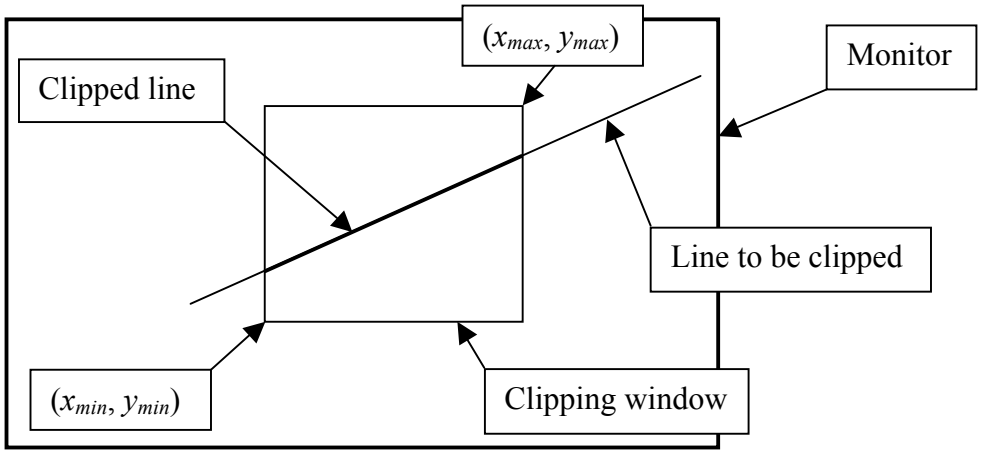
Clipping – Points, Lines, and Polygons
In Aligned Rectangular Windows

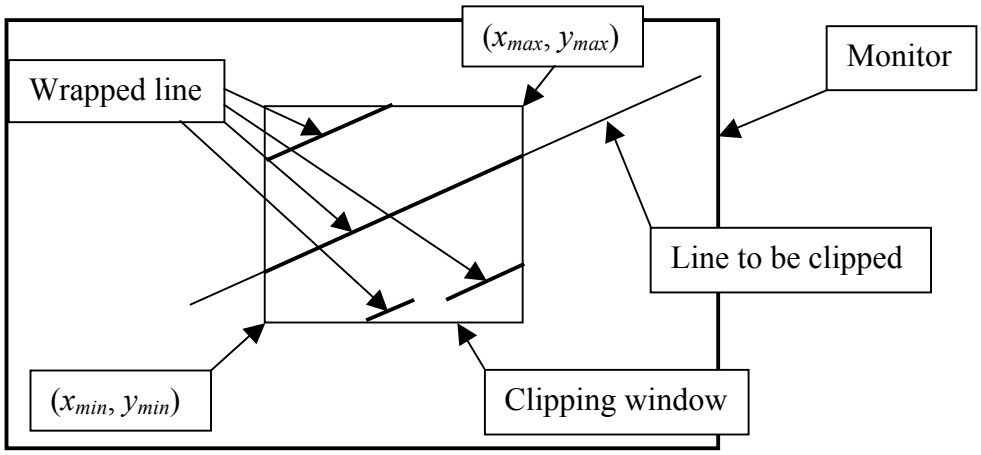
1. Things that Are, and Aren't, Covered Here

In these notes I'll only be discussing clipping relative to aligned, rectangular windows. The window could be a window within the computer display or it could be the whole display. Aligned means that the edges of the window will have constant x (vertical) or y (horizontal) screen coordinates. Put differently, this means that we can fully define the window with two screen coordinates, (x_{min}, y_{min}) and (x_{max}, y_{max}). E.g., the typical situation for a line clip is:



I'll only be looking at three kinds of clipping, point clipping (which is trivial), line clipping, and polygon clipping. Probably the most important thing that I won't be discussing is character clipping. Also I'll only describe one algorithm for each, although in the last section I'll mention some others.

Historically clipping used to be done by the user, not the graphics system. If you didn't you'd get overflow wrapping. So the picture above becomes:



2. Point Clipping

To clip the point (x, y) against the window is trivial, since there isn't a better way to do it than just performing the test $x_{min} \leq x \leq x_{max}$ **and** $y_{min} \leq y \leq y_{max}$ and only display the point if this is true. However isolated points rarely occur in computer graphics, so this is not only trivial but also fairly unimportant.

