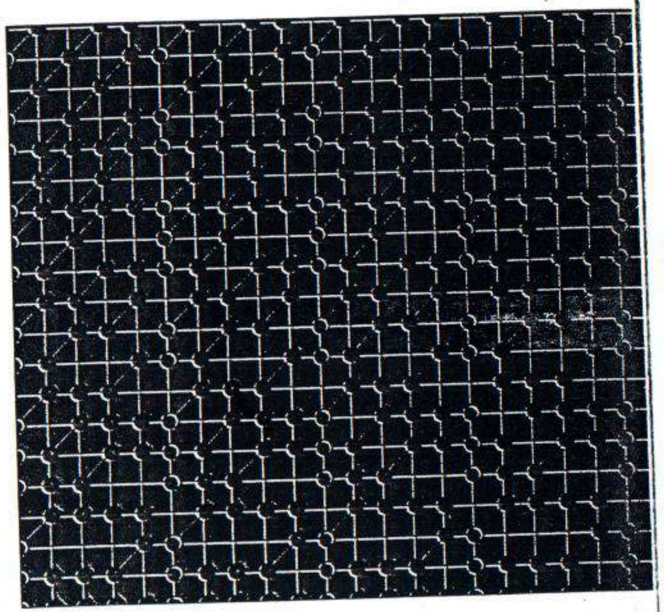
N-Step Incremental Straight-Line Algorithms

Graeme W. Gill
Labtam Australia



This class of algorithms extends Bresenham's integer straight-line algorithm to generate more than one pixel per inner loop, thus reducing inner loop overhead.

Although the emphasis in computer graphics research has shifted away from line-based object representations, users still expect all modern graphics display devices to draw straight lines efficiently. The problem of digitizing a line into a uniform grid (raster) has many solutions. Bresenham's algorithm[1] uses only integer arithmetic and generates one pixel per inner loop. There are many acceleration techniques for it.

The emergence of RISC microprocessors offers new choices for implementing cost-effective, high-performance graphical display systems.[2] The straight-line drawing algorithm I describe here was developed to help make a general-purpose CPU competitive in speed with more specialized graphics processing hardware. It is based on Bresenham's single-step integer algorithm, but it improves pixel-generation efficiency by generating four pixels per loop. It also permits the writing of adjacent horizontal pixels to be compressed into more efficient multipixel write instructions that are available in the most popular CPU/frame buffer architectures. The algorithm achieves still greater efficiency by exploiting the observation that some four-pixel sequences are more likely to occur than others.

Bresenham's straight-line algorithm is the first in a family of *N*-step algorithms. In addition to illustrating the form of these algorithms with the quad-step case ($N = 4$), this article describes a technique for generating other members of the family. Figure 1 lists the variables used here, together with an outline of Bresenham's algorithm for lines in the first octant. The nomenclature is similar to that used by Bresenham in a 1987 *CG&A* article.[3]

Quad-step algorithm

Bresenham's algorithm shows that the sign of E (the error term) exactly predicts the pixel to be written at any step in the process of rasterizing a line. Therefore, to illustrate the quad-step case, we want a test set that predicts the axial and diagonal moves up to four pixels in advance. The possible combinations of four axial or diagonal moves are

XXXX XXXD XXDX XXDD XDXX XDXD XDDX XDDD
DXXX DXXD DXDX DXDD DDXX DDXD DDDX DDDD

where X represents an axial move and plot, and D represents a diagonal move and plot. At each step in the four moves, E must be of the appropriate sign to give the next step. In addition, the range of E must stay within the bounds $K2 \le E < K1$ as is the case in the single-step algorithm.

Note: An early version of this article, titled "Another Quad Step Incremental Line Algorithm," was published in the proceedings of Ausgraph 90, held in Melbourne, Australia, Sept. 1990.

