

A New Coding Scheme for Line Segment Clipping in E2

Vaclav Skala(✉)

Department of Computer Science and Engineering, University of West Bohemia,
301 00 Pilsen, Czech Republic
skala@kiv.zcu.cz
http://www.VaclavSkala.eu

Abstract. This contribution presents a new coding scheme based on Cohen-Sutherland line segment clipping algorithm, which enables to distinguish all possible cases easily. It leads to more efficient algorithm for a line segment clipping in E2. It also presents importance of a detailed analysis in algorithm development, if the algorithm robustness and efficiency is required.

1 Introduction

Many algorithms for a line clipping or a line segment clipping by a rectangular window have been published already. Probably the Cohen-Sutherand's (C-S) [7], Cyrus-Beck (CB) [3] and Liang-Barsky (LB) [11] algorithms are the most known for line segment and line segment clipping in E^2 and used in computer graphics courses. The CS algorithm uses end-point position coding to detect some cases, which leads to more efficient computation. However, in some cases 4 intersections of the line and the clipping window are computer; two of those are actually unnecessary. The CB algorithm was actually designed for a line clipping by a convex polygon. Several improvements of the C-B algorithm were published, e.g. Nicholl-Lee-Nicholl [12], Bui [2], Skala [28].

The line and line segment clipping are fundamental and critical operations in the computer graphics pipeline as all the processed primitives have to be clipped out of the drawing area to decrease computational requirements and also respect the physical restrictions of the hardware. The clipping operations are mostly connected with the Window-Viewport and projection operations. There are many algorithms developed recently with many modifications, see Andreev [1], Day [4], Dörr [6], Duvalenko [5], Kaijian [9], Krammer [10], Liang [11], Skala [27], Sobkow [29].

However, those algorithms have been developed for the Euclidean space representation in spite of the fact, that geometric transformations, i.e. projection, translation, rotation, scaling and Window-Viewport etc., use homogeneous coordinates, e.g. projective representation. This results into necessity to convert the

Supported by the University of West Bohemia - Institutional research support No. 1311.

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12953, pp. 16–29, 2021.
https://doi.org/10.1007/978-3-030-86976-2_2

results of the geometric transformations to the Euclidean space using division operation as follows:

$$\mathbf{X} = (X, Y) \quad \mathbf{x} = [x, y : w]^T \qquad X = \frac{x}{w} \quad Y = \frac{y}{w} \quad w \neq 0 \tag{1}$$

where X, Y are the point coordinates in the Euclidean space E^2, while $x, y : w$ are in the homogeneous coordinates [23,24]; similarly in the E^3 case.

If a point is given in the Euclidean space the homogeneous coordinates are given as $\mathbf{x} = [X, Y : 1]^T$. The homogeneous coordinates also enable to represent a point close or in infinity, i.e. when $w \to 0$, and postpone the division operations. It leads to better numerical robustness and computational speed-up, in general.

2 Line Segment Clipping

The line segment clipping operation in the E^2 and E^3 space is a fundamental problem in Computer Graphics and it has been already deeply analyzed. The line and line segment clipping algorithms against a rectangular window in E^2 are probably the most used algorithms and any improvements or speed up can have a significant influence on efficiency of the whole graphics pipeline. Many algorithms have been developed, e.g. the Cohen-Sutherland (CS) [7] for a line segment clipping against the rectangular window, the Liang-Barsky (LB) [11] and Cyrus-Beck (CB) [3] (extensible to the E^3 case) algorithms for clipping a line against a convex polygon and the Nichol-Lee-Nichol (LNL) [12] (modified by Skala [28]) are the most used algorithms.

However, some more sophisticated algorithms or modification of the recent ones have been developed recently, e.g. line clipping against a rectangular window, see Bui [2], Skala [15,17], line clipping by a convex polygon with $O(lgN)$ complexity, Skala [18] (based on Rappaport [14]) using ordering of the vertex indices, or algorithm with $O_{exp}(1)$ complexity using pre-processing Skala [20], line clipping by a window with quadratics arcs Skala [16], etc.

In the E^3 case, the algorithms have computational complexity $O(N)$ as there is no ordering of the vertices ordering in the E^3 case, however, the algorithm with $O_{expected}(\sqrt{N})$ have been developed by Skala [19–21].

