_{min} \le t_{max}$ then DRAW_LINE ($\mathbf{x}(t_{min}),\mathbf{x}(t_{max})$);
end { SQRT_CLIP };
```

Algorithm 9.3: $O(\sqrt{N})$ algorithm.

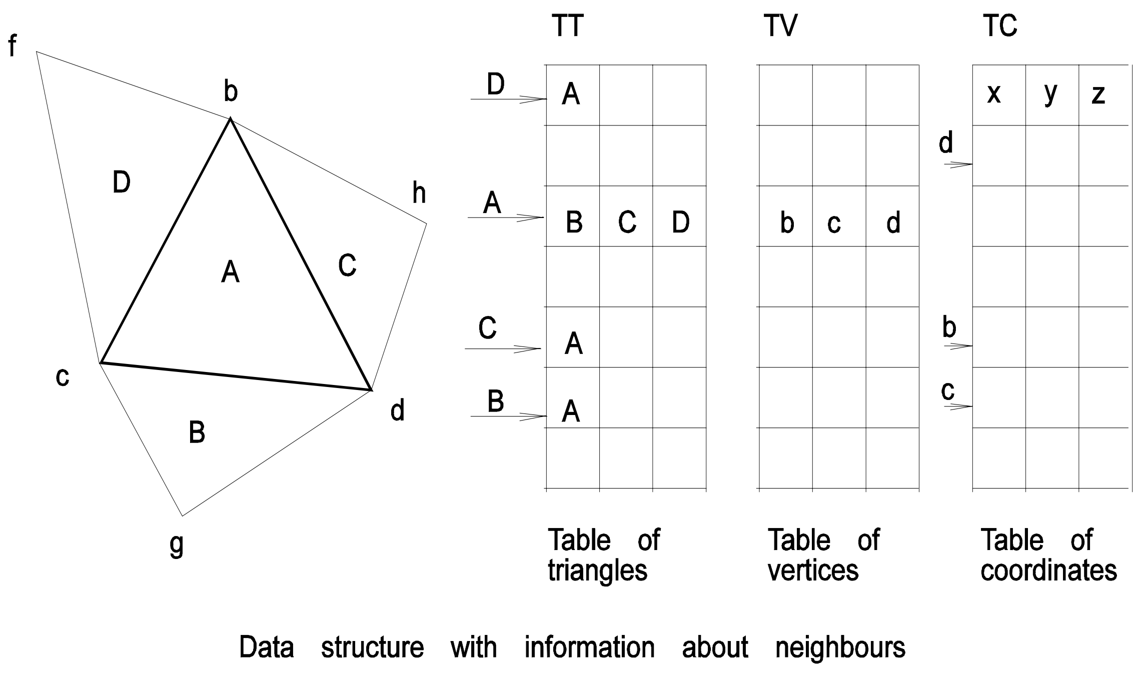


Figure 9.5

9.4. 3D *O(1)* algorithm

Let the given polyhedron $P \in E^3$ is projected to E^2, see Figure 9.6 (only the front facets are shown) and line *p* is the projection of plane ρ_1 (the given line lies on plane ρ_1). This line intersects many facets (triangles), in our case line p intersects the facets 9,11,12,13,14,15,16,17,18,19. The plane ρ_1 described as

$$y = kx + q$$

Let us assume the semidual representation for *(k,q)* values. Then the semidual space can be split into small rectangles using space subdivision technique. Each rectangle in semidual space represents an infinite „butterfly" zone in E^2 space. There are AFL lists (Active Facets List) of facets associated with each „butterfly" zone. The AFL list contains information on all facets that interfere with the zone. The AFL list can be implemented by a list of pointers but such implementation would be quite memory demanding as its length can be estimated as $O(\sqrt{N})$. Therefore, it is more convenient to use binary maps [Ska93b]. This technique is based on a binary vector in which the *i*-th bit is set to „1" if the *i*-th facet is in the AFL list. Using this technique, the memory requirements are small and the intersection operation is implemented as the bitwise operation **and**.

Using this approach we can expect that 4 - 6 facets will be necessary to test in detail if semidual space is subdivided enough. It is necessary to point out that we must prepare both *(k,q)* and *(m,p)* semidual representations for all three planes ρ_i^+ , $i = 1,2,3$, i.e. $AFL_1,...,AFL_j$ list, $j = 1,...,6$. It means that we need six semidual representations altogether. For each clipped line we must select two planes ρ_{i_1} and ρ_{i_2} and appropriate semidual representations, i.e. *(k,q)* or *(m,p)* , for each selected plane. The algorithm can be described by Algorithm 9.4.

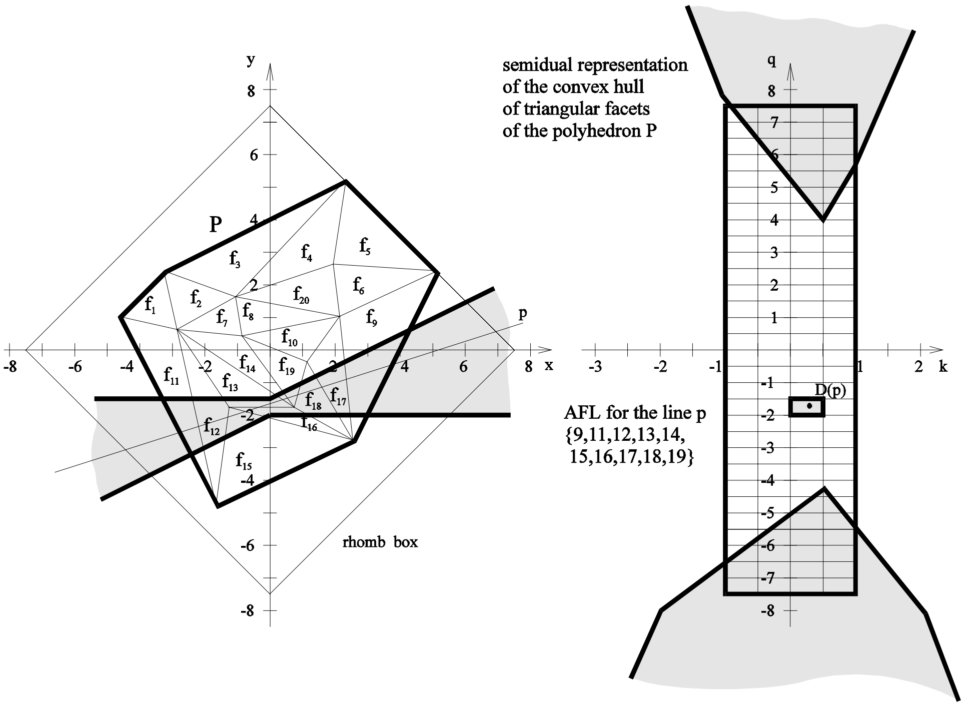


Figure 9.6: Semidual representation of convex polyhedron.

