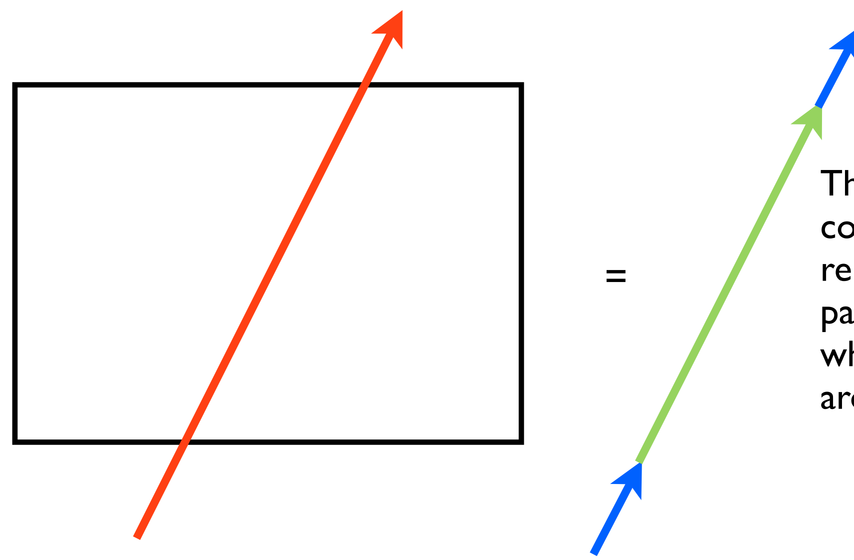
Challenge #2 - Line Clipping Algorithms
Cohen-Sutherland / Liang-Barsky
Computer Graphics

Presented by:
Santiago Zubieta / Jose Cortes

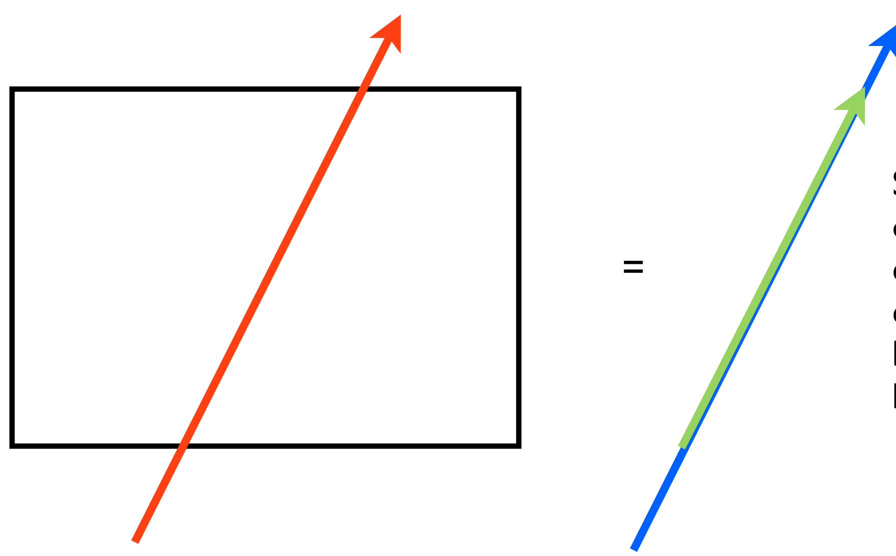
Teacher:
Helmuth Trefftz

Universidad EAFIT

With these clipping algorithms we intend to not only find the sub-line lying within the clipping box/area, but also create other sub-lines that are part of the original line but outside the clipping area. This is why our clipping algorithm implementations return not only a Line, but a List of variable size of Lines, each one identified as being inside or outside.



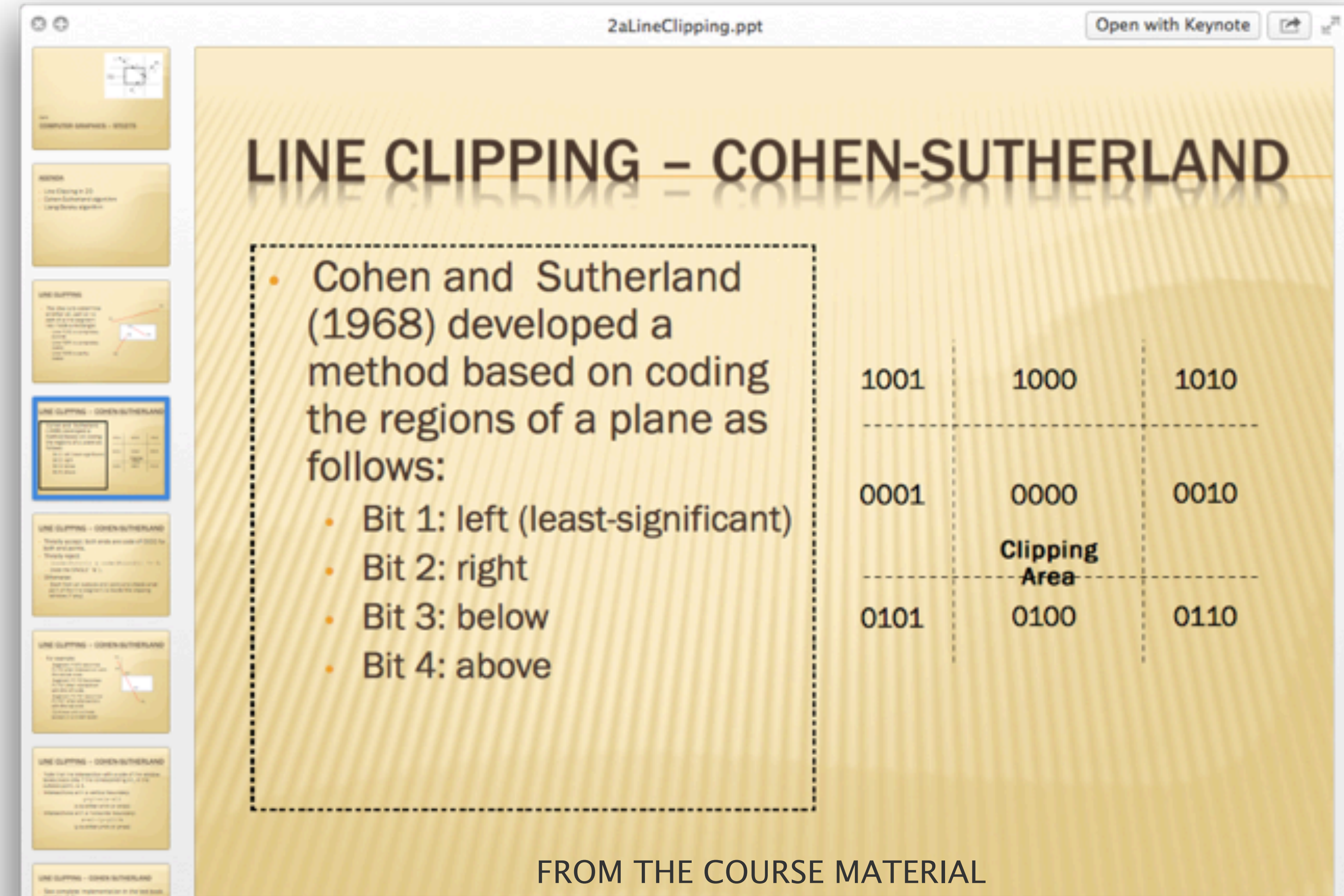
Three sub-lines that together compose the original line, to represent visually the clipped part of the line, which is cool when moving the clipping area around with the pointer!



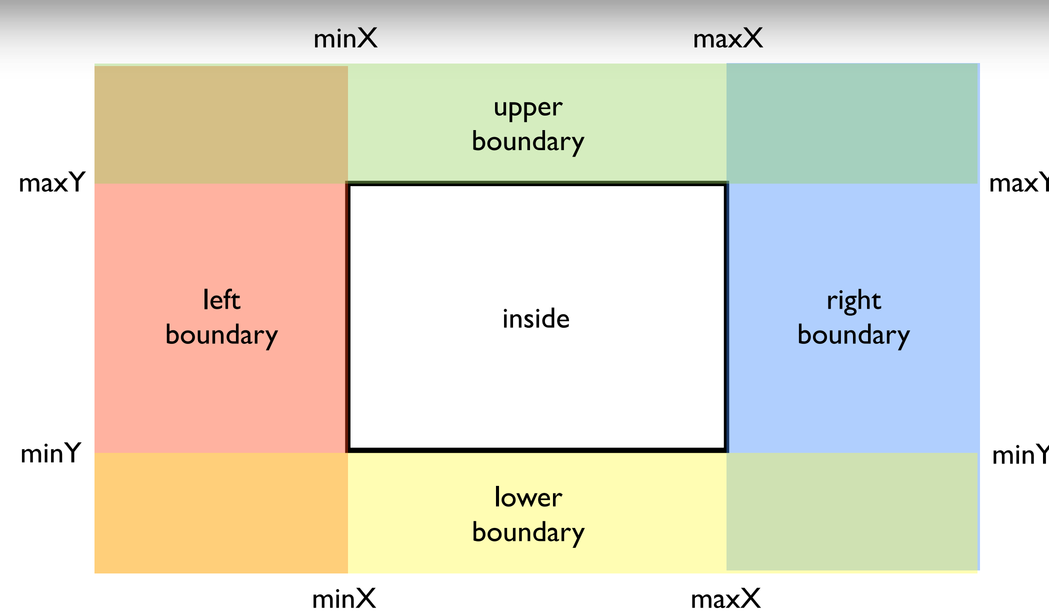
Some may find only the clipped line, redraw the original line in the “outside” color and draw the clipped line over it in the “inside” color. This is cheating! May look ugly according to aliasing, and be terribly slower (drawing a part that will just be overlapped later).

The outer Lines are easily created in the Cohen-Sutherland Algorithm by taking the pieces of outer Line it trims in each iteration (until reaching trivial cases) or in the Liang-Barsky algorithm by making lines between the starting/ending points defining the original Line, and the starting/ending points defining the clipped Line.

1. Cohen-Sutherland Algorithm



First a List is generated with all the lines (by default 1000) to be clipped. Both algorithms run on the same List.

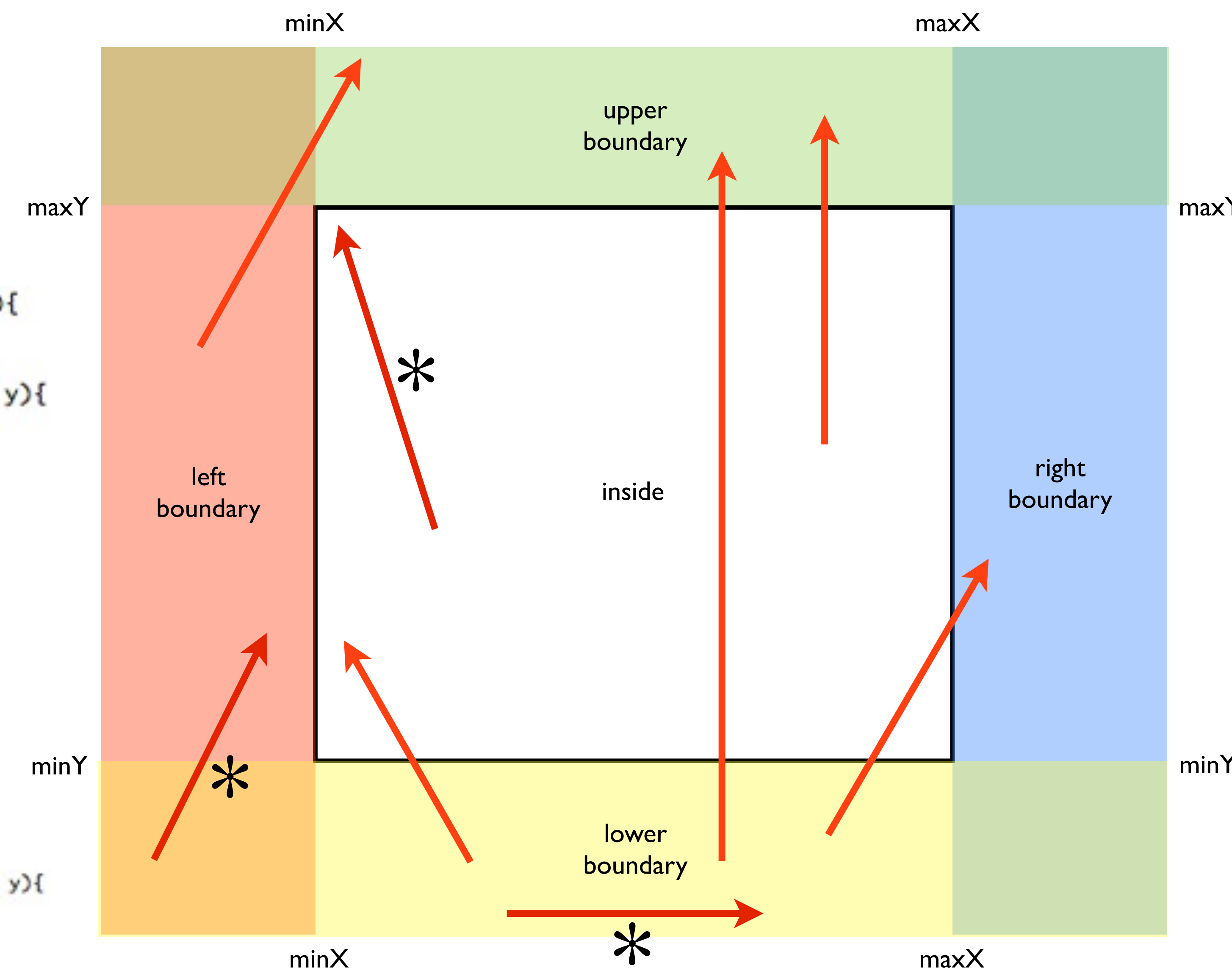


For each line, the algorithms will return a List containing its sub-lines according to the clipping. May be 1, 2 or 3 sub-lines.