In the following, the Cohen-Sutherland (CS) and the S-Clip algorithms based on implicit formulation using projective representation will be presented. The CS algorithm is based on end-points classification, while the S-Clip is primarily based on the classification of the window corner against the given line.

2.1 Cohen-Sutherland Algorithm

The Cohen-Sutherland (CS) algorithm for line segment clipping is a fundamental algorithm presented in computer graphics courses. It splits the 2D space into 9 areas defined by the rectangular clipping window, see Fig. 1. The line segment given by its end-points $\mathbf{x}_A$, $\mathbf{x}_B$ is classified and if not directly accepted or rejected, intersections with lines in which the window edges lie are computed

and the intersection point found is then classified end the process is repeated, see Huges et al, see [8] for details. It is an analogy of the root bisection method in numerical mathematics.

2.2 End-Points Coding

A line segment is defined by its end-points $\mathbf{x}_A$ and $\mathbf{x}_B$. The end-point position classification was used in the CS algorithms developed by Cohen-Sutherland [7]. Some additional coding for speedup were introduced in Bui [2], originally for the Euclidean space representation. The classification of the line segment end-points and the corners of the window mutual positions enables faster processing. It enables simple rejection of line segments not intersecting the window and direct acceptance of segments totally inside of the window, see Fig. 1.

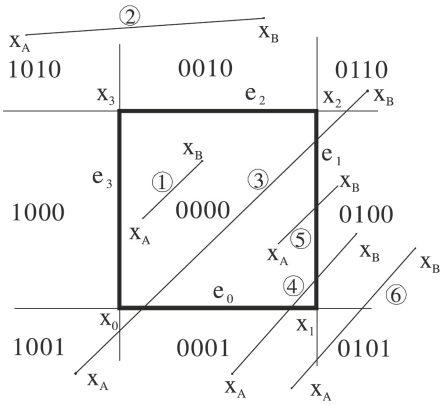

Fig. 1. The codes of line segment end-points

The end-points classification introduced in CS is represented by the Algorithm 1. If $\mathbf{c}_A$ and $\mathbf{c}_B$ are the codes of the end-points then the sequence catching those cases can be expressed as:

- **if** $(\mathbf{c}_A$ **lor** $\mathbf{c}_B) = [0000]$ **then** the line segment is totally inside
- **if** $(\mathbf{c}_A$ **land** $\mathbf{c}_B) \neq [0000]$ **then** the line segment is totally outside

If the end-points of a line are given in the Euclidean space, i.e. $w = 1$, then the codes of the end-points are determined as in the Algorithm 1. However, in the general case, i.e. when $w \neq 1$ and $w > 1$, the conditions must be modified using multiplication, e.g. to $x\ w_{min} < x_{min}\ w$, etc., and then no division operations are needed.

Algorithm 1. End-point code computation

```
1: procedure CODE (c, x);          ▷ code c for the position x = [x, y : 1]^T
2:     c := [0000]^T;                                        ▷ initial setting
3:     if x < x_min then c := [1000]^T                       ▷ according to x coordinate
4:         if x > x_max then c := [0100]^T;
5:     if y < y_min then c := c lor [0001]^T                 ▷ according to y coordinate
6:         if y > y_max then c := c lor [0010]^T;
7:                          ▷ lor is all bits or - instead of algebraic + operation
8: end procedure
```

It can be seen, that other cases, see Fig. 1, cannot be directly distinguished by the CS algorithm coding, see the cases 4 and 6, and intersection points have to be computed, including the invalid ones, e.g. in the case 3 probably 4 intersections will be computed and 2 of those will be invalid. It is necessary to note, that the CS algorithm uses division operations in the floating point, which is the most time consuming operation.