In Algorithm 9.4, function *Detail E^3 Test* is based on the CB algorithm that is performed only for those facets that are included in the final set Ω. It is obvious that the algorithm complexity does not depend on the number of polyhedron facets but on the length of the set Ω. If the rectangles are "small enough" then 4 - 6 facets can be expected in the final set Ω nearly for all cases. Since all steps in Algorithm 9.4 have *O(1)* complexity the whole algorithm has *O(1)* expected complexity, too. It is necessary to point out that number of members in AFL list depends on subdivision in *(k,q)*, *(m,p)* spaces respectively and also on **geometric shape** of the given polyhedron, see Ska96b].

Experimental results proved that the *O(1)* algorithm is faster than CB-3D algorithm when the number of facets is greater than 24 (if at least 1000 lines are processed). For detailed analysis, see [Ska96c].

```
global constants:
                 a - size of bounding rhomboid box for the given polyhedron P,
                 $N_q$ - number of subdivision for q axis in semidual space representation,
                 $N_k$ - number of subdivision for k axis in semidual space representation,
                 for all other spaces assume $N_q = N_p$ , $N_k = N_m$ .
procedure CLIP_3D_O1 ($x_A$, $x_B$);
begin
    Select two planes $\rho_{i_1}$ and $\rho_{i_2}$ , $i_1 \neq i_2$ for the given line $p \in E^3$
    kk := 1;
    for i := $i_1$, $i_2$ do  { plane index $i \in \{1,2,3\}$ }
    begin
        COMPUTE line equation of $p_i$; { projected plane $\rho_1$ $ax + by + c = 0$ }
        { index of the $AFL_j$ list }
        { $AFL_1$ is the (k,q) semidual space for xy plane }
        { $AFL_2$ is the (m,p) semidual space for xy plane, etc. }
        if $|\Delta x| \geq |\Delta y|$ then j := 2*i - 1 else j := 2*i;
        { index zone determination }
        ii := ( q + a ) * $N_q$ / (2*a) + 1 ;      jj := ( k + 1) * $N_k$ / 2 + 1 ;
        $\Omega_{kk}$ := $AFL_j$[ii,jj];      kk := kk + 1;
    end;
    $\Omega := \Omega_1 \cap \Omega_2$;
    for i := 1 to N do { N - number of polyhedron facets }
                    if $\Omega$[i] = 1 then        Detail $E^3$ Test (facet$_i$, p);
end
```

Algorithm 9.4: 3D *O(1)* algorithm.

Nevertheless the algorithm has not "clear" *O(1)* complexity. In spite of high effort this algorithm has not been derived in the Cartesian co-ordinate system. Nevertheless the *O(1)* run-time complexity can be also got by using the space subdivision in spherical co-ordinate system [Ska99b]. In spherical co-ordinate system, the plane Σ that passes the system origin and contains the given line *p* is unambiguously determined by the two angles φ and ϑ, see Figure 9.7. On plane Σ, if (ρ, φ') is polar co-ordinate of system origin's projection on line *p* then (ρ, φ') unambiguously determines line *p*. Therefore, an arbitrary line in E^3 can be represented by four values $(\rho, \varphi', \varphi, \vartheta)$. Now, it is very clear that we can use the infinite space subdivision for $(\rho, \varphi', \varphi, \vartheta)$ to get the *O(1)* run-time complexity. Number of members in AFL list depends not only on subdivision in $(\rho, \varphi', \varphi, \vartheta)$ space but also on **geometric shape** of the given polyhedron.

It can be shown that computation of the AFL for all regions (pre-processing) runs in $O(N* n_\rho* n_{\varphi'}* n_\varphi* n_\vartheta)$ time, where:

- N is the number of edges of the given polygon,
- n_φ, n_ϑ is the number of subdivisions in the direction φ and ϑ, respectively,
- n_ρ, $n_{\varphi'}$ is the number of subdivisions in the direction ρ and φ', respectively.

Using the pre-constructed AFL then we can clip lines or line segments in *O(1)* expected time, see Algorithm 9.5. The COMPUTE function is based on the CB-3D algorithm and is performed only for facets included in the AFL associated with the selected region (*i,j,k,l*).