Trivial Cases

-Both ends completely Inside
-Both ends outside but share at least one boundary

Remove them and classify accordingly (inside, outside)




```
private boolean upOfBox(int y){
    return y > maxY;
}
private boolean downOfBox(int y){
    return y < minY;
}
```

```
private boolean leftOfBox(int x){
    return x < minX;
}
private boolean rightOfBox(int x){
    return x > maxX;
}
```

*may be faster if you use binary literals but somehow Java won't let me

Trivial Accept:

this is true for both points

```
public boolean isInsideBox(int x, int y){
    //p is 0000
    return rightOfBox(x) == false &&
           downOfBox(y)  == false &&
           leftOfBox(x)  == false &&
           upOfBox(y)    == false;
}
private boolean bothInside(int x1, int y1, int x2, int y2){
    //trivial accept
    //p1 and p2 are both 0000 / 0000
    return isInsideBox(x1,y1) == true &&
           isInsideBox(x2,y2) == true;
}
```

* : current trivial cases

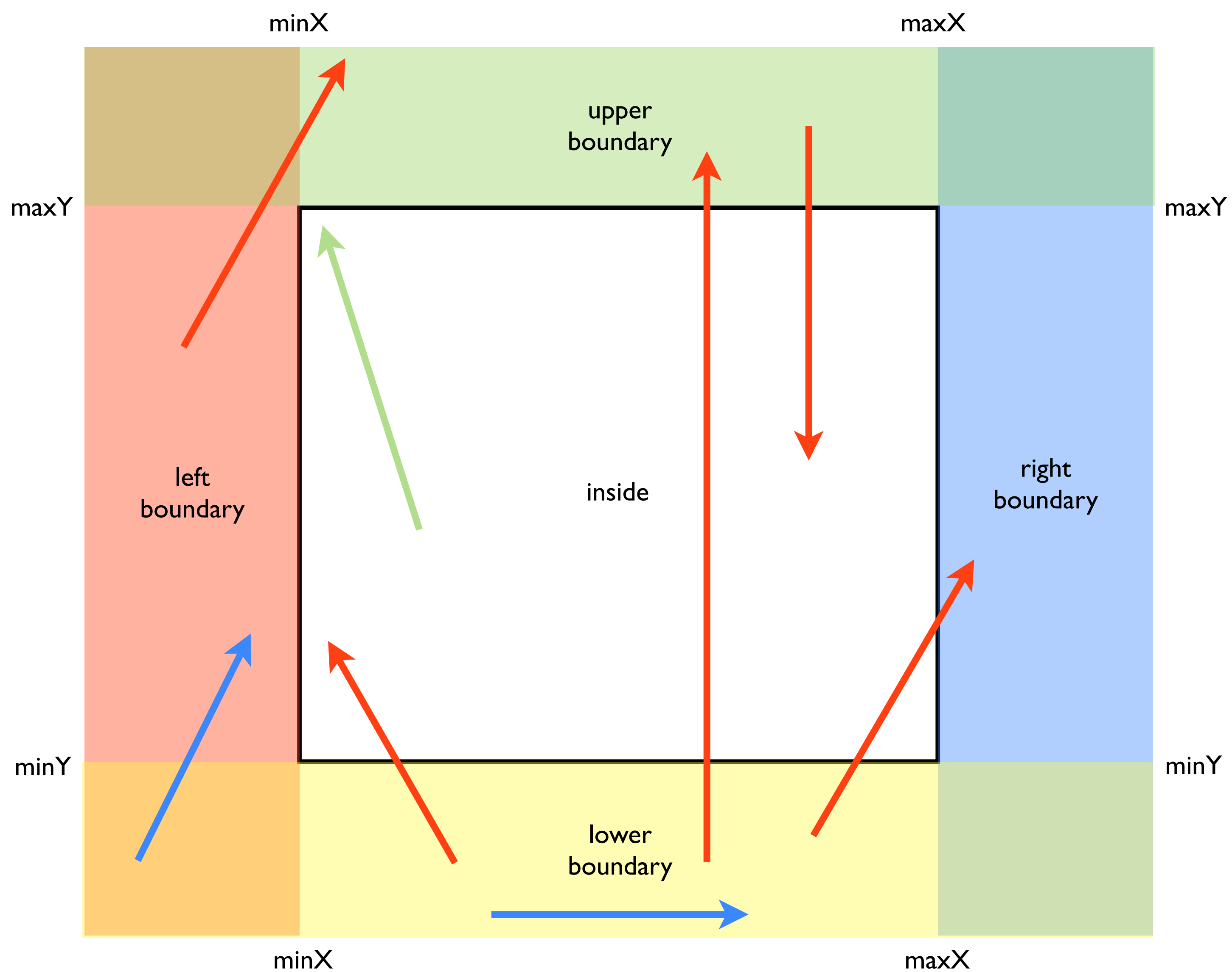
Trivial Reject:

this is true for both points

```
private boolean bothSameSide(int x1, int y1, int x2, int y2){
    //trivial reject: p1 & p2 != 0
    //p1 and p2 have at least a 1 in the same position
    return rightOfBox(x1) == true && rightOfBox(x2) == true ||
           downOfBox(y1)  == true && downOfBox(y2)  == true ||
           leftOfBox(x1)  == true && leftOfBox(x2)  == true ||
           upOfBox(y1)    == true && upOfBox(y2)    == true;
}
```

Non-trivial Cases

Each line will have at least one point outside the box. Lets make sure that the starting point is outside, swapping places if needed. This will make the segment of the line that is trimmed at a boundary in each iteration to be assured to be outside.

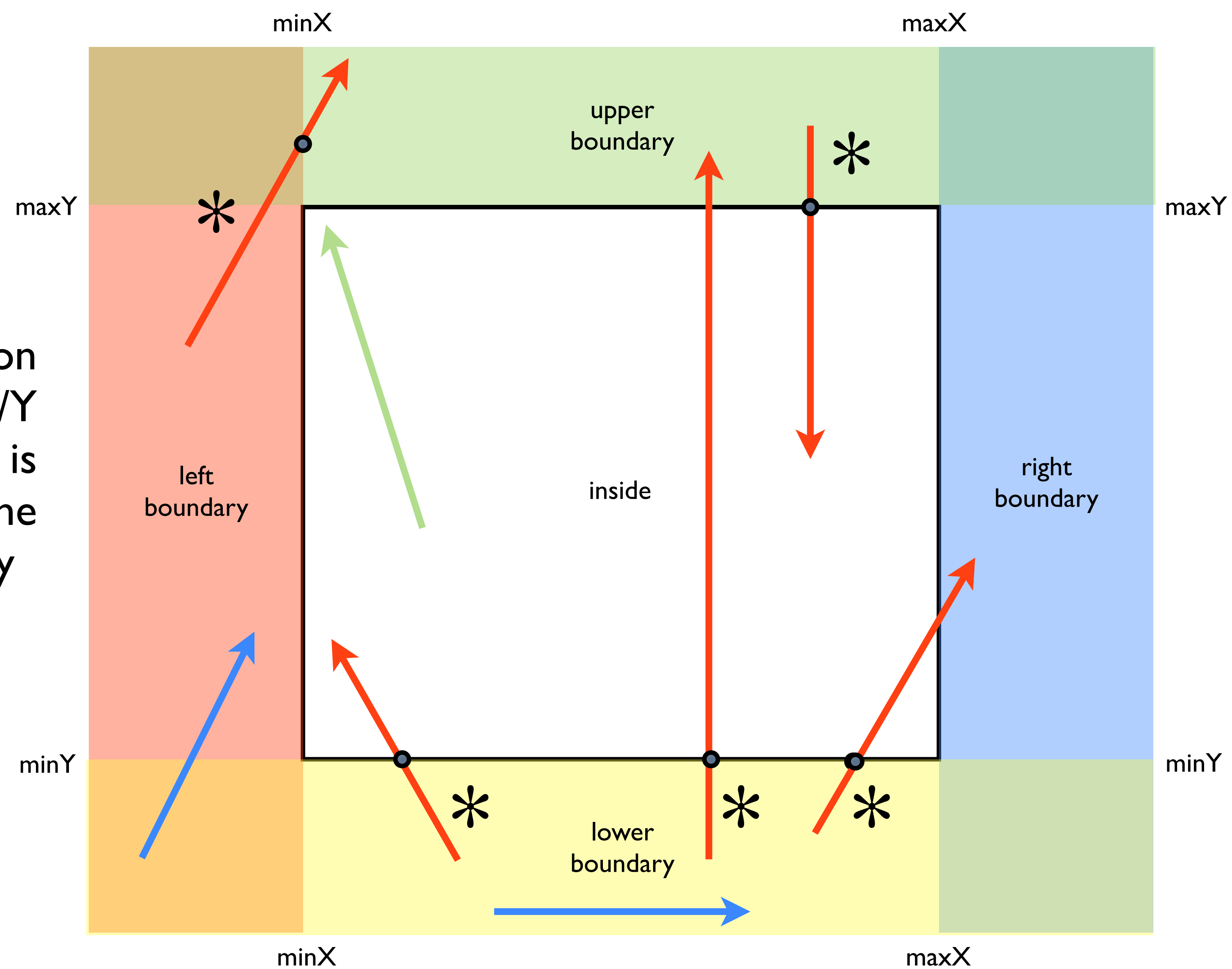



```
if(isInsideBox(x1, y1) == true){
    int temp;
    temp = x1; x1 = x2; x2 = temp;
    temp = y1; y1 = y2; y2 = temp;
}
```

Non-trivial Cases

Now check at what boundary of the box the starting point lies

According to the position of the point, a new X/Y will be found, which is the point moved to the corresponding boundary



The line from the starting point to this new point is assured to be lying outside the box, so clip and color accordingly

```
if(upOfBox(y1) == true){
    nx = x1 + (dx / dy) * (maxY-y1);
    ny = maxY;
}
else if(downOfBox(y1) == true){
    nx = x1 + (dx / dy) * (minY-y1);
    ny = minY;
}
```

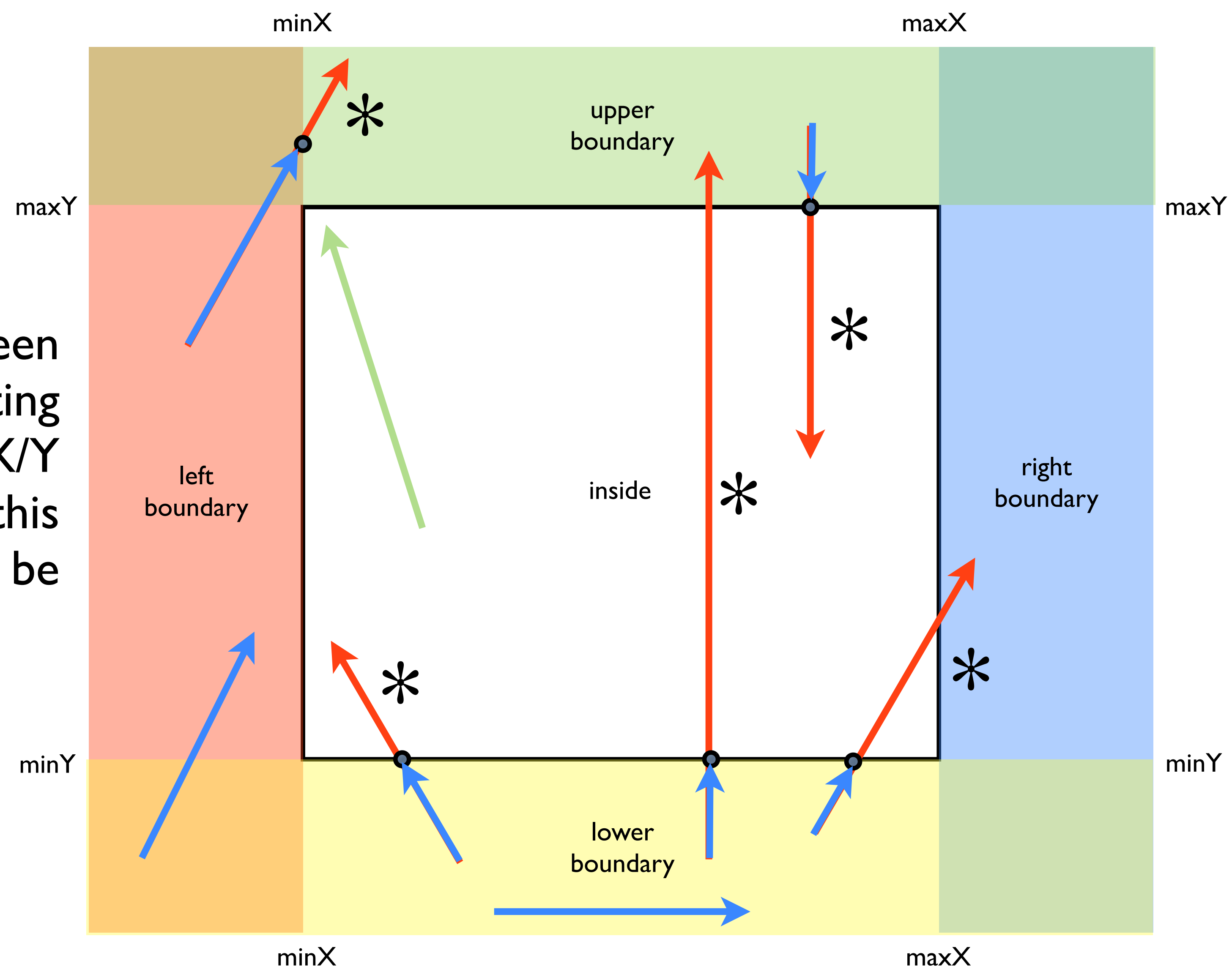
*: segments assure to be outside

```
else if(rightOfBox(x1) == true){
    ny = y1 + (dy / dx) * (maxX-x1);
    nx = maxX;
}
else if(leftOfBox(x1) == true){
    ny = y1 + (dy / dx) * (minX-x1);
    nx = minX;
}
```

Non-trivial Cases

The line from the starting point to the new point is assured to be outside, but the line from the new point to the ending point can't be assured to be completely inside.

Add the Line between the original X/Y starting point and the new X/Y to the result List, this Line is assured to be completely outside.



Add the line between the new X/Y and the original X/Y ending point to a stack, which is used within a loop, the algorithm will be repeated with this new line.