2.3 S-Clip

Let us consider a typical example of a line clipping by the rectangular clipping window, see Fig. 2, and a line p given in the implicit form using projective notation:

$$p : ax + by + cw = 0, \quad i.e. \quad \mathbf{a}^T\mathbf{x} = 0 \tag{2}$$

where $\mathbf{a} = [a, b : c]^T$ are coefficients of the given line p, $\mathbf{x} = [x, y : w]^T$ is a point on this line using projective notation (w is the homogeneous coordinate). Advantage of the projective notation is, that a line p passing two points $\mathbf{x}_A$, $\mathbf{x}_B$ or an intersection point $\mathbf{x}$ of two lines p_1, p_2 can be computed as:

$$\mathbf{p} = \mathbf{x}_A \wedge \mathbf{x}_B, \quad \mathbf{x} = \mathbf{p_1} \wedge \mathbf{p_2} \tag{3}$$

where $\mathbf{a} \wedge \mathbf{b}$ is the outer product application on the vectors $\mathbf{a}$, $\mathbf{b}$ (actually the cross-product is used in this case, i.e. $\mathbf{a} \times \mathbf{b}$), see Skala [25].

The S-Clip algorithm [22] and its optimization for the normalized clipping window [26] use the window corners classification. This enables to determine which window edges are intersected by a line directly without additional intersection computation.

Let us consider an implicit function $F(\mathbf{x}) = \mathbf{a}^T\mathbf{x}$ representing a line, Eq. 2. The clipping operation should determine intersection points $\mathbf{x}_i = [x_i, y_i : w_i]^T$, $i = 1, 2$ of the given line p with the window edges, if any. The line splits the plane into two parts, see Fig. 2. The corners of the window are split into two groups according to the sign of the function $F(\mathbf{x})$ value. This results into Smart-Line-Clip (S-L-Clip) algorithm [22], see Algorithm 2.

The S-Clip line segment algorithm [22] is a slight modification of the S-L-Clip algorithm, which respect position of the line segment end-points and uses the end-point positions classification.

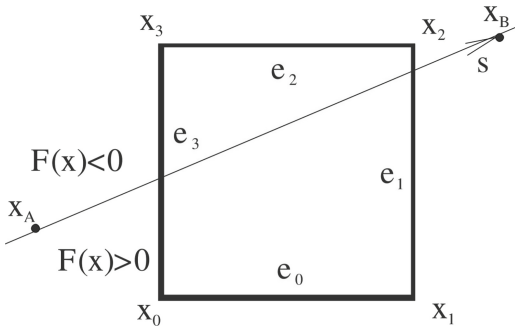

Fig. 2. Clipping against the rectangular window in E^2

Table 1. All cases; N/A - Non-Applicable (impossible) cases the MASK column is used for line segment clipping Skala [22]

c	**c**	TAB1	TAB2	MASK
0	0000	None	None	None
1	0001	0	3	0100
2	0010	0	1	0100
3	0011	1	3	0010
4	0100	1	2	0010
5	0101	N/A	N/A	N/A
6	0110	0	2	0100
7	0111	2	3	1000

c	**c**	TAB1	TAB2	MASK
15	1111	None	None	None
14	1110	3	0	None
13	1101	1	01	0100
12	1100	3	1	0010
11	1011	2	1	0010
10	1010	N/A	N/A	N/A
9	1001	2	0	0100
8	1000	3	2	1000

It means that the ith corner is classified by a bit value c_i as:

$$c_i = \begin{cases} 1 & if\ F(\mathbf{x}_i) \geq 0 \\ 0 & otherwise \end{cases} \quad i = 0, ..., 3 \tag{4}$$

The Table 1 presents the codes for all the situations (some of those are not possible). The columns **TAB1** and **TAB2** contain indices of edges of the window intersected by the given line (values in the **MASK** column are used in the S-Clip algorithm [22] for the line segment end-points).

It can be seen, that the S-L-Clip algorithm (see Algorithm 2) is quite simple and easily extensible for a line and line segment clipping by a convex polygon with $O(N)$ complexity as the Table 1 can be generated synthetically [22]. It is significantly simpler than the Liang-Barsky algorithm [11].

In the following, a new line segment clipping algorithm based on the Cohen-Sutherland and S-Clip algorithms.

