Raster Graphics

Victor Eijkhout

August 2004

1 Vector graphics and raster graphics

We may use fancy techniques such as Bezier curves for describing shapes, but at some point the graphic needs to be rendered on an actual device with pixels or ink dots. Thus we need algorithms for deciding which pixels to turn on or off, and, in case the device has a larger bitdepth, with what intensity.

The technical terms here are 'vector graphics' for a description of the lines and curves, and 'raster graphics' for the pixel-by-pixel description. A description of a raster is also called 'bitmap'. Common bitmap-based formats are JPEG, GIF, TIFF, PNG, PICT, and BMP.

Vector graphics make for a much more compact file, but they rely on the presence of a final rendering stage. For instance, Macromedia's Flash format (now an open Internet standard) is a vector graphics format, that relies on a browser plugin. However, presence of a Flash renderer is pretty much standard in browsers these days. Adobe Postscript is also a vector format. The first popular printer to incorporate it, the Apple Laserwriter, had a Motorola 68000 processor, exactly as powerful as the Macintosh computer it connected to.

Two vector standards are being proposed to the W3C: the Precision Graphics Markup Language and the Vector Markup Language. PGML is backed by Adobe Systems, IBM, Netscape, and Sun Microsystems. VML is supported by Microsoft, Hewlett-Packard, Autodesk, Macromedia, and Visio. Both standards are based on Extensible Markup Language (XML).

2 Basic raster graphics

2.1 Line drawing

We limit the discussion to lines with slope less than 1. For those, on a grid of pixels, one pixel per column will be switched on, and the question to be addressed is which pixel to turn on in each column.

Probably the simplest way to draw a line is by using an 'incremental' drawing algorithm. Let a line $y = mx + B$ be given, and we write the slope as $m = \delta y/\delta x$. In the case of pixel graphics, we set $\delta x \equiv 1$, so $\delta y = m$ and we can recursively state

$$y_{i+1} = y_i + \delta y.$$

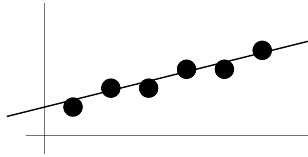

Figure 1: Line with slope ≤ 1 and one pixel per column on

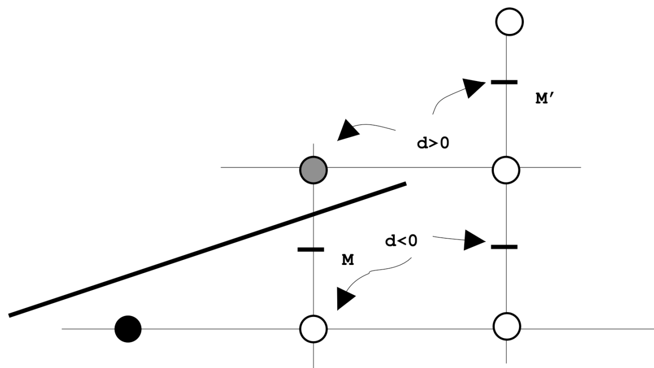

Figure 2: The midpoint algorithm for line drawing

The simplest implementation of this is

let x_0, y_0 and m be given, then
for $i = 0 \ldots n - 1$
  `WritePixel`$(x_i, \text{Round}(y_i))$
  $x_{i+1} = x_i + 1$
  $y_{i+1} = y_i + m$

Since (x_i, y_i) is never related to the actual formula for the line, there will be an accumulation of round-off error. However, this will be negligible. More seriously, the rounding operation is relatively expensive. In the next algorithm we will eliminate it. If possible, we want to operate completely in integer quantities.

The 'midpoint algorithm' proceeds fully by integer addition. First we write the equation for the line in two different ways as

$$y = \frac{dy}{dx}x + B, \qquad F(x, y) = ax + by + c = 0.$$

Clearly, $a = dy, b = -dx, c = B$, and we can derive dx, dy from the end pixels of the line. The function F is zero on the line, positive in the half plane under it, and negative above it.

