Using Program Transformations to Derive Line-Drawing Algorithms

ROBERT F. SPROULL
Carnegie-Mellon University

A wide variety of line-drawing algorithms can be derived by applying program transformations to a simple, obviously correct algorithm. The transformations increase the speed of the algorithm and eliminate the need for floating-point computations. We show how Bresenham's algorithm can be derived in this way. The transformations are then used to derive several variants of Bresenham's algorithm, designed for use on displays that can generate more than one pixel at a time. The treatment shows a complete, extended example of the practical use of program transformations.

Categories and Subject Descriptors: D.1 [**Software**]: Programming Techniques; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Program transformation, line-drawing, raster graphics

1. INTRODUCTION

Many computer graphics devices use line-drawing algorithms to approximate straight lines by displaying dots that are constrained to lie on a discrete grid. Incremental pen plotters that move a pen in small steps require such a line-generation algorithm. Point-plotting CRT displays and electrostatic plotters use the algorithms to approximate straight lines. More recently, frame buffer, raster scan displays use these algorithms to identify the picture elements (pixels) that should be illuminated to display a line.

Simplicity and speed are the key design criteria for line-drawing algorithms because the computations are often implemented in hardware in order to achieve high line-generation speeds. It appears that the early popularity of the binary rate multiplier (BRM) was due entirely to simplicity, for it generates rather poor approximations to straight lines. The digital differential analyzer (DDA) gener-

---

This research was sponsored by the Defense Advanced Research Projects Agency, ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551.

The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Author's present address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0730-0301/82/1000-0259 $00.75

ates better approximations to the true line, but requires an iterative loop that may average almost two cycles to generate each point. An algorithm devised by J. E. Bresenham [4] dominates the DDA: it generates the optimal line, in the sense described in Section 2; it requires only integer additions and subtractions; and it generates one output point for each iteration of the inner loop.

To achieve very high line-generation speeds, an algorithm must compute the location of several points on a line at once. Such multipoint algorithms have several applications, chiefly in raster scanned systems that can write more than one pixel at a time into the image. The investigation of multipoint algorithms was motivated by the 8×8 frame-buffer display [14], which can in one memory cycle write a square region with 8 pixels on a side located anywhere on the screen.

This paper shows how simple program transformations are used to derive all of these algorithms, starting from obviously correct algorithms based on simple analytic geometry. These transformations assure us that the more efficient but more complex algorithms are correct, because they have been derived by correct transformations from a correct algorithm. Transformation techniques of different sorts are used in optimizing compilers [1, 2], are recommended to programmers for improving their programs [3, 11], and are part of research into automatic program improvement [7].

2. LINE-DRAWING PRELIMINARIES

The line-drawing problem is to determine a set of pixel coordinates (x, y), where x and y are integers, that closely approximates the line from the point (0, 0) to the point (dx, dy), for integer values of dx and dy. The assumption that one line endpoint is at the origin loses no generality because lines with other origins are simply translations of the line with origin (0, 0). Additionally, lines are restricted to the first octant: $0 \leq dy \leq dx$. Again, this assumption loses no generality because an arbitrary line can be generated by transposing the canonical line or by reflecting it about one of the principal axes.

The objective of a line-drawing algorithm is to enumerate those pixels that lie close to the *true line*, the mathematical line from (0, 0) to (dx, dy). Figure 1 illustrates a typical line, showing with circles the pixels that correspond either to spots illuminated by a CRT beam on a raster display or to the swath of a plotter pen. Notice that integral values of coordinates locate pixel centers.

While a line may be displayed using many different pixel configurations, one configuration is usually preferred. The preference arises because some configurations approximate the true line more closely than others, some appear to have more uniform pixel density, or brightness, than others, and so on. Many of the algorithms presented in this paper generate the *optimal line*, defined as follows:

1. The optimal line illuminates exactly one pixel in each vertical column. This assumption depends on the fact that the line's extent in x exceeds its extent in y. The purpose of this choice is to limit variations in pixel spacing.
2. Within each column, the pixel illuminated is the one closest to the true line.

