Antialiasing polygon scan conversion algorithm

Arnar M. Hrafnkelsson

January 17, 2000

1 Goal

The idea is to create a polygon scan conversion algorithm that operates on the subpixel level. If we have 256 different levels of coverage we need 255 pixels that could be either on or off. It is not very convenient to split a square in to even 255 slices. It is however nice to split it into 256 pixels. So it's possible to use that instead. It would take 256 times the needed memory to do the razterization at that level. This requirement could be lowered by using bits so it would only take 32 times the memory. But this is sort of ridiculous rendering a box with this on a 400×300 image would take $32 * 400 * 300/1024 = 3.75$ megs.[1] The Goal is to achieve the optimal quality and make this run at reasonable speed and not require all to much memory.

2 Basic concept

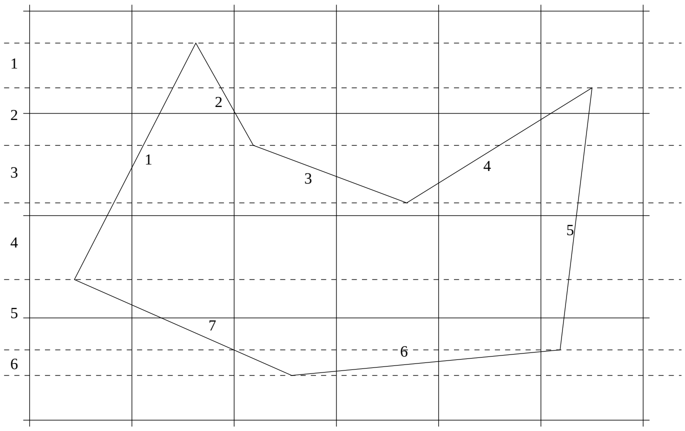

Figure 1: Coarse example of a small polygon

[1] Bear in mind that this grows as a square of the sidelengths.

The idea is to convert the pixels that lie on the inside of the polygon boundary (the ones that dont actually need special care) with a list of on off marks. Refer to figure 1. For the first scanline[2] the 2^{nd} pixel is partialy covered. In the start-stop list line 1 will be a start - but line 2 is a stop. So the stop means that all pixels up to the next start are skipped. So the next pixel to consider is pixel 6 but it's also a start and stop in the same pixel.

For another example look at scanline 3. Both lines 1 and 7 have are start on pixel 1. This means that all pixels thru 4 are filled and 5 is partially covered.

3 The boundaries

This is probably more or less the standard way of doing this. Next we need to consider the partial filled pixels.

3.1 One line through a single pixel

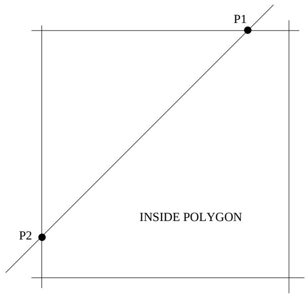

Figure 2: Rendering of a pixel with a single line through it

Here we see the case where a single line crosses the polygon. This is a particularly simple case and is easy to handle. If the intersection points between the boundary of the pixel with the line is on adjacent sides the area is $xy/2$ where x and y are the distances from the joint corner. If the outside of the polygon is toward that corner then the area is $1 - xy/2$. If the intersection points are on opposite sides then the area is $(x + y)/2$, in this case we just measure from the inside part to the boundary.

3.2 More than one line in the same pixel

This is where things get a tad complicated - up to now we have just been able to use the fraction of area covered to get an exact value.

Now lets look at a more complete problem (but still not the worst possible).

[2] The scanlines are bounded by the nondashed lines.

