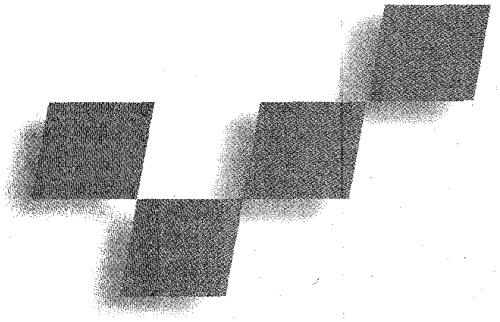


Pixel-Processing Fundamentals

Jack Bresenham
Winthrop University

Key to developing line-drawing algorithms is an explicit reference model for quantifying a "good" raster approximation. This tutorial looks at old practices, then demonstrates development of line- and edge-drawing algorithms.

Pictures of line drawings and text characters can be displayed using raster devices such as laser printers, cathode ray tubes, and incremental plotters. A laser printer fuses a small, discrete dot on a page. A cathode ray tube's electron gun excites a discrete spot of phosphor to glowing visibility. An incremental plotter steps an inked pen between conceptual reference points to leave, say, a 0.001-inch line segment drawn at an integer multiple of 45 degrees. Raster devices, therefore, typically employ a discrete rectilinear representational grid in which an integer-valued coordinate mesh point is a conveniently addressable entity.

In computer graphics, the conceptual mesh point is the *pixel*. Pixel and its predecessor *pel* are each a shortened common expression for picture element. Some find it useful to conceptualize raster devices as chessboards in which a pixel is represented by the round base of a king piece as it moves from square to square. In this analogy, a pixel address is the coordinate location of the center of each chessboard square. Drawing a curve typically progresses step by step along a pixel pathway constrained by the unit steps of the king piece. Even though the physical length of a diagonal step differs by a factor of $\sqrt{2}$ from the length of an axial step, each of the eight possible directions constitutes a single step. Hence, the term *unit step* describes movement to one of the eight neighboring mesh points.

Old practices applied in a new context

Forming pictures from pixels is an old practice. The rastered displays presented in Figures 1 and 2 illustrate noncomputer-drawn rastered pictures. Needlepoint or counted cross-stitch, such as that popularized by the image on a box of Whitman Sampler chocolates, uses essentially the same technique as does an incremental plotter; small pieces of colored thread are exposed to compose a scene from tiny line segments. In the 1950s and 1960s, card stunt displays in the spectator stands during football game halftimes used colored cardboard squares as pixels to present clever pictures and captions.

Other examples include the scoreboard at a sports stadium, which uses discrete light bulbs to form numbers, letters, and pictures as does the big moving news display at Times Square in New York City. The French pointillist painter Georges Pierre Seurat (1859-91) perfected a painting style in which he composed pictures strictly from discrete dots. A close look at your favorite newspaper will reveal its black-and-white photos to be collections of dots. Variable spacing in black dot clusters produces the visual effect of various shades of gray called *dithering*.

Thus, *quantized* picture composition from discrete dot patterns certainly is nothing conceptually new. Computer technology has, however, provided a means for faster generation of more complex and comprehensive visual displays. Computer programs can synthesize and manipulate picture drawing abstractly and use rasterization algorithms to pick appropriate pixels at rendering time. Computer peripherals can then display the generated image as a dot composition of discrete pixels.

Pixel generation or *rendering* is often separated from picture presentation by rastering pixel selections into an intermediate memory buffer from which a peripheral can independently control the timing of its extraction of one or more horizontal scan lines of pixels for actual display and viewing. Dual-ported memory capable of accepting pixels written by the rendering process at the same time the CRT refresh circuitry is reading pixels for display is known as video random access memory, or VRAM.

The pixel representation in such an intermediate memory or frame buffer typically implies its raster presentation coordinate location by its memory address. The address itself is calculated from the discrete pixel integer (X, Y) location. For example, a 256-color display 640 pixels wide by 480 pixels high might use one byte

0272-17-16/96/$5.00 © 1996 IEEE

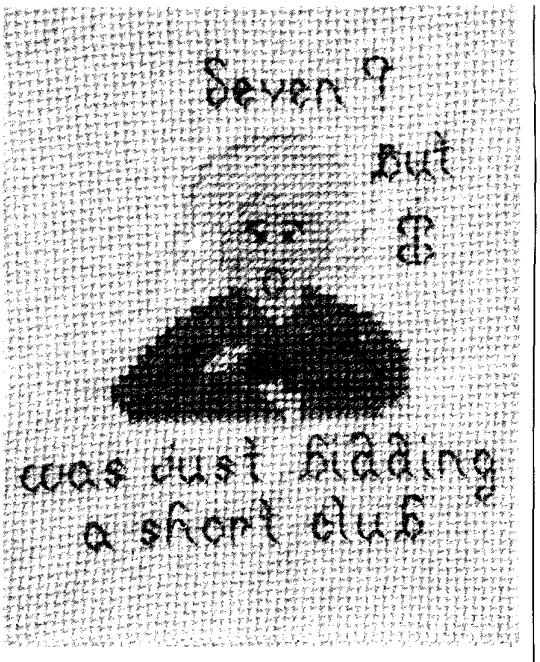
per pixel to hold each 8-bit color value. The frame-buffer memory address for a pixel at (X, Y) would be calculated as

$$\text{address} = \text{base} + X + 640Y$$

where $X \in \{0, \ldots, 639\}$ and $Y \in \{0, \ldots, 479\}$. Alternatively, pixel coordinates (X, Y) can be recovered from the memory address as

$$X = ((\text{address} - \text{base}) \text{ MOD } 640)$$
$$Y = ((\text{address} - \text{base}) \text{ DIV } 640)$$

Not unlike a paint-by-numbers strip in the Sunday newspaper comics, number values held in the frame buffer specify pixel color. The pixel's value explicitly stored in memory can be as simple as a dichotomous 0 for "off" and 1 for "on" convention that requires only one bit per pixel for a bilevel or monochrome display. Alternatively, a pixel's stored value could be a 24-bit, three-byte-partitioned value representing a specific color shade specified by its triplet red, green, and blue (RGB) component mix from a palette of 16,777,216 color choices. Typically, the number held in the frame buffer is simply a positional index into a color table or palette from which the actual RGB color value is read. Separating the actual displayed color from the frame buffer's pixel value provides a late color-binding mechanism that lets you dynamically or interactively redefine palette colors to, say, highlight or more widely separate selected colors after a first viewing of the data.