0272-17-16/94/$4.00 © 1994 IEEE

Figure 1. Variables and outline of Bresenham's algorithm for lines in the first octant (inner loop).

```
Q   = max |x1 - x0|, |y1 - y0|
P   = min |x1 - x0|, |y1 - y0|
Da  is the frame store pixel address.
M1  is the address increment needed to move the pixel address
      in the axial direction.
M2  is the address increment needed to move the pixel address
      in the diagonal direction.
E   is the discriminator or error term.
    = 2P - Q
K1  is the error increment for an axial move (K1 will be ≥ 0).
    = 2P
K2  is the error increment for a diagonal move (K2 will be ≤ 0).
    = 2(P - Q)
C   is the loop counter.
    = Q.

write pixel at (Da)
while C != 0 {
    while E < 0 and C != 0 {          /* do an axial move */
        C = C - 1
        Da = Da + M1                  /* move the address axially */
        E = E + K1                    /* adjust the error term */
        write pixel at (Da)
    }
    while E ≥ 0 and C != 0            /* do a diagonal move */
        C = C - 1
        Da = Da + M2                  /* move the address diagonally */
        E = E + K2                    /* adjust the error term */
        write pixel at (Da)
    }
}
```

For each quad move—called a *glyph*—we can generate a set of equations that must be satisfied for that glyph to be next in the sequence making up a line. For example, XDXX must satisfy the equations

$E < 0$	for X
$E + K1 \ge 0$	for X, D
$E + K1 + K2 < 0$	for X, D, X
$E + 2K1 + K2 < 0$	for X, D, X, X
$K2 \le E < K1$	at all times

These conditions can be simplified to the following:

if $K1 \le -K2 < 2K1$
  then $-K1 \le E < -2K1 - K2$
else if $2K1 \le -K2 < 3K1$
  then $-K1 \le E < 0$

The structure of the simplified conditions shows that choosing a glyph depends on the ratio of $K1$ to $K2$—that is, on the slope of the line. This implies that we must classify lines not only into their appropriate octant but also into their appropriate suboctant to ensure an unambiguous choice of glyph. The conditional equations for the glyphs XXDD and DDXX show that these conditions can never be met. In other words, these glyphs are never generated in drawing a straight line with the same pixelization as Bresenham's algorithm. This leaves 14 usable glyphs. Figure 2 illustrates them, together with the change in error term when the glyph is plotted.

Figure 2. The 14 usable glyphs.

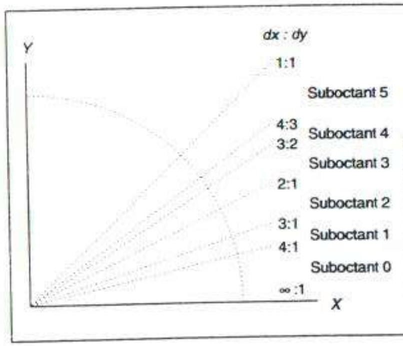
Glyph	Error change
DDDD	4K2
DDDX	K1 + 3 K2
XDDD	K1 + 3 K2
DDXD	K1 + 3 K2
DXDD	K1 + 3 K2
XDDX	2 K1 + 2 K2
DXDX	2 K1 + 2 K2
XDXD	2 K1 + 2 K2
DXXD	2 K1 + 2 K2
XDXX	3 K1 + K2
XXDX	3 K1 + K2
DXXX	3 K1 + K2
XXXD	3 K1 + K2
XXXX	4 K1

Pattern

Doing the arithmetic for all the glyphs ultimately generates the suboctant classification illustrated on the next page by Figure 3 and its associated tabular data and by Table 1, which shows the conditions for choosing the next glyph in each suboctant. Figure 4 shows the overall organization of the algorithm, and how a line is classified first into its octant, then into its suboctant, with each suboctant making use of one of five possible glyphs.

There are some restrictions on the sequence of glyphs within a suboctant. For instance, a line in suboctant 1 will never have four or five Xs in a row, because such a line would be in suboctant 0. So XDXX is never followed by XXXD or XXDX. We can work this out by looking at the condition equations for the concatenation of all combinations of suboctant glyphs; but additional information is obtained by simulating the quad-step routine, running through a set of lines with a desired distribution of slopes, and keeping a record of which glyph in a suboctant follows another. The examples in this article use a uniform distribution of slopes, in the absence of definitive slope-distribution information for "typical" applications.