Now we consider how to progress given that we have switched on a pixel at (x_p, y_p). With the initial assumption that the line has a slope between 0 and 1, the next pixel will be either $(x_P + 1, y_P + 1)$, or $(x_p + 1, y_p)$, depending on which is closer to the line.

Instead of measuring the distance of the candidate next pixels to the line, we decide whether

their midpoint M is above or under the line. For this, we use the function $F(\cdot,\cdot)$ introduced above, and evaluate the 'decision value' of the midpoint:

$$d = F(x_p + 1, y_p + 1/2).$$

The two cases to consider then are

$d < 0$**:** M lies over the line, so we take $y_{p+1} = y_p$;
$d \geq 0$**:** M lies under the line, so we take $y_{p+1} = y_p + 1$.

Similarly we update the mid point: if $d \geq 0$, the midpoint moves up. Note that the new midpoint is at $x_{p+1} + 1$.

Now, we do not actually use the midpoint, only the value of d. The algorithm is then complete once we find a way to update d cheaply. For this we look at its next value

$$d' = F(x_{p+1} + 1, y_{p+1} + 1/2).$$

Corresponding to the above two cases:

$$\begin{array}{rll} d' = & a(x_{p+1}+1) + b(y_{p+1}+1/2) + c = & \\ d < 0: & = a(x_p+2) + b(y_p+1/2) & = d + a = d + dy \\ d \geq 0: & = a(x_p+2) + b(y_p+3/2) + c & = d + a + b = d + dy - dx \end{array}$$

In other words, we update d with dy or $dy - dx$ depending on whether it's negative or non-negative.

To start off the algorithm, dx and dy are computed from the endpoints, and the initial value of d follows from

$$d_0 = F(x_0 + 1, y_0 + 1/2) = F(x_0, y_0) + a + b/2 = 0 + dy - dx/2.$$

To get rid of the division by 2, which would cause real rather than integer values to be used throughout the algorithm, we can redefine $\tilde{F}(x, y) = 2F(x, y)$; correspondingly we update d with $2dy$ and $2(dy - dx)$ in the two cases.

Exercise 1. Can you modify the DDA line drawing algorithm so that it works (as well as possible) when the line is given between points that are not on pixels?

These algorithms are sometimes referred to as 'Digital Differential Analyzers', since they trace out a curve by proceeding with small differences. The line algorithm was first derived by Bressenham.

2.2 Circle drawing

Circles can be drawn with a similar algorithm. We observe that, because of 8-fold symmetry, we can limit the algorithm to the part of a circle from $x = 0$ to $x = y$. The midpoint argument is now slightly more complicated. The function for the circle is

$$F(x, y) = x^2 + y^2 - R^2,$$

and the decision value in the midpoint M is

$$d = F(x_p + 1, y_p + 1/2) = x^2 + 2x + y^2 + y + 5/4.$$

The two cases to consider are similar to before:

$d < 0$**:** M lies in the circle, so we take $y_{p+1} = y_p$;
$d \geq 0$**:** M lies outside the circle, so we take $y_{p+1} = y_p + 1$.

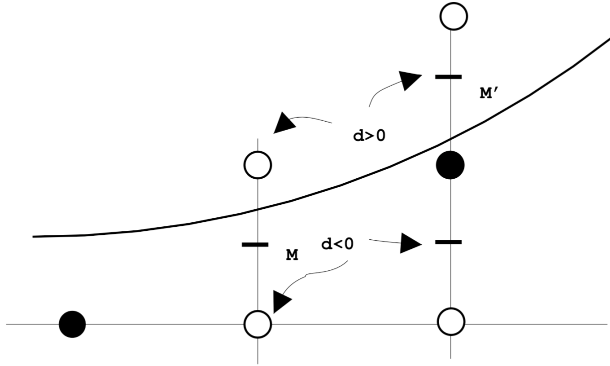

Figure 3: The midpoint algorithm for circle drawing

To update the decision value we get the two cases

$$\begin{array}{rll} d' = & F(x_{p+1}+1, y_{p+1}+1/2) = & \\ d<0: & = x^2+4x+y^2+y+4\,1/4 & = d+2x+3 \\ d\geq 0: & = x^2+4x+y^2+3y+6\,1/4 & = d+2(x+y)+5 \end{array}$$

Exercise 2. Why is there no need to consider bigger increments of y_p in the circle drawing algorithm? After all, a circle has curvature so the slope increases.