1 Nancy's needlepoint (courtesy of Nancy Bull Bresenham; photo by Joel Nichols).

2 Stanford football halftime (photo courtesy of Larry Tessler).

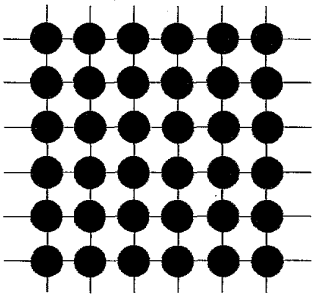
Rasterization: sampling approximation

Many drawing applications, such as engineering design or business chart preparation, use a geometric line as a fundamental graphical shape. More complex shapes, such as splines or boxes or character fonts, are then formed using many short lines or filled polygonal areas, often simply triangles, bounded by small line segments. Computer graphics processing must therefore provide fast, accurate, consistent line-drawing capability at the elementary systems level. Rasterization will simply be the process of sampling the "true" line to approximate its continuous representation, using discrete dots or lattice points for integer coordinate pixels chosen to be close to the locus of the true line.

How will we know when we have a good approximation? What specifically constitutes "close"? To answer such questions, we need explicit assumptions and a conceptual reference model. Let's start with some coordinate assumptions. Our underlying grid for discretization is a rectilinear Cartesian coordinate system with uniformly spaced X and Y grid lines, that is, square spacing between pixels. Pixel-coordinate addresses represent the center of a pixel and correspond to integer mesh points on the uniform grid. Figure 3 on the next page illustrates pixels in such a raster space.

Different assumptions, not considered here, could use alternatives such as a hexagonal grid[1] or pixel addresses corresponding to the lower left corner of an open, rectangular pixel.[2] An unfortunate choice would be a truly rectangular grid in which the horizontal distance between pixels differs from the vertical distance. Early IBM PC graphics used such an *anisotropic grid*; later IBM PC graphics have begun to use the preferred square spacing for the addressable pixel grid.

Recall that in ordinary geometry, points have no size and hence occupy no area. A displayed pixel does cover some small, but finite, area. Likewise, a geometric line has only length with no width or thickness. Our displayed rastered lines will be composed of pixels and will



3 Lattice points address the center of a pixel in a Cartesian coordinate system raster grid.

therefore have at least single-pixel thickness. Later we'll want to purposely specify lines of multiple-pixel widths. These acknowledged discrepancies in area between conceptual points and lines and physical pixels present opportunities for visual anomalies, which we'll need to handle in as consistent a fashion as possible.

Our rastering strategy for lines of the form

$$F(x, y) = Y - sX = 0$$

with $0 < s < 1$ and nonnegative X, will be to step abscissa integer X values by unity from 0 and find the "closest" ordinate integer Y value for each successive value of X. That our representational grid in raster space requires pixel addresses to be integer coordinate lattice points does not mean that we can specify only lines with integer endpoints. However, lines with integer endpoints are typically rendered much faster and more compactly, so it is worthwhile understanding what is traded in accuracy to achieve integer speed in pixel generation.

We'll get to the simplified integer-endpoint line, first-octant form, $Y = sX$, by considering various simplifications for approximating the general form

$$\Delta y(x_1 - x_0) = \Delta x(y_1 - y_0)$$

for a directed line segment from a noninteger point (x_0, y_0) to a noninteger point (x_1, y_1). First, though, let's explicitly, but somewhat arbitrarily, specify and quantify what we want to accept as a "good" fit for each "close" pixel in a discrete raster approximation of a continuous true line.

Measuring closeness of a delicatessen to an apartment in Manhattan would likely require laying out a staggered route with rectangular corners to account for sidewalk pathway constraints imposed by city street layouts. In a different context, measuring closeness to a neighboring ranch house in the *llano estacado* plains of eastern New Mexico more likely would simply use linear distance as the crow flies or the horse gallops. Let's look at some alternative measures of closeness in rastering or scan converting geometric primitives such as lines and circles.

Closeness: objective measurement for a subjective assumption

With hand-crafted, low-volume cross-stitched pictures, each artisan could subjectively decide what close meant in approximating continuous scenes by sampled, discrete smidgens of thread placed close to original picture geometric features. With computer-generated drawing, precise predictability, guaranteed repeatability, and systematic selection of pixels are desirable characteristics. We'll want to use an *algorithm* or set of rules to specify how to pick pixels consistently.

Before developing an algorithmic rasterization methodology for pixel generation, we'll need to subjectively pick a quantifiable, measurable means to define what is meant by close or best choice between two candidate pixels at each step. Having subjectively picked a closeness measure for its perceived aesthetical implications or its computational tractability, we'll then objectively pick pixels by precise measurement. We'll also want to devise a methodology that efficiently and rapidly calculates our objective measure for selecting each pixel. In this tutorial, I discuss only picking a best pixel location at each step, saving color shade and other considerations such as antialiasing and line width for another time.

One possible measure of closeness could be the magnitude of realized line length minus true line length. Figure 4 shows an example approximation of the line segment $F(x, y) = Y - \frac{1}{2}X = 0$ for $0 \le X \le 8$ that uses this measure. It consists of four axial steps followed by four diagonal steps. No other discrete approximation for $X \in \{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\}$ can provide a closer approximation than that shown in Figure 4. True length is $4\sqrt{5}$. Realized length shown is $4(1 + \sqrt{2})$.

The approximations shown in Figure 5 seem aesthetically closer to the original continuous line, yet each length is still $4(1 + \sqrt{2})$. So, by our "goodness" choice of difference between approximated line length and true line length, the representation in Figure 4 is OK! If it is not OK, then we can always change our error metric or measure of goodness so that calculation of a revised closeness is something more to our liking. We'll certainly want a computationally tractable measure for easy implementation.

The important matter, I suggest, is not so much which numerical measure of goodness or closeness you employ. Instead, the important matter is to pick a quantifiable measure overtly. Then we can work to understand all its implications and limitations. If we find undesirable side effects accompanying our choice, we can either patch matters up—for example, by later incorporating provision for line retraceability—or discard the measure and explicitly choose an alternative quantification.

