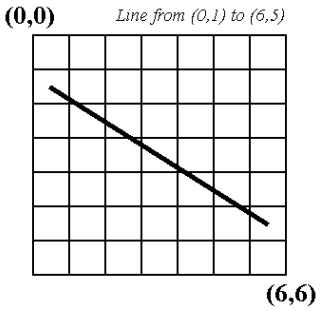
Drawing Lines in 2 Dimensions

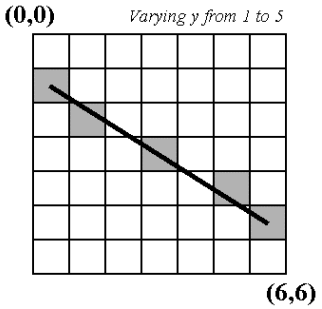
Drawing a straight line (or an arc) between two end points when one is limited to discrete pixels requires a bit of thought.

Consider the following line superimposed on a 2 dimensional 7x7 pixel grid –

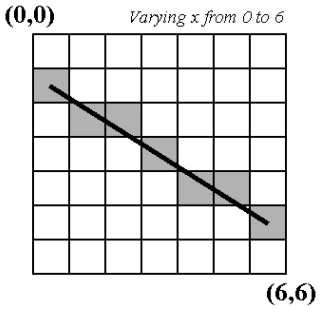


The pixels are indexed with the usual screen convention of ***x*** running from left to right and ***y*** running from top to bottom. Which pixels should be set in order to represent the line?

If we increment our way vertically downwards through the ***y*** values, setting the pixel in the appropriate ***x*** column as we go, we get –



The pixellated line has gaps in it. If, on the other hand, we increment our way horizontally across through the ***x*** values, setting the pixel in the appropriate ***y*** scan-line as we go, we get –



This is clearly a much more satisfactory solution.

We note, however, that had the line been longer in ***y*** than in ***x*** then the first solution would have delivered the better result. The magnitude of the slope of the line is the key here. If the ***x*** range exceeds the ***y*** range then the magnitude of the slope is less than 1 and we should increment our way through ***x*** (and vice versa).

In fact, there are 4 cases to consider when selecting which pixels should be used to construct a continuous line –

1. Positive slope less than 1
2. Positive slope greater than or equal to 1
3. Negative slope less than or equal to -1
4. Negative slope greater than -1

If we know the end points of the line to be drawn we can always calculate the slope –

$$m = \frac{y_e - y_s}{x_e - x_s}$$

A positive slope (in screen co-ordinates) means that the line is monotonic increasing (as ***x*** increases so does ***y***) whilst a negative slope indicates the opposite (as ***x*** increases ***y*** decreases).

The intercept, where the line meets (or would meet if long enough) the ***y*** axis is also readily obtained –

$$c = y_s - m \cdot x_s$$

Of course, this relationship holds for all pairs of ***x*** and ***y*** values on the line so we can determine a ***y*** value for any ***x*** value and vice versa –

$$y_k = c + m \cdot x_k$$

or,

$$x_k = \frac{y_k - c}{m} = \frac{y_k}{m} - \frac{c}{m}$$

Now, if we consider only *changes* in ***x*** and ***y*** we don't need to worry about the intercept except when determining the first pixel to be plotted –

$$\Delta y = m \cdot \Delta x$$

and,

$$\Delta x = \frac{\Delta y}{m}$$

We shall consider two algorithms for drawing a 2D line on a display screen based on the above principles; the *Digital Differential Analyser* and *Bresenham's* algorithms.

The Digital Differential Analyser (DDA) Algorithm

The DDA is a scan-conversion algorithm which samples a line at unit intervals along one axis and determines integer values nearest the line for the other axis.