```
procedure Spher_CLIP_3D_O1 ($\mathbf{x}_A$, $\mathbf{x}_B$);
begin
    $q_\rho := n_\rho / r$;   $q_{\varphi'} := n_{\varphi'} / (2*Pi)$;   $q_\varphi := n_\varphi / Pi$;   $q_\vartheta := n_\vartheta / Pi$;
    $t_0 := +\infty$;                $t_1 := -\infty$;         { initialisation - interval $< t_0 , t_1 > = \emptyset$ }
    Compute $(\varphi, \vartheta)$ co-ordinates of plane $\Sigma$ passing the origin and containing line $p$;
    Compute $(\rho, \varphi')$ polar co-ordinates of line $p$ in plane $\Sigma$;
    $i := \vartheta * q_\vartheta + 1$;  $j := \varphi * q_\varphi + 1$; $k := \varphi' * q_{\varphi'} + 1$;  $l := \rho * q_\rho + 1$;
    {test all facets in the AFL for region(i,j,k,l);compute the appropriate value of t}
    COMPUTE (AFL[i,j,k,l], $t_0$, $t_1$)
    if line segment clipping then $<t_0,t_1> := <t_0,t_1> \cap <0,1>$;
    if $< t_0 , t_1 > \neq \emptyset$ then DRAW_LINE($\mathbf{x}(t_0)$, $\mathbf{x}(t_1)$) {an intersection exists}
end
```

Algorithm 9.5: *O(1)* algorithm using spherical co-ordinate.

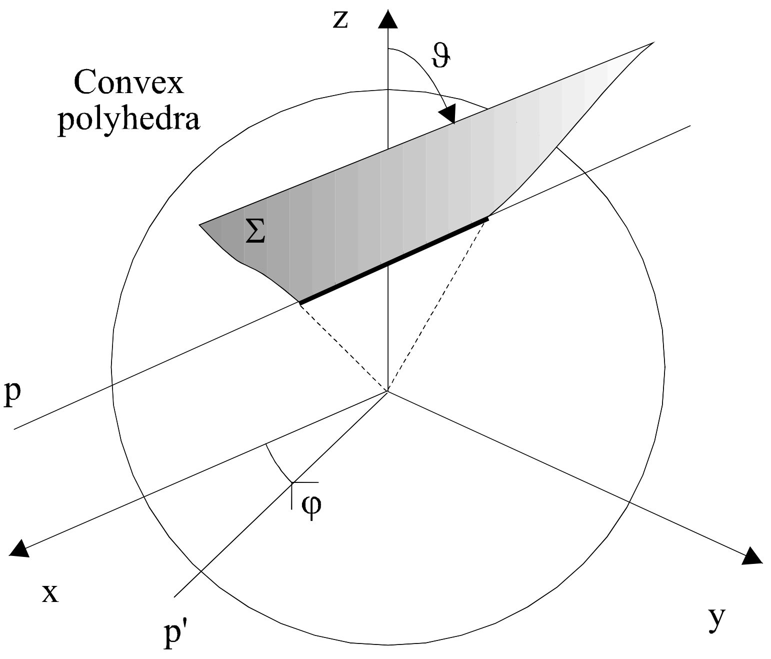


Figure 9.7: (φ, ϑ) representation of plane Σ containing line p.

10. Clipping in homogeneous co-ordinates

There are two reasons to clip in homogeneous co-ordinates. The first has to do with efficiency: It is possible to transform the perspective-projection canonical view volume into the parallel-projection canonical view volume, so a single clipping procedure, optimized for the parallel-projection canonical view volume, can always be used. However, the clipping must be done in homogeneous co-ordinates to ensure correct results. A single clipping procedure is typically provided in hardware implementations of the viewing operation. The second reason is that points that can occur as a result of unusual homogeneous transformations and from use of the rational parametric splines [Fol90a] can have negative *W* co-ordinate and can be clipped properly in homogeneous co-ordinate but not in 3D.

With regard to clipping, it can be shown that the transformation from the perspective-projection canonical view volume to the parallel-projection canonical view volume is:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1 - z_{\min}} & \dfrac{-z_{\min}}{1 - z_{\min}} \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad z_{\min} \neq 1$$

Figure 10.1 shows the results of applying matrix *M* to the perspective-projection canonical view volume. By using matrix *M*, we can clip against the parallel-projection canonical view volume rather than against the perspective-projection canonical view volume.

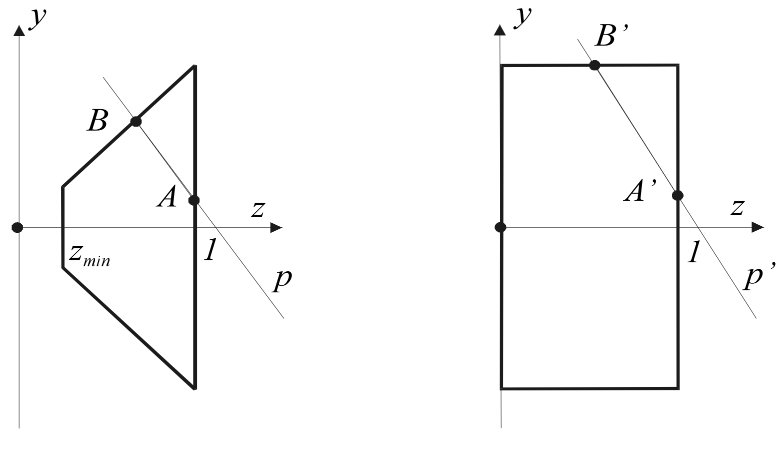


Figure 10.1: Side views of normalized perspective view volume before and after application of matrix *M*.

Therefore, the clipping against the perspective-projection canonical view volume can be represented as multiplication with matrix M^{1}. $Clip_{par}$. M ($Clip_{par}$ represents the clipping against the parallel-projection canonical view volume).