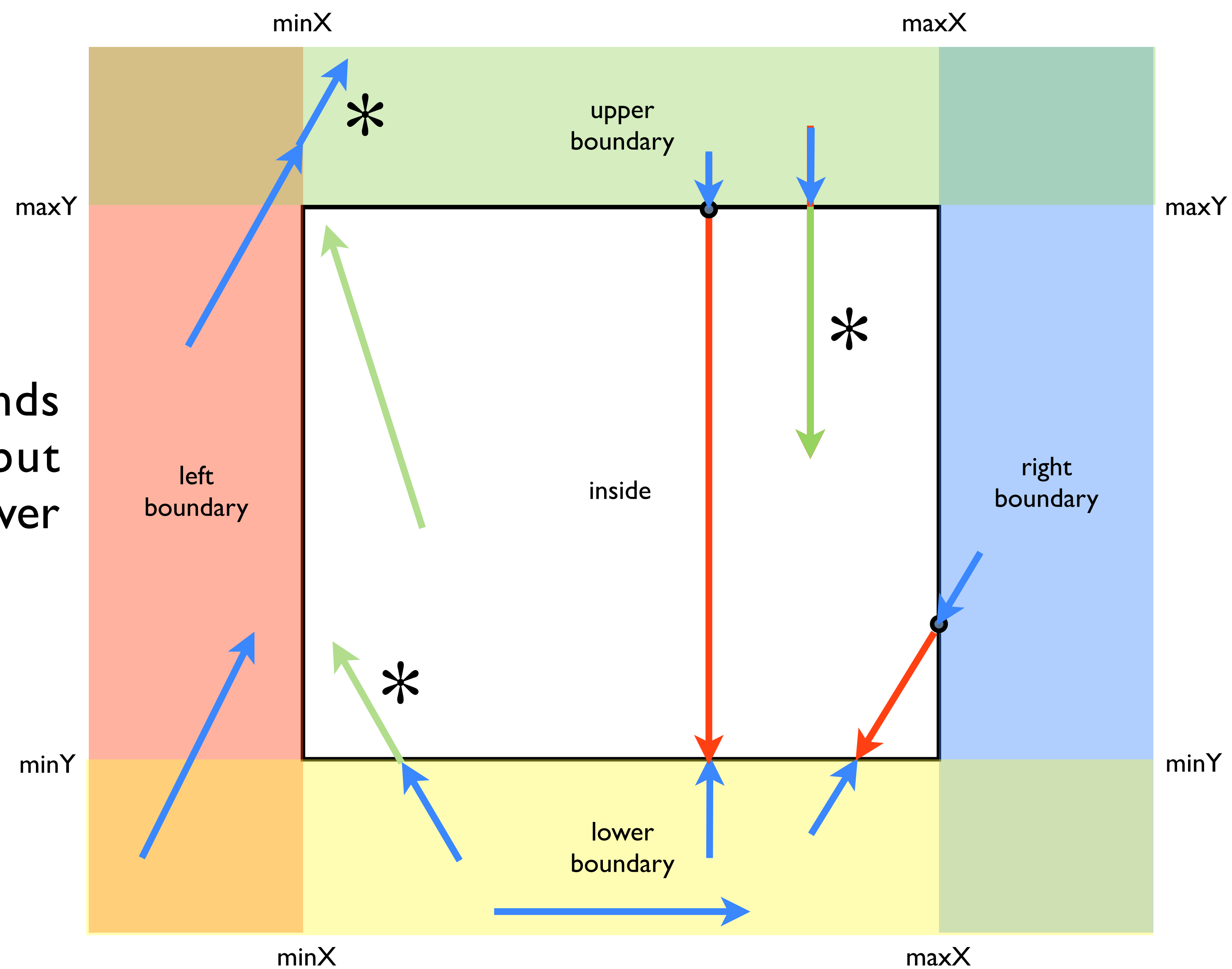
```
line = new Line(xx, yy, x1, y1);
arr.add(line);
stack.push(new Line(xx, yy, x2, y2));
```

* : segments that will be used on next iteration

Repeat Algorithm Again

Trivial accepts/rejects are more likely at this stage.

Lines with both ends outside the box, but crossing it, will go over the algorithm 3 TIMES



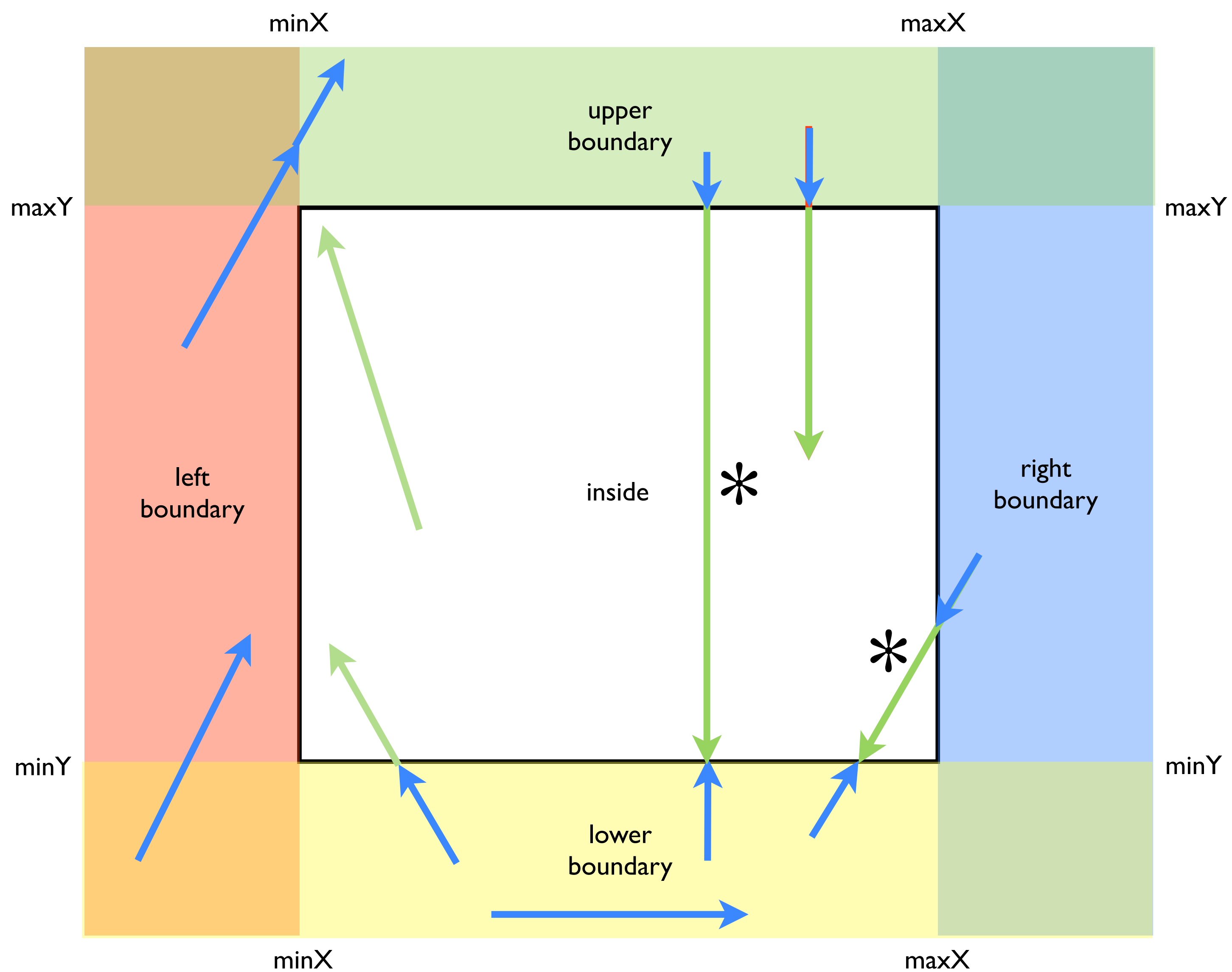
Lines with an end inside and other end outside, will go over the algorithm 2 TIMES

Lines that are trivial from the beginning with go over the algorithm just 1 TIME

*: current iteration trivial cases

Repeat Algorithm Again

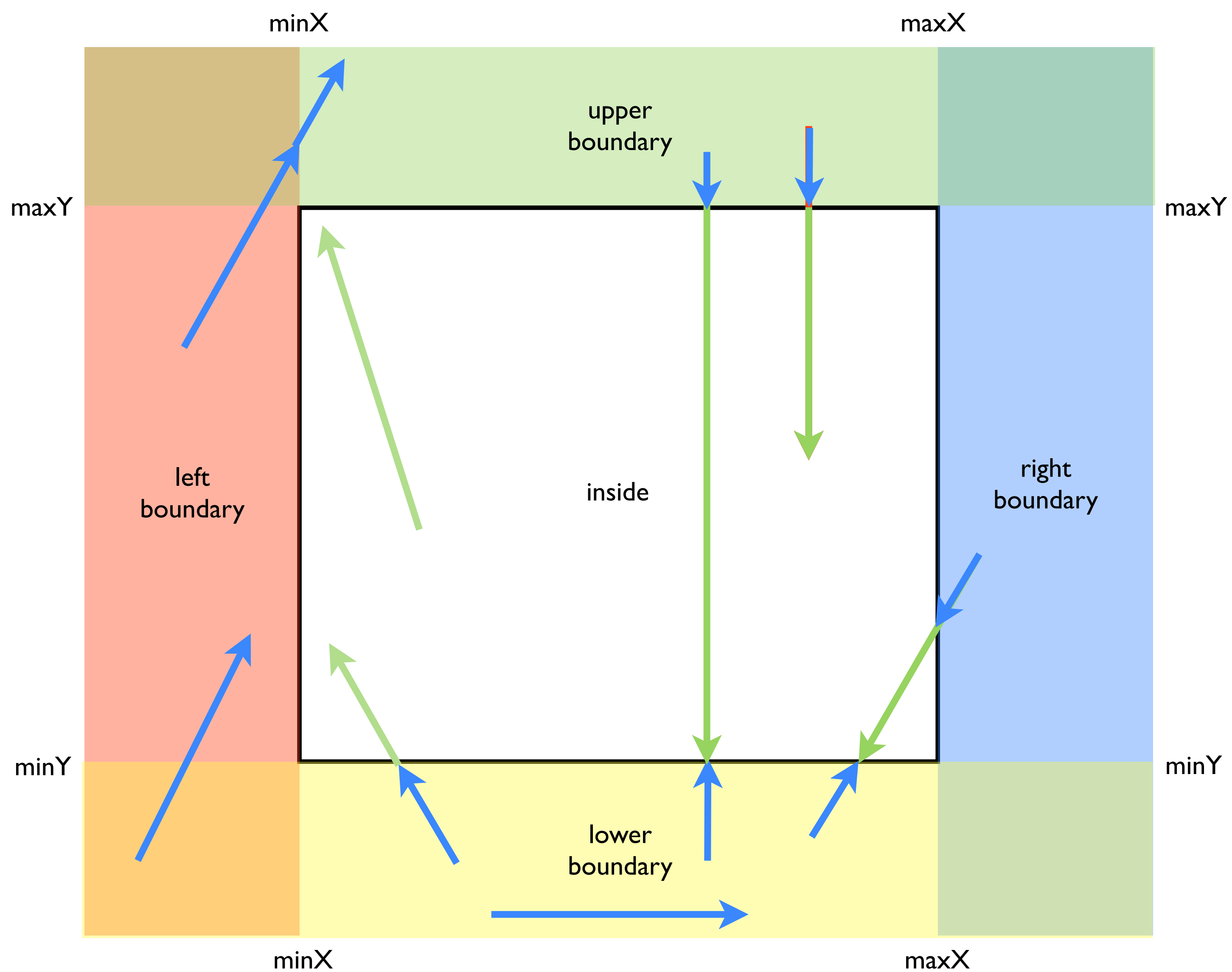
Every line will go through at most 3 iterations of the algorithm, always ending in the usual accept/reject trivial cases.



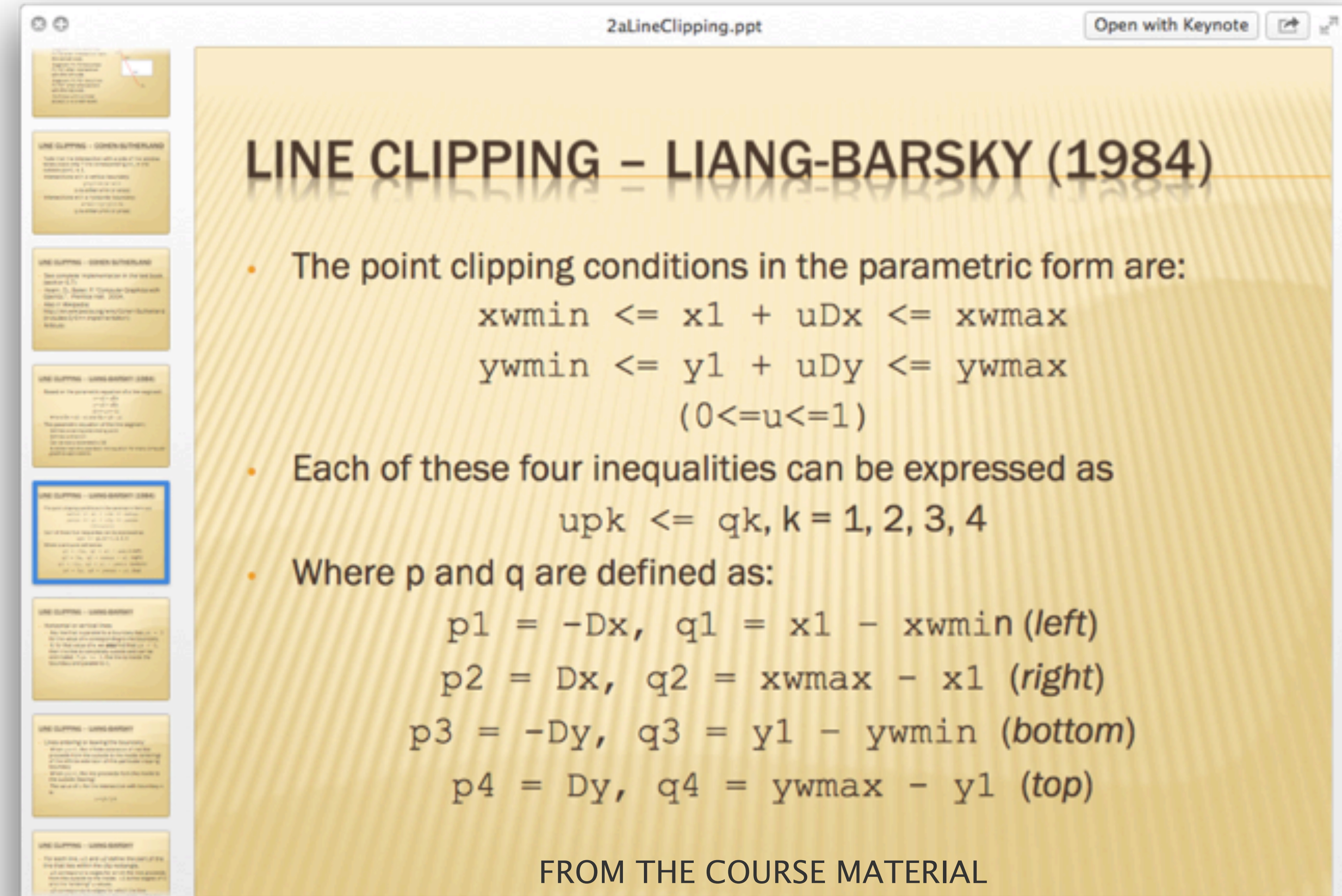
*: current iteration trivial cases

Conclusions

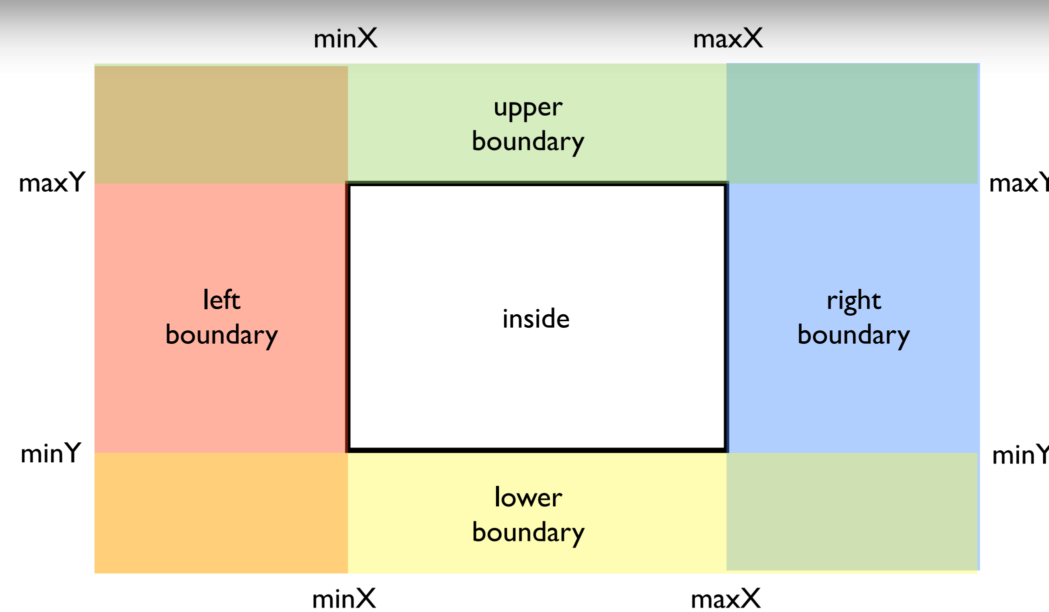
Too slow due to repeated going over the algorithm up to 3 times, it does this to reduce every line to the 2 trivial cases.