Length is an appealing closeness measure when tying bed sheets together to make a quick exit from the third floor of a burning hotel. However, it is probably not a good measure for judging line rastering choices. Three measures more commonly used for this purpose are magnitude or absolute values of *axial distance*, *normal distance*, and *function residue*. For a terse nomenclature, let's use single-word descriptors—axial, normal, and residual, respectively —for these three measures of closeness.[3,4]

Axial displacement error in our first-octant case is a candidate pixel to true-line distance measured parallel to the y-axis. Normal error is candidate pixel to curve-locus distance measured along the line perpendicular to the curve $F(x, y) = 0$. Residual error is the absolute value or magnitude of the scalar obtained by evaluat-

ing the curve's defining function $F(x, y)$ at a pixel integer mesh point coordinate $(x=X, y=Y)$—that is, upper case X and Y are the integer values of the candidate pixel coordinate location. The residue will be zero if and only if the pixel lies exactly on the curve.

Since the objective function to be minimized at each successive step in rastering often seems to be misunderstood, a line example and a circle example can illustrate these three alternative measures. For lines, the three measures are always equivalent; for circles, the three measures can differ in the pixel selected.

Quantified error measures illustrated

The floor of a number n is shown as $\lfloor n \rfloor$ and is the largest integer less than or equal to n. The ceiling of a number n is shown as $\lceil n \rceil$ and is the smallest integer greater than or equal to n. For each integer X and its associated or dependent y value, the two integer coordinate candidate pixels are the "lower" pixel $(X, \lfloor y \rfloor)$ and the "upper" pixel $(X, \lceil y \rceil)$. For each integer Y and its associated x dependent value, the two integer coordinate candidate pixels are the "left" pixel $(\lfloor x \rfloor, Y)$ and the "right" pixel $(\lceil x \rceil, Y)$. We'll assume lines are oriented in a relative first octant with $0 < \Delta y < \Delta X$.

Our rastering strategy is to fix one independent variable, say X, to be an integer, then solve $F(x, y) = 0$ for the dependent variable y. Typically, y will be a noninteger, so the two nearest integers $\lceil y \rceil$ and $\lfloor y \rfloor$ bounding y are of interest. When $\lceil y \rceil = \lfloor y \rfloor = Y$, an integer, for an integer value of X, the pixel at (X, Y) lies precisely on the curve $F(x, y) = 0$. Clearly, the pixel choice in the special case is (X, Y). In general, a choice will have to be made between point $P_{\text{upper}} = (X, \lceil y \rceil)$ and point $P_{\text{lower}} = (X, \lfloor y \rfloor)$ when stepping the abscissa by unity.

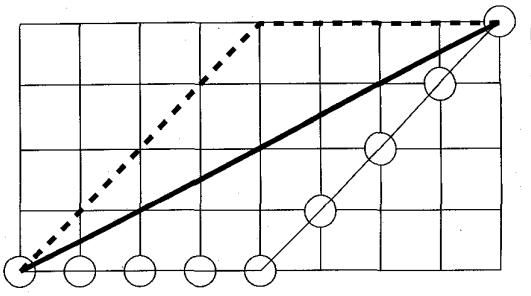
Implicit in this rastering strategy is an assumption that the curve $F(x, y) = 0$ can be treated in regions such that it is a single-valued function of one variable within the region of interest. Lines are inherently single valued for either x or y. Integer-centered circles must be done by quadrant or by octant to satisfy our single-valued curve assumption, but they can use symmetry to place eight pixels per selection. Noninteger-centered circles cannot rely upon multiquadrant or multioctant symmetry to "broadcast" one pixel selection to other quadrants or octants. While this tutorial examines noninteger instances, a common practice is to round real numbers so that only integer endpoint lines and integer-center, integer-radius circles are actually rendered.

For each candidate pair of pixels, we'll want to systematically pick the one pixel closest to the true curve. The question, of course, is what specifically constitutes close. Let's examine choosing $X = 1$ to see what's close for the line segment from (0.1, 0.4) to (9.8, 2.6) and $X = 2$ for the origin-centered circle of radius 4.925. The functional forms for the line are

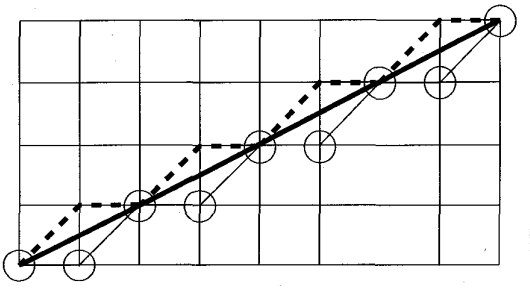
$$F(x, y) = 0 = \Delta x(y - y_0) - \Delta y(x - x_0)$$
$$= (9.8 - 0.1)(y - 0.4) - (2.6 - 0.4)(x - 0.1)$$
$$F(x, y) = 0 = 9.7y - 2.2x - 3.66$$

or

$$y = (2.2/9.7)x + (3.66/9.7)$$



4 *One raster approximation based on realized minus true line length for the segment* $F(x, y) = Y - \frac{1}{2}X = 0$ **for** $0 \le X \le 8$.



5 **A second raster approximation based on realized minus true line length for the segment** $F(x, y) = Y - \frac{1}{2}X = 0$ **for** $0 \le X \le 8$.

evaluated at $X = 1$ to yield $y = 0.60412$ so that the two candidate pixels are

$$(X, \lfloor y \rfloor)) = (1, 0) \text{ and } (X, \lceil y \rceil) = (1, 1)$$

The functional form for the circle is

$$F(x, y) = 0 = (x - x_{\text{center}})^2 + (y - y_{\text{center}})^2 - r^2$$
$$F(x, y) = 0 = x^2 + y^2 - (4.925)^2$$

or

$$y = ((4.925)^2 - x^2)^{1/2}$$

evaluated at $X = 2$ to yield $y = 4.500625$ so that the two candidate pixels are

$$(X, \lfloor y \rfloor)) = (2, 4) \text{ and } (X, \lceil y \rceil) = (2, 5)$$

The axial distance from the pixel at (1, 0) is just the vertical distance to the true-line intercept at $X = 1$ or a distance of $0.60412 - 0 = 0.60412$. Axial distance from the pixel at (1, 1) to the true line is $1.0 - 0.60412 = 0.39588$; hence, (1, 1) would be the selected pixel. The axial distance thus amounts simply to ordinary rounding.