In addition to showing which glyphs never follow one another, a simulation gives some indication of the probability of one particular glyph following another. This allows the deci-



Suboctant number	K1 / K2	Condition
5	3 to ∞	$-3K2 < K1$
4	2 to 3	$-2K2 < K1 \le -3K2$
3	1 to 2	$-K2 < K1 \le -2K2$
2	1/2 to 1	$K1 \le -K2 < 2K1$
1	1/3 to 1/2	$2K1 \le -K2 < 3K1$
0	0 to 1/3	$3K1 \le -K2$

Figure 3. Suboctant classification.

Table 1. Glyph decision conditions within suboctants plus next-glyph probability.

Suboctant	Glyph	Decision Point	Glyph	Decision Point	Glyph	Decision Point	Glyph	Decision Point	Glyph
0	$E <$ XXXX 0,4,3,2,1	$-3K1$	$\le E <$ XXXD 0,1	$-2K1$	$\le E <$ XXDX 0,1,2	$-K1$	$\le E <$ XDXX 0,2,3,1	0	$\le E$ DXXX 0,3,4,2,1
1	$E <$ XXXD 1,0	$-2K1$	$\le E <$ XXDX 2,1	$-K1$	$\le E <$ XDXX 2,4,3	0	$\le E <$ DXXX 4,3	$-2K1-K2$	$\le E$ DXXD 1,0
2	$E <$ XXDX 3,4,1,2	$-K1$	$\le E <$ XDXX 4,3	$-2K1-K2$	$\le E <$ XDXD 2,0,1	0	$\le E <$ DXXD 0,1,2	$-K1-K2$	$\le E$ DXDX 4,2,3,1
3	$E <$ XDXD 0,2,1,3	$-K1-K2$	$\le E <$ XDDX 4,3,2	0	$\le E <$ DXDX 2,4,3	$-K1-2K2$	$\le E <$ DXDD 0,1	$-K2$	$\le E$ DDXD 1,0,3,2
4	$E <$ XDDX 3,4	$-K1-2K2$	$\le E <$ XDDD 0,1	0	$\le E <$ DXDD 2,0,1	$-K2$	$\le E <$ DDXD 2,3	$-2K2$	$\le E$ DDDX 3,4
5	$E <$ XDDD 4,1,0,2,3	0	$\le E <$ DXDD 4,2,1,3	$-K2$	$\le E <$ DDXD 4,3,2	$-2K2$	$\le E <$ DDDX 4,3	$-3K2$	$\le E$ DDDD 4,0,1,2,3

sion tree after each glyph plot to test E against glyphs in decreasing order of probability. Looking again at Table 1, we can see a summary of this information from a simulation for lines in a 2,048-square grid. The information appears as a list of the number of the glyphs in that suboctant, in the order of their frequency of following the given glyph. For example, in suboctant 1, glyph 2 (XDXX) has the list 2, 4, 3. This means that it is most frequently followed by XDXX, DXXD, and DXXX, and that it is never followed by XXXD and XXDX. I have run a similar simulation on a 16,000-square grid to compare the results pixel by pixel with the output of a conventional single-step routine. The results verified that the quad-step implementation behaves the same as Bresenham's integer line algorithm.

An important advantage in drawing more than one pixel at a time is that if we have separate code for the x- and y-driven octants, then in the x-driven octants it becomes possible to process a series of plots connected by axial moves as a single multipixel plot. For example, if we use a 32-bit word, byte-addressable machine to execute the quad algorithm and if there is one pixel per byte, then we could implement the glyph DXXX as a diagonal move, followed by a four-pixel write, followed by a triple axial move. This means that near-horizontal lines will execute with fewer internal steps and memory operations. By prealigning x-driven lines to 32-bit boundaries, we can also do this on machine architectures that enforce strict data alignment.