The 3D parallel-projection view volume is defined by $-1 \le x \le 1$, $-1 \le y \le 1$, $0 \le z \le 1$. We find the corresponding inequalities in homogeneous co-ordinate by replacing x by X/W, y by Y/W, and z by Z/W, which results in

$$-1 \le X/W \le 1, \qquad -1 \le Y/W \le 1, \qquad 0 \le Z/W \le 1$$

It can be seen that, we must consider separately the cases of $W > 0$ and $W < 0$. In the first case, we can multiply the inequalities by W without changing the sense of the inequalities. In the second case, the multiplication changes the sense. This result can be expressed as

$$W > 0: -W \le X \le W, \qquad -W \le Y \le W, \qquad 0 \le Z \le W \qquad (10.1)$$

$$W < 0: -W \ge X \ge W, \qquad -W \ge Y \ge W, \qquad 0 \ge Z \ge W \qquad (10.2)$$

In many cases, only the region given by the Equation (10.1) needs to be used, because prior to application of M, all visible points have $W > 0$ (normally $W = 1$). However, it is sometimes desirable to represent points directly in homogeneous co-ordinate with arbitrary W co-ordinate. Hence, we might have a $W < 0$, meaning that the clipping must be done against the regions given by Equations (10.1) and (10.2). Figure 10.2 shows these as region R_1 and region R_2, and also shows why both regions must be used.

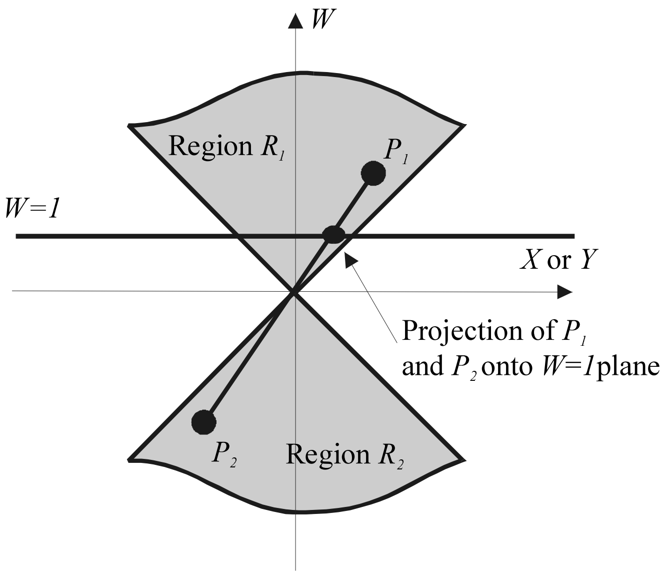


Figure 10.2: Points P_1 and P_2 both map into the same point on the $W = 1$ plane.

Let us assume two points P_1 and P_2 whose homogeneous co-ordinates differ by a constant multiplier. These homogeneous points correspond to the same 3D point (on the $W = 1$ plane of homogeneous space) although P_1 is in region R_1 and P_2 is in region R_2. If clipping were only to region R_1, then point P_2 would be discarded incorrectly.

There are two solutions to problem of points in region R_2. The first is to clip all points twice, once against each region, but doing two clips is expansive. A better solution is to negate points that have negative W, and then to clip them. Similarly, we can clip properly a line whose end-points are both in region R_2 by multiplying both end-points by -1, to place the points in region R_1.

Another problem arises with lines such as P_1P_2, shown in Figure 10.2, whose end-points have opposite values of W. The projection of line onto the $W = 1$ plane is two segments, one of which goes to $+\infty$, the other to $-\infty$. The solution now is to clip twice, once against each region, with the possibility that each clip will return a visible line segment. A simple way to do this is to clip the line against region R_1, to negate both end-points of the line, and to clip against region R_1. This approach preserves one of the original purposes of clipping in homogeneous co-ordinates: using a single clipping region, see [Bli78a] for further discussion.

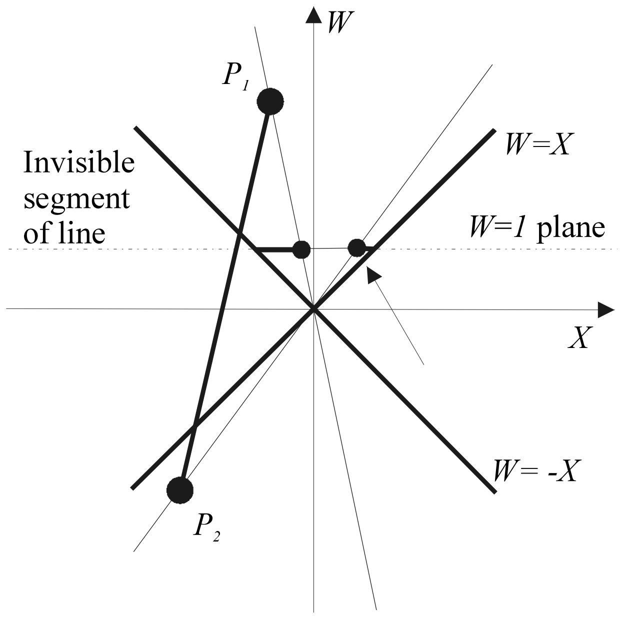


Figure 10.3: Line P_1P_2 projects onto two line segments.

11. Summary

In this thesis we have explored the fundamental algorithms for line clipping problem in E^2 and E^3. Some newer algorithms and approaches to improve the algorithms have been shown, too. Algorithms CS, LB, CB and their 3D extension are most often used and can be found in most textbooks. The others have been developed in the Computer Graphics Laboratory of Department of Computer Science and Engineering at the University of West Bohemia. The author has assisted in their verification and testing. The author is also co-author of LSSB, LSB, modified *O(log N)*, SF, PC and SF-3D algorithms. Algorithms LSSB and LSB are also original and have been published in ***Proceedings of the international conference SCCG'97,*** Slovak republic [Bui97a] and in ***The Visual Computer,*** Springer Verlag [Bui98a]. Modified *O(log N)* algorithm has been published in ***Proceedings of the international conference SCCG'99,*** Slovak republic [Bui99a].