The circle algorithm can be further improved by observing that the quantities $2x$ and $2y$ can themselves easily by constructed by updating. This removes the need for any multiplication.

2.3 Cubics

Suppose we have a cubic function $f(t) = at^3 + bt^2 + ct + d$. Instead of evaluating this polynomial directly, using Horner's rule, we compute the value $f(t+\delta)$ by updating:

$$f(t+\delta) = f(t) + \Delta f(t).$$

We find

$$\begin{array}{rcl} \Delta f(t) & = & f(t+\delta) - f(t) \\ & = & a(3t^2\delta + 3t\delta^2 + \delta^3) + b(2t\delta + \delta^2) + c\delta \\ & = & 3a\delta t^2 + (3a\delta^2 + 2b\delta)t + a\delta^3 + b\delta^2 + c\delta \end{array}$$

This still leaves us with a quadratic function to evaluate, so we define

$$\begin{array}{rcl} \Delta^2 f(t) & = & \Delta f(t+\delta) - \Delta f(t) \\ & = & 3a\delta(2t\delta + \delta^2) + (3a\delta^2 + 3b\delta)\delta \\ & = & 6a\delta^2 t + 6a\delta^3 + 2b\delta^2 \end{array}$$

Finally, we derive $\Delta^3 f(t) = \Delta^2 f(t+\delta) - \Delta^2 f(t) = 6a\delta^2$. Taken all together, we can now compute $f_{n+1} \equiv f((n+1)\delta)$ by initializing

$$\Delta^3 f_0 = 6a\delta^2, \quad \Delta^2 f_0 = 6a\delta^3 + 2b\delta^2, \quad \Delta f_0 = a\delta^3 + b\delta^2 + c\delta$$

and computing by update

$$f_{n+1} = f_n + \Delta f_n, \quad \Delta f_{n+1} = \Delta f_n + \Delta^2 f_n, \quad \Delta^2 f_{n+1} = \Delta^2 f_n + \Delta^3 f_0$$

The advantage of this algorithm is its low operation count, and the fact that it works fully by integer operations.

3 Rasterizing type

Typefaces can be described by curves, but several aspects to them make it necessary to do more than just rendering these curves, when rasterizing them. Part of the problem is that characters in a font are relatively small, and satisfy all sorts of constraints that both may be hard to satisfy (especially at low resolution), and are immediately noticed when rendered incorrectly.

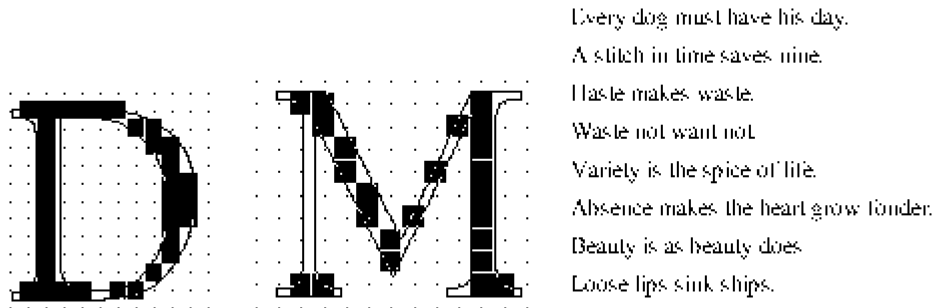

Figure 4: Problems in rasterizing type, and resulting illegible output

Such problems result from using too simple algorithms for converting the character outlines to rasters. For instance, an obvious algorithm is

> A pixel is turned on if its center is within the curve.

Now consider the case where a curve with large radius exceeds location $y = n + 1/2$ for only one x. This results in the 'pimple' on top of the 'e' in figure 5. On the other hand, if such a curve stays just under such a halfpoint, we get a long plateau, as in the left side curve of the 'e'.