II. Liang-Barsky Algorithm



First a List is generated with all the lines (by default 1000) to be clipped. Both algorithms run on the same List.



For each line, the algorithms will return a List containing its sub-lines according to the clipping.

Non-trivial Cases

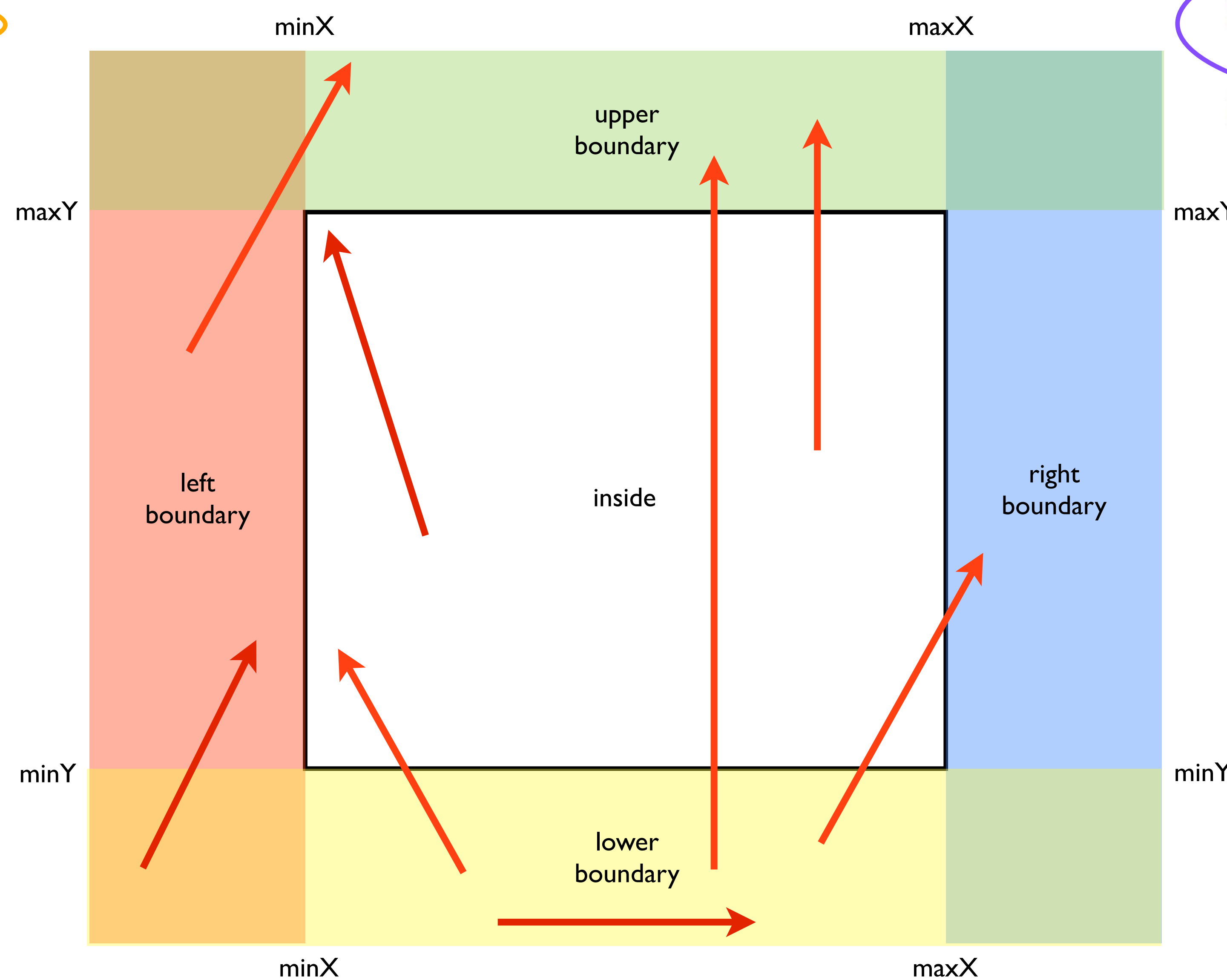
There's only one Special Case, and its a line parallel to a boundary, and completely outside of it. Parallel lines but within boundaries, are fixed by not taking into account $+\infty$/$-\infty$ values of u.

```
if(p[i] == 0 && q[i] < 0){
```

draw original line,
its completely outside

```
u = q[i]/p[i];
if(u == Double.POSITIVE_INFINITY ||
    u == Double.NEGATIVE_INFINITY){
    continue;
}
```

Ignore this value of u. The u's from the other axis will be used.



Us:
originally 0

```
// x1 = x1 + 0*(x2-x1)
```

Ut:
originally 1

```
// x2 = x1 + 1*(x2-x1)
```

Find new U's that satisfy that they the biggest but not lower than 0 for Us, the smallest but not higher than 1 for Ut, and that Us is lower than Ut

Otherwise the algorithm will proceed to find the best values of us - ut for the parametric equation

Mathematics

Parametric Equations

$$0 \le u \le 1$$

$$x_1 + u*dx = X \qquad \boxed{\begin{matrix} x_1 & x_2 \\ y_1 & y_2 \end{matrix}}$$
$$y_1 + u*dy = Y$$

$$x_1 + u_s*dx = nx_1$$
$$y_1 + u_s*dy = ny_1$$
$$x_1 + u_t*dx = nx_2$$
$$y_1 + u_t*dy = ny_2$$

$$\frac{\overset{Qi}{@ - @_1}}{\underset{Pi}{d@}} = u$$

```
p[0] = -dx; p[1] = dx;
p[2] = -dy; p[3] = dy;
q[0] = x1 - minX; q[1] = maxX - x1;
q[2] = y1 - minY; q[3] = maxY - y1;
```

for (0..3) ~ i

(after ignoring infinite values of **u** and the special case)

```
if(p[i] < 0){
    ini = Math.max(ini, u);
    // The furthest entering value
}
else{
    end = Math.min(end, u);
    // The closest exiting value
}
```

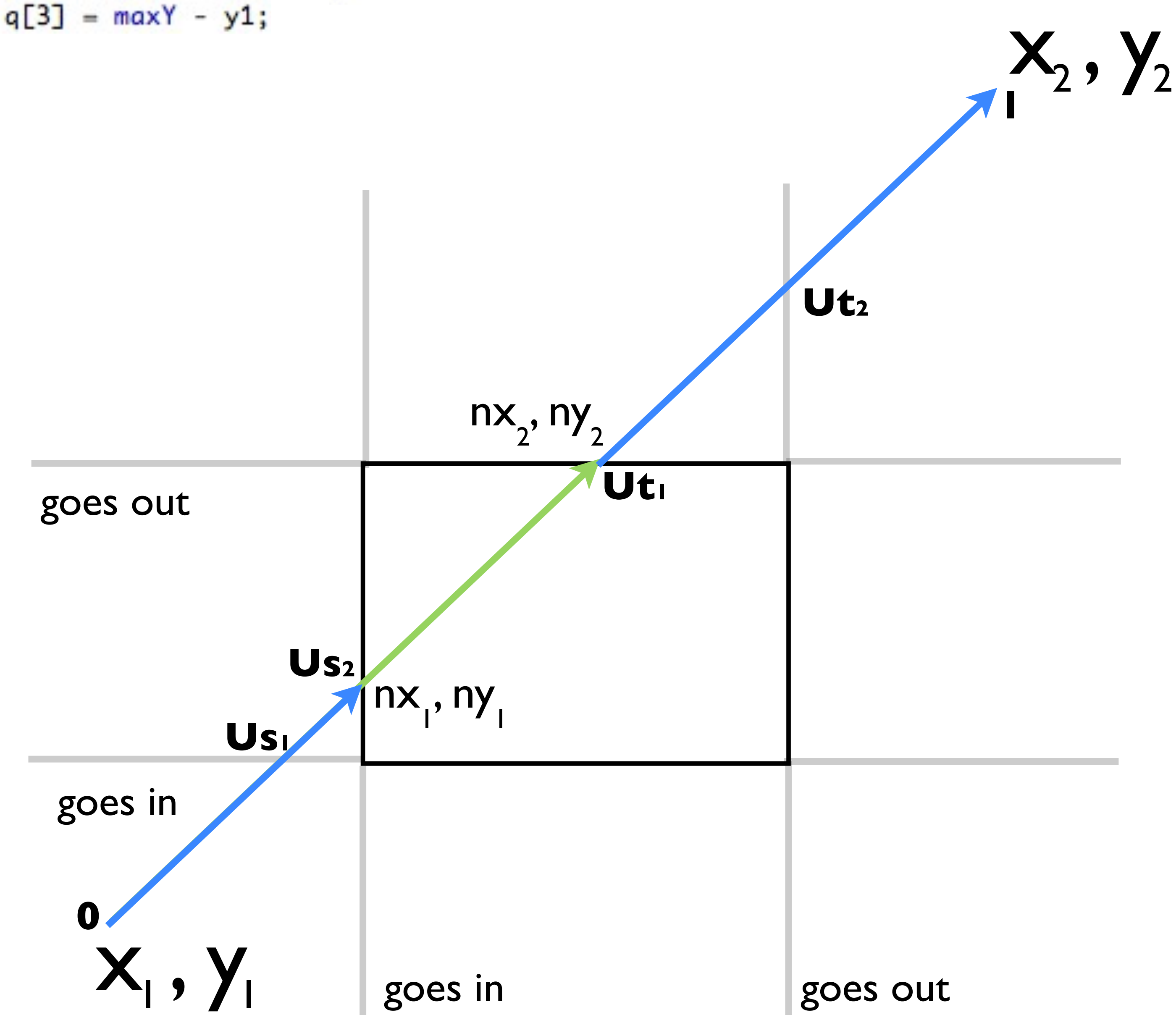
Us must be the biggest Us found among incoming boundaries, also bigger than 0. Any smaller value will give a point outside the Line.

Ut must be the smallest Ut found among outgoing boundaries, also smaller than 1. Any bigger value will give a point outside the Line.

$\mathrm{Min}(Ut_1, Ut_2, 1) = Ut_1$
$\mathrm{Max}(Us_1, Us_2, 0) = Us_2$
$end = Ut = Ut_1$
$ini = Us = Us_2$

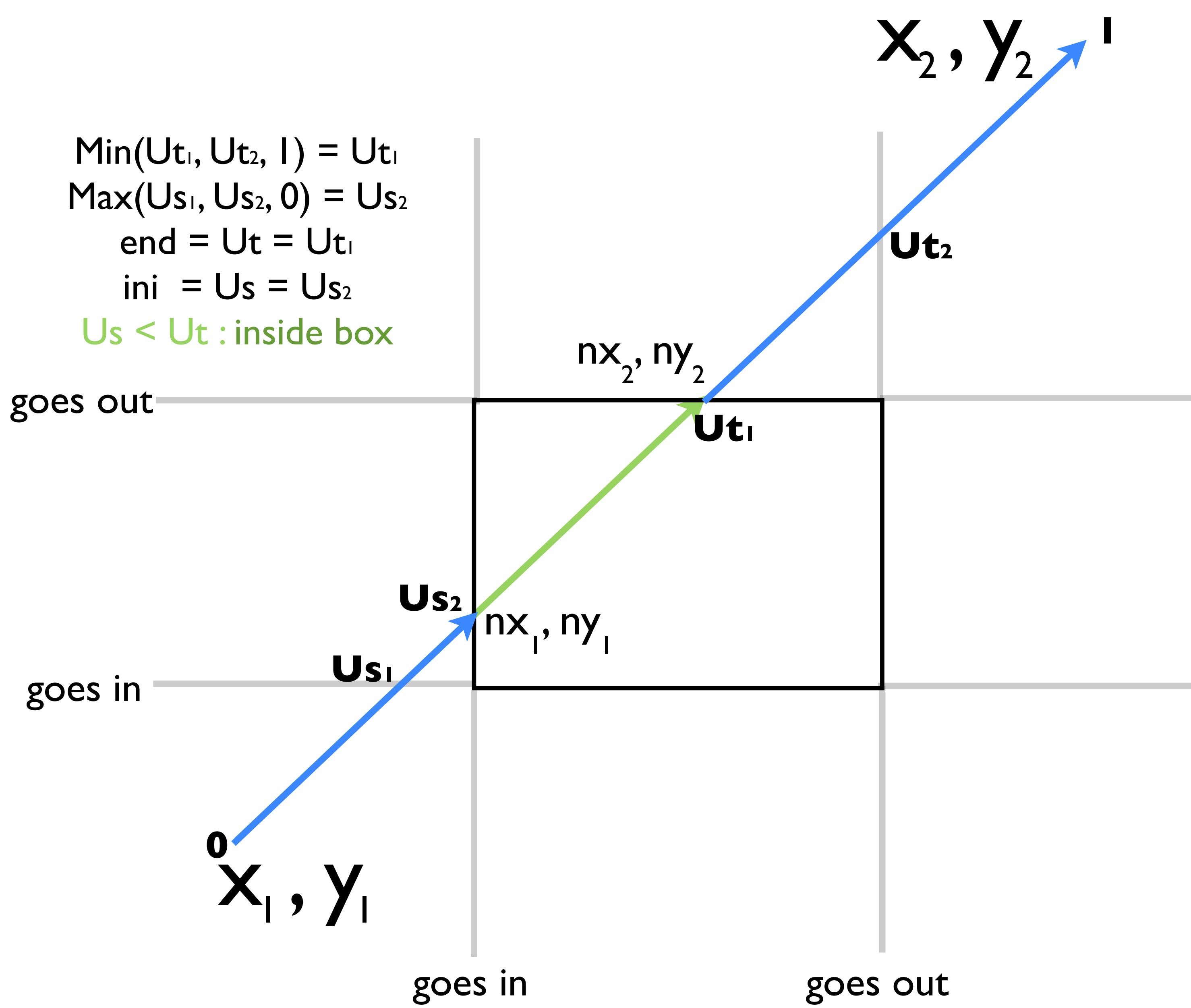
Use to find new starting/ending points.

```
int nx1, nx2, ny1, ny2;
nx1 = (int)(Math.round(x1 + ini*dx));
nx2 = (int)(Math.round(x1 + end*dx));
ny1 = (int)(Math.round(y1 + ini*dy));
ny2 = (int)(Math.round(y1 + end*dy));
```