Table 11.1 and Table 11.2 give an overview of mentioned algorithms and their complexity.

2D clipping algorithms	**Complexity**
Cohen-Sutherland algorithm	*O(N)*
LSSB algorithm	*O(N)*
Liang-Barsky algorithm	*O(N)*
LSB algorithm	*O(N)*
SF algorithm	*O(N)*
Cyrus-Beck algorithm	*O(N)*
ECB algorithm	*O(N)*
O(log N) algorithm	*O(log N)*
Modified *O(log N)* algorithm	*O(log N)*
O(1) algorithm	expected *O(1)*
Sutherland-Hodgman algorithm	*O(M*N)*

Table 11.1: Complexity of presented clipping algorithms in E^2.

3D clipping algorithms	**Complexity**
CS-3D algorithm	*O(N)*
PC algorithm	*O(N)*
LB-3D algorithm	*O(N)*
SF-3D algorithm	*O(N)*
CB-3D algorithm	*O(N)*
Algorithm based on two planes use	*O(N)*
$O(\sqrt{N})$algorithm	expected $O(\sqrt{N})$
O(1) algorithm	expected *O(1)*

Table 11.2: Complexity of presented clipping algorithms in E^3.

It is necessary to point out that some selected algorithms has been tested on Pentium II, 350MHz, 64MB RAM, 512KB CACHE and Pentium III, 500MHz, 64MB RAM, 512KB CACHE just at the last moment and the experimental verification proved trend of increasing.

12. Conclusion and Future work[*]

It is well known that the fundamental problem in algorithm design is to use all known data properties as much as possible in order to get an algorithm with better efficiency. If this factor is used in algorithm design it is possible not only to improve algorithms property but sometimes also to change the algorithms complexity. This contention is proved very clearly in this thesis by analyzing algorithms for line clipping. Generally, we can say that if we want to improve an algorithm for the given problem, we need to consider the following issues:

- the trade-off between memory, run-time and pre-processing complexities,
- which kind of pre-processing complexity (time or memory) we can expect if we need faster solution of the given problem,
- the use of pre-processing or parallel and distributed.

The presented algorithms proved that applying these approaches can bring a significant speed-up even with the known algorithms. We have concentrated in detail on the problem of line clipping in E^2 and E^3. It is clear that the mentioned approaches can be applied not only in the clipping problem solution but also in many other areas of computer graphics. The problem is how we can apply them. We would like to draw attention to this problem in the next period. We also intend to integrate our algorithms to the MESA, which is a graphics library and can be used in many computer graphics applications.

[*] Some figures in this thesis were taken up from the used sources

13. References

Some papers are available in on-line version from http://herakles.zcu.cz/publication.htm

[Aho74a] Aho,A.V., Hopcroft,J.E., Ullman,J.D.: *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[And89a] Andreev,R.: Algorithm for Clipping Arbitrary Polygons, *Computer Graphics Forum*, Vol.8, No.3, pp.183-192, 1989.

[And91a] Andreev,R., Sofianska,E.: New Algorithm for 2-Dimensional Line Clipping, *Computers & Graphics*, Vol.15, No.4, pp.519-526, 1991.

[Aro89a] Arokiasamy,A.: Homogeneous Coordinates and The Principle of Duality in Two Dimensional Clipping, *Computers & Graphics*, Vol.13, No.1, pp.99-100, 1989.

[Bal94a] Balaban,I.J.: An Optimal Algorithm for Finding Segments Intersections, *MGV*, Vol.3, Nos.1/2, pp.403-410,1994.

[Bal94b] Ballo,M.: A New Approach to Non Self-intersecting Polygon Clipping, *MGV*, Vol.3, Nos.1/2, pp.111-122,1994.

[Ben79a] Bentley,J.L.: An Introduction to Algorithm Design, *Computer*, pp.66-87, 1979.

[Bli78a] Blinn, J.F., Newell,M.E.: Clipping Using Homogeneous Coordinates, *ACM Computer Graphics*, Vol.12, No.3, pp. 245-251, 1978.

[Bui97a] Bui,D.H., Skala,V.: Fast Modifications of Cohen-Sutherland algorithm for Line Segment and Line Clipping in E^2, *SCCG'97 International Conference proceedings*, Bratislava-Budmerice, pp.205-212, 1997.

[Bui98a] Bui,D.H., Skala,V.: Fast Algorithms for Clipping Lines and Line Segments in E^2, *The Visual Computer*, Vol.14, No.1, pp.31-37, 1998.

[Bui98b] Bui,D.H., Skala,V.: Line Clipping Algorithms in E^2, *SCG'98 Proceedings of Seminars on Computational Geometry*, Kočovce, pp.52-57, 1998.

[Bui99a] Bui,D.H., Skala,V.: New Fast Line Clipping Algorithm in E^2 with O(lgN) Complexity, *SCCG'99 International Conference proceedings*, Bratislava-Budmerice, pp.212-219, 1999.

[Bui99b] Bui,D.H., Skala,V.: Two New Algorithms for Line Clipping in E^2 and Their Comparison, *TR 108/99*, University of West Bohemia, Plzeň, 1999.

[Bui99c] Bui,D.H., Skala,V.: A New Algorithm for Pyramidal Clipping of Line Segments in E^3, *TR 109/99*, University of West Bohemia, Plzeň, 1999.

[Bui99d] Bui,D.H., Skala,V.: New Algorithm for Line Clipping against a Pyramid in E^3, *TR 110/99*, University of West Bohemia, Plzeň, 1999.

[Bur88a] Burkert,A., Noll,S.: Fast Algorithm for Polygon Clipping with 3D Windows, *Eurographics '88 International Conference proceedings*, Nice, pp.405-419, 1988.