The normal distance to line $\Delta x(y - y_0) - \Delta y(x - x_0) = 0$ from any point (u_0, v_0) can be found using the equation of the perpendicular line:

$$\Delta x(x - u_0) + \Delta y(y - v_0) = 0$$

For (1 , 0), solving simultaneously $0 = 9.7y - 2.2x - 3.66$ and $9.7(x - 1) + 2.2(y - 0) = 0$ yields an intersection point (0.86969, 0.57457) from which the perpendicular or normal distance is seen to be

$$((1 - 0.86969)^2 + (0.57457 - 0)^2)^{1/2} = 0.58916$$

For (1, 1), solving simultaneously $0 = 9.7y - 2.2x - 3.66$ and $9.7(x - 1) + 2.2(y - 1) = 0$ yields an intersection point (1.08539, 0.62349) from which the perpendicular or normal distance is seen to be

$$((1.08539 - 1.0)^2 + (1.0 - 0.62349)^2)^{1/2} = 0.38607$$

As was the case for an axial distance measure, the pixel (1, 1) is again selected, now using normal distance as our error metric. In fact, if $\theta = \arctan(\Delta y/\Delta x)$, then normal distance is always just axial distance multiplied by cosine(θ) for any first-octant line.

For function residue, you simply evaluate

$$F(x, y) = 9.7y - 2.2x - 3.66$$

at the two points (1, 0) and (1, 1) so that the two signed residues are

$$F(1, 0) = 9.7(0) - 2.2(1) - 3.66 = -5.86 \text{ and}$$
$$F(1, 1) = 9.7(1) - 2.2(1) - 3.66 = 3.84$$

We then compare the absolute values of each residue to select, again, the pixel located at (1, 1).

For the circle example, $F(x, y) = 0 = (x - x_{center})^2 + (y - y_{center})^2 - r^2$, the axial distance is

$$4.500625 - 4.0 = 0.500625$$

from candidate pixel $(2, 4) = (X, \lfloor y \rfloor)$ and

$$5.0 - 4.500625 = 0.499375$$

from candidate pixel $(2, 5) = (X, \lceil y \rceil)$. Using axial distance as our measure, we'll pick pixel (2, 5). The normal distance is measured along a radial ray so that the perpendicular distance from (2, 4) is the magnitude of

$$\sqrt{(2-0)^2 + (4-0)^2} - 4.925$$

or $|4.472136 - 4.925| = 0.452864$. The normal distance from (2, 5) is

$$\sqrt{(2-0)^2 + (5-0)^2} - 4.925$$

or $|5.385165 - 4.925| = 0.460165$. Using normal distance as our measure, we'll pick pixel (2, 4).

The function residue would use

$$|F(2, 4)| = |2^2 + 4^2 - 4.925^2| = 4.255625 \text{ and}$$
$$|F(2, 5)| = |2^2 + 5^2 - 4.925^2| = 4.744375.$$

Using magnitude of function residue, we'll pick pixel (2, 4). For circles, then, the choice of error measure can make a difference.

For lines, all three measures—axial, normal, and residual—are equivalent in that identical pixels always will be selected. For circles, as demonstrated above, the three measures are not necessarily equivalent. When the circle center and radius are restricted to integers, the three measures will always coincide for circles as well as lines.

A line-drawing algorithm derived

For a general approach, consider the following pseudocode. In practice, you look at the specific class of curves in context to simplify calculations and avoid actually running both loops shown in the pseudocode. For curves of any complexity, it is common to approximate the curve polygonally by short chords and use only line rasterization. Conceptually, a rasterization in a single-valued region of a "true" curve could be of the form

```
FOR X := 0 TO (M - 1) DO
  BEGIN
    y := "true" solution of F(X, y) = 0
    Y := integer ordinate of closer
           of pixel (X, ⌊y⌋)
           or pixel (X, ⌈y⌉)
  {"close" can be by axial, normal,
    or residue measure}
    pixel_X[X] := X
    pixel_Y[X] := Y
  END;
FOR Y := 0 TO (N - 1) DO
  BEGIN
    x := "true" solution of F(x, Y) = 0
    X := integer abscissa of closer
           of pixel (⌊x⌋, Y)
           or pixel (⌈x⌉, Y)
    pixel_X[M + Y] := X
    pixel_Y[M + Y] := Y
  END;
Cull (pixel_X, pixel_Y) pairs so that any one
  coordinate value 2-tuple location appears
  once only.
Display minimum spanning set of pixels.
```

For directed line segments from (x_0, y_0) to (x_1, y_1) such that $F(x, y) = \Delta x(y - y_0) - \Delta y(x - x_0) = 0$ and in a relative first octant with $(x_1 - x_0) > (y_1 - y_0) > 0$, it suffices to use increasing integer values of X to obtain the minimum spanning set of pixels that will approximate the line. For a circle of radius r centered at (a, b) such that $F(x, y) = (x - a)^2 + (y - b)^2 - (r)^2 = 0$ and moving clockwise through the octant in which $(y - b) > (x - a) > 0$, it also suffices to use successively increasing integer values of X.

Since the three measures are equivalent for rastering lines, let's use axial displacement as our quantitative measure of closeness to derive an incremental algorithm. As seen earlier, axial displacement error from an integer coordinate pixel at (X, Y) to a point (x_d, y_d) on the continuous true line (or other curve given by the function $F(x, y) = 0$) is distance measured parallel to a coordinate axis. A nongraphics analogy is simple rounding of a scalar number. The displacement from a number to the integer just above and just below it is essentially how the decision is made to pick a closest integer in this one-dimensional analogy.

For nomenclature, let's assign uppercase variables to be integer and lowercase variables to be real or integer:

- A is always an integer (A never has a fractional value component other than zero) and

- a can be either an integer or a real number (a can have a zero or non-zero fractional value component).