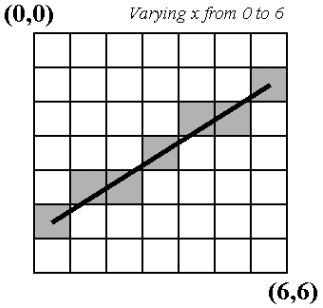
Consider the line we have been looking at. The DDA will sample along the ***x*** axis in this case determining ***y*** values as it goes –

$$y_{k+1} = y_k + m$$

If our line had a slope whose magnitude was greater than 1 (45º) then the DDA would need to sample along the ***y*** axis determining ***x*** values using –

$$x_{k+1} = x_k + \frac{1}{m}$$

These equations will work whether the slope is positive or negative. When the slope is negative the ***x*** and ***y*** values will vary inversely but ***m*** will have the opposite sign so all will be well –



Note that ***m*** is not necessarily an integer, so the DDA requires *floating point arithmetic* and *rounding* of the result. This will slow up the drawing process – even if implemented in hardware.

Bresenham's Algorithm

Bresenham's algorithm scan-converts lines more efficiently than the DDA by using only *integer arithmetic*. It can also be adapted to display curves.

Consider, once again, a line with positive slope less than 1 so we sample along x determining which y to plot.

If we have just plotted the pixel (x_k, y_k) we now need to decide whether to plot (x_{k+1}, y_k) or (x_{k+1}, y_{k+1}) as the next pixel on the line. Note that $x_{k+1} = x_k + 1$ and $y_{k+1} = y_k + 1$. Let the vertical disparities of the pixel positions from the actual mathematical line be given by d_1 and d_2.

Then,

$$d_1 = y - y_k = m(x_k + 1) + c - y_k$$

and,

$$d_2 = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - c$$

Consider now the difference between these two values –

$$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2c - 1$$

If we let the differences between the x and y co-ordinates of the end points of the line be given by –

$$\Delta x = x_e - x_s$$

and,

$$\Delta y = y_e - y_s$$

and if we insist that these endpoint co-ordinates are integers (which they will be if they are specified as pixels) then we can write the slope of the line as a ratio of two integers –

$$m = \frac{\Delta y}{\Delta x}$$

If we now substitute this ratio for m above we get –

$$d_1 - d_2 = 2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2c - 1$$

and so,

$$\Delta x\left(d_1 - d_2\right) = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + \left[2\Delta y + \Delta x\left(2c - 1\right)\right]$$

where the term in square brackets is constant and independent of the pixel position. We shall call this term ***b*** from now on.

If we now define a decision parameter as follows –

$$p_k = \Delta x\left(d_1 - d_2\right)$$

$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + b$$

Then when p_k is positive we should choose (x_k+1, y_k+1) as the pixel to plot and when it is negative we should choose (x_k+1, y_k).

Everything apart from the constant, ***b***, in this decision can be calculated using integer arithmetic so if we cam eliminate the need for ***b*** we have a method for plotting lines using integer arithmetic only.

Consider the difference –

$$p_{k+1} - p_k = 2\Delta y\left(x_{k+1} - x_k\right) - 2\Delta x\left(y_{k+1} - y_k\right)$$

We have already noted that $x_{k+1} = x_k+1$ so we can re-write this as –

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x\left(y_{k+1} - y_k\right)$$

and where $(y_{k+1}-y_k)$ is either 0 or 1 depending on the sign of p_k.

So we have a recurrence equation which involves only integer calculations and all that is left is to determine a starting value p_0 for it.

Using the starting point of the line (x_s, y_s) we can determine the intercept –

$$c = y_s - mx_s = y_s - \left(\frac{\Delta y}{\Delta x}\right)x_s$$

and substituting this into the equation for p_k we obtain –

$$p_0 = 2\Delta y - \Delta x$$

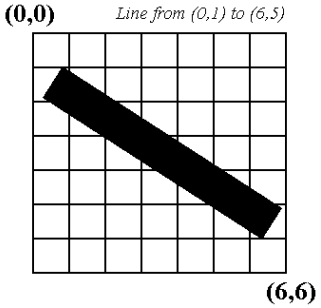
==========

Aliasing and Anti-aliasing

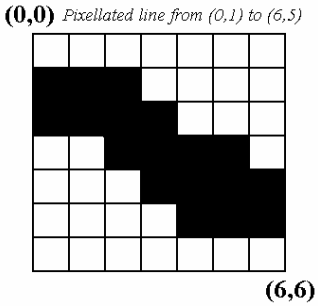
When we produce lines with Bresenham's algorithm, whilst we might be quite impressed with the speed at which we can generate them, we will not be overly enamoured with the way they look. In general they will appear jagged, stepped, rasterised. This effect is known as *aliasing*, a term derived from sampling signals at discrete points in time and thus losing any information conveyed between those discrete times.