[Che88a] Cheng,F., Yen,Y.KA Parallel Line Clipping Algorithm and its Implementation, *Parallel Processing for Computer Vision and Display International Conference proceedings*, 1988.

[Cyr78a] Cyrus,M., Beck,J.: Generalized Two and Three Dimensional Clipping, *Computers & Graphics*, Vol.3, No.1, pp.23-28,1978.

[Day92a] Day,J.D.: An Algorithm for Clipping Lines in Object and Image Space, *Computers & Graphics*, Vol.16, No.4, pp.421-426,1992.

[Day96a] Day,J.D.: Image Space Algorithms for Line Clipping, *WSCG'96 International Conference proceedings*, Plzeň, pp.47-54, 1996.

[Don94a] Donovan,W., Hook,T.V.: Direct Outcode Calculation for Faster Clip Testing, *Graphics Gems IV*, pp.125-131, 1994.

[Dor90a] Dorr,M.: A New Approach to Parametric Line Clipping, *Computers & Graphics*, Vol.14, No.3, pp.449-464, 1990.

[Duv93a] Duvanenko,V.J., Robbins,W.E., Gyurcsik,R.S.: Simple and efficient 2D and 3D Span Clipping Algorithms, *Computers & Graphics*, Vol.17, No.1, pp.39-54, 1993.

[Duv96a] Duvanenko,V.J., Robbins,W.E., Gyurcsik,R.S.: Line Segment Clipping Revisited, *Dr Dobbs Journal*, Vol.21, No.1, pp.107-112, 1996.

[Fel83a] Feliziani,S., Franchina,V.: An Algorithm for Rectangular Clipping, *Pixel. Computer Graphics, CAD/CAM, Image Process.*, Vol.4, pp.37-40, 1983.

[Fol90a] Foley,D.J., van Dam,A., Feiner,S.K., Hughes,J.F.: *Computer Graphics Principles and Practice*, Addison Wesley, 2nd ed., 1990.

[Fun90a] Fung,K.Y., Nicholl,T.M., Tarjan,R.E., Van Wyk,C.J.: Simplified Linear Time Jordan Sorting and Polygon Clipping, *Information Processing Letters*, Vol.35, pp.85-92, 1990.

[Gre98a] Greiner,G., Hormann,K.: Efficient Clipping of Arbitrary Polygons, *ACM Transaction on Graphics*, Vol.17, No.2, pp.71-83, 1998.

[Gut78a] Guttmann,H., Weiss,J.: Clipping in the View of Decentralization of Computer Graphics, *Interactive Techniques in Computer Aided Design International Conference proceedings*, Bologna, pp.235-240, 1978.

[Han84b] Hanrahan,P.: A Note Concerning Polygon and Line Clipping, *Technical Report*, No.8, NYIT Computer Graphics Lab, 1984.

[Her88a] Herman,I., Reviczky,J.: Some Remarks on the Modelling Clip Problem, *Computer Graphics Forum*, Vol.7, No.4, pp.265-272, 1988.

[Hua96a] Hua,S., Tokuta,A.: Generalized Clipping of a Polygon Against a 2D Arbitrary Window and 3D Non-Convex Volume, *WSCG'96 International Conference proceedings*, Plzeň, pp.257-266, 1996.

[Hub90a] Hubl,J., Herman,I.: Modelling Clip: Some More Results, *Computer Graphics Forum*, Vol.9, No.2, pp.101-107, 1990.

[Hub93a] Hubl,J.: A Note on 3D-Clip Optimisation, *Computer Graphics Forum*, Vol.12, No.2, pp.159-160, 1993.

[Kai90a] Kaijian,S., Edwards,J.A., Cooper,D.C.: An Efficient Line Clipping Algorithm, *Computers & Graphics*, Vol.14, No.2, pp.297-301,1990.

[Kil87a] Kilgour,A.: Unifying Vector and Polygon Algorithms for Scan Conversion and Clipping, *Eurographics '87 International Conference proceedings*, Amsterdam, pp.363-375, 1987.

[Kol94a] Kolingerová,I.: Dual Representation and Its Usage in Computer Graphics, *PhD Thesis* (in Czech), University of West Bohemia, Plzeň, 1994.

[Kol97a] Kolingerová,I.: Convex polyhedron-line intersection detection using dual representation *The Visual Computer*, Vol.13, No.1, pp.42-49, 1997.

[Kra92a] Krammer,G.: A Line Clipping Algorithm and Its Analysis, *Computer Graphics Forum*, Vol.11, No.3, pp.253-266, 1992.

[Kuz95a] Kuzmin,Y.P.: Bresenham's Line Generation Algorithm with Built-in Clipping, *Computer Graphics Forum*, Vol.14, No.5, pp.275-280, 1995.

[Lew78a] Lewis,H.R., Papadimitriou,C.H.: The Efficiency of Algorithms, *Scientific American*, pp.96-108, 1978.

[Lia83a] Liang,Y.D., Barsky,B.A.: An Analysis and Algorithms for Polygon Clipping, *CACM*, Vol.26, No.11, pp.868-876, 1983.

[Lia84a] Liang,Y.D., Barsky,B.A.: A New Concept and Method for Line Clipping, *ACM Transaction on Graphics*, Vol.3, No.1, pp.1-22, 1984.

[Lia92a] Liang,Y.D., Barsky,B.A.: The Optimal Tree Algorithm for Line Clipping, *Technical paper distributed at Eurographics'92 Conference*, Cambridge, 1992.

[Lia92b] Liang,Y.D., Barsky,B.A., Slater,M.: Some Improvements to a Parametric Line Clipping Algorithm, *Technical Report*, No.92/688, University of California at Berkeley, 1992.

[Mai92a] Maillot,P.G.: A New, Fast Method For 2D Polygon Clipping: Analysis and Software Implementation, *ACM Transaction on Graphics*, Vol.11, No.3, pp.276-290, 1992.