To display the optimal line, the line-drawing algorithm must compute, for each integer x_i, the coordinate y_i of the pixel that should be illuminated. The coordinate y_t of the true line is simply $y_t = (dy/dx)x_i$. Illuminating a pixel centered at y_i

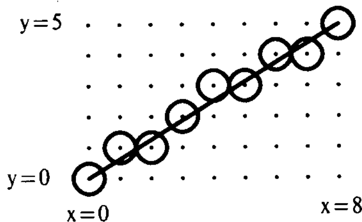

Fig. 1. The line from (0, 0) to (8, 5). Small dots represent pixel centers. The solid line represents the true line. Circles show the pixels that are illuminated to display the optimal line.

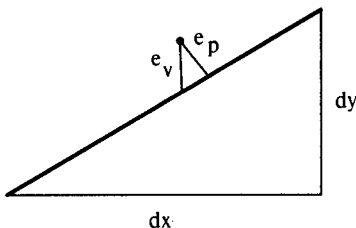

Fig. 2. Illustration of the relationship between the vertical distance e_v and the perpendicular distance e_p.

introduces an error $e_v = y_i - y_t = y_i - (dy/dx)x_i$, measured along the y-axis. The error e_p measured perpendicular to the line can be determined using similar triangles (Figure 2): $e_p = (dx/(dx^2 + dy^2)^{\frac{1}{2}})e_v$. Thus, for any given line, e_p is simply a constant times e_v. Consequently, determining y_i by minimizing the error e_v will identify the pixel that is closest to the line, using either vertical or perpendicular distance measures.

The errors can be minimized if y_i is computed by rounding y_t: $y_i = \text{round}(y_t)$, or equivalently, $y_i = \text{trunc}(y_t + 1/2) = \lfloor y_t + 1/2 \rfloor$. (Recall that the floor function, $\lfloor x \rfloor$, denotes the greatest integer less than or equal to x.) With this choice, $e_v = \lfloor y_t + 1/2 \rfloor - y_t$, so $-1/2 < e_v \leq 1/2$.

3. DERIVATION OF THE BRESENHAM ALGORITHM

The minimum error formulation of the optimal line leads directly to a simple algorithm that enumerates all the points on the optimal line, and which can be expressed in a PASCAL-like language:

A1

```
var yt: exactreal; dx, dy, xi, yi: integer;
for xi := 0 to dx do begin
    yt := [dy/dx]*xi;
    yi := trunc(yt + [1/2]);
    display(xi,yi)
end
```

Although this procedure is expressed using programming language constructs, it requires that precise real arithmetic be used; floating-point approximations are

not permitted. To emphasize this precise arithmetic, variables that use it are declared to have type *exactreal.* Square brackets enclose expressions whose values do not change during iteration of the loop; these expressions can be computed only once, before the loop is entered, and saved in temporary variables. We shall also maintain that multiplications by a power of two do not require multiplication operations, but can be achieved by addition or arithmetic shifting.

Strength reduction. The next version of the algorithm is derived from A1 by observing that y_t can be calculated *incrementally* by adding the quantity (dy/dx) on each iteration. Converting multiplications into repeated additions is one of a number of incremental techniques used frequently in computer graphics algorithms. These techniques make incremental changes to the state of an algorithm or data structure rather than recomputing a result from scratch.

A2

```
var yt: exactreal; dx, dy, xi, yi: integer;
yt := 0;
for xi := 0 to dx do begin
    yi := trunc(yt +[1/2]);       * assert yt = (dy/dx)xi *
    display(xi,yi);
    yt := yt + [dy/dx]
end
```

Substitution of variable. A simple transformation substitutes

$$y_s = y_t + 1/2. \tag{1}$$

A3

```
var ys: exactreal; dx, dy, xi, yi: integer;
ys := 1/2;
for xi := 0 to dx do begin
    yi := trunc(ys);      * assert ys = (dy/dx)xi + 1/2 = yt + 1/2 *
    display(xi,yi);
    ys := ys + [dy/dx]
end
```