Let $A = \lfloor a \rfloor$ so that A is the floor of a and A is the largest integer less than or equal to a. Note that for $0 \le \delta < 1$ and $a = A + \delta, A = \lfloor a \rfloor$. For example,

$$\lfloor 4 \rfloor = \lfloor 4 + 0.0 \rfloor \text{ is } 4$$
$$\lfloor -4 \rfloor = \lfloor -4 + 0.0 \rfloor \text{ is } -4$$

$$\lfloor 4.1 \rfloor = \lfloor 4 + 0.1 \rfloor \text{ is } 4$$
$$\lfloor -4.1 \rfloor = \lfloor -5 + 0.9 \rfloor \text{ is } -5$$

$$\lfloor 4.9 \rfloor = \lfloor 4 + 0.9 \rfloor \text{ is } 4$$
$$\lfloor -4.9 \rfloor = \lfloor -5 + 0.1 \rfloor \text{ is } -5$$

Let $A = \lceil a \rceil$ so that A is the ceiling of a and A is the smallest integer greater than or equal to a. Note that for $0 \le \delta < 1$ and $a = A - \delta, A = \lceil a \rceil$. For example,

$$\lceil 4 \rceil = \lceil 4 - 0.0 \rceil \text{ is } 4$$
$$\lceil -4 \rceil = \lceil -4 - 0.0 \rceil \text{ is } -4$$

$$\lceil 4.1 \rceil = \lceil 5 - 0.9 \rceil \text{ is } 5$$
$$\lceil -4.1 \rceil = \lceil -4 - 0.1 \rceil \text{ is } -4$$

$$\lceil 4.9 \rceil = \lceil 5 - 0.1 \rceil \text{ is } 5$$
$$\lceil -4.9 \rceil = \lceil -4 - 0.9 \rceil \text{ is } -4$$

We can round to the "closest" integer by

$$\text{Round}(a) = \lfloor a + \tfrac{1}{2} \rfloor \text{ or } \text{Round}(a) = \lceil a - \tfrac{1}{2} \rceil$$

The results will agree for all instances except when a is an exact half point, that is, its fractional component δ is one-half. Let's choose the form $\text{Round}(a) = \lfloor a + \tfrac{1}{2} \rfloor$.

For a line segment from (x_0, y_0) to (x_1, y_1), consider the case at an integer abscissa value X_i with $\Delta x = x_1 - x_0$ and $\Delta y = y_1 - y_0$. Incrementation is $X_{i+1} = X_i + 1$ when $0 < \Delta y < \Delta x$ (the first-octant case) and $x_i, y_i, \Delta x, \Delta y$, and r_i are real while X_i, Y_i are integer. To select the pixel at X_i, Y_i, we first calculate the "true," typically noninteger value y from the line equation

$$y = (\Delta y / \Delta x)x + b$$

where $b = y_0 - (\Delta y / \Delta x)x_0$. We then round y to find Y_i, the "closest" ordinate, by axial displacement error measure.[5]

$$Y_i = \lfloor y_i + \tfrac{1}{2} \rfloor \qquad \text{round to integer ordinate}$$
$$Y_i = \lfloor (\Delta y / \Delta x)X_i + b + \tfrac{1}{2} \rfloor$$
$$Y_i = \lfloor (2\Delta y / 2\Delta x)X_i + (2\Delta x / 2\Delta x)b + (\Delta x / 2\Delta x) \rfloor$$
$$Y_i = \lfloor ((2\Delta y X_i + 2\Delta x b + \Delta x)/(2\Delta x)) \rfloor$$
$$Y_i = \lfloor Y_i + (r_i / 2\Delta x) \rfloor$$

where Y_i is an integer and $0 \le r_i < 2\Delta x$.

To relate current ordinate selection to the next ordinate integer value selected, observe that

$$Y_{i+1} = \lfloor y_{i+1} + \tfrac{1}{2} \rfloor$$
$$Y_{i+1} = \lfloor (\Delta y / \Delta x)X_{i+1} + b + \tfrac{1}{2} \rfloor$$
$$Y_{i+1} = \lfloor (2\Delta y / 2\Delta x)X_{i+1} + (2\Delta x / 2\Delta x)b + (\Delta x / 2\Delta x) \rfloor$$

Recall that $X_{i+1} = X_i + 1$ is the first-octant stepping strategy:

$$Y_{i+1} = \lfloor (2\Delta y / 2\Delta x)(X_i + 1) + (2\Delta x / 2\Delta x)b + (\Delta x / 2\Delta x) \rfloor$$
$$Y_{i+1} = \lfloor ((2\Delta y X_i + 2\Delta x b + \Delta x)/2\Delta x) + (2\Delta y / 2\Delta x) \rfloor$$
$$Y_{i+1} = \lfloor Y_i + (r_i / 2\Delta x) + (2\Delta y / 2\Delta x) \rfloor$$
$$Y_{i+1} = \lfloor Y_i + ((r_i + 2\Delta y)/2\Delta x) \rfloor$$
$$Y_{i+1} = Y_i + \lfloor ((r_i + 2\Delta y)/2\Delta x) \rfloor$$

Recall that $0 \le r_i < 2\Delta x$ and $0 < \Delta y < \Delta x$, so

$$0 < (r_i + 2\Delta y) < 4\Delta x \text{ and}$$
$$0 < (r_i + 2\Delta y)/2\Delta x < 2$$

Hence $\lfloor (r_i + 2\Delta y)/(2\Delta x) \rfloor$ must be either 0 or 1, and for $X_{i+1} = X_i + 1$ we must have either $Y_{i+1} = Y_i + 0$ or $Y_{i+1} = Y_i + 1$ so that from any pixel coordinate, we step to the next pixel coordinate along either a unit axial step or a unit diagonal step. We can make the choice easily by looking at a running error measure r_i.

For our first-octant case,

$$X_{i+1} = X_i + 1 \text{ and } Y_{i+1} = Y_i + \lfloor (r_i + 2\Delta y)/(2\Delta x) \rfloor$$

Thus,

$$\text{if } (r_i + 2\Delta y) < 2\Delta x, \text{ then } \lfloor (r_i + 2\Delta y)/(2\Delta x) \rfloor = 0$$
$$\text{if } (r_i + 2\Delta y) \ge 2\Delta x, \text{ then } \lfloor (r_i + 2\Delta y)/(2\Delta x) \rfloor = 1$$

Note that a simpler, equivalent test is to look just at the sign of $e_i = (r_i + 2\Delta y - 2\Delta x)$ and then update our location and axial error measure accordingly.

If $e_i < 0$, then

$$Y_{i+1} = Y_i + 0 \text{ and } r_{i+1} = r_i + 2\Delta y$$

so that $e_{i+1} = r_{i+1} + 2\Delta y - 2\Delta x$ and $e_{i+1} = e_i + 2\Delta y$ with $X_{i+1} = X_i + 1$ and $Y_{i+1} = Y_i$.

If $e_i \ge 0$, then

$$Y_{i+1} = Y_i + 1 \text{ and } r_{i+1} = r_i + 2\Delta y - 2\Delta x$$

so that $e_{i+1} = r_{i+1} + 2\Delta y - 2\Delta x$ and $e_{i+1} = e_i + 2\Delta y - 2\Delta x$ with $X_{i+1} = X_i + 1$ and $Y_{i+1} = Y_i + 1$.