[Mat85a] Mathew,A.J.: Polygonal Clipping of Polylines, *Computer Graphics Forum*, Vol.4, No.4, pp.407-414, 1985.

[Mer84a] Meriaux,M.: A Two-Dimensional Clipping Divider, *Eurographics '84 International Conference proceedings*, Copenhagen, pp.389-395, 1984.

[Nar96a] Narayanaswami,C.: A parallel polygon- clipping algorithm, *The Visual Computer*, Vol.12, No.3, pp.147-158, 1996.

[Nic87a] Nicholl,T.M., Lee,D.T., Nicholl,R.A.: An Efficient New Algorithm for 2D Line Clipping: Its Development and Analysis, *ACM Computer Graphics*, Vol.21, No.4, pp.253-262, 1987.

[Nic91a] Nicholl,R.A., Nicholl,T.M.: A Definition of Polygon Clipping, *Technical Report,* No.281, Computer Sci. Dept.,University of West Ontario, 1991.

[Nie95a] Nielsen,H.P.: Line Clipping Using Semi-Homogeneous Coordinates, *Computer Graphics Forum*, Vol.14, No.1, pp.3-16, 1995.

[Pre85a] Preparata,F.P., Shamos,M.I.: *Computational Geometry: An Introduction*, Springer Verlag, New York, 1985.

[Rap91a] Rappaport,A.: An Efficient Algorithm for Line and Polygon Clipping, *The Visual Computer*, Vol.7, No.1, pp.19-28, 1991.

[Rog85a] Rogers,D.F., Rybak,L.M.: A Note on An Efficient General Line Clipping Algorithm, *IEEE Computer Graphics & Applications*, Vol.5, No.1, pp.82-86, 1985.

[Sch98a] Schneider,B.O., Welzen,J.V.: Efficient Polygon Clipping for an {SIMD} Graphics Pipeline, *IEEE Transactions on Visualization and Computer Graphics*, Vol.4, No.3, 1998.

[Sha92a] Sharma,N.C., Manohar,S.: Line Clipping Revisited: Two Efficient Algorithms Based on Simple Geometric Observations, *Computers & Graphics*, Vol.16, No.1, pp.51-54, 1992.

[Ska89a] Skala,V.: Algorithms for 2D Line Clipping, *CGI'89 Conference Proceedings*, pp.121-128, 1989.

[Ska89b] Skala,V.: Algorithms for 2D Line Clipping, *EG'89 Conference Proceedings*, pp.355-367, 1989.

[Ska93a] Skala,V.: Algorithm for Line Clipping in E^2 for Convex Window (in Czech), *Algorithms'93 Conference Proceedings*, Bratislava, 1993.

[Ska93b] Skala,V.: An Efficient Algorithm for Line Clipping by Convex Polygon, *Computers & Graphics*, Vol.17, No.4, pp.417-421, 1993.

[Ska93c] Skala,V.: Memory Saving Technique for Space Subdivision Technique, *Machine Graphics and Vision,* Vol.2, No.3, pp.237-250, 1993.

[Ska94a] Skala,V.: O(lg N) Line Clipping Algorithm in E^2, *Computers & Graphics*, Vol.18, No.4, pp.517-524, 1994.

[Ska94b] Skala,V.: Point-in-Polygon with O(1) Complexity, *TR 68/94*, University of West Bohemia, Plzeň, 1994.

[Ska95a] Skala,V., Kolingerová,I., Bláha,P.: A Comparison of 2D Line Clipping Algorithms, *Machine Graphics and Vision*, Vol.3, No.4, pp. 625-633, 1995.

[Ska96a] Skala,V.: An Efficient Algorithm for Line Clipping by Convex and Non-convex Polyhedrons in E^3, *Computer Graphics Forum*, Vol.15, No.1, pp.61-68, 1996.

[Ska96b] Skala,V.: Line Clipping in E^2 with Suboptimal Complexity O(1), *Computers & Graphics*, Vol.20, No.4., Pergamon Press, pp.523-530, 1996.

[Ska96c] Skala,V., Lederbuch,P., Sup,B.: A Comparison of O(1) and Cyrus-Beck Line Clipping Algorithms in E^2 and E^3, *SCCG'96 International Conference proceedings*, June 5-7, Bratislava-Budmerice, pp.27-44, 1996.

[Ska96d] Skala,V., Lederbuch,P.: A Comparison of a New O(1) and the Cyrus-Beck Line Clipping Algorithms in E^2, *COMPUGRAPHICS'96 International Conference proceedings*, Paris, pp.281-287, 1996.

[Ska97a] Skala,V.: A Fast Algorithm for Line Clipping by Convex Polyhedron in E^3, *Computers & Graphics*, Vol.21, No.2, pp.209-214, 1997.

[Ska97b] Skala,V.: Algorithms Complexity and Line Clipping Problem Solutions, **invited talk**, *Proceedings of EDU+COMPUGRAPHICS'97*, Algarve, Portugal, pp.30-34, 1997.

[Ska97c] Skala,V.: Line Clipping Algorithm Complexity in E^2 and E^3 , **invited talk**, *SCCG'97 International Conference proceedings*, Bratislava-Budmerice, 1997.

[Ska99a] Skala,V.: Non-Orthogonal Co-ordinates in Computer Graphics, *GRAPHICON'99 International Conference proceedings*, Moscow, Russia, pp.45-50, 1999.

[Ska99b] Skala,V.: Non-linear Co-ordinate Systems, *Syllabuses of the Talk at the University of Loannina,* Greece, May 1999.

[Ska99c] Skala,V.: Algorithm Complexity in Computer Graphics, *Talk at the University of Girona,* Spain, June 1999.