Our mathematically ideal line describes a function whose range is continuous but which our pixellated raster screen requires to become discrete. In effect we are sampling at discrete points along the line and plotting the most appropriate pixel for each sample. We have no choice of course; the screen resolution constrains our sampling rate so aliasing is inevitable. Furthermore, our ideal mathematical line has zero thickness but the lines we wish to display must have a thickness to be seen. Indeed, we make a virtue of this requirement and put lines of varying thickness to good use. In computer graphics a straight line is, in reality, a thin rectangle and we must treat it as such.

Consider the following line superimposed on a 2 dimensional 7x7 pixel grid –



This line has a very clear thickness and is quite obviously a filled rectangle. If we pixellate it, as we did with our mathematically ideal line previously, we get –



The pixellated line displays clear evidence of aliasing. Can we make it more aesthetically appealing? Of course, the answer is yes. We can *anti-alias* it.

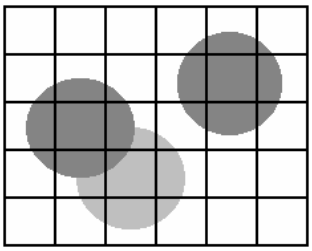
Anti-aliasing algorithms work by partially illuminating pixels which are adjacent to those plotted by the line drawing algorithm (e.g. the DDA or Bresenham's algorithms). This means that anti-aliased lines will be wider than was probably intended.

The two most common methods for anti-aliasing are *filtering* and *supersampling*.

Filtering

This technique is based on the concept that each pixel has an area of influence around it. If our rectangular-shaped line overlaps the area of influence of a pixel then that pixel should be illuminated by an amount proportional to the degree of overlap. This amount is determined by a *filter function*.

One popular filtering scheme is based on circular areas of influence which are one pixel in radius. I.e. the area of influence of each pixel extends as far as the centres of its four nearest neighbours –



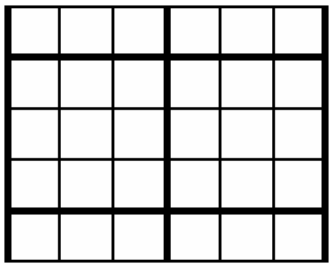
The filter function is defined to be a cone with its base being the area of influence and its apex lying directly above the pixel's centre, at a height which produces an overall volume for the cone of unity.

The intersection of a pixel's area of influence with the rectangular line is then determined and the volume of the cone directly above that intersection yields the amount of illumination to be applied to that pixel. Clearly the amount of illumination applied to any pixel will range from 0 (pixel's area of influence totally outwith the line) up to 1 (pixel totally within the line).

This is clearly a computationally expensive process. In order to keep the computation time down it is normal to pre-calculate tables of values for the filter function. The downside to this is that the resulting algorithms are not very flexible – the filter function, area of influence and palette size are all fixed in a given table and any changes to these values will require a new table to be produced.

Supersampling

Another technique for anti-aliasing uses the concept of *sub-pixels*. Each pixel is divided into, say nine, sub-pixels –



We then count the number of sub-pixels that the line overlaps. We need a reference point within each sub-pixel to use in this counting process. It doesn't matter where it is as long as we use it consistently for all sub-pixels. A corner, such as the top-left, is a popular choice. Centres would produce a more aesthetically pleasing result but at the cost of computational efficiency.

The proportion of sub-pixels which the line overlaps is then used to determine the illumination, in discrete steps from 0 through to 1. Clearly the larger the number of sub-pixels which the pixel is divided into, the finer the resolution that can be accomplished in the illumination. Once again we have to strike a balance between aesthetics and computation time. This will be determined by the needs of the particular application.