If it is important to always draw a line from the start point to the end point, then three rather than two different versions of the code are required: one for y-driven lines, one for x-driven lines toward the right ($+x$), and one for x-driven lines toward the

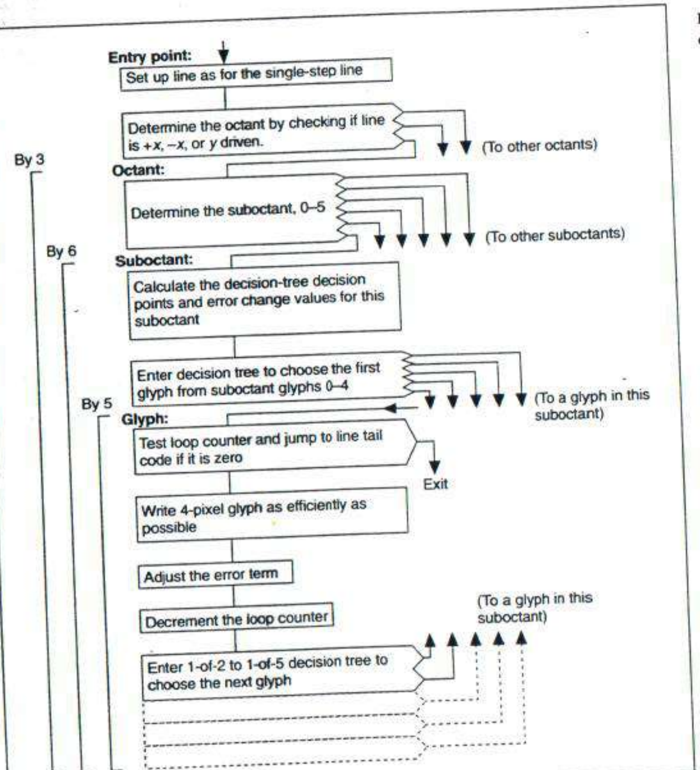


Figure 4. Structure of the full quad-step algorithm.

left ($-x$). This is because a processor will write multiple pixels in one direction from a given address. The full quad-step algorithm referred to in the rest of this article is this more general y, $+x$, and $-x$ version.

There is symmetry between the X and D moves. For example, suboctants 0, 1, and 2 mirror octants 5, 4, and 3, respectively. Bresenham[4] demonstrated this analytically. This symmetry works as a coding cross-check; it can also halve the code size needed for the algorithm. If the consecutive horizontal pixel code is not implemented, then we can fold suboctants 3, 4, and 5 into suboctants 2, 1, and 0. This variation is referred to as the compact quad-step algorithm. The following code fragment illustrates this folding:

```
if -K2 ≥ K1 then   /* if we are in suboctants 5, 4, or 3 */
   K1 = -K1         /* reflect suboctant about x = 2y axis */
   K2 = -K2
   exchange (K1, K2)
   exchange (M1, M2)
   E = -E -1        /* reflect error */
```

One advantage of using the same discriminator as Bresenham's algorithm is that we can use the same techniques for making the line retraceable. For instance, we could subtract 1 from E when $dy < 0$.[2,3]

Figure 5 on the next page illustrates the glyph decision conditions graphically for the quad-step case. We can visualize the cases of glyph lengths from 1 to 3 within this illustration by coalescing the regions that contain the shorter glyphs. From this, we can see that a triple-step algorithm needs eight glyphs and four suboctants, and a double-step algorithm needs four glyphs and two suboctants.

Performance

Table 2 presents various metrics of a single-step Bresenham algorithm, the compact quad-step algorithm, the full quad-step algorithm, and the eight-suboctant algorithm.[5] The estimated costs per pixel equal the total of accumulated primitive counts per pixel, recorded in software simulations of the algorithms. The primitive unit costs very roughly correspond to those of an Intel 80960KA/B processor,[6,7] but should apply to many others.