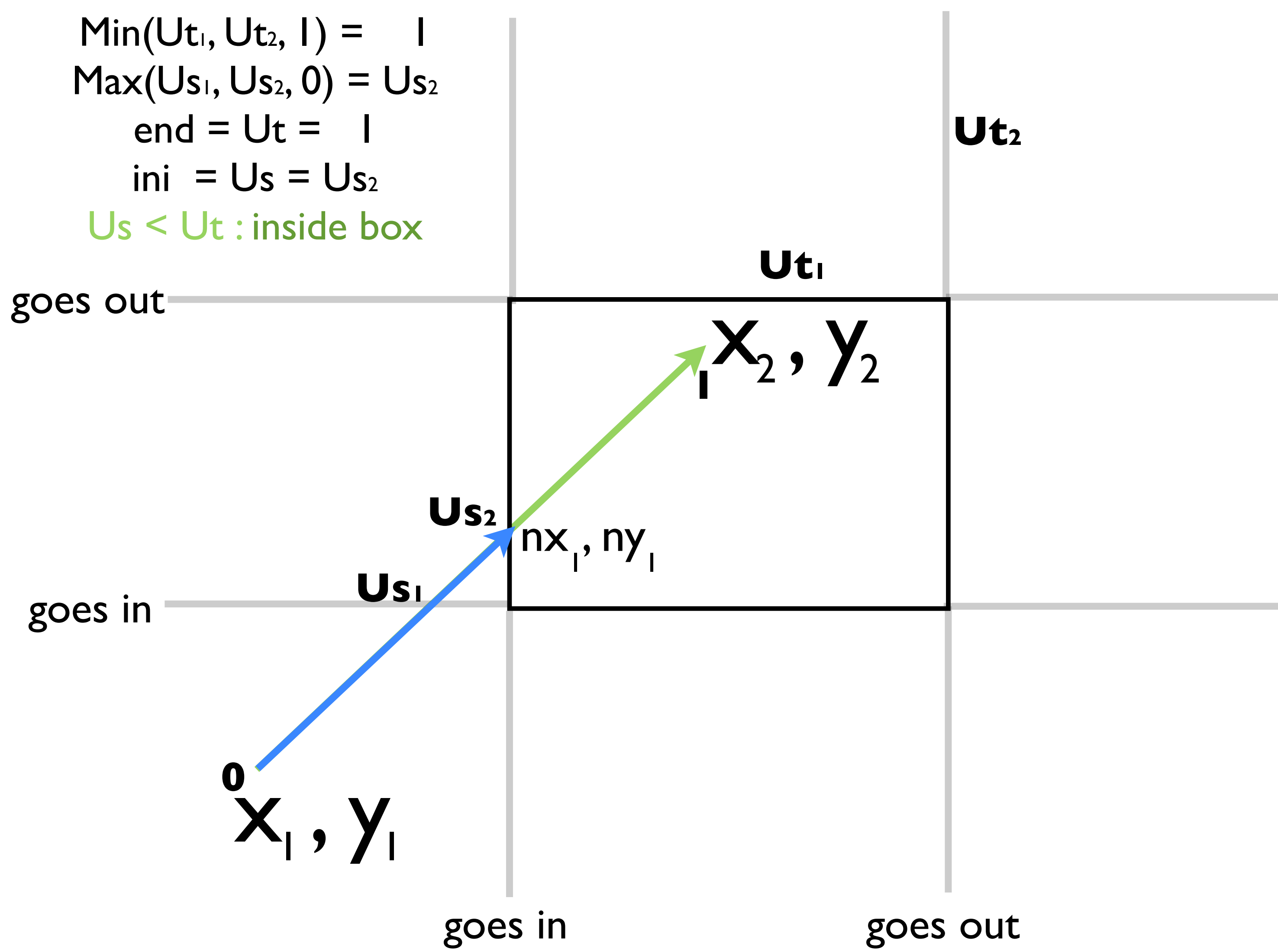
Possible Cases

Diagonal Line completely crossing the Area



Possible Cases

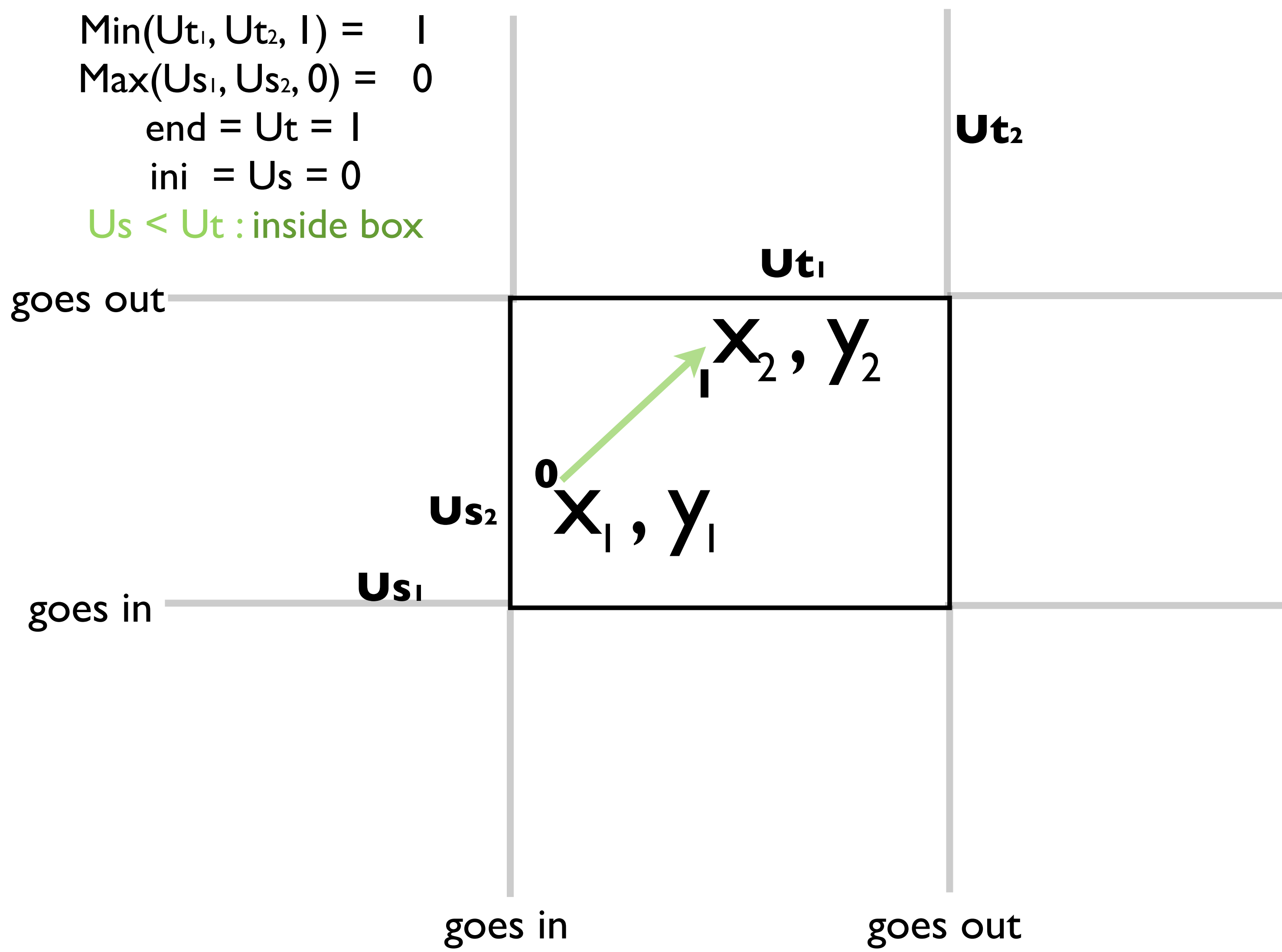
Diagonal Line partially crossing the Area



$nx_2, ny_2 = x_2 , y_2$

Possible Cases

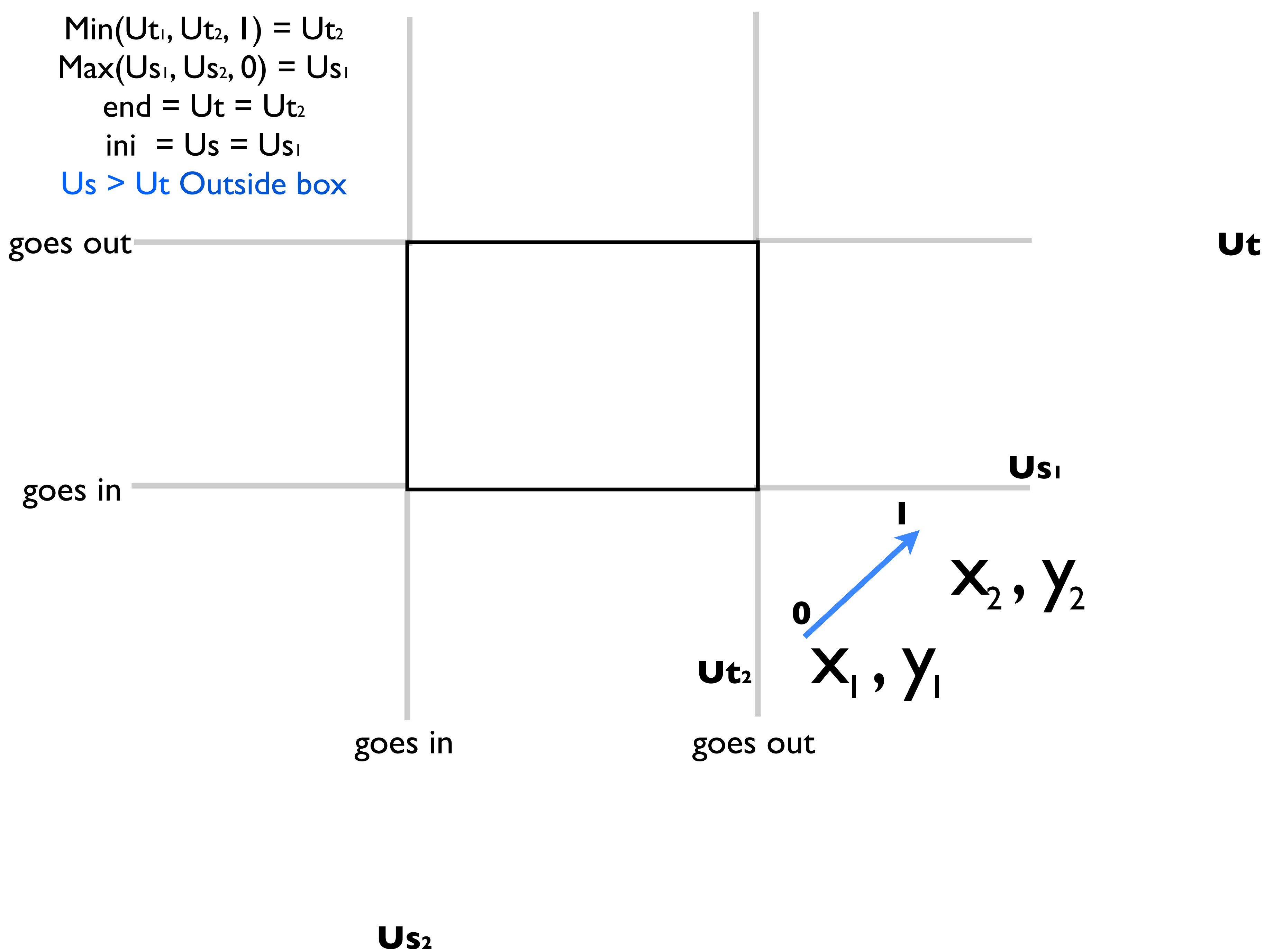
Diagonal Line completely inside the Area



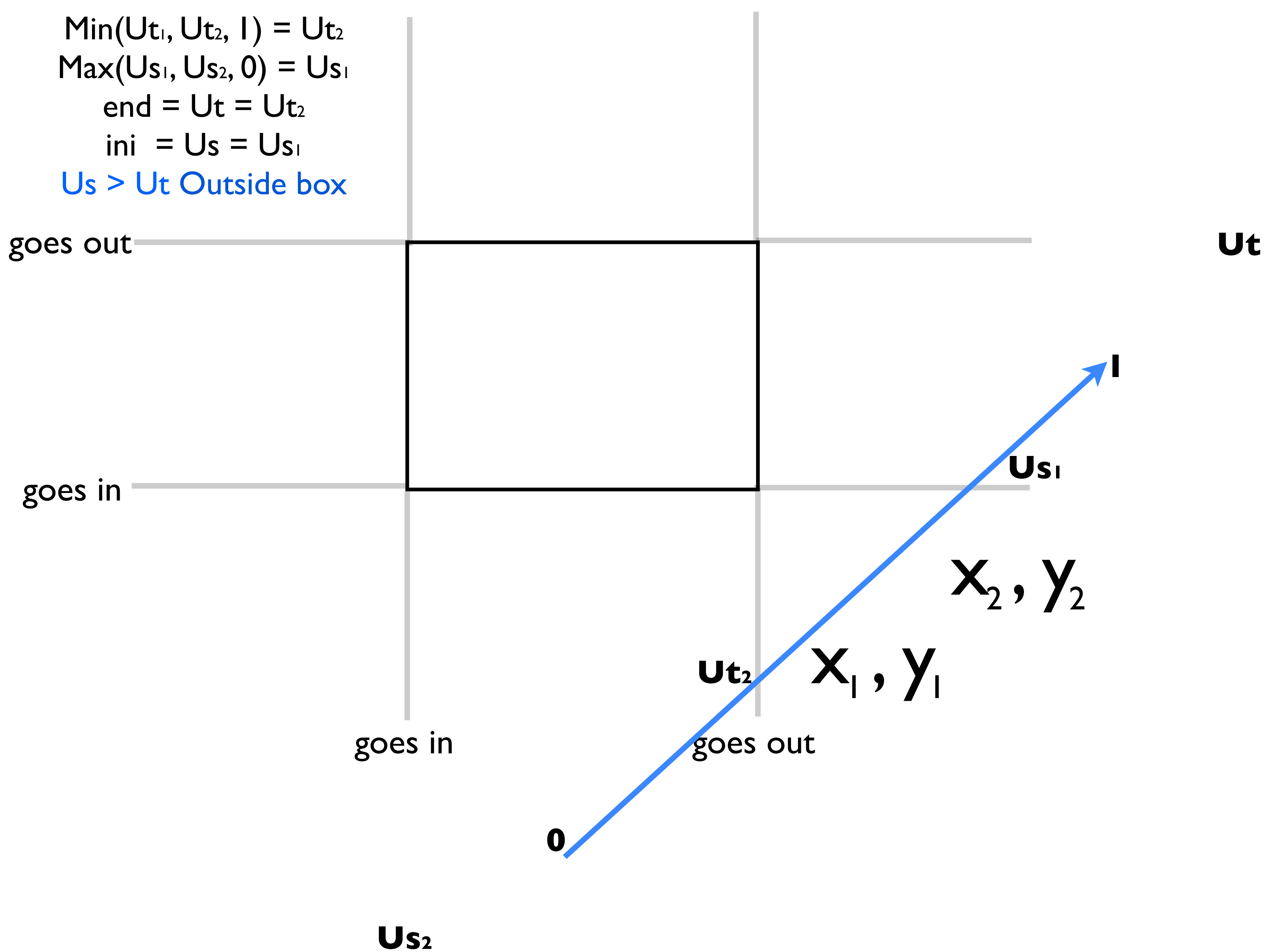
$$x_1 , y_1 = nx_1, ny_1 \qquad nx_2, ny_2 = x_2 , y_2$$