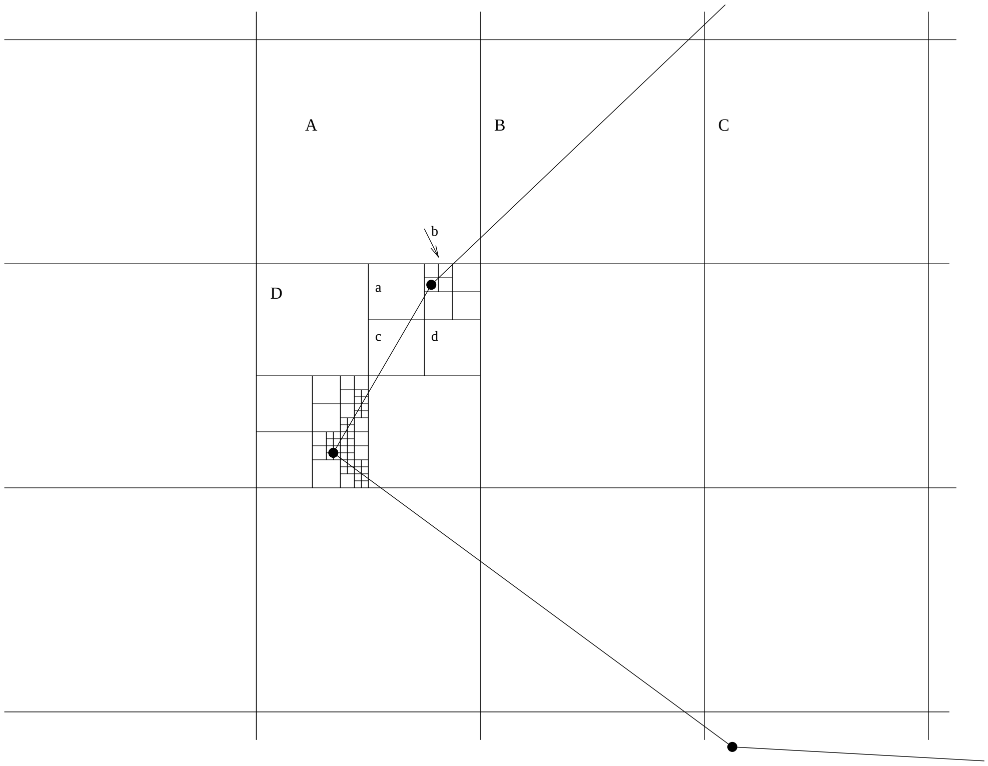

Figure 3: Rendering of a segment where multiple lines are in the same pixel

If we look at figure 3 we see the tesselation of pixels to approximate the area coverage of the polygon within the pixel.

Polygons A and B are both of the form discussed in 3.1. Polygon C is completely covered and the same goes to the all pixels to the right of it.

Now look closely at polygon D. It has been subdivided into 4 quadrants. We identify which case each of them belong to. The upper left is blank the lower right is of the type where a single line goes through the pixel. The remaining two are of the same type as D is. We iterate this process - for example the upper right is split into a, b, c, d. Of those a and c are of the simple kind and d is filled.

We define the weight of a subpixel area by the subdivision level as $1/2^{2l}$. For example the subpixels a, b, c, d belong to level 2 so their weight is $1/2^{2*2} = 1/16$. We see that to only 4 subdivision levels are needed to achieve $2^{2*4} = 256$ graylevels. In the lowest subdivision level we can simply assume the subpixel area as covered since two lines meet at that point and it's effect on the final pixel value is negible.

It is not necessary to keep all of the data in memory - instead it is possible to traverse the pixel quadtree and accumulate with:

if pixel is covered thenpixel_total += coverage_ratio * weight(subdivisionlevel)

4 Faster method for the worst case

The worst case isn't all to nice to handle with the previous method. Especially if there are more than one corners of the polygon meeting in the same pixel.

A better and probably faster way to deal with this is to use this different method.

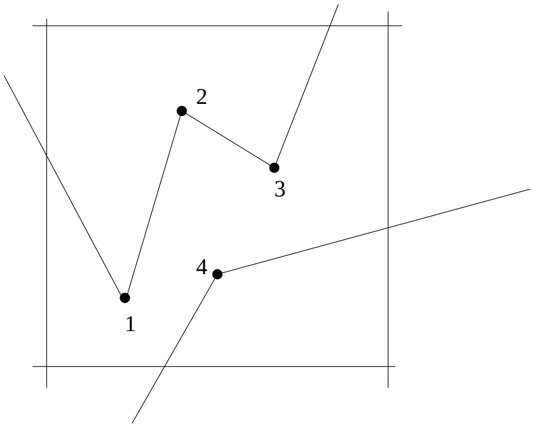

Figure 4: Rendering of a pixel where multiple segments enter the pixel

1. Start by finding a list of points that are inside the pixel
2. Pop a point of the list. Trace the boundary from it and pop and remember all points along the way untill a boundary is hit. Then in the same manner trace from the starting point the other way to a boundary. Now the two paths and the boundary define a polygon. The area of this segment can be calculated and added to the total with the formula

$$A_s = A_{s-1} + \frac{1}{2}\left|\sum_{n=1}^{N} \left|\begin{array}{cc} x_i & x_{i+1} \\ y_i & y_{i+1} \end{array}\right|\right|$$

3. Iterate step 2 untill no points are left in the list.