3. Some Basic Line Segment Math

Most people have been trained that when dealing with line computations it is best to use the slope-intercept form, which is $y = mx + b$. This is rarely the best approach in graphics. The slope-intercept form describes an infinite line, whereas what we'll have in graphics is a finite line segment between two points (x_0, y_0) and (x_1, y_1). Most of the time it is far better to use the parametric form of the line, where both x and y are defined in terms of a parameter t. E.g., the line above will be described using two equations:

$$x(t) = x_0 + (x_1 - x_0) * t$$
$$y(t) = y_0 + (y_1 - y_0) * t$$
$$\text{where } 0 \le t \le 1$$

Alternatively we can just say

$$P(t) = P_0 + (P_1 - P_0) * t \quad \text{where } 0 \le t \le 1$$

Note that when $t = 0$, $P = P_0$, and when $t = 1$, $P = P_1$. Also, if we look at the slope of the line,

$$\frac{dy}{dx} = \frac{dy/dt}{dx/dt} = \frac{y1 - y0}{x1 - x0},$$

it is constant, so this is a straight line segment between P_0 and P_1.

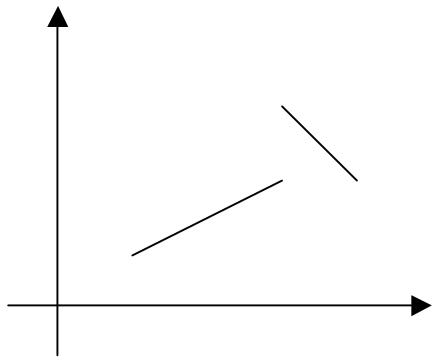
To use this form, say we want to know whether the line segment from (1, 1) to (3, 2) intersects the line segment from (3, 3) to (4, 2). The line equations (being careful to use different parameter names for each line) are

$$x(t) = 1 + 2t \qquad x(s) = 3 + s$$
$$y(t) = 1 + t \qquad y(s) = 3 - s$$

The segments intersect when they have the same x and y values, which gives

$$1 + 2t = 3 + s$$
$$1 + t = 3 - s$$

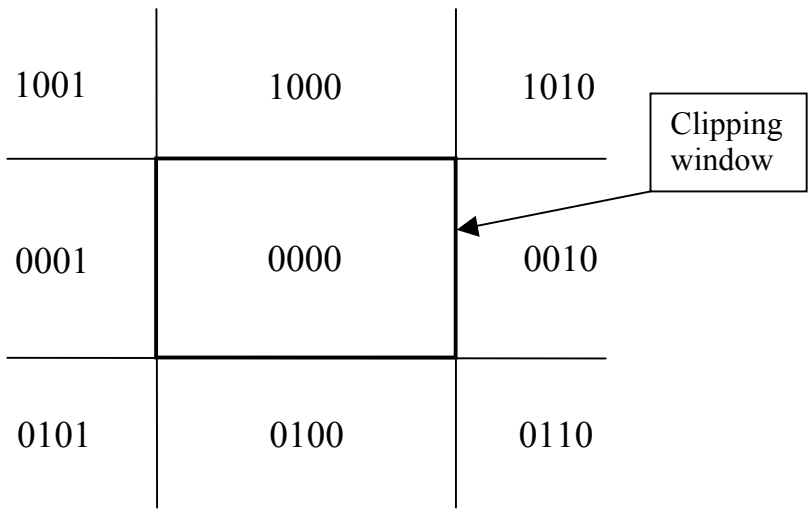
These simultaneous equations have the solution $t = \frac{4}{3}$ and $s = \frac{2}{3}$, and so the line segments don't intersect since both parameters aren't between 0 and 1. Looking at a graph of this, these values make sense.



The major advantage of this approach over the traditional slope-intercept approach is that it includes the fact that these are segments, not lines. With slope-intercept we'd have found that these lines intersect since they aren't parallel, but would then have had to check to see where the intersection was relative to the segments.