Possible Cases

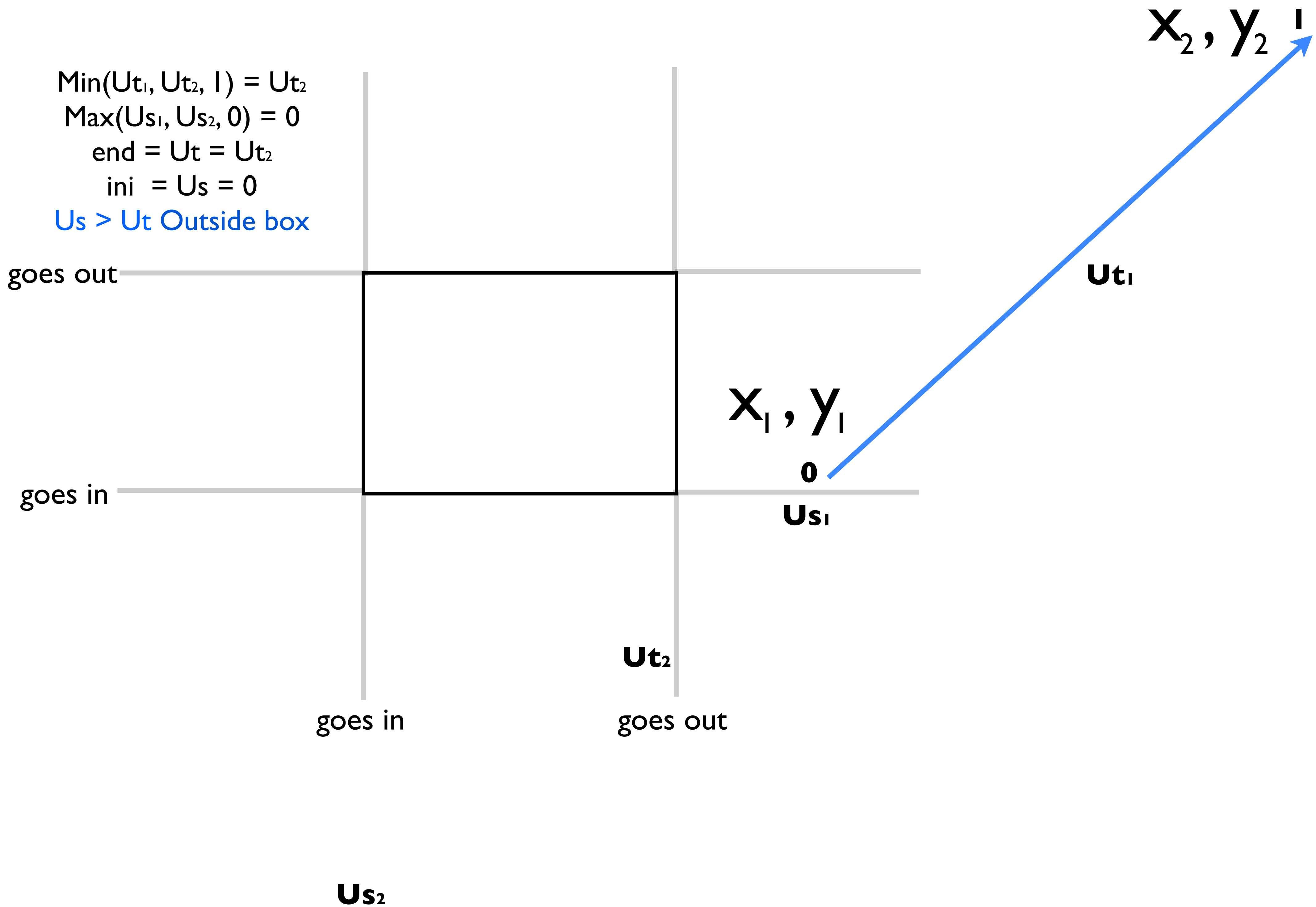
Diagonal Line completely outside the Area



Possible Cases Diagonal Line completely outside the Area



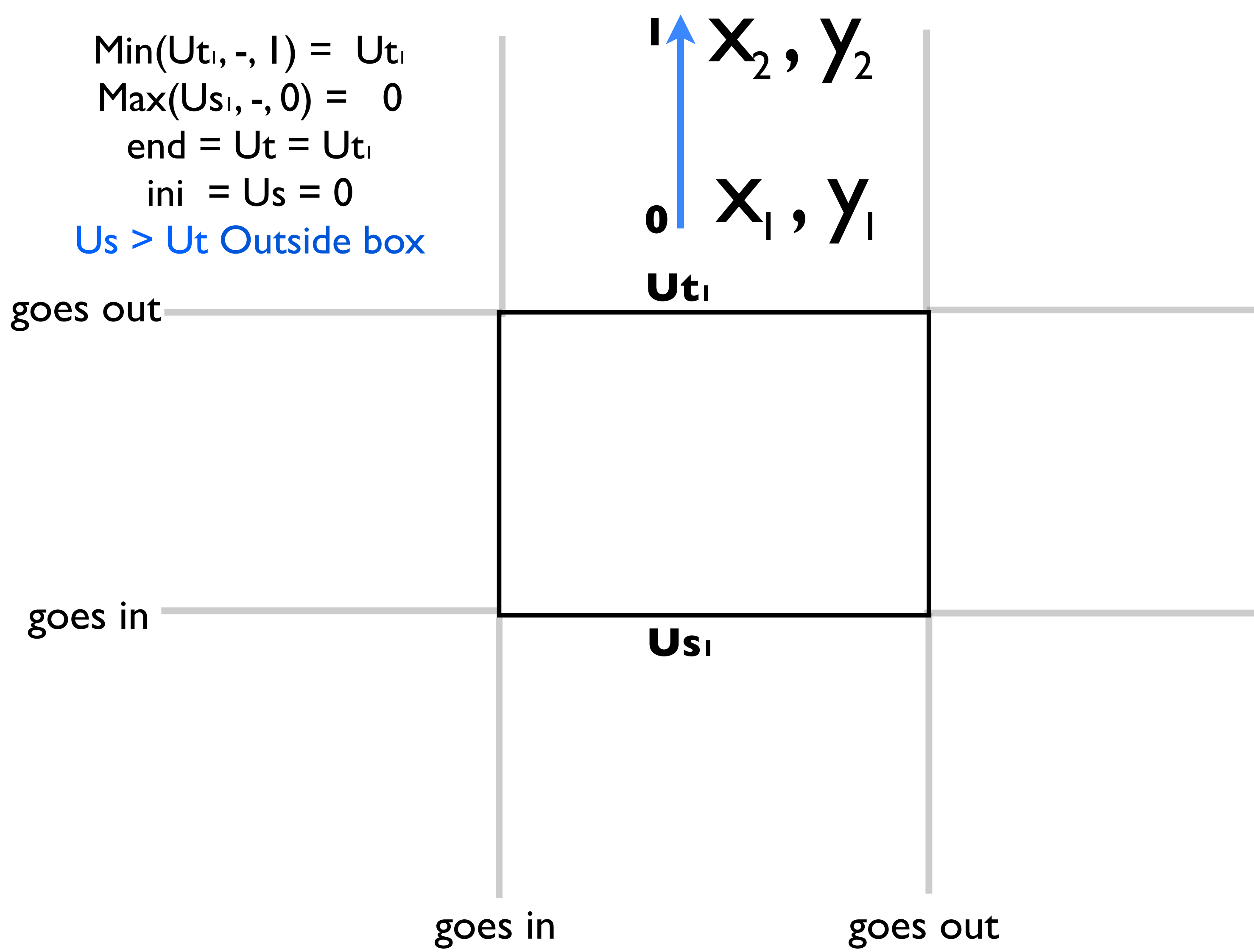
Possible Cases Diagonal Line completely outside the Area



Possible Cases

Line parallel to a boundary
completely outside the Area
but within the parallel boundaries

Us_2 Ut_2 Are 'infinite', there's no way this line will ever touch both the parallel boundaries



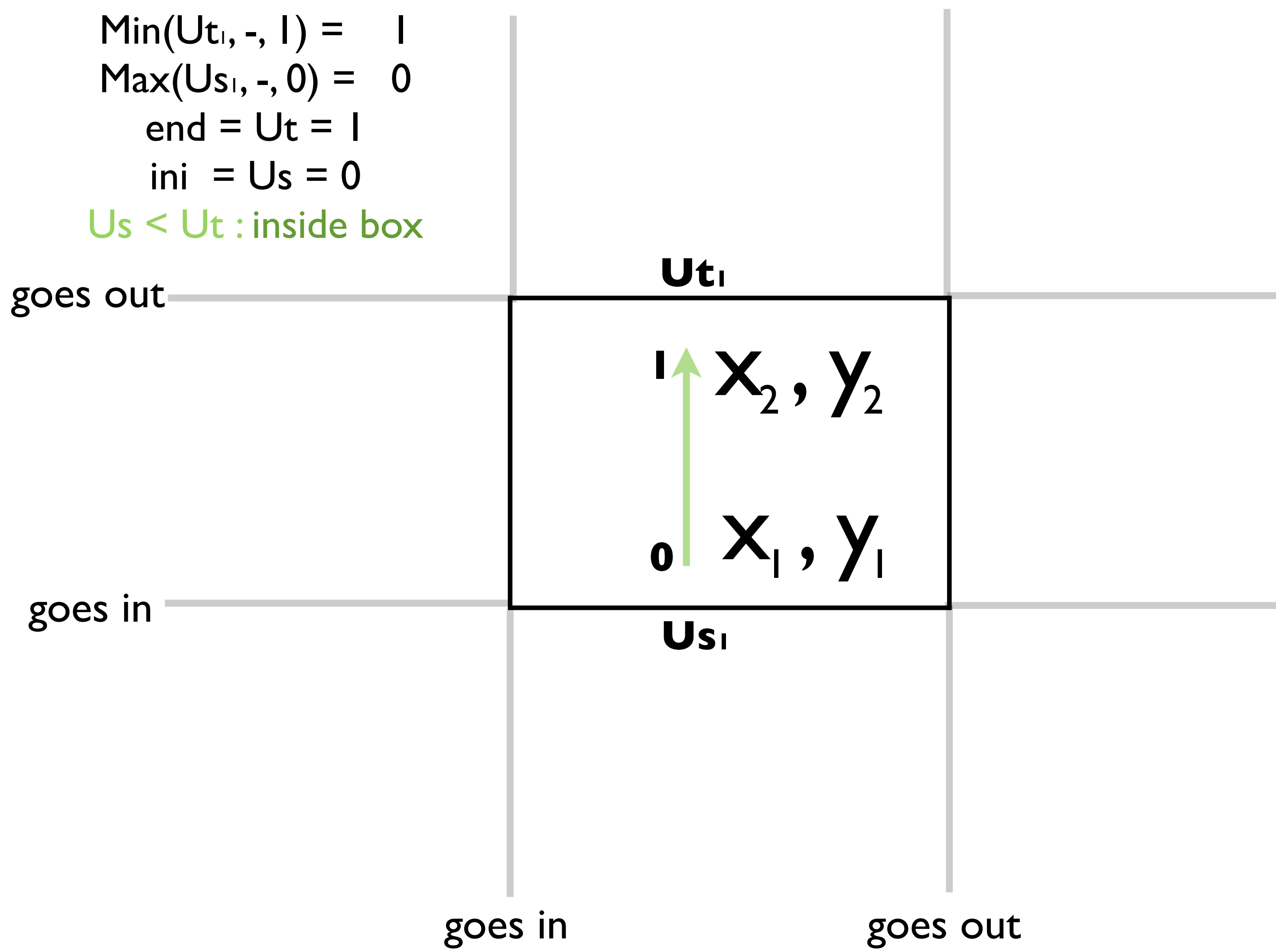
$x_1 , y_1 = nx_1, ny_1$ $\qquad$ $nx_2, ny_2 = x_2 , y_2$

Possible Cases

Line parallel to a boundary completely inside the Area

U_{s2} **U_{t2}** Are 'infinite', there's no way this line will ever touch both the parallel boundaries