3 Proposed Algorithm

The proposed algorithm is based on full classification of all possible cases based on the Cohen-Sutherland coding scheme. However, if the codes of the end-points

Algorithm 2. S-L-Clip - Line clipping algorithm by the rectangular window

```
1: procedure S-L-Clip(x_A, x_B);                    ▷ line is given by two points
2:     p := x_A ∧ x_B;                              ▷ computation of the line coefficients
3:     for i := 0 to 3 do                           ▷ can be done in parallel
4:         if p^T x_i ≥ 0 then c_i := 1 else c_i := 0;   ▷ codes computation
5:     end for
6:     if c ≠ [0000]^T and c ≠ [1111]^T then        ▷ line intersects the window
7:         i := TAB1[c];   x_A := p ∧ e_i;          ▷ first intersection point
8:         j := TAB2[c];   x_B := p ∧ e_j;          ▷ second intersection point
9:         output(x_A, x_B)   ▷ operator ∧ means the cross-product application
10:    else
11:        NOP                                      ▷ line does not intersect the window
12:    end if
13: end procedure
```

C_A and C_B are taken as integer numbers, see Table 2, and summed as $C_{AB} = C_A + C_B$, then the code C_{AB} gives us a composed code differentiating all the different cases Table 3 and the table is symmetrical. As the line segment $\mathbf{x}_A\,\mathbf{x}_B$ is actually oriented its orientation has to be respected in intersection computations.

Table 2. Numerical summation codes $C_{AB} = C_A + C_B$, IN - inside area, C - corner area, S - side area, n/a - non-applicable cases or outside case

			IN	C	S	C	S	C	S	C	S
	C_{AB}	C_B	0	5	4	6	2	10	8	9	1
	C_A		0000	0101	0100	0110	0010	1010	1000	1001	0001
IN	0	0000	IN	5	4	6	2	10	8	9	1
C	5	0101	5	n/a	n/a	n/a	7	15	13	n/a	n/a
S	4	0100	4	n/a	n/a	n/a	6	14	12	13	5
C	6	0110	6	n/a	n/a	n/a	n/a	n/a	14	15	7
S	2	0010	2	7	6	n/a	n/a	n/a	10	11	3
C	10	1010	10	15	14	n/a	n/a	n/a	n/a	n/a	11
S	8	1000	8	13	12	14	10	n/a	n/a	n/a	9
C	9	1001	9	n/a	13	15	11	n/a	n/a	n/a	n/a
S	1	0001	1	n/a	5	7	3	11	9	n/a	n/a

3.1 Classification of Possible Cases

The code C_{AB} gives us additional information on positions of the line segment end-points. If all the possible positions of a line segment are analyzed, then the following cases can be distinguished:

Table 3. Possible cases: n/a - non-applicable or solved by the C-S coding C - corner area, S - side area, IN - inside area End-points: IC - inside-corner, IS - inside-side; Cases: SS - side-side, SnCS - side-near corner - side, SdC - side-distant corner-side, CoC - corner-opposite corner, id: case re-indexing

		id	-1	0	1	2	3	4	5	6	7
	Case		IN	C	S	C	S	C	S	C	S
		C_B	0	5	4	6	2	10	8	9	1
	C_A		0000	0101	0100	0110	0010	1010	1000	1001	0001
IN	0	0000	IN	IC	IS	IC	IS	IC	IS	IC	IS
C	5	0101	IC	n/a	n/a	n/a	SdC	CoC	SdC	n/a	n/a
S	4	0100	IS	n/a	n/a	n/a	SnCS	SdC	SS	SdC	SnCS
C	6	0110	IC	n/a	n/a	n/a	n/a	n/a	SdC	CoC	SdC
S	2	0010	IS	SdC	SnCS	n/a	n/a	n/a	SnCS	SdC	SS
C	10	1010	IC	CoC	SdC	n/a	n/a	n/a	n/a	n/a	SdC
S	8	1000	IS	SdC	SS	SdC	SnCS	n/a	n/a	n/a	SnCS
C	9	1001	IC	n/a	SdC	CoC	SdC	n/a	n/a	n/a	n/a
S	1	0001	IS	n/a	SnCS	SdC	SS	SdC	SnCS	n/a	n/a

- INSIDE (IN) - the both end-points are inside
- OUTSIDE (n/a) - the line segment does not intersect the window
- Inside-Side (IS) - one end-point is inside, the second one is inside of a side area, see Fig. 3
- Inside-Corner (IC) - one end-point is inside, the second one is inside of a corner area, see Fig. 3
- Side - Side (SS) - the end-points are in the opposite side areas, see Fig. 4
- Side - near Corner - Side (SnCS) - both end-points are in the side areas sharing a common corner, see Fig. 4
- Side - distant Corner (SdC) - one end-point is inside of the side area, the second one is in the distant(opposite) corner area, see Fig. 5
- Corner - opposite Corner (CoC) - the both end-points are in the opposite corner areas, see Fig. 6