4. Cohen-Sutherland Line Clipping

There have been a number of algorithms developed for line clipping, so I'll describe the most popular, Cohen-Sutherland. The underlying idea is to first divide 2D space up into nine regions, shaped like an infinite tic-tac-toe board, and assign four-bit codes to each region, where the center is the clipping window, as shown below:



If you look carefully at the pattern of the four bits $b_3b_2b_1b_0$, you'll see that they have a pattern. If a point (x, y) satisfies $x < x_{min}$, then $b_0 = 1$ (where b_0 is the rightmost bit) otherwise $b_0 = 0$, and the same edge rules apply to the other four edges. I.e.,

$$b_0 = x < x_{min}$$
$$b_1 = x > x_{max}$$
$$b_2 = y < y_{min}$$
$$b_3 = y > y_{max}$$

There are times when including the edges in the clip makes sense, and then the comparison operators will change with $\leq$ replacing $<$ and $\geq$ replacing $>$.

Given any line segment from $P_0 = (x_0, y_0)$ to $P_1 = (x_1, y_1)$, the algorithm first assigns these four-bit codes to P_0 and P_1 based on their locations in this grid. There are three main steps to the algorithm:

1. If code(P_0) = code(P_1) = 0000, then both endpoints are in the window, so display the line without clipping and exit.

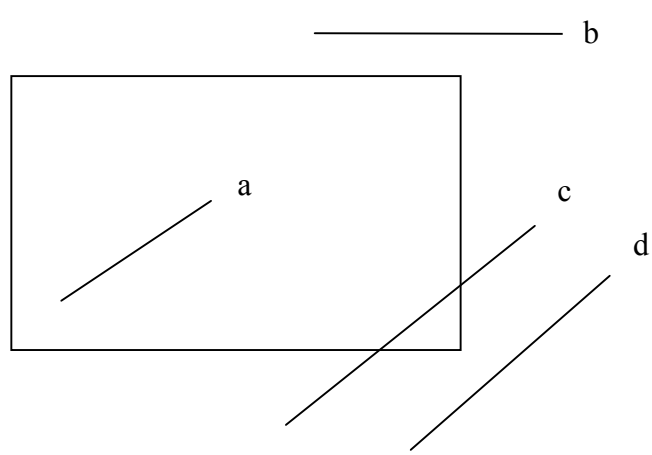
2. If (code(P_0) **and** code(P_1)) $\neq$ 0000, where **and** is bitwise and, then this line is completely outside the window, so reject it and exit.
3. Otherwise, we have two choices on how to handle the current line, which might or not cross the window. The two choices, which we'll discuss further below, are to either divide it into two equal pieces and run the algorithm again, or to compute edge intersections with the appropriate edges (they can be found through (code(P_0) **or** code(P_1))) which gives the clipped line.

So the goal of the algorithm is to rapidly draw lines that are fully in the window or can be easily detected as being fully outside the window, and then handle the other cases with more work in step 3.

Step 1 is obvious.

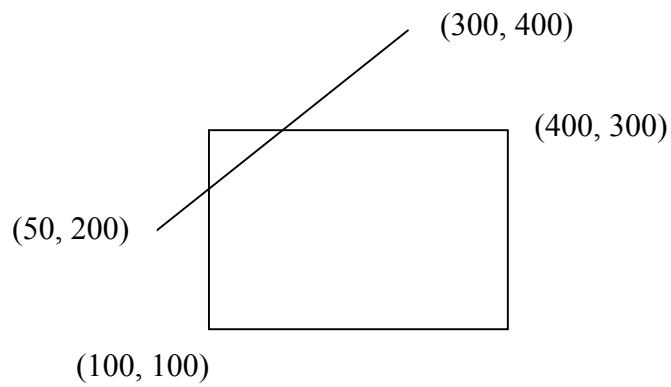
Step 2 uses the fact that if the **and** of the two four-bit codes is not 0000, then somewhere there must be a 1 in the same bit location in both codes. Say, for example, that it is bit b_0 that is a 1 in both. This means that both points are to the right side, outside the window, so cannot intersect the window. Obviously similar arguments apply to the other three bits.

Consider the diagram below, the codes on the endpoints of line *a* are both 0000 and so the line will be drawn in step 1. The endpoint codes of line *b* are 1000 and 1010, so since their **and** is 1000 the line will be rejected in step 2. The only difficulty occurs with lines like *c* and *d*, which share the same endpoint codes (0100 and 0010), but one needs clipping and the other can be rejected.