3.1 Scaled fonts

These rasterizing problems are largely due to the facts that

- Characters are scalable, so the relations between top/bottom or left/right are not always mapped the same way to a pixel grid;
- Even if characters are used at the same size, they need to be displayed on various kinds of rasters (printer, screen);
- Characters can be placed in any horizontal or vertical location, so relations to pixel boundaries are also flexible.

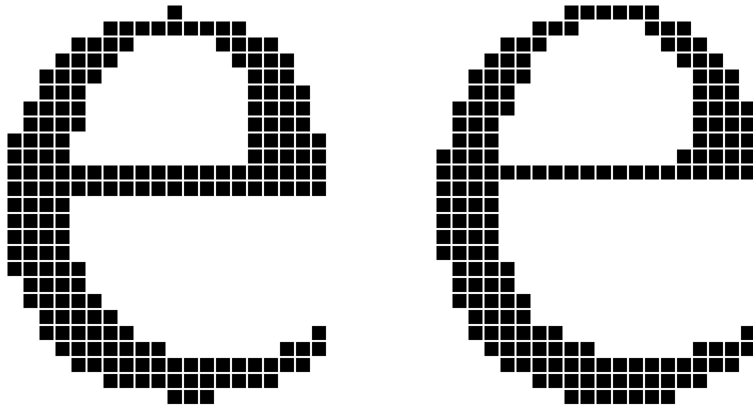

Figure 5: A bad and a good way of rendering a Times Roman ‘e’ at low resolution

The conversion process goes more or less as follows[1]:

Figure 6: Scaled and rasterized character outline

- Character outlines are based on coordinates on some grid, often expressed as fixed point, or other scheme with a finite mesh size.
- The character is scaled to the raster on which it will be rendered.
- The result is rounded to the raster, in figure 6 this puts the left and right sides on pixel boundaries; note that other vertical parts of the character are not necessarily pixel-aligned.
- The scaled and rounded outline is then rasterized by turning a set of pixels on.

We see that in both final steps, rounding to the raster, and switching on pixels, we can have unwanted effects.

1. Much of this discussion is based on the definition of TrueType fonts.

3.2 Pixelation

Above we said that pixels are switched on if their center falls within the curve. There are two problems with this:

- Sometimes not enough pixels are turned on to render the shape accurately, and
- Deciding whether a pixel is within the shape is not trivial to begin with. For instance, letters such as ‘o’ or ‘p’ have an inner region that should not be filled. In another example, sometimes it is convenient to describe a shape as non-disjoint union of other shapes; see figure 7

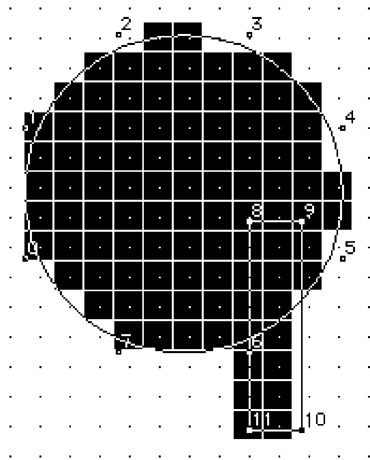

Figure 7: A shape consisting of overlapping contours

The second problem can be solved a number of ways. We could for instance look at a scan line, and switch to on/off mode every time we cross a curve boundary. This approach does not work for intersecting curves.

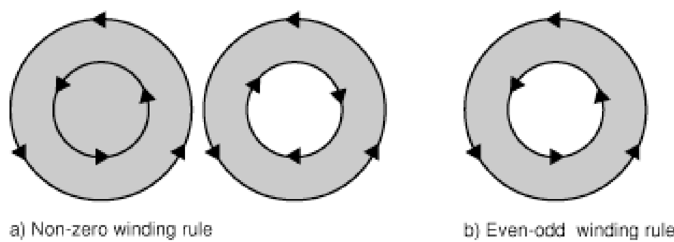

Figure 8: The effects of different winding number rules

Better solutions are based on looking at the so-called ‘winding number’. This number counts, for a given point in the plane, how often a curve winds around the point. If this is zero, the point is outside the curve, otherwise it is inside it.

That implementing winding number rules is not trivial can be seen from two screen shots of Acrobat Reader version 4; figure 9.