For initial conditions, pick an integer starting abscissa X_s, then calculate the dependent integer starting ordinate Y_s and initial decision variable e_s. That is, with X_s an integer, find integer Y_s and initial e_s. To begin, let

$$\Phi = 2\Delta y X_s + 2\Delta x y_0 - 2\Delta y x_0 + \Delta x$$
$$= 2\Delta y(X_s - x_0) + 2\Delta x(y_0 + \tfrac{1}{2})$$

Then $Y_s = \lfloor \Phi / 2\Delta x \rfloor$ and $r_s = \Phi - 2\Delta x Ys$. Therefore,

$$e_s = r_s + 2\Delta y - 2\Delta x = \Phi - 2\Delta x Ys + 2\Delta y - 2\Delta x$$

or

$$e_s = 2((\Delta y(X_s - x_0)) - (\Delta x(Y_s - y_0))) + (2\Delta y - \Delta x)$$
$$= 2((\Delta y(X_s - x_0 + 1)) - (\Delta x(Y_s - y_0 + \tfrac{1}{2})))$$

For the special case of an integer starting point, $X_s = x_0$

6 Simplified line-drawing algorithm.

```
Calculate
  DX = Xt − Xs, DY = Yt − Ys, and DZ = |DX| − |DY|
    if DX < 0
      then M21 = −1
      else M21 = +1
    if DY < 0
      then M22 = −1
      else M22 = +1
    if DZ < 0
      then P = |DX|, Q = |DY|, M11 = 0, M12 = M22
      else P = |DY|, Q = |DX|, M11 = M21, M12 = 0
Set initial values
  {X, Y}  ←{Xs, Ys}     Current point
  C       ←Q            Loop count
  K1      ←2P           Axial step constant
  E       ←2P − Q       Decision variable or error term
  K2      ←2P − 2Q      Diagonal step constant
  M1      ←{M11, M12}   Axial step "unit" increment {±1, 0} or {0, ±1}
  M2      ←{M21, M22}   Diagonal step "unit" increment {±1, ±1}
Select pixels
  Write pixel at {X, Y} and decrement C ← C − 1
  While C ≥ 0 execute the following loop
    if (E < 0)
      then {X, Y} ← {X, Y} + M1 and E ← E + K1
      else {X, Y} ← {X, Y} + M2 and E ← E + K2
    Write pixel at {X, Y} and decrement C ← C − 1
```

and $Y_s = y_0$ or any other starting point in which the integer point X_s, Y_s lies on the line, we can use the simplified form $e_{s\_0} = 2\Delta y - \Delta x$. For the general case of noninteger endpoints and starting anywhere along our first-octant line, we can raster to the closest pixel by the axial, normal, or residual measures of error by first setting (X_i, Y_i) to (X_s, Y_s) and e_i to e_0. Then, until the pixel with the ending value of X is written, loop writing a pixel, testing the error term e sign, and updating the coordinate location and error term as follows:

```
Display the pixel at (Xi , Yi) then
  quit if Xi = X ending_integer_value
  otherwise update as
    if ei < 0,
      then ei+1 = ei + 2Δy, Xi+1 = Xi + 1,
        and Yi+1 = Yi
    if ei ≥ 0,
      then ei+1 = ei + (2Δy − 2Δx),
        Xi+1 = Xi + 1, and Yi+1 = Yi + 1
```

Notice that for any given line, the two addends used to update the error term e are constants that can be set outside the loop. Should rounding as $\lceil a - \frac{1}{2} \rceil$ be preferred, then the $e = 0$ case and $e < 0$ instances can be combined so that a diagonal step is taken only when $e > 0$.

All-octants line-drawing algorithm

To see an algorithm that draws lines of all orientations, not just first-octant lines, let's assume the common circumstance of integer endpoint lines. We'll assume that exact half points are an arbitrary choice and retraceability is not a requirement. For a line segment from an integer starting grid point $\{X_s, Y_s\}$ to an integer terminating grid point $\{X_t, Y_t\}$, all variables are integer and the all-octants line-stepping algorithm[6] simplifies to what appears in Figure 6.

Note that the incremental lines for this algorithm are not retraceable. To modify the algorithm for drawing retraceable lines, set the initial value of E as follows: If $(\Delta Y < 0)$ then $E = 2P - Q - 1$; otherwise, $E = 2P - Q$. To see the effect, draw the line from (0, 0) to (2, 1) and back from (2, 1) to (0, 0).

Differences between edges and lines

Edges in raster space differ from mere lines. Runs of pixels along a horizontal, vertical, or diagonal path mean rastered lines can intersect in more than one pixel, while geometric lines would have had only one point in common. Edges bound an area so that pixels on one side of the edge are "inside" pixels while pixels on the other side of the edge are "outside" pixels. A single horizontal run from a rastered line can include both inside and outside pixels, so accounting for the runs is a modification needed if we're to consistently fill adjacent areas.

To illustrate algorithm modification or tailoring to fit a specific new circumstance, consider an edge from an integer-valued starting point (X_0, Y_0) to an integer-valued terminating point (X_t, Y_t) with $0 < \Delta Y < \Delta X$, where pixels to the left of the "true line" are considered "inside" pixels while those to the right are considered "outside" pixels (Figure 7).

Assuming integer endpoints, a change of variable can simplify matters. Let $u = x - X_0$ and $v = y - Y_0$ so that the edge becomes $v = (Y_1 - Y_0)/(X_1 - X_0)u = (\Delta Y/\Delta X)u$ with starting-point coordinates $U_0 = 0$ and $V_0 = 0$.

Here our edge-drawing objective is to find the succession of pixels closest to the intersection of a horizontal line $V = k$ and our edge line $v = (\Delta Y/\Delta X)u$ as k runs from 0 to $V_t = (X_t - X_0)$. This will provide the rightmost pixel along each successive one-pixel-wide horizontal strip across a bounded area having our edge as its rightmost boundary segment. Knowing that $V = k$, we need to find the closest integer U_k that keeps the pixel at $\{U_k, V_k\}$ inside the area or on the edge itself. That's just the floor $U_k = \lfloor u_k \rfloor$. So

$$U_k = \lfloor (\Delta x/\Delta y) V_k \rfloor \text{ or}$$
$$U_k = \lfloor (\Delta x/\Delta y) k \rfloor \text{ or}$$
$$U_k = \lfloor U_k + G_k/\Delta y \rfloor$$

where U_k is an integer, $0 \le G_k < \Delta y$, and

$$U_{k+1} = \lfloor (\Delta x/\Delta y)(V_k + 1) \rfloor \text{ or}$$
$$U_{k+1} = \lfloor (\Delta x/\Delta y)(k + 1) \rfloor \text{ or}$$
$$U_{k+1} = \lfloor U_k + H + (G_k + S)/\Delta y \rfloor$$

where $0 \le S < \Delta y$ and $\Delta x = (H\Delta y) + S$.