Step 3 has to handle cases like this. There are two approaches. The simplest in terms of hardware accelerators is to just divide the line into two, and run

the algorithm again on both halves. In some cases this will immediately terminate, but in others it will lead to repeated calls. The usual rule is to terminate this after at most 11 calls, since by then you will be at pixel resolution on a screen with approximately $1K \times 1K$ resolution and a longest possible (diagonal) line. The second approach is to compute intersections with the edges using the parametric form of the line segment, which will immediately determine whether the line misses the window or, if it crosses the window, will give the piece to display. Using bitwise or on the codes will determine which edge intersections need to be considered. E.g., for *c* and *d* the endpoint codes were 0100 and 0010. So do intersections against the edges related to bits 1 and 2 of the code. E.g., consider the two lines below:



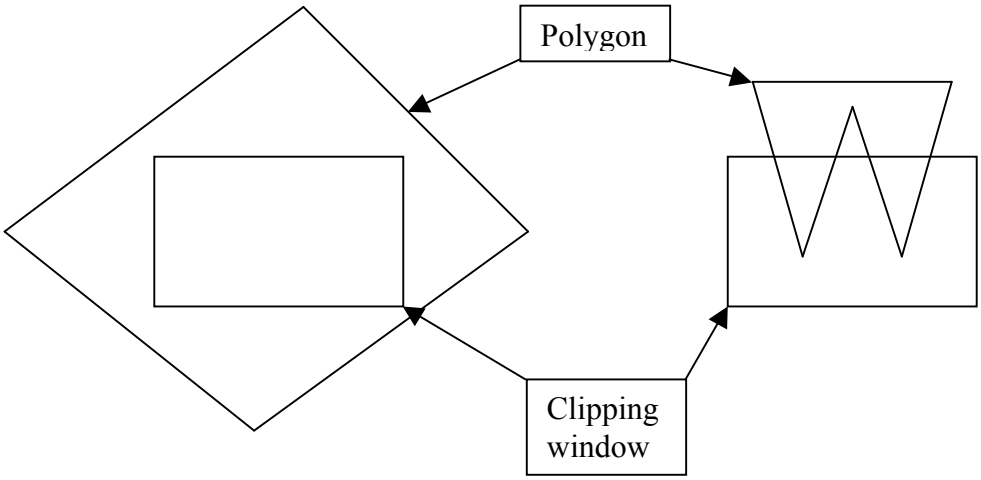
The line from (50, 200) to (300, 400) has parametric equation

$$x(t) = 50 + 250t$$
$$y(t) = 200 + 200t \quad 0 \le t \le 1$$

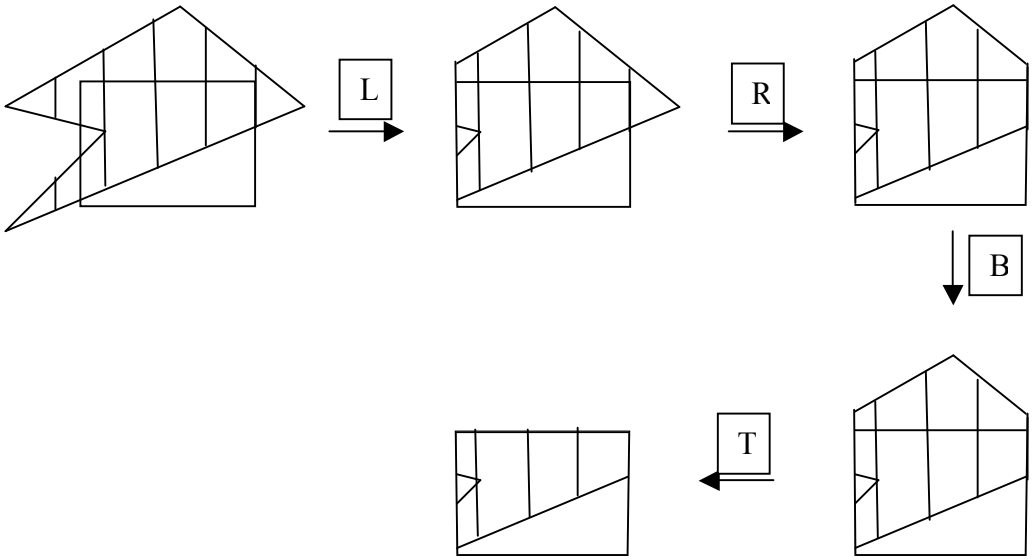
The endpoint codes, 0001 and 1000, tell us that we need to compute intersections with the left and top edges, which have equations $x = 100$ and $y = 300$. Substituting $x = 100$ into the first equation gives $t = 0.2$, and using this in the second equation gives $y = 240$. So the first intersection is at the point (100, 240). Similarly, intersecting with $y = 300$ gives $t = 0.5$, and so the intersection point is at (175, 300). The clipped line is now just the line between (100, 240) and (175, 300).