Representation change. Algorithm A3 is further transformed by breaking y_s into integer and fractional parts: y_{si}, which will take on only integer values, and y_{sf}, which will hold only fractional values. Thus

$$y_s = y_{si} + y_{sf} \tag{2}$$

$$0 \leq y_{sf} < 1. \tag{3}$$

This substitution requires that the incremental step $(y_s := y_s + [dy/dx])$ be changed to add the increment to the fractional part (y_{sf}) and then test whether the result exceeds 1 (i.e., to see if it is no longer fractional).

A4

```
var ysf: exactreal; dx, dy, xi, ysi: integer;
ysi := 0; ysf := 1/2;
```

```
for xi :=0 to dx do begin
   * assert ysi + ysf = yt + 1/2 *
   display(xi,ysi);
   if ysf + [dy/dx] ≥ 1 then begin
      ysi := ysi + 1;
      ysf := ysf + [dy/dx −1]
   end else begin
      ysf := ysf + [dy/dx]
   end
end
```

Substitution of variable. Algorithm A4 can be transformed into the Bresenham algorithm by replacing the use of y_{sf} with that of a variable r:

$$r = 2\,dy + 2(y_{sf} - 1)\,dx. \tag{4}$$

The objectives of this transformation are to change the comparison in the inner loop to a sign check (i.e., a comparison with 0), and to eliminate division operations by scaling by 2 dx. Making the appropriate substitution of r into A4 yields the Bresenham algorithm:

A5

```
var dx, dy, xi, ysi, r: integer;
ysi := 0; r := 2*dy − dx;
for xi := 0 to dx do begin
   * assert yt + 1/2 = ysi + ysf = ysi + :(r + 2dx − 2dy)/2dx *
   display(xi,ysi);
   if r ≥ 0 then begin
      ysi := ysi + 1;
      r := r − [2*dx − 2*dy]
   end else begin
      r := r + [2*dy]
   end
end
```

The Bresenham algorithm is ideal for implementation in hardware or microprocessors with limited arithmetic power. The algorithm requires neither division nor multiplication, and requires no floating-point approximations because all variables take on only integer values. Moreover, r is not required to hold large values. Equations (3) and (4) imply

$$2\,dy - 2\,dx \le r < 2\,dy. \tag{5}$$

If $0 \le dy \le dx \le 2^n - 1$, r is bounded by

$$-2^{n+1} + 2 \le r < 2^{n+1} - 2. \tag{6}$$

Thus if dx and dy are n-bit positive integers, r requires at most $n + 2$ bits in a two's complement representation.

Interpretation of r. The value of r is related to the vertical error e_v, the distance from the pixel center to the true line. The errors will be identical for all algorithms given above, because the same sequence of points is generated. When *display* is called, $e_v = y_{si} - y_t$. Using eq. (1) to substitute for y_t, and then eq. (2) to substitute

for y_s, we have

$$e_v = y_{si} - \left(y_s - \frac{1}{2}\right) = y_{si} - \left(y_{si} + y_{sf} - \frac{1}{2}\right) = \frac{1}{2} - y_{sf}.$$

Applying transformation of eq. (4) yields

$$e_v = \frac{-r}{2\,dx} - \frac{1}{2} + \frac{dy}{dx}.$$

The value r is thus linearly related to e_v, but is offset by 1/2 due to the loop's initial conditions, offset by (dy/dx) because r has already been computed for the next point along the line when *display* is called, and scaled by 2 dx to require only integral values of r.

Summary. All of the algorithms developed in this section compute the same sequence of points (x_i, y_i) that approximate the true line. Mathematical and program transformations are used to derive efficient implementations. The algorithms are usually adapted to draw lines in any octant by making separately customized versions for each octant.