Therefore

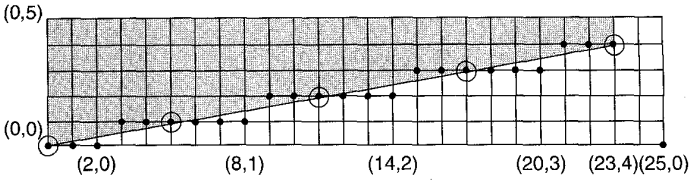
$$U_{k+1} = U_k + H + \lfloor (G_k + S)/\Delta y \rfloor$$

and either $\lfloor (G_k + S)/\Delta y \rfloor = 0$ with $G_{k+1} = G_k + S$ or $\lfloor (Gk + S)/\Delta y \rfloor = 1$ with $G_{k+1} = G_k + S - \Delta y$.

More or less as we did before with line rastering, we can establish a decision variable or error-tracking term $E = ((G + S) - \Delta Y)$ and state an edge-drawing algorithm for this case of first-octant slope with its inside to the left of the edge. The generalization for all slopes and inside to the right as well as to the left is a bit messier than plain line drawing, but still straightforward and not covered here.

Also not addressed here is the conceptual reference question of how to treat pixels exactly on an edge. An exact-edge pixel is common to two area boundaries, so you need to explicitly resolve the matter of a shared pixel and the tri-state distinction of inside, on-edge, and outside that was avoided in traditional geometry with zero-area, uncolored, abstract edges. My point is to demonstrate that modified algorithms are appropriate for modified objectives and to emphasize that it is always necessary to understand clearly any objective function. Before using any algorithm, always understand what it does and what it does not do! In practice, you would concurrently run an edge algorithm for a current left edge and current right edge so that all pixels in between would be filled as the figure—likely a triangle—is traversed by horizontal slices.

Figure 8 presents the rightmost edge form of the edge-drawing algorithm. If you find the broken edges from this algorithm unappealing aesthetically, you can always draw a line boundary to separate filled areas.



7 Edge contrasted to line.

Set initial values

H	$\leftarrow (X_t - X_0)$ DIV $(Y_t - Y_0)$	Basic run-length step size in pixels
S	$\leftarrow (X_t - X_0)$ MOD $(Y_t - Y_0)$	Slope residue
$\{X, Y\}$	$\leftarrow \{X_0, Y_0\}$	Current point
C	$\leftarrow Y_t - Y_0$	Loop count
K_1	$\leftarrow S$	H pixel step constant
E	$\leftarrow S - (Y_t - Y_0)$	Decision variable or error term
K_2	$\leftarrow S - (Y_t - Y_0)$	$(H + 1)$ pixel step constant
M_1	$\leftarrow \{M_{11}, M_{12}\} = \{H, 1\}$	Short step {horizontal, unit diagonal} increment
M_2	$\leftarrow \{M_{21}, M_{22}\} = \{(H + 1), 1\}$	Long step {horizontal, unit diagonal} increment

Select pixels

Write final pixel of horizontal run at $\{X, Y\}$ and decrement $C \leftarrow C - 1$

While $C \ge 0$ execute the following loop

If $(E < 0)$

then $\{X, Y\} \leftarrow \{X, Y\} + M_1$ and $E \leftarrow E + K_1$

otherwise $\{X, Y\} \leftarrow \{X, Y\} + M_2$ and $E \leftarrow E + K_2$

Write final pixel of horizontal run at coordinate point $\{X, Y\}$

Decrement $C \leftarrow C - 1$ and return to test of C at top of loop

8 Edge-drawing algorithm.

Attention to detail

It can be tempting to simplify a drawing algorithm, for example, by swapping endpoints of a line to always draw with a positive Y displacement. It works fine for solid lines and saves a minor bit of memory or circuitry in the rastering program or hardware adapter. When line styles such as dot dashed or user-specified patterns are added, then swapping the endpoints disrupts the correct pattern and incorrect lines are drawn with the pattern reversed. An interesting example I always use in my graphics classes is the mixed implementation seen in Borland's Turbo Pascal. Line drawing for horizontal lines is correctly done for patterned lines. For any line orientation other than horizontal, however, line drawing is incorrectly implemented for lines having a negative delta y displacement, that is, $(Y_{terminate} - Y_{start}) < 0$.

To demonstrate the visual effect produced by endpoint swapping, try the Pascal procedure in Figure 9 (next page) after you first initialize the graphics mode in an IBM-compatible PC. Notice that for horizontal lines the pattern is correctly reversed for the yellow right-to-left and the green left-to-right horizontal lines. For vertical lines, though, the patterned lines are seen to be identical for both the green bottom-to-top and the yellow top-to-bottom lines. The two vertical lines have different directions, so the yellow vertical pattern should be reversed and the vertical green and yellow patterns should differ. The green vertical line is incorrectly drawn.

Conclusion

Rastered pictures have been around a long time. Computer-driven raster displays simply introduce a higher performance, more demanding technology that benefits from systematic rules. To examine the development of a compact, integer-arithmetic algorithm for line rastering, this tutorial looked at some pixel space basics, then considered alternative measures by which closeness to a true line's locus could be evaluated.