The most simple cases, i.e. the line segment is totally inside, resp. totally outside are easily detected by the bit-wise condition, i.e. by the original Cohen-Sutherland end-pointposition classification.

$$(C_A \textbf{ lor } C_B) = [0000], \quad \text{resp.} \quad (C_A \textbf{ land } C_B) \neq [0000]$$

After this, the simple cases as Inside-Side (IS) or Inside-Corner (IC) cases are detected by the logical condition

$$(C_A = [0000]) \textbf{ or } (C_B = [0000])$$

Now, the more complex cases are to be solved, i.e. Side-near Corner Side (SnCS), Side-distant Corner (SdC), Corner-opposite Corner (CoC) cases.

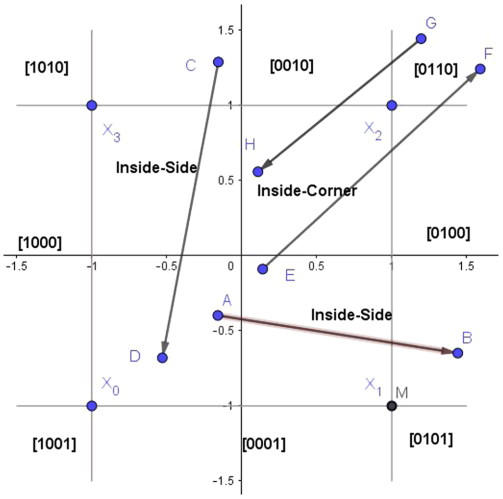

Fig. 3. Inside-Side (IS) and Inside-Corner (IC) cases

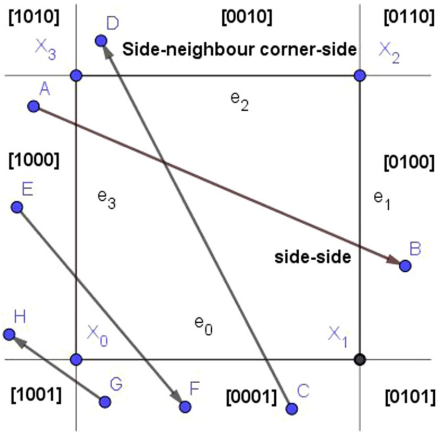

Fig. 4. Side-Side(SS) and Side-near Corner (SnC)