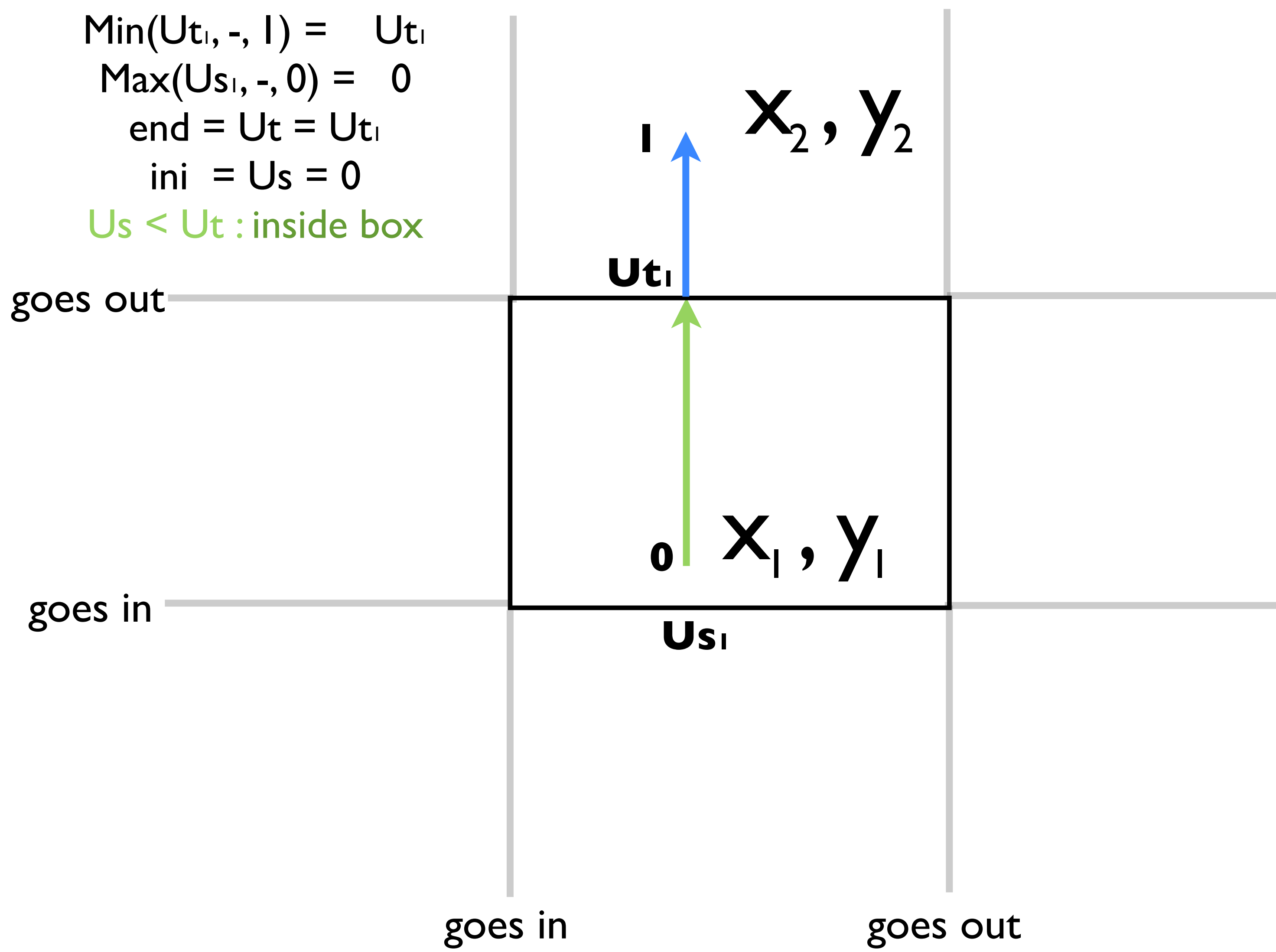
Min(U_{t1}, -, 1) = 1
Max(U_{s1}, -, 0) = 0
end = Ut = 1
ini = Us = 0
Us < Ut : inside box



Possible Cases

Line parallel to a boundary partially crossing the Area

Us_2 / Ut_2: Are 'infinite', there's no way this line will ever touch both the parallel boundaries



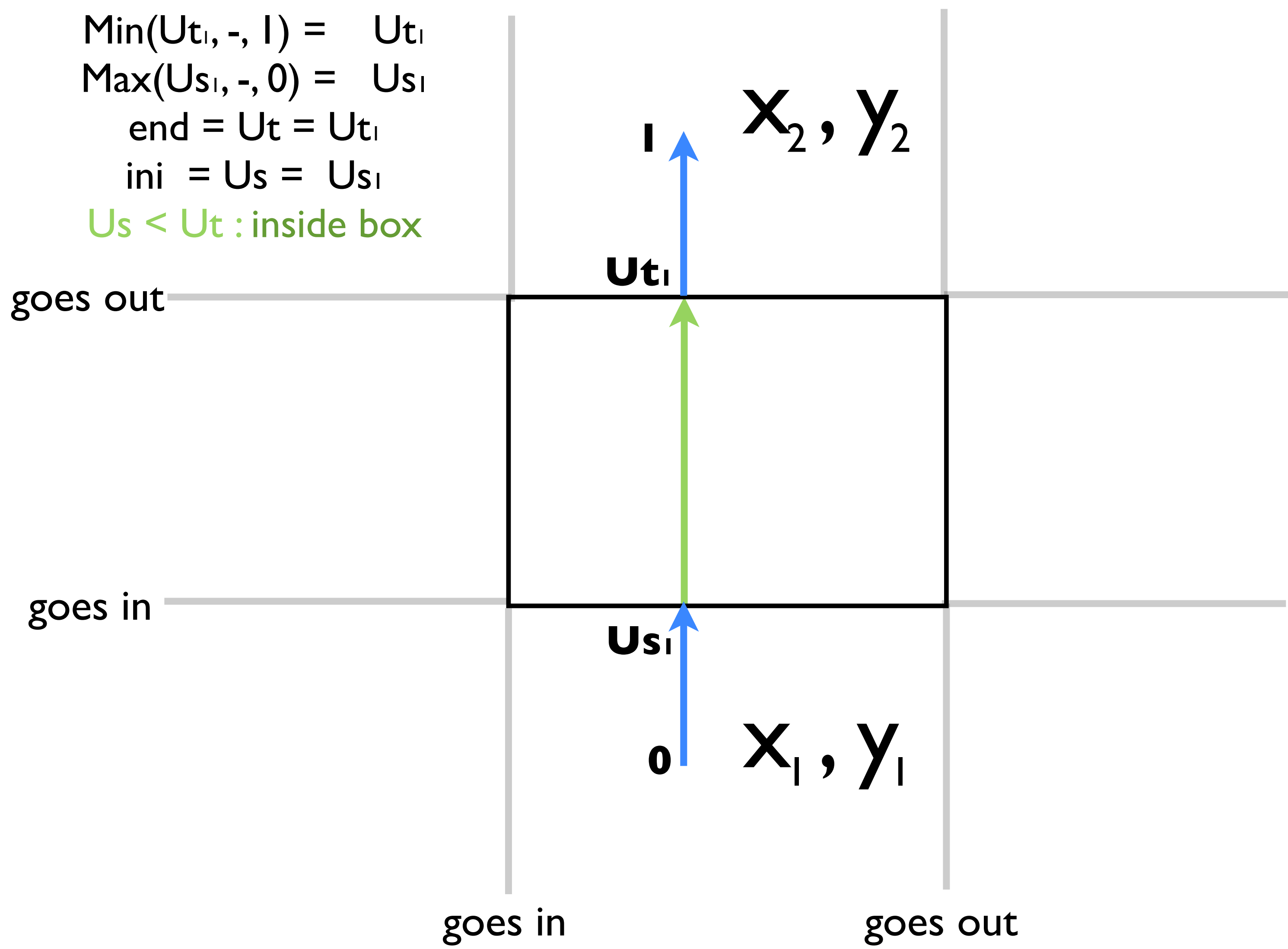
$$x_1 , y_1 = nx_1, ny_1$$

Possible Cases

Line parallel to a boundary completely crossing the Area

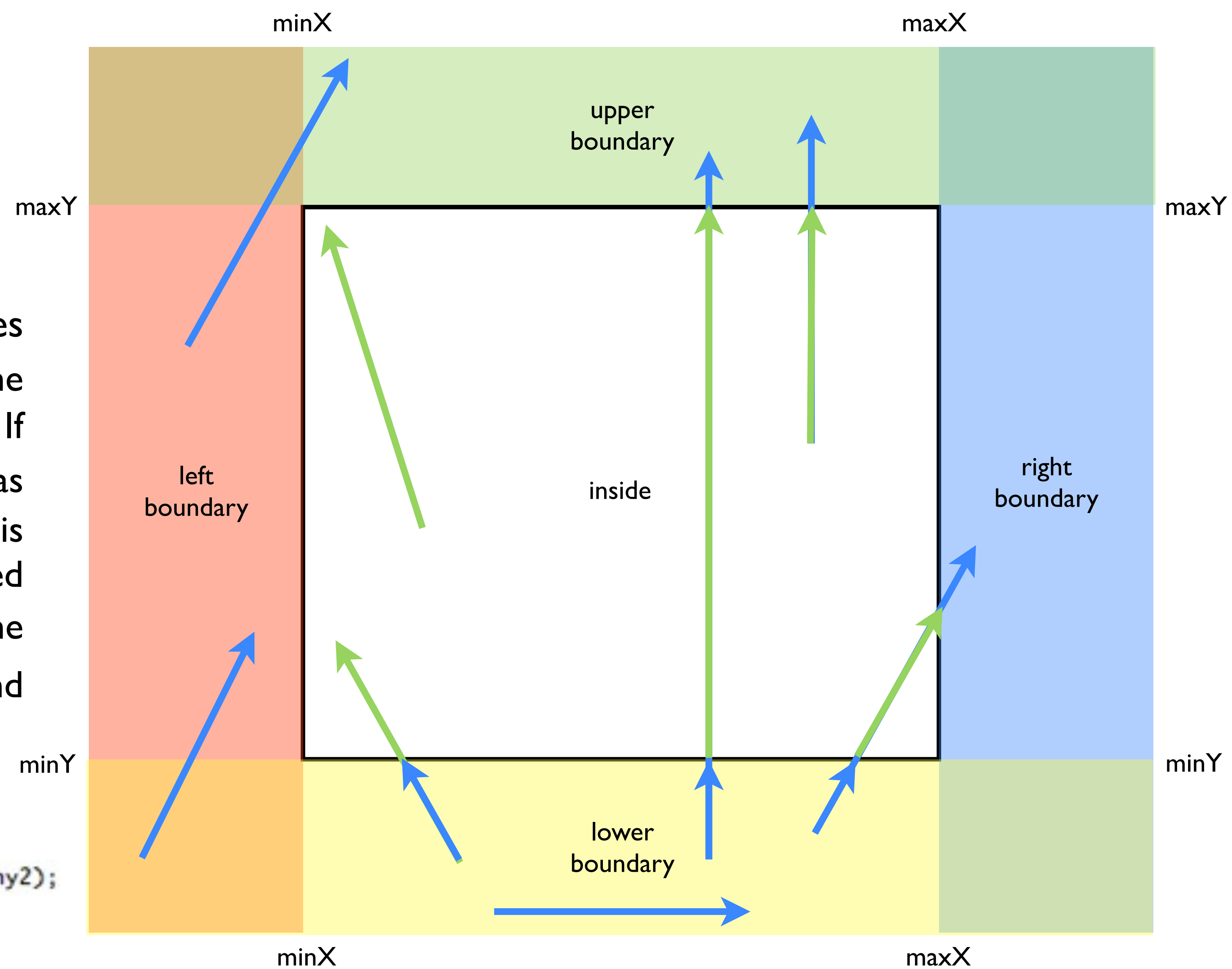
Us_2 Ut_2 Are ‘infinite’, there’s no way this line will ever touch both the parallel boundaries

Min(Ut_1, -, 1) = Ut_1
Max(Us_1, -, 0) = Us_1
end = Ut = Ut_1
ini = Us = Us_1
Us < Ut : inside box



Non-trivial Cases

In a single pass of the algorithm, it will find the values of u for the line to cross a boundary. All boundaries are tested, and the lower ut and bigger us are used.



If if happens that ut < us then for sure the line lies outside. If ut < 1 then the line was clipped at exiting. If us > 0 then the line was clipped at entering. With this we can now make a clipped line from us to ut and the outer lines from 0 to us and from ut to 1.

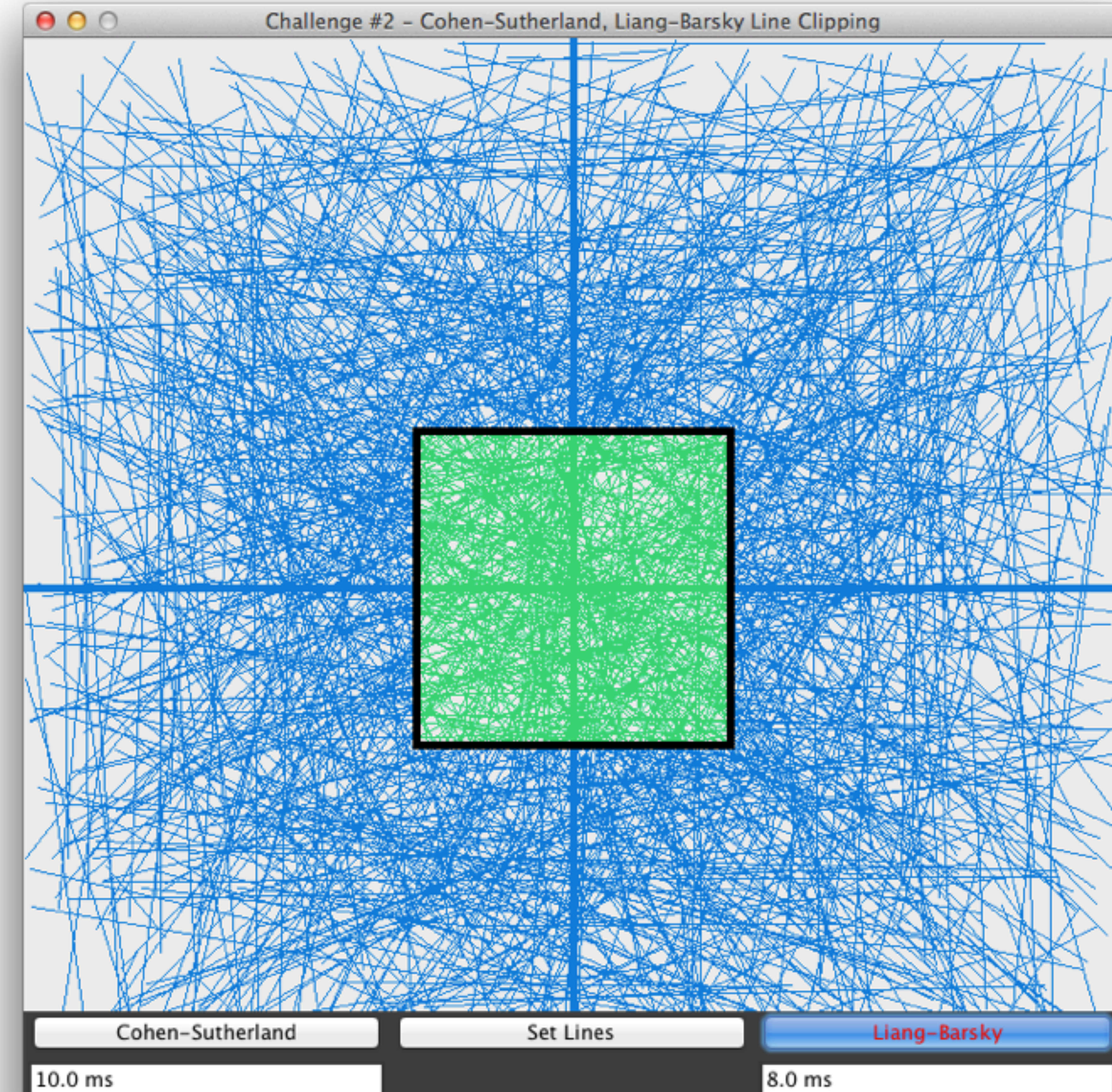
The previous algorithm required up to 3 passes to trim all the parts of the line outside each boundary and get it down to a trivial case. This algorithm in just 1 pass, determines the values of the parametric equation in which the Line is inside the area, and with this values we can immediately obtain the clipped Line and the outer parts of the original Line.

```
if(ini < end){
    l1 = new Line(nx1, ny1, nx2, ny2);
    l1.setType("INNER");
    arr.add(l1);

    if(ini > 0){
        l2 = new Line(x1, y1, nx1, ny1);
        l2.setType("OUTER");
        arr.add(l2);
    }
    if(end < 1){
        l3 = new Line(nx2, ny2, x2, y2);
        l3.setType("OUTER");
        arr.add(l3);
    }
```