[Sla94a] Slater,M, Barsky,B.A..: 2D Line and Polygon Clipping Based on Space Subdivision, *The Visual Computer*, Vol.10, No.7, pp.407-422, 1994.

[Sob87a] Sobkow,M.S., Pospisil,P., Yang,Y.-H.: A Fast Two-dimensional Line Clipping Algorithm via Line Encoding, *Computers & Graphics*, Vol.11, No.4, pp.459-467, 1987.

[Soj96a] Sojka,E.: Two Simple and Efficient Algorithms for Jordan Sorting and Polygon Cutting and Clipping, *COMPUGRAPHICS'96 International Conference proceedings*, Paris, pp.241-252, 1996.

[Sto89a] Stolfi,J.: Primitives for Computational Geometry, *Report 36*, SRC DEC System Research Center, 1989.

[Sut74a] Sutherland,I.E., Hodgman,G.W.: Reentrant Polygon Clipping, *Communications of the ACM*, 1974.

[Tou85a] Toussaint, G.T.: A Simple Linear Algorithm for Intersecting Convex Polygons, *The Visual Computer*, Vol.1, pp.118-123, 1998.

[The89a] Theoharis,T., Page,I.: Two Parallel Methods for Polygon Clipping, *Computer Graphics Forum*, Vol.8, No.2, pp.107-114, 1989.

[Vat92a] Vatti, B.R., A Generic Solution to Polygon Clipping, *Communications of the ACM*, Vol.35, No.7, pp.56-63, 1992.

[Zal98a] Zalik,B., Gombosi,M., Podgorelec,D.,: A Quick Intersection Algortihm for Arbitrary Polygons, *SCCG'98 International Conference proceedings*, April 23-25, Bratislava-Budmerice, pp.195-204, 1998.

[Zac95a] Zachariáš,S.: Duality and Complexity (in Czech), TR 81/95, University of West Bohemia, Plzeň, 1995.

[Zac96a] Zachariáš,S.: Projection in Barycentric Coordinates, *WSCG'96 International Conference proceedings*, Plzeň, pp.409-418, 1996.

[Zac89a] Zachrisen,M.: Yet another Remark on the Modelling Clip Problem, *Computer Graphics Forum*, Vol.8, No.3, pp.237-238, 1989.

[Zwa95a] Zwaan,M., Reinhard,E., Jansen,F.: Pyramid Clipping for Efficient Ray Traversal, *Eurographics Rendering Workshop*, 1995.

14. Author's publications & research work

14.1. Related publications to this thesis

[Bui97a] Bui,D.H., Skala,V.: Fast Modifications of Cohen-Sutherland algorithm for Line Segment and Line Clipping in E^2, *SCCG'97 International Conference proceedings*, Bratislava-Budmerice, pp.205-212, 1997.

[Bui98a] Bui,D.H., Skala,V.: Fast Algorithms for Clipping Lines and Line Segments in E^2, *The Visual Computer*, Vol.14, No.1, pp.31-37, 1998.

[Bui98b] Bui,D.H., Skala,V.: Line Clipping Algorithms in E^2, *SCG'98 Proceedings of Seminars on Computational Geometry*, Kočovce, pp.52-57, 1998.

[Bui99a] Bui,D.H., Skala,V.: New Fast Line Clipping Algorithm in E^2 with O(lgN) Complexity, *SCCG'99 International Conference proceedings*, Bratislava-Budmerice, pp.212-219, 1999.

[Bui99b] Bui,D.H., Skala,V.: Two New Algorithms for Line Clipping in E^2 and Their Comparison, *TR 108/99*, University of West Bohemia, Plzeň, 1999, submitted to ***GPKO'2000,*** *International Conference*, Kroczyce, Poland, 2000.

[Bui99c] Bui,D.H., Skala,V.: A New Algorithm for Pyramidal Clipping of Line Segments in E^3, *TR 109/99*, University of West Bohemia, Plzeň, 1999, submitted to ***The Visual Computer, Springer Verlag***.

[Bui99d] Bui,D.H., Skala,V.: New Algorithm for Line Clipping against a Pyramid in E^3, *TR 110/99*, University of West Bohemia, Plzeň, 1999, submitted to ***Computers & Graphics, Pergamon Press***.

[Bui99e] Bui,D.H., Skala,V.: Algorithms for Line Clipping and Their Complexity, *I&IT'99 International Conference proceedings*, Banska Bystrica, Slovakia, 1999.

[Bui99f] Bui,D.H., Skala,V.: Algorithms Complexity and Its Decrement, *CE&I'99 International Conference proceedings*, Kosice, Slovakia, 1999.

In all publications listed above the author's contribution is about 50%.

14.2. Research work - VolVis version for Windows NT

VolVis is a **Volume Visualization System** that unites numerous visualization methods within a comprehensive visualization system, providing a flexible tool for the scientist and engineer as well as the visualization developer and researcher.

The underlying principle behind VolVis is to provide as diverse an array of algorithms and capabilities as possible. This makes the VolVis system useful within any field, which has the need to manipulate, render and measure volumetric data including medicine, biology, geology, and physics.

VolVis is developed primary by the research group headed by Prof. Arie Kaufman at the State University of New York at Stony Brook for the **UNIX** system.

More information about VolVis can be found at the VolVis homepage that is at the URL: http://www.cs.sunysb.edu/~vislab/start_volvis.html.

In this project the kernel of system is analyzed and partly converted to the **MS Windows NT** by our research group headed by Prof. Vaclav Skala. The aim of this project is to obtain the information how this type of programming packages can be implemented and verify the function of kernel of the system. The first version of system has been devolved to Prof. Arie Kaufman. Prof. Arie Kaufman and his research group will improve this version further and the whole system can be received from him.