5. Sutherland-Hodgman Filled Polygon Clipping

It might appear that we can solve polygon clipping by just using line clipping on the edges, but unfortunately this doesn't work. E.g., consider the case where a red polygon completely covers the viewing window. If we clip its edges they will disappear, and the polygon will vanish. There is also a problem if the clipped polygon fragments into multiple small pieces. E.g., both of the two cases below cause problems.

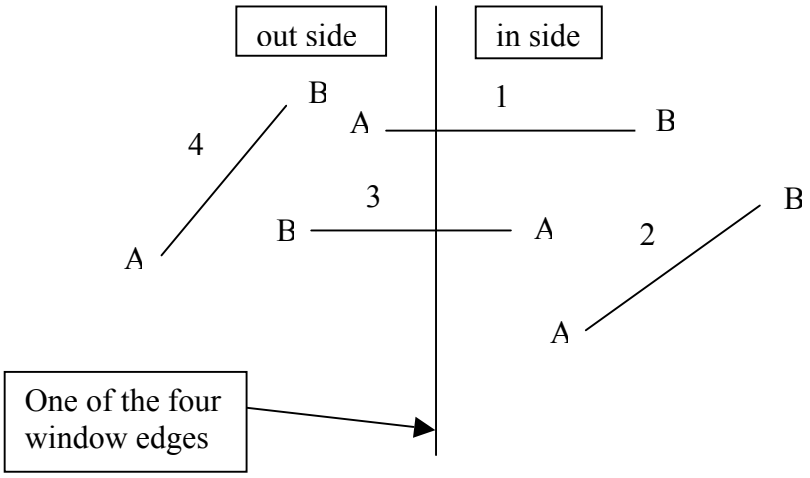


Sutherland-Hodgman works by taking the polygon description as a list of vertices, and then building a new polygon description where the part outside one edge is discarded, then repeats with a second edge, then a third, and finally the remaining edge, so that the final polygon is strictly inside the clipping window. This process is shown on the next page. In this picture we have a polygon shaded with vertical lines and a clipping window. First we clip against the left window edge, then against the right edge, then against the bottom edge, and finally against the top edge, leaving the clipped polygon shown.



The goal is to do these edge clipping operations as efficiently as possible. The Sutherland-Hodgman algorithm does this by looking at the relationship between individual polygon edges, as we go around the polygon, and the current window edge, where the current edge is first the left, then the right, then the bottom, and finally the top. Obviously any edge order will work, but I'll stick with this one for consistency.

So, say we have a polygon edge from vertex A to vertex B and we are seeing how it relates to the window side (the in side) and the non-window side (the out side) of the current window edge. There are four possibilities, below:

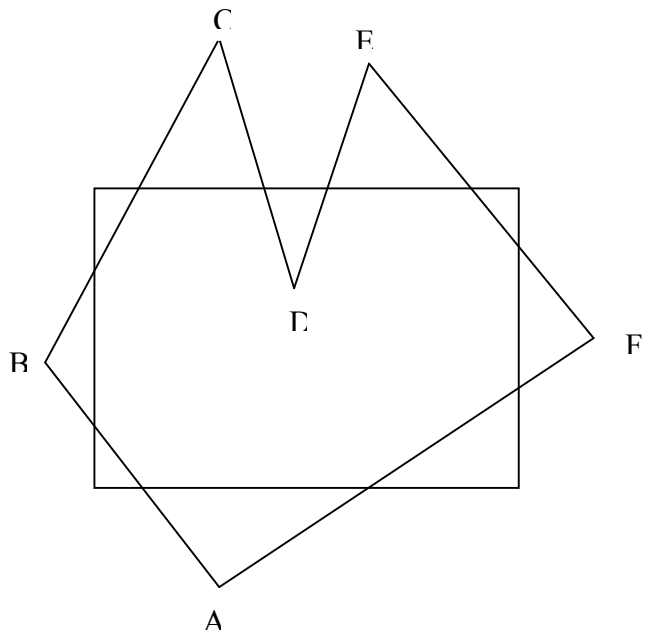


I.e., case 1 is when the polygon edge is going from outside to inside, case 2 from inside to inside, case 3 from inside to outside, and case 4 from outside to outside. The algorithm takes a polygon description as an ordered list of

vertices that go around the polygon, and outputs a new polygon list clipped to this list. The output rules that it uses to do this are:

Case 1: output I, B, where I is the intersection of the line and edge.
Case 2: output B.
Case 3: output I, where I is the intersection of the line and edge.
Case 4: output nothing.

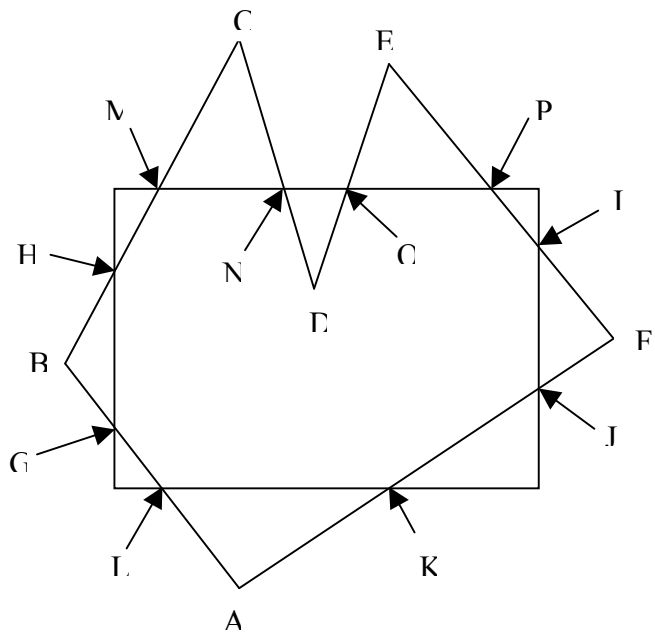
To see how this works, see the polygon and clipping window below:



The polygon is described by the vertex list ABCDEF, where there is the assumption that it wraps back from F to A. Clipping against the left edge first, where the in side is to the right and the out side is to the left, we run the algorithm from A to B, then B to C, …, and finally F to A.

A→B is in side to out side, case 3, so output the intersection G shown in the figure on the top of the next page. (I'll use the next available letter for new vertices.)
B→C is out side to in side, case 1, so output H (intersection) and C.
C→D is in side to in side, case 2, so output D.
Similarly, D→E, E→F, and F→A are all in side to in side relative to the left edge, so will output E, F, and A.

The output list from this is GHCDEFA, which is our first new polygon, which will be used for the next pass, which is clip against right edge.



Since we are clipping against the right edge, the in side is now to its left and the out side to its right. So G→H, H→C, C→D, and D→E are all in side to in side, and will output H, C, D, and E, respectively. E→F is in to out, so the output is the intersection, I. F→A (out to in) outputs J and A, and A→G outputs G. So the new polygon is HCDEIJAG.

Clipping against the bottom, A is the only point on the out side, and so the output will be CDEIJKLGH.

Finally, clipping against the top, C and E are on the out side, so we get our final polygon NDOPIJKLGHC, which is clipped against the window.

6. Other Line and Polygon Clipping Systems

The Liang-Barsky method is based on the same approach as Cohen-Sutherland, but takes advantage of some efficiencies that can be made using parametric equations. The Nicholl-Lee-Nicholl method is quite a bit more complicated than the Cohen-Sutherland based systems, and is based on increasing the complexity of the region structure. We will discuss both of these further in CS 525.

Weiler-Atherton is the main competition for Sutherland-Hodgman for clipping polygons. The algorithm, which we'll also look at in more detail in CS 525, traces around the polygon looking for intersections with the clipping polygon edges and ultimately jumping to reentry points after it leaves the clipping area.