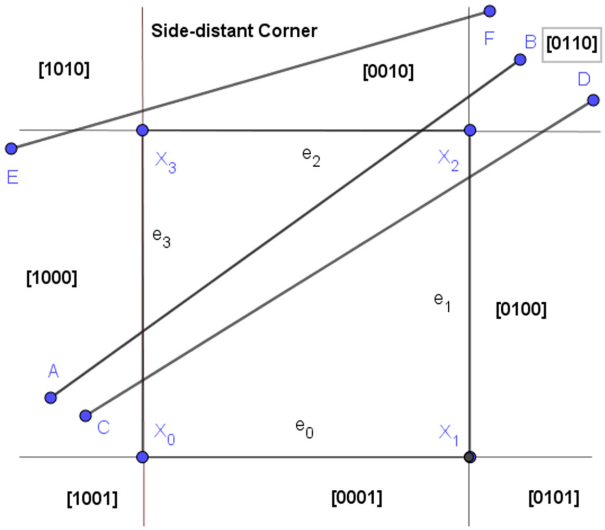

Fig. 5. Side-distant Corner (SdC) case

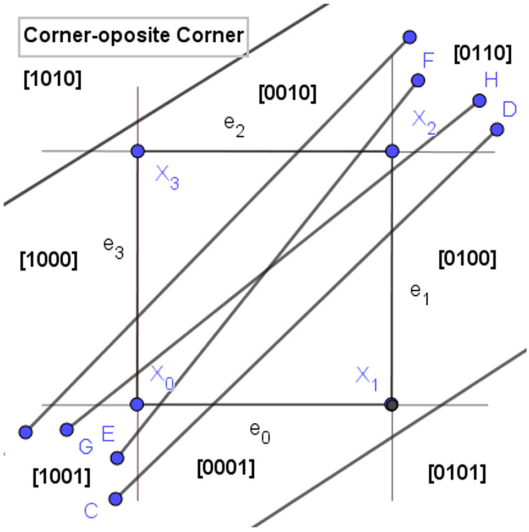

Fig. 6. The Corner-opposite Corner (CoC) case

The Table 3 presents all the cases for the summation code $C_{AB} = C_A + C_B$, while the Table 2 presents the C_{AB} values for each the case. It can be seen that non-trivial cases are actually formed by sub-tables 3×3 and the table itself is formally symmetrical. As a line segment is oriented actually, its orientation is to be respected in the algorithm. Unfortunately, the codes for C_A and C_B in the table Table 3 are not ordered according to the numerical values of the codes. As

the simple cases, when at least one point is inside of the clipping window, are easily detectable, only more complex cases are to be distinguished and they are re-indexed.

The table Table 4 represents re-indexing of the code C_A, resp. C_B, so that the index of an area is ordered anti-clockwise starting at 0 from the right bottom side area; the *id* of the clipping window area is set to -1 for code efficiency only.

It can be seen, that the re-indexed value *id* gives also additional information, whether the end-point is inside of the side area or corner one, i.e. *id* is even or odd (except of the INSIDE case).

$$id \textbf{ and } [0001] = \begin{cases} [0000] \ corner\ area \\ [0001] \ side\ area \end{cases}$$

The one-dimensional array is used to end-points code re-indexing for cases of the codes C_A and C_B (non-applicable cases 11–15 removed).

$$id_A = TAB_CODE_INDEX[C_A] \quad id_B = TAB_CODE_INDEX[C_B]$$

Table 4. TAB_CODE_INDEX: Re-indexing table of edges and corners using [Left, Right, Top, Bottom] coding

C_A	C_A	$Type$	id	C_A	C_A	$Type$	id
0	0000	IN	-1	8	1000	S	5
1	0001	S	7	9	1001	C	6
2	0010	S	3	10	1010	C	4
3	0011	n/a	n/a	11	1011	n/a	n/a
4	0100	S	1	12	1100	n/a	n/a
5	0101	C	0	13	1101	n/a	n/a
6	0110	C	2	14	1110	n/a	n/a
7	0111	n/a	n/a	15	1111	n/a	n/a

It means, that the index id_A is giving a row in the Table 3 for the code C_A and the id_B is giving the column for the code C_B. As a consequence, if $id_A < id_B$ then the only upper triangle of the Table 3 is used, i.e. significant number of cases are reduced. The value of the code C_{AB} enables to distinguish different possible cases easily. However, as the line segment orientation is to be respected, the proposed algorithm, described in the next, has to respect it.

Algorithm 3. Q-CLIP

```
1: Global variables:
2:    real: x_min, y_min, x_max, y_max,          ▷ x_min, x_max the window's corner
3:    array TAB_CODE_INDEX[0:10] = [ -1, 7, 3, -9, 1, 0, 2, -9, 5, 6, 4 ];
4:                ▷ the value −1 means end-point is inside; −9 means the n/a case
5:    array TAB_CODE_CASE[0:15] =
6:      [ -1, -1, -1, 0, -1, 1, 1, 2, -1, 1, 1, 2, -1, 2, 2, 3 ];   ▷ C_AB re-indexing
7: procedure Q_CLIP(x_A, x_B);
8:                    ▷ All other procedures for clipping should be declared here
9:    C_A := CODE(x_A); C_B := CODE(x_B);                          ▷ set the C-S codes
10:                ▷ all logical operations land, lor, lxor are bit-wise operations
11:
12:                                                  ▷ the whole line segment is inside
13:    if (C_A lor C_B) = [0000] then { DRAW(x_A, x_B); EXIT; }
                                                  ▷ the whole line segment is outside
14:    if (C_A land C_B) ≠ [0000] then  EXIT;
15:    C_AB := C_A + C_B;                              ▷ one end-point is inside cases
16:    # precompute directional vector s = x_B − x_A for better efficiency
17:    if (C_A = [0000]) or (C_B = [0000]) then {
18:       id_AB := TAB_CODE_INDEX[C_AB];          ▷ now, only the cases: IS or IC
19:       if (id_AB land [0001])= [0000] then
20:          { SOLVE_IS; EXIT }                   ▷ the IS case; C_AB ∈ {4, 2, 8, 1}
21:       else
22:          { SOLVE_IC; EXIT }                   ▷ the IC case; C_AB ∈ {5, 6, 10, 9}
23:       endif }
24:    endif
25:    # complex cases with two possible intersections
26:    id_case := TAB_CODE_CASE[C_AB];          ▷ re-indexing of non-trivial cases
27:    switch id_case do                         ▷ all complex cases classification
28:       case 0: { SOLVE_SS; EXIT; }                  ▷ SS cases; C_AB ∈ {3, 12}
29:       case 1: { SOLVE_SnCS; EXIT; }        ▷ SnCS cases; C_AB ∈ {6, 5, 10, 9}
30:       case 2: { SOLVE_SdC; EXIT; }         ▷ SdC cases; C_AB ∈ {7, 13, 14, 11}
31:       case 3: { SOLVE_CoC; EXIT; }                 ▷ CoC cases; C_AB = 15
32:    end switch
33: end procedure
```