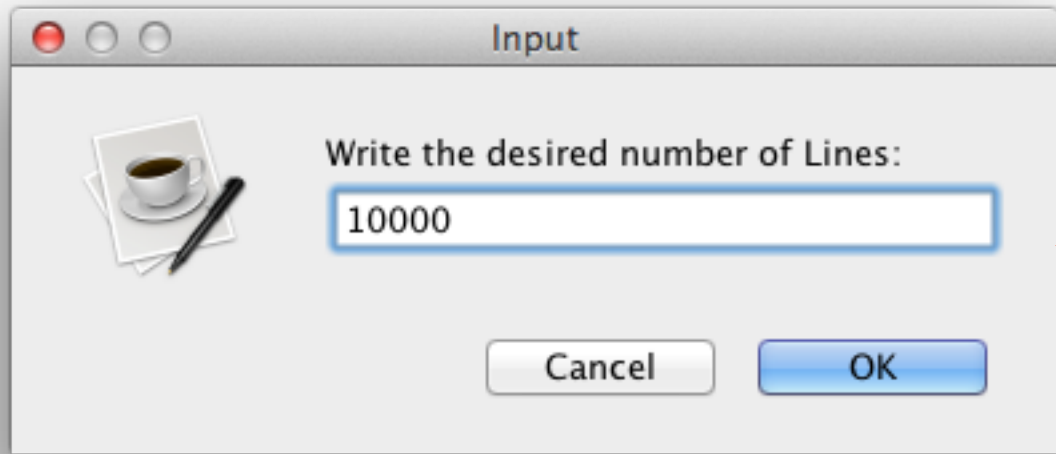
```
else{
    l1 = new Line(x1, y1, x2, y2);
    l1.setType("OUTER");
    arr.add(l1);
}
```

Java Application

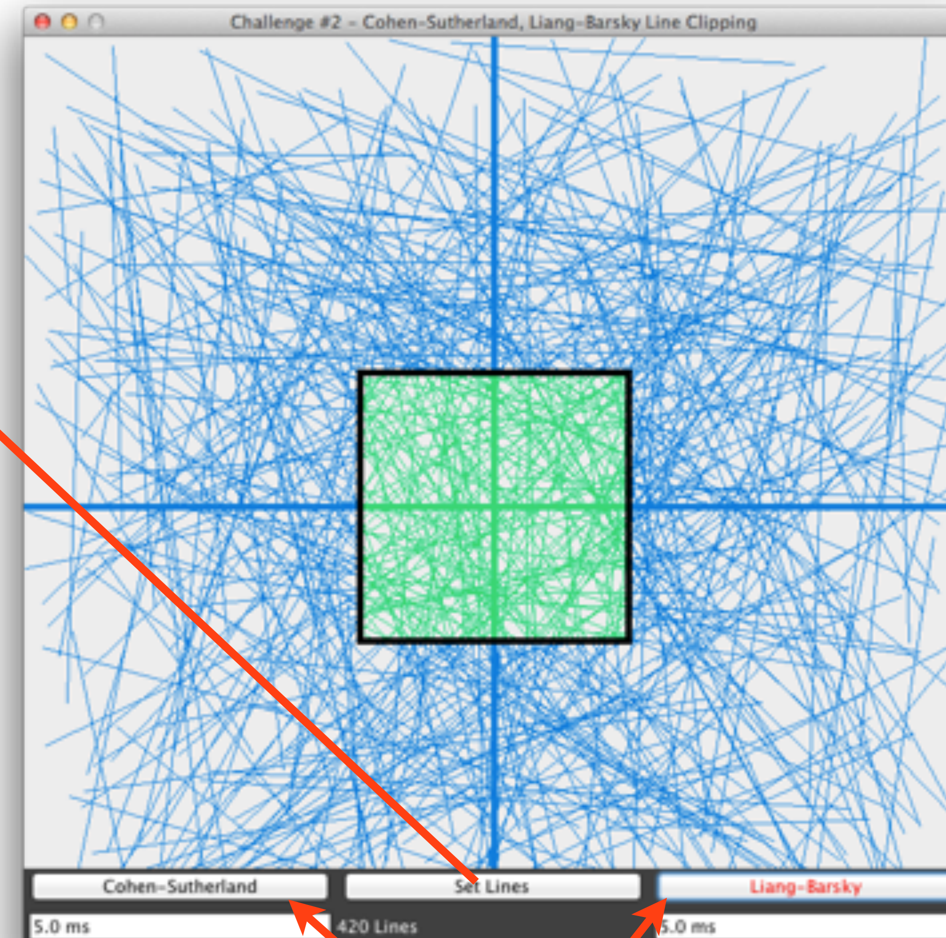


Java Application

By default starts in Cohen-Sutherland Mode

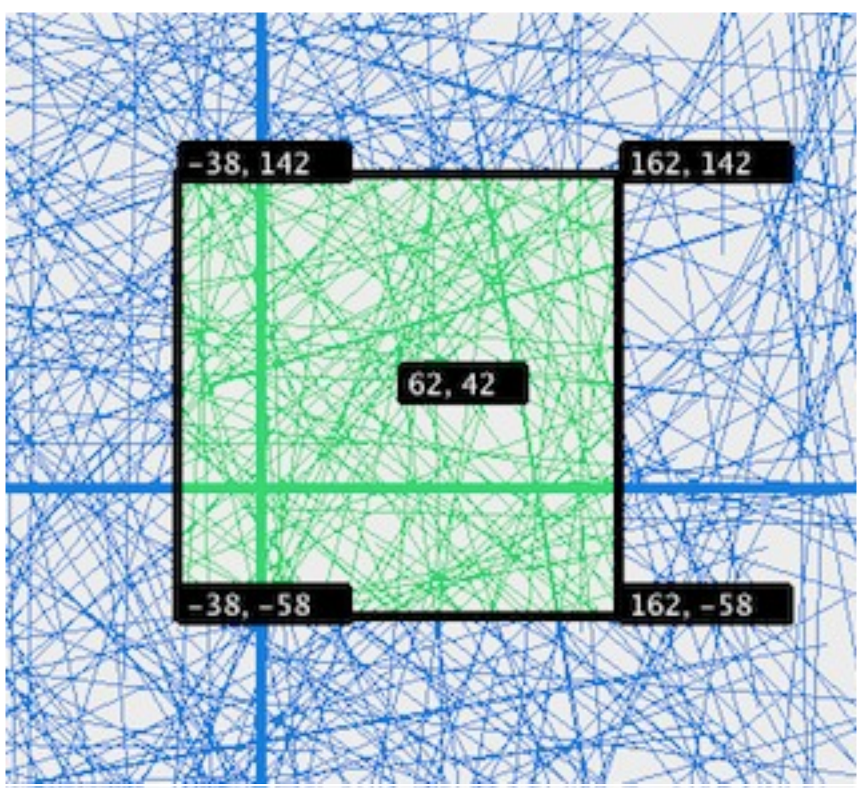


Ask for desired amount of lines to be clipped
(If invalid value is inputted, will be 1000 by default)

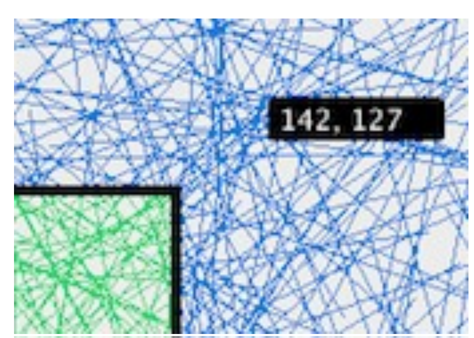


Clicking either will reset clipping-box position and give time for each algorithm, try with different numbers of lines! Moving the box, resulting in re-clipping, will also display times for the selected algorithm.

Clipping-box can be dragged around!



You will see the corner coordinates and the center coordinates



Moving the pointer while not moving the clipping-box will display the pointer coordinates. In this case a buffered image will be used to prevent needlessly re-clipping the lines.

Running Time

For the time we're taking into account only the time it takes to clip a Line (from an existing List of randomly generated Lines) and the time it takes to draw the resulting Lines from the clipping process.

For each Line in the original List of Lines which both algorithms use for a valid comparison

```
if(status == "SUTHER"){
    t1 = new Date().getTime();
    newLines = box.splitCohenSuther(line);
    t2 = new Date().getTime();
}
if(status == "BARSKY"){
    t1 = new Date().getTime();
    newLines = box.splitLiangBarsky(line);
    t2 = new Date().getTime();
}
totalTime += t2 - t1;
```

the algorithms will return a List, and for every Line in that List

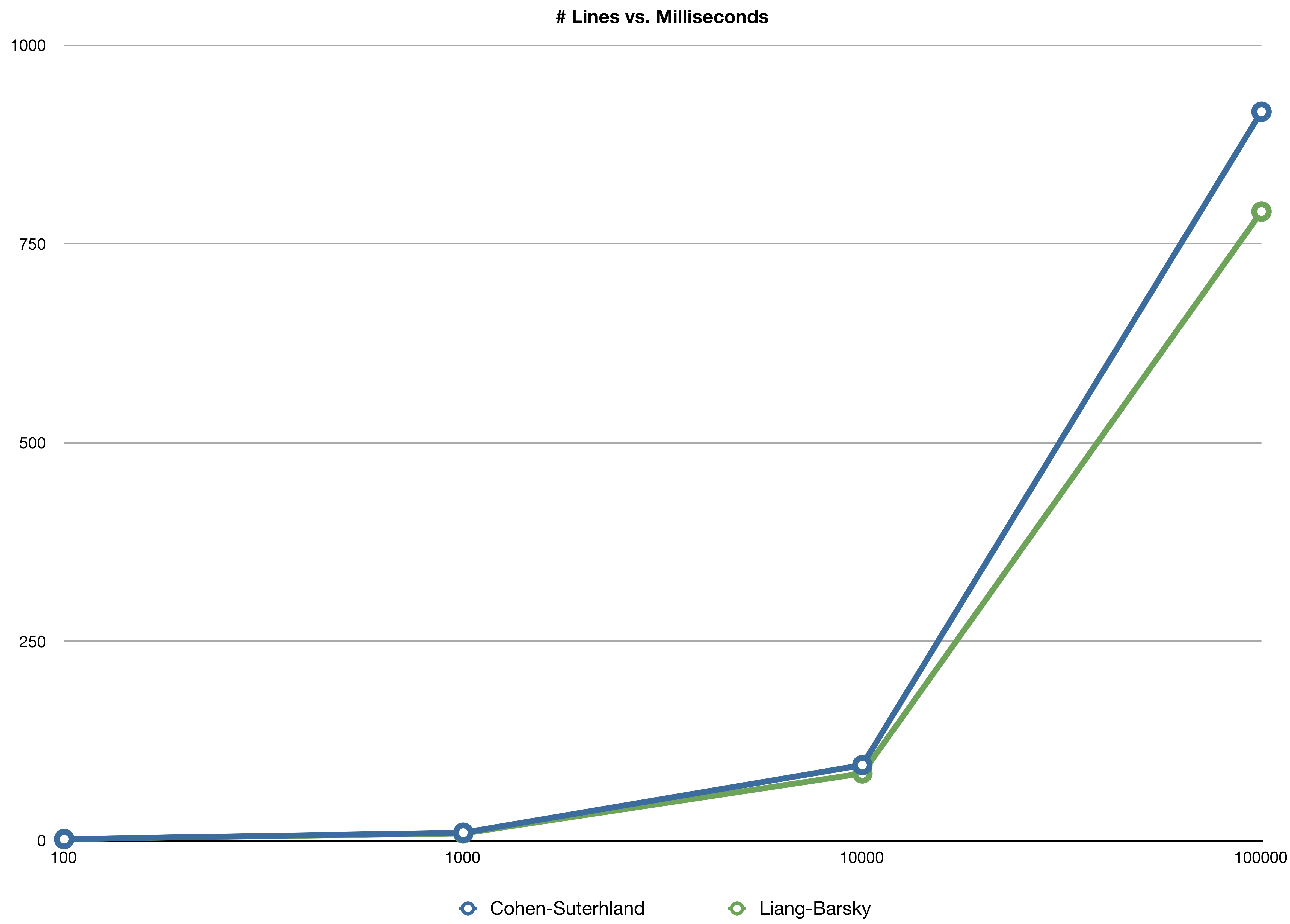
```
t1 = new Date().getTime();
g2d.drawLine(x1, y1, x2, y2);
t2 = new Date().getTime();
totalTime += t2 - t1;
```

totalTime is reseted every time the original List of randomly generated Lines is going to be traversed and displayed when all the Lines are done.

So this pretty much ignores stuff that isn't involved in the clipping and painting of the lines. The resulting time is in Milliseconds. For very small values of Lines to clip, there's not a noticeable difference, and sometimes an algorithm is faster than the other and then slower. The difference in speeds is more noticeable at higher amount of Lines to clip.

	100	1000	10000	100000
CS	2.0	10.0	103.0	910.0
CS	1.0	11.0	89.0	896.0
CS	1.0	9.0	82.0	916.0
CS	2.0	9.0	92.0	950.0
CS	3.0	9.0	90.0	925.0
CS	1.0	10.0	115.0	934.0
CS	1.0	9.0	91.0	882.0
Average	1.57142857143	9.57142857143	94.5714285714	916.142857143
LB	2.0	9.0	80.0	813.0
LB	0.0	6.0	77.0	779.0
LB	2.0	13.0	88.0	814.0
LB	2.0	8.0	77.0	829.0
LB	1.0	8.0	93.0	769.0
LB	2.0	5.0	87.0	759.0
LB	3.0	12.0	87.0	771.0
Average	1.71428571429	8.71428571429	84.1428571429	790.571428571

Running Time



Thanks for your time!

More challenges to follow!