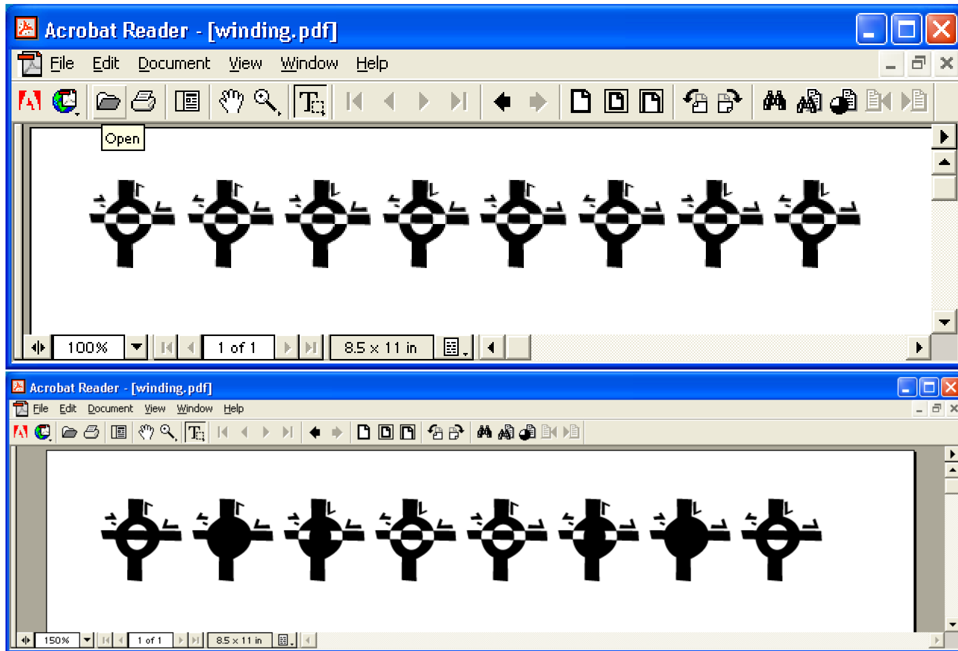

Figure 9: Acrobat 4 rendering of a complicated figure at different magnifications

3.3 Font hinting / instructing

To prevent some of the problems indicated above, characters in a font file consist of more than just the outline description. Additionally, each character can have a short program in a language defined by the font file format. Such a program can enforce that certain distances in the font as exact multiples of pixel distances.

For instance, the letter 'O' in figure 10 has the following constraints

1. A certain amount of white space buffering the character; distance 3 is the 'transport';
2. The width of the band
5,6 Visual under and overshoot.
7 The height of the band

Distances 5 and 6 are over and undershoot: a letter with curved top/bottom like 'O' would seem too small if it stayed between the baseline and the cap height. To compensate for that, the letter is made slightly higher and deeper. However, for small sizes and low resolutions, this compensation needs to be switched off, since it would look too drastic.

3.4 Dropouts

In certain cases, pixels can not be switched on that are necessary for continuity of the figure drawn. Figure 11 shows a case where a connecting bit is too thin to hit the centers of any pixels. To cover such cases, the renderer usually has an algorithm that detects when a scan

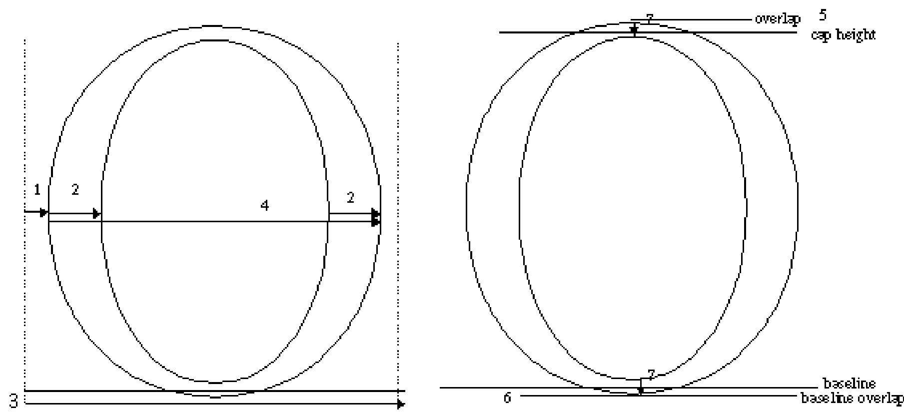

Figure 10: Constraints on the letter 'O'

line enters and leaves a contour without setting any pixels. It will then set, for instance, the left pixel.