3.2 Q-CLIP Algorithm

The proposed clipping Q-CLIP algorithm is described by the Algorithm 3. The Q-CLIP algorithm solves the trivial cases first, i.e. the whole segment acceptance or rejection, then more complex cases are solved. It can be seen, that the Q-Clip algorithm is free of cycles, i.e. **while** and/or **for** cycle constructions, etc.

The implementation of the proposed algorithm is simple and straightforward, however, it should be noted, that:

– careful implementation is needed to solve each case, e.g. IC, IS ..., as it influences efficiency of the algorithm significantly.

– use of **array of function** in-line construction might be more computationally efficient than the**switch** construction
– the bit-wise condition $(C_A$ **lor** $C_B) = [0000]$ differs from the condition $(C_A = [0000])$ **or** $(C_B = [0000])$ as it is the logical operation

It should be noted that the **Switch** instruction is to be implemented as an array of inline functions in order to avoid multiple **if** instructions in which the **Switch** instruction is actually translated.

The presented Q-CLIP algorithm can be easily modified for the case, when a line segment and vertices of the clipping window are given in homogeneous coordinates in general, i.e. when $w \neq 1$.

The proposed Q-CLIP algorithm was implemented in C and Pascal languages on 64bit MS Windows 10 operating system. Experiments made proved its superiority over the original Cohen-Sutherland algorithm, especially in the SdC and CoC cases. For each case, i.e. IC, IS, SnC, SdC, CoC the same number of line segments were generated randomly. The average speed up was over 15% nearly independent of the programming language used.

However, if vector operations with the homogeneous coordinate representation is used, similarly as in Nielsen [13], Skala [26], additional significant speed up can be expected if SSE instructions are used.

4 Conclusion

This contribution describes shortly a new coding scheme for the line segment clipping algorithm based on Cohen-Sutherland's algorithm using arithmetic operations to distinguish the fundamental cases eliminating unnecessary computations with clipping window edges. All the cases are easy to implement. However, computational efficiency is to be kept in mind in coding.

The experiments made proved the speedup over 10–15% against the original Cohen-Sutherland algorithm. Additional speed up can be expected if vector notation and vector operations are used for intersection computation.

The proposed algorithm presents a new coding scheme for distinguishing all the cases in line segment clipping in E^2. Similar approach can be taken for the line segment clipping in the E^3 case.

Acknowledgment. The author would like to thank to colleagues at the University of West Bohemia in Plzen for fruitful discussions and to anonymous reviewers for their comments and hints, which helped to improve the manuscript significantly.