4. AN *n*-STEP ALGORITHM

Before exploring multipoint algorithms, we illustrate the transformation techniques developed in the previous section by deriving an algorithm that takes horizontal steps of n units in x. Such an algorithm will generate every nth point on the line. We start with an obvious variant of A1:

N1

```
var yt: exactreal; dx, dy, xi, yi, n: integer;
for xi := 0 to dx by n do begin
    yt := [dy/dx]*xi;
    yi := trunc(yt + 1/2);
    display(xi,yi)
end
```

Computing y_t incrementally, and substituting $y_s = y_t + 1/2$, we have a variant of A3:

N3

```
var ys: exactreal; dx, dy, xi, yi, n: integer;
ys := 1/2;
for xi := 0 to dx by n do begin
    yi := trunc(ys);
    display(xi,yi);
    ys := ys + [n*(dy/dx)]
end
```

When y_s is broken into integer part y_{si} and fractional part y_{sf}, $n(dy/dx)$ may also have an integer and fractional part. Define the integer part s so that $0 \le n(dy/dx) - s \le 1$; the fractional part is then $n(dy/dx) - s$, which although called fractional, may actually equal 1. A value of s that meets these constraints is $s =$

$\lfloor n(dy/dx) \rfloor$. The algorithm becomes:

N4

```
var ysf: exactreal; dx, dy, xi, ysi, n, s: integer;
* assume s has been computed *
ysi := 0; ysf := 1/2;
for xi := 0 to dx by n do begin
    display(xi,ysi);
    if ysf + [n*(dy/dx) − s] ≥ 1 then begin
      ysi := ysi + [s + 1];
      ysf := ysf + [n*(dy/dx) − s − 1]
    end else begin
      ysi := ysi + s;
      ysf := ysf + [n*(dy/dx) − s]
    end
end
```

The next step is to apply a transformation that makes a Bresenham-like algorithm: $r = 2n\ dy + 2(y_{sf} - 1 - s)\ dx$.

N5

```
var ysf: exactreal; dx, dy, xi, ysi, n, s, t: integer;
                    * assume s and t = 2n dy − 2s dx have been computed *
r := t − dx;        * ysf = 1/2 implies r = 2ndy + 2(1/2 − 1 − s)dx *
ysi := 0;
for xi := 0 to dx by n do begin "N5loop"
    display(xi,ysi);
    if r ≥ 0 then begin
      ysi := ysi + [s + 1];
      r := r − [2*dx − t]
    end else begin
      ysi := ysi +s;
      r := r + t
    end
end "N5loop"
```

Note that this algorithm is identical to A5 if $n = 1$, $s = 0$. The attentive reader will question what happens if $dy = dx$, $n = 1$. Note that s is not *defined* to be $\lfloor n(dy/dx) \rfloor$. So by setting $s = 0$ in this case, the assumption $0 \le n(dy/dx) - s \le 1$ is not violated. The other possibility for $dy = dx$, namely $s = 1$, generates the same points, although the algorithm is then not identical to A5. It is important to remember that the n-step algorithm generates the same optimal points as the Bresenham algorithm.

A minor difficulty with N5 is the need to compute $s = \lfloor n(dy/dx) \rfloor$ and $t = 2n\ dy - 2s\ dx$. Although this could be done with multiply and divide operations, a small incremental algorithm can be used to compute s by interleaving the multiplication and division, developed using the same principles shown in A1 through A5:

```
var sf: exactreal; s, i, n: integer;
s := 0; sf := 0;
for i := 0 to n − 1 do begin
    * assert i*(dy/dx)= s + sf *
```

```
    if sf + [dy/dx] ≥ 1 then begin
      s := s + 1;
      sf := sf + [dy/dx − 1]
    end else sf := sf + [dy/dx]
end
```

This program is transformed by substituting $sp = (sf - 1)\ dx + dy$ and including obvious calculations for t in the following prologue for insertion in algorithm N5:

N5p

```
var dx, dy, s, t, sp, i, n: integer;
begin "N5prologue"
s := 0; t := 0;
sp := dy − dx;
for i := 0 to n − 1 do begin
                                 * assert i(dy/dx) = s + (sp + dx − dy)/dx *
    t := t + [dy + dy];          * to form 2ndy term *
    if sp ≥ 0 then begin
      s := s + 1;
      t := t − [dx + dx];        * to form −2sdx term *
      sp:= sp − [dx − dy]
    end else sp := sp + dy
end
end "N5prologue"
```

This prologue has the effect of one division and three multiplications, all of which are interleaved in a single loop. For further efficiency, the loop may be unwound; for small n, it may be unwound entirely.

5. MULTIPOINT ALGORITHMS

This section develops line-drawing algorithms that are capable of high speed by generating several points on a line at once. These algorithms are useful if a frame buffer display can write several pixels in one operation, or if lines must be approximated with special characters [10]. The transformations illustrated in the preceding sections are used extensively in deriving these algorithms. They allow the algorithm to be stated in conceptually simple terms and then transformed into one that can be efficiently implemented with integer arithmetic. To save space, the derivations in this section skip many of the steps illustrated in previous sections, but the techniques are the same.

A related class of algorithms generates multiple points on a line in order to form a compact encoding of the point sequences, using techniques akin to run coding [5, 6, 8, 12, 13]. The lengths of multipoint sequences selected by these algorithms are determined by the length of repeating patterns in the point sequences. The objective of multipoint algorithms given in this section is rather different, namely to generate points on the line in fixed length sequences; the length is determined by the number of pixels that can be written into the frame buffer at once. While variable length sequences obtained from a run-coding algorithm could be broken down into fixed length sequences that match the configuration of the frame buffer memory, more efficient algorithms are obtained by designing the algorithms from the outset to generate fixed length sequences.

5.1 The (n, n) Algorithm

The n-step algorithm developed in Section 4 is the basis for a parallel algorithm: operate n copies of the procedure, each generating points spaced n units apart; hence the name (n, n). Each copy of the algorithm is *phased* slightly differently: the copy with *phase* = 0 generates points at $x = 0, n, 2n, \ldots$; the copy with *phase* = 1 generates points at $x = 1, n + 1, 2n + 1, \ldots$; and so on. This technique (cf. A1) is simply expressed as:

P1

```
var phase: integer;
for phase := 0 to n − 1 do parbegin
    var xi, yi: integer; yt: exactreal;    * These variables are duplicated for each phase. *
    for xi := 0 + phase to dx by n do begin
      yt := [dy/dx]*xi;
      yi := trunc( yt + [1/2]);
      display(xi, yi)
    end
parend
```

The bracketing **parbegin** and **parend** mean that there are n parallel copies of the inner loop, each operating with a different value of *phase* and with separate copies of the local variables *xi*, *yi*, and *yt*. We now proceed with transformations demonstrated in Sections 3 and 4, obtaining first P3, a variant of N3:

P3

```
var phase: integer
for phase := 0 to n − 1 do parbegin
    var xi, yi: integer; ys: exactreal;    * These variables are duplicated for each
                                             phase.*
    ys := [dy/dx]*phase + 1/2;
    for xi := 0 + phase to dx by n do begin
      yi := trunc(yt);
      display(xi, yi);
      yt := yt + [n*(dy/dx)]
    end
parend
```

The inner loop is now transformed into one almost identical to the inner loop of N5; only the iteration of *xi* is different. The initial computation for *ys* requires a multiply/divide, which is transformed into a loop executed *phase* times to compute initial values for *ysi* and *r*. This loop is combined with the prologue (N5p) to compute values for *s* and *t*. The final result is P5:

P5

```
var dx, dy, s, t, n, sp, i, phase: integer;
begin "N5prologue"    * Prologue is identical to N5p, above. *
s := 0; t := 0;
sp := dy − dx;
for i := 0 to n − 1 do begin
    * assert i(dy/dx) = s + (sp + dx − dy)/dx *
    t := t + [dy + dy];
```

```
    if sp ≥ 0 then begin
      s := s + 1;
      t := t − [dx + dx];
      sp := sp − [dx − dy]
    end else sp := sp + dy
end
end "N5prologue"
for phase := 0 to n − 1 do parbegin
    var xi, ysi, r, i: integer;      * These variables are duplicated for each phase. *
    begin "P5init"      * Computes initial values for r and ysi *
      r := t − dx;
      ysi := 0;
      for i := 0 to phase − 1 do
        if r ≥ [t − 2* dy] then begin
          ysi := ysi + 1;
          r := r − [2* dx − 2* dy]
        end else r := r + [2* dy]
    end "P5init"
    for xi := 0 + phase to dx by n do begin "P5loop"
      * Identical to N5loop, above. *
      display(xi, ysi);
      if r ≥ 0 then begin
        ysi := ysi + [s + 1];
        r := r − [2* dx − t]
      end else begin
        ysi := ysi + s:
        r := r + t
      end
    end "P5loop"
parend
```

For greater efficiency, both the N5prologue and P5init loops may be profitably unwound. The P5init block computes initial values for r and ysi for each phase; for more efficiency, the P5init loop can be executed n times outside the *phase* loop and can pass to the P5loop values for r and ysi corresponding to each phase. Notice that the P5init loop bears a strong resemblance to the one-step Bresenham algorithm, A5; the difference arises because of the slightly different expressions for r.

5.2 The (1, n) Algorithm

The second algorithm capable of exploiting parallelism uses the n-step algorithm to find points on the line at n-unit intervals and fills points in between with a stroke. The n pixels in each stroke can be written in parallel. This technique is useful when lines must be approximated with characters because a raster display or printer is controlled by a character generator; the characters are simply short strokes.

The algorithm is easily derived from N5. In the inner loop, the test on r determines whether the line rises by $s + 1$ or s units for a move of n units in x. If the line rises by $s + 1$ units, a stroke that rises $s + 1$ units in n is drawn from the current (x, y) point. The stroke is determined by an index i that gives its rise in y, $i = 0, 1, \ldots, n$. The strokes may be precomputed using the Bresenham algorithm, as shown in Figure 3 for $n = 8$. Note that each stroke has only n points $(x = 0, 1, \ldots, n - 1)$, but that the rise associated with a stroke is that of the

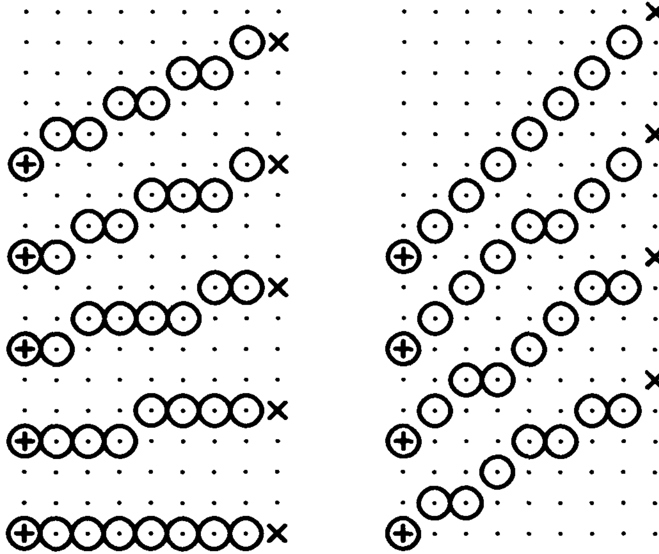

Fig. 3. The nine different strokes for $n = 8$. The left column shows rises of 0 (bottom), 1, 2, 3, and 4 (top). The right column shows rises of 5 (bottom), 6, 7, and 8 (top). The origin of a stroke is marked with a + and the origin of the next stroke with a ×.

(n + 1)th point ($x = n$). This convention is adopted because algorithm N5 computes the rise to the origin of the next stroke rather than the rise to the end of the current stroke.

In order to draw lines of arbitrary length, the last stroke on the line may be only a partial stroke. The standard stroke is simply truncated: only the first