The key to effective use of all algorithms is a clear understanding of your objective function and the implications that can arise from incomplete or unintentionally fuzzy assumptions. For example, the edge-drawing algorithm developed here from the line-drawing

```
PROCEDURE DrawPatternedLineThickly;
 {test Borland's Turbo Pascal v.7 patterned lines}
CONST
  MyPattern = $501F;
      {VGA pixel pattern:  0101 0000 0001 1111}
  msg1 = 'Lower horiz. lines: from left to right';
  msg2 = 'Upper horiz. lines: from right to left';
  msg3 = 'horizontal OK per specification';
  msg4 = 'vertical incorrect as end points swapped';
VAR
  i  : INTEGER;
BEGIN
  SetBkColor(Blue);
  ClearDevice;
  SetLineStyle(UserBitLn, MyPattern, NormWidth);
  SetTextStyle(DefaultFont, HorizDir, 2);
  SetColor(LightGreen);
  OutTextXY(10, GetMaxY-85, msg1);
  OutTextXY(360, 45, 'Right Vertical:');
  OutTextXY(345, 65, 'from bottom to top');
  OutTextXY(375, 85, 'drawing');
  OutTextXY(375, 105, 'specification');
  FOR i := 100 TO 107 DO
    Line(10,GetMaxY-i, GetMaxX-10, GetMaxY-i);
  FOR i := 1 TO 6 DO
    Line((i + GetMaxX DIV 2), GetMaxY-125,
         (i + GetMaxX DIV 2), 10);
  SetTextStyle(DefaultFont, HorizDir, 2);
  SetColor(Yellow);
  OutTextXY(10, GetMaxY-150, msg2);
  OutTextXY(20, 45, 'Left Vertical:');
  OutTextXY(10, 65, 'from top to bottom');
  OutTextXY(35, 85, 'drawing');
  OutTextXY(35, 105, 'specification');
  FOR i := 110 TO 114 DO
    Line(GetMaxX-10, GetMaxY-i, 10,GetMaxY-i);
  FOR i := 1 TO 5 DO
    Line((-i + GetMaxX DIV 2), 10,
         (-i + GetMaxX DIV 2), GetMaxY-125);
  SetTextStyle(DefaultFont, HorizDir, 2);
  SetColor(White);
  OutTextXY(375, 205, 'Line Patterns');
  OutTextXY(35, 205, 'Line Patterns');
  OutTextXY(50, GetMaxY-55, msg3);
  OutTextXY(0, GetMaxY-30, msg4);
END; {..quit PROCEDURE DrawPatternedLineThickly..}
```

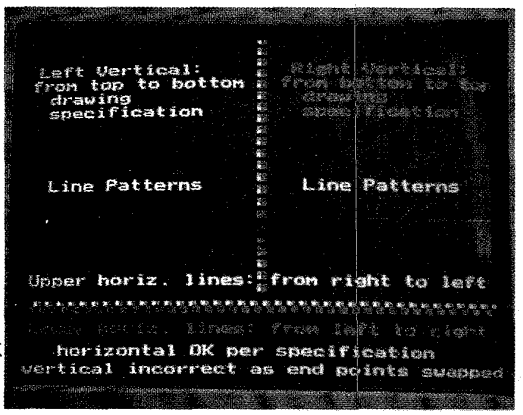


Photo by Joel Nichols

9 Pascal procedure (left) and procedure output (right) showing the effect of inattention to correctly tying together pixel selection and the attribute of line style. (Download Pascal source code from http://www.computer.org/.)

algorithm showed how raster space differs from traditional geometric space. In raster space, lines intersect in multiple pixels, so clipping must be carefully chosen to be by closest pixel or by theoretical intersection. Do you scissor-clip postrastering or geometric-line-clip pre rastering? The visual result can differ. Attributes such as line thickness and patterns also affect the implementation of drawing algorithms.

The computer graphics literature offers many enhancements to simple single-pixel-per-loop iteration. For example, you can select iterations of more than one pixel per loop.[7-10] Beyond the scope of this tutorial is a class of algorithms that set up parallel processing pixel selection.[11-12] All approaches share a need to clearly understand an explicit reference model, as illustrated here. ■

References

1. C.A.Wüthruch and P. Stucki, "An Algorithmic Comparison Between Square- and Hexagonal-Based Grids," *CVGIP*, Vol. 53, No. 4, July 1991, pp. 324-339.
2. ISO Info. Processing Systems, *Interfacing Techniques for Dialogue with Graphical Devices*, Parts 1-6, IS9636, 1991.
3. J.E. Bresenham, "Incremental Circles," in *Display Algorithms–Computer Studies*, R. Earnshaw, ed., Tech. Report 189, Leeds Univ., UK, 1983.
4. M.D. McIlroy, "Getting Raster Ellipses Right," *ACM Trans. Graphics*, Vol. 11, No. 3, July 1992, pp. 259-275.
5. L. Dorst and A.W.M. Smeulders, "Discrete Representation of Straight Lines," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, No. 4, July 1984, pp. 450-463.
6. J.E. Bresenham, "Ambiguities in Incremental Line Rastering," *IEEE CG&A*, Vol. 7, No. 5, May 1987, pp. 31-43.
7. G.W. Gill, N-Step Incremental Straight-Line Algorithms," *IEEE CG&A*, Vol. 14, No. 3, May 1994, pp. 66-72.
8. P. Graham and S.S. Iyengar, "Double- and Triple-Step Linear Interpolation," *IEEE CG&A*, Vol. 14, No. 3, 1994, pp. 49-53.
9. J.G. Rokne, B. Wyvil, and X. Wu, "Fast Line Scan Conversion," *ACM Trans. Graphics*, Vol. 9, Oct. 1990, pp. 356-338.
10. X. Wu and J.G. Rokne, "Double-Step Incremental Generation of Lines and Circles," *CVGIP*, Vol. 37, 1987, pp. 331-334.
11. O. Lathrop, D. Kirk, and D. Voorhies, "Accurate Rendering by Subpixel Addressing," *IEEE CG&A*, Vol. 10, No. 5, Sept. 1990, pp. 45-53.
12. W.E. Wright, "Parallelization of Bresenham's Line and Circle Algorithms," *IEEE CG&A*, Vol. 10, No. 5, 1990, pp. 60-67.

Jack Bresenham *is a graphics consultant to industry and, since 1987, a professor of computer science at Winthrop University. Undergraduate teaching is his primary academic interest. His career at IBM from 1960-1987 included work on line- and circle-drawing algorithms, S/360 RPG, unbundling, token ring, and the IBM 3270 PC-GX graphics display at development laboratories in the US, England, and Italy. Bresenham earned his PhD at Stanford. He is a member of Phi Kappa Phi, Sigma Xi, and ACM, and a senior member of IEEE. He has been a member of the IEEE CG&A editorial board since 1990.*

Readers may contact Bresenham at Winthrop University, Computer Science Department, Thurmond Hall, Rock Hill, SC 29733, e-mail bresenhamj@winthrop.edu.