References

1. Andreev, R., Sofianska, E.: New algorithm for two-dimensional line clipping. Comput. Graph. **15**(4), 519–526 (1991)
2. Bui, D., Skala, V.: Fast algorithms for clipping lines and line segments in E2. Vis. Comput. **14**(1), 31–37 (1998)

3. Cyrus, M., Beck, J.: Generalized two- and three-dimensional clipping. Comput. Graph. **3**(1), 23–28 (1978)
4. Day, J.: A new two dimensional line clipping algorithm for small windows. Comput. Graph. Forum **11**(4), 241–245 (1992)
5. Duvanenko, V., Robbins, W., Gyurcsik, R.: Line-segment clipping revisited. Dr. Dobb's J. **21**(1), 107–110 (1996)
6. Dörr, M.: A new approach to parametric line clipping. Comput. Graph. **14**(3–4), 449–464 (1990)
7. Foley, D., van Dam, A., Feiner, S., Hughes, J.: Computer Graphics: Principles and Practice. Addison-Wesley (1990)
8. Hughes, J.F., et al.: Computer Graphics: Principles and Practice, 3 edn. Addison-Wesley (2013)
9. Kaijian, S., Edwards, J., Cooper, D.: An efficient line clipping algorithm. Comput. Graph. **14**(2), 297–301 (1990)
10. Krammer, G.: A line clipping algorithm and its analysis. Comput. Graph. Forum **11**(3), 253–266 (1992)
11. Liang, Y.-D., Barsky, B.: A new concept and method for line clipping. ACM Trans. Graph. (TOG) **3**(1), 1–22 (1984)
12. Nicholl, T.M., Lee, D., Nicholl, R.A.: Efficient new algorithm for 2D line clipping: its development and analysis. Comput. Graph. (ACM) **21**(4), 253–262 (1987)
13. Nielsen, H.: Line clipping using semi-homogeneous coordinates. Comput. Graph. Forum **14**(1), 3–16 (1995)
14. Rappoport, A.: An efficient algorithm for line and polygon clipping. Vis. Comput. **7**(1), 19–28 (1991)
15. Skala, V.: Algorithm for 2D line clipping. In: Earnshaw, R.A., Wyvill, B. (eds.) New Advances in Computer Graphics, NATO ASI, pp. 121–128 (1989). https://doi.org/10.1007/978-4-431-68093-2_7
16. Skala, V.: Algorithms for clipping quadratic arcs. In: Chua, T.-S., Kunii, T.L. (eds.) CG International '90. pp, pp. 255–268. Springer, Tokyo (1990). https://doi.org/10.1007/978-4-431-68123-6_16
17. Skala, V.: An efficient algorithm for line clipping by convex polygon. Comput. Graph. **17**(4), 417–421 (1993)
18. Skala, V.: O(lg N) line clipping algorithm in E2. Comput. Graph. **18**(4), 517–524 (1994)
19. Skala, V.: An efficient algorithm for line clipping by convex and non-convex polyhedra in E3. Comput. Graph. Forum **15**(1), 61–68 (1996)
20. Skala, V.: Line clipping in E2 with O(1) processing complexity. Comput. Graph. (Pergamon) **20**(4), 523–530 (1996)
21. Skala, V.: A fast algorithm for line clipping by convex polyhedron in E3. Comput. Graph. (Pergamon) **21**(2), 209–214 (1997)
22. Skala, V.: A new approach to line and line segment clipping in homogeneous coordinates. Vis. Comput. **21**(11), 905–914 (2005)
23. Skala, V.: Length, area and volume computation in homogeneous coordinates. Int. J. Image Graph. **6**(4), 625–639 (2006)
24. Skala, V.: Barycentric coordinates computation in homogeneous coordinates. Comput. Graph. (Pergamon) **32**(1), 120–127 (2008)
25. Skala, V.: Intersection computation in projective space using homogeneous coordinates. Int. J. Image Graph. **8**(4), 615–628 (2008)
26. Skala, V.: Optimized line and line segment clipping in E2 and geometric algebra. Ann. Math. Inform. **52**, 199–215 (2020)

27. Skala, V.: A novel line convex polygon clipping algorithm in E2 with parallel processing modification. In: Gervasi, O., et al. (eds.) ICCSA 2021. LNCS, vol. 12953, pp. 3–15. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86976-2_1
28. Skala, V., Bui, D.: Extension of the Nicholls-Lee-Nichols algorithm to three dimensions. Vis. Comput. **17**(4), 236–242 (2001)
29. Sobkow, M., Pospisil, P., Yang, Y.-H.: A fast two-dimensional line clipping algorithm via line encoding. Comput. Graph. **11**(4), 459–467 (1987)