4 Anti-aliasing

In the algorithms so far, the decision was between switching a pixel on or off. There are a few problems with this. For instance, for certain slopes the rendered curve can have a very 'jagged' look. Also, a curve at slope 1 will have the same number of pixels on as a curve at slope 0, but on a path that is longer by $\sqrt{2}$. Thus, it may look lighter or thinner.

If the display supports a range of values ('grayscale'), we can try to use this and find a better visual rendering. First we will look at an example, then briefly consider the general theory of this phenomenon.

4.1 Raster graphics with larger bitdepths

In the algorithms above, pixels close to a line or other curve were switched on. If the display supports it, we can compute a measurement of proximity of the pixel to the line, and set a brightness based on that. There are various ways to implement this. For instance, one could consider the line to have an area, and to compute the intersection area of the line and the pixel box. Here we will use a 'filter function'. The support of this function will be larger than the pixel box.

We will modify the midpoint method for line drawing so that in each column three pixels will be nonzero, with an intensity based on the Euclidean distance to the line. Let v be the (signed) vertical distance from the midpoint to the line, and D the euclidean distance, then $D = vdx/\sqrt{dx^2 + dy^2}$. The denominator we can compute once and for all at the start of the algorithm, but a simple way of computing vdx is needed.

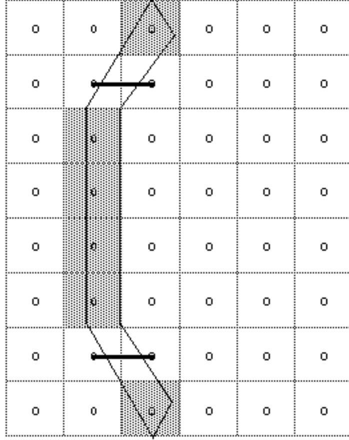

Figure 11: A case of 'dropout': missing pixels give a disconnected curve

Consider the case where $d < 0$, in which case we choose $y_{p+1} = y_p$. Now we have

$$0 = F(x_p + 1, y_p + v) = F(x_p + 1, y_p) + 2bv \Rightarrow 2vdx = F(x_p + 1, y_p),$$

and

$$d = F(M) = F(x_p + 1, y_p + 1/2) = F(x_p + 1, y_p) + b$$

so $2vdx = d + dx$. Likewise, if $d \geq 0$, $2vdx = d - dx$. Since we know how to update d cheaply, we can now iteratively compute D.

For the top and bottom point, $D = 2(1-v)dx/\sqrt{\ldots}$ and $D = 2(1+v)/\sqrt{\ldots}$, respectively.

4.2 The general idea

Smoothing out a picture with grayscales is usually called 'anti-aliasing'. To see why, consider how the parts of a graphics system fit together. After we derive a curve or other two-dimensional object (this could be a projected surface) to display, the exact values are sampled according to the pixel resolution. By computing a Fourier transform of these samples, we get an impression of the visual 'frequencies' in the picture.

If, instead of simply switching on pixels based on the sampled curve, we compute pixel values so that sampling them reproduces the frequency spectrum, we get a picture that looks closer to the intended one.

Contents

1 **Vector graphics and raster graphics** *1*
2 **Basic raster graphics** *1*
2.1 *Line drawing* *1*
2.2 *Circle drawing* *3*
2.3 *Cubics* *4*
3 **Rasterizing type** *5*
3.1 *Scaled fonts* *5*
3.2 *Pixelation* *7*
3.3 *Font hinting / instructing* *8*
3.4 *Dropouts* *8*
4 **Anti-aliasing** *9*
4.1 *Raster graphics with larger bitdepths* *9*
4.2 *The general idea* *10*