Initially, I implemented the algorithm for an Intel 80960K series processor. Items 8 to 10 of Table 2 show the results of running at 20 MHz with a packed 8-bit/pixel frame store, the code and frame store memory system having three wait states. These results are the ultimate pixel-rendering rate (factoring out the

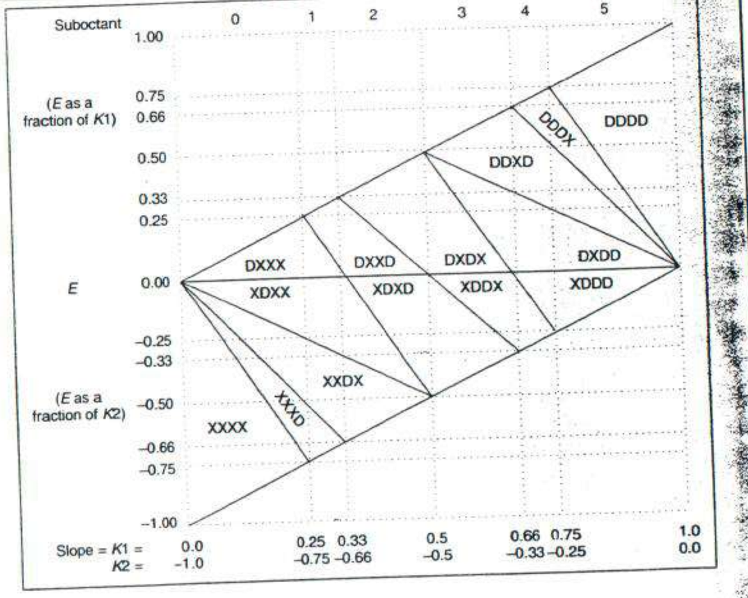


Figure 5. Quad-step decisions within an octant.

setup overhead) for lines enumerated on a large grid. Items 11 to 13 show the results of a subsequent implementation, fine-tuned for the more recent 80960CA processor. The 80960CA ran at 25 MHz with a memory system having two wait states. The improvement in performance between the single-step and quad-step algorithms is close to the rough estimate predicted by the primitive costing. The compact quad-step algorithm is approximately one-fourth the size and operates with some performance penalty compared to the full algorithm.

The actual implementation tested uses the quad-pixel algorithm only for lines longer than a minimum length, for example, 32 pixels. This approach minimizes the impact of the routine's slightly greater setup overhead on short lines. A conventional one-pixel-at-a-time routine is needed anyway, to clean up after the maximum four pixels have been processed, and possibly to prealign the start of the x-driven lines, thereby avoiding word boundary crossings.

On the machine tested, all the code for a suboctant fits into the on-chip instruction cache. This frees the memory interface for pixel writes. It also ensures that the speed is not limited by the rate at which instructions can be fetched from main memory—an important consideration if larger numbers of steps per inner loop are contemplated.

Note that the 80960K results are limited primarily by the decision-making (rasterizing) process rather than the memory bandwidth (pixel writing). The 80960CA, on the other hand, represents a trend in modern processors in which improvements in on-chip operational speed outstrip improvements in off-chip access time. Thus, the 80960CA performance uses 96 percent of the theoretical memory bandwidth, while the 80960K uses only 54 percent. The 82 percent usage by the full 80960CA quad-step algorithm indicates that internal operation overhead is still significant. Future developments in processors seem likely to further improve internal speed without proportionally improving memory access. Thus, the most important attribute of the full quad-step line algorithm might be the extra memory bandwidth it provides.

Comparison with other algorithms

An algorithm that takes advantage of pixel runs in the axial or diagonal direction is generally least efficient when the slope of the line is such that the runs become very short.[4] It is most efficient when the runs are long. In contrast, an N-step algorithm provides an almost constant speedup in all directions and can still take some advantage of runs by plotting pixels N at a time.

A symmetry-based speedup can halve the pixel address calculation time. It can also be combined with other speedup techniques,[8] but it is complicated by the ambiguous case of lines crossing exactly midway between two integer coordinates. Further, symmetry-based speedup applies only to lines with integer end points. Within a windowing graphics system, all primitives might be clipped to arbitrary boundaries. In the worst case (a line cut in half by a window boundary), the symmetry property cannot be used at all. The symmetry-based speedup reduces scan conversion costs, but it does nothing about memory bandwidth limits. This is its principle disadvantage.

Bao and Rokne[5] developed a quad-step algorithm in a somewhat different manner. Table 2 makes some comparisons between their eight-suboctant algorithm and the six-suboctant algorithm described here. In the 80960 architecture, comparisons are cheap and taken jumps are expensive. This is representative of many current machines (both RISC and CISC designs) and tends to indicate that the six-suboctant routine will be faster, since it uses fewer taken jumps. The difference is slight, however, and in practice the two algorithms will run at comparable speeds.

Table 2. Algorithm metrics.

Performance Factors	Single-Step (Bresenham's) Algorithm	Compact Quad-Step Algorithm	Full Quad-Step Algorithm	Eight-Suboctant Quad-Step Algorithm
Suboctant code size (Bytes)		390	390	702
Total code size (Bytes)	160	1,740	7,020	14,976
Arithmetic (2 units) (Primitives/pixel)	3.0		1.5	1.5
Writes (3 units) (Primitives/pixel)	1.0		1.0	1.0
Taken jumps (6 units) (Primitives/pixel)	0.75		0.316	0.471
Not-taken jumps (3 units) (Primitives/pixel)	1.5		0.438	0.25
Cost/pixel (Units/pixel)	18		9.21	9.58
80960KA/B performance (Mpixels/second)	1.24	2.18	2.39	
Bandwidth limit (Mpixels/second)	4.00	4.00	5.24	
Utilization (Percent of limit)	31	54	46	
80960CA performance (Mpixels/second)	2.68	6.02	6.74	
Bandwidth limit (Mpixels/second)	6.25	6.25	8.19	
Utilization (Percent of limit)	43	96	82	

The size estimates in Table 2 assume that the eight- and six-suboctant routines are similarly implemented and hence use a similar amount of code per glyph write routine. The eight-suboctant algorithm has eight suboctants of eight glyphs to choose from, while the six-suboctant routine has six suboctants of five glyphs; hence, the 2:1 code size difference. Simulation results indicated that the different choice of discriminator in the eight-suboctant [algorithm] leads to different pixelization from that of Bresenham-based algorithms. There is no indication that the eight-suboctant algorithm is any less accurate, just that it handles cases where $E = 0$ differently and therefore cannot be used as a transparent replacement for Bresenham's line routine.

The eight-suboctant algorithm's different discriminator also makes it necessary to formulate a single-step or per-pixel clipped version of the algorithm to complete the tail case. Using Bresenham's algorithm for this purpose would introduce extra complexity in calculating a discriminator and lead to unacceptable results in a windowing system where region clipping might cause a given straight line to be drawn in several small pieces. In addition, a programmer using the system might find it disconcerting to have, say, a dashed line use different pixelization from a solid line. Finally, the eight-suboctant algorithm does not seem to be symmetrical about the arctan(1/2) line, which means it is impossible to develop a compact version of the algorithm.

N-step algorithms

The process used to generate the quad-step algorithm can be generalized for any number of pixels per inner loop. I have implemented the techniques described below in a program that automatically generates algorithm information equivalent to that in Table 1 for N from 1 to 32. I then used the tables in line-drawing simulations. These techniques can also determine which glyphs within a suboctant may legally follow each other.

The basic technique is a number sieve. For a given N, all possible combination of X and D are generated and sieved to discover the glyphs that could be part of a Bresenham-algorithm-generated line and the range of line slopes over which the glyphs could be used. The sieve test examines all possible substrings within the string of length N and computes either a lower or

Pixels/ Step	Suboctants	Glyphs/ Suboctant	Average Decision Tree Size	[illegible] Glyphs
4	6	5	3.07	14
8	22	9	3.13	76
16	80	17	3.32	498
32	324	33	3.26	3,650

Table 3. Larger numbers of pixels/step.

upper slope limit depending on whether the string ends in an X or a D. The slope is computed as the ratio of $K1$ over $-K2$ and therefore varies from 0 to ∞. If the string being tested ends in an X, then a possibly new upper limit is computed as

(number of Ds in the string + 1)/(number of Xs in the string − 1)

If the string being tested ends in a D, then a possible new lower limit is computed as

(number of Ds in the string − 1)/(number of Xs in the string + 1)

Any glyph with an upper limit less than or equal to its lower limit cannot be part of a Bresenham algorithm straight line. The boundaries of the suboctants are the slopes at which any glyph enters or exits its usable range. A glyph's usable range often covers several suboctants.

If we regard a glyph as a binary number with the most significant bit on the left, with X taking the value 1, and with D taking the value 0, then we can order the glyphs for testing against the lowest to highest values of E by listing them from greatest to least binary value (see the rows in Table 1 for $N = 4$). Note that the number of glyphs within a suboctant is always $N + 1$.

The decision condition between two glyphs within a suboctant is the negative value of the left-most common string of Xs and Ds, when expressed as the sum of $K1$s and $K2$s, respectively (again, see Table 1 for the $N = 4$ case.) When a glyph is plotted, we calculate the change in the error term simply as the number of Xs and Ds expressed as the sum of $K1$s and $K2$s, respectively. We can check the legality of one glyph in an octant following another by applying the sieve test to the concatenation of the glyph sequence being considered, restricted to the suboctant range in question.

Conclusion

The quad-step algorithm is too large to justify its use in older hardware with limited memory space, but it can be viable in the context of modern memory and software sizes. Because the algorithm reduces both calculation overhead and the number of memory accesses for adjacent pixels, it can improve the performance of current systems that are limited in their processor speed and of future systems that might be limited in their memory speed. The algorithm gives results identical to those from Bresenham's single-step routine while drawing pixels in the expected direction from start to end point—advantages not shared by all fast line-drawing algorithms. Furthermore, as the gradual trend towards more bits per pixel continues, a processor supporting multi-word burst data instructions could make good use of this algorithm in speeding up line drawing into a 24-bits-per-pixel, one-pixel-per-word color frame buffer.

I chose to implement the value $N = 4$ because it gave a useful performance improvement without exceeding the resources (cache and register space) of the target processor, and it was small enough to hand code. However, the techniques described here can be used to construct a straight-line algorithm that generates more than four steps per loop. For larger values of N, it seems desirable to generate code automatically using the output of the N-step sieve. Table 3 summarizes the scope of algorithms with values of N that are greater than four in power-of-two increments (to satisfy alignment restrictions). The relatively small average decision tree sizes indicate that algorithms of greater than four pixels per step might further improve line-drawing efficiency. ❑

References

1. J.E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems J.*, Jan. 1965, pp. 25-30.
2. J. Boothroyd and P.A. Hamilton, "Exactly Reversible Plotter Paths," *Australian Computer J.*, Jan. 1965, pp. 25-30.
3. J.E. Bresenham, "Ambiguities in Incremental Line Rastering," *IEEE CG&A*, Vol. 7, No. 5, May 1987, pp. 31-43.
4. J.E. Bresenham, "Run Length Slice Algorithm for Incremental Lines," in *Fundamental Algorithms for Computer Graphics*, R.A. Earnshaw, ed., Springer-Verlag, New York, 1985.
5. G. Bao and J. G. Rokne, "Quadruple-Step Line Generation," *Computers & Graphics*, Vol. 13, No. 4, 1989, pp. 461-469.
6. *80960KB Programmers Reference Manual*, Intel Literature Sales, Santa Clara, Calif., 1988.
7. G.J. Myers, D.L. Budde, The 80960 Microprocessor Architecture, Wiley, New York, 1988.
8. J. G. Rokne, B. Wyvill, and X. Wu, "Fast Line Scan-Conversion," *ACM Trans. Graphics*, Vol. 9 No. 4, Oct. 1990, pp. 376-388.

Graeme W. Gill is a design engineer at Labtam Australia. His work focuses on high-performance digital design and the optimization of graphics software and hardware architecture for Labtam's X11 server-based products. In his spare time, he maintains xli, an X11 image viewer. Other technical interests include audio and video technologies. Gill received his BE from Royal Melbourne Institute of Technology in 1984. He is a member of IEEE and ACM.

Readers can contact Gill at Labtam Australia Pty. Ltd., PO Box 297 Mordialloc, Victoria, Australia, 3195; e-mail graeme@labtam.oz.au.

Readers interested in a copy of the author's implementation of the N-pixel/step table generator should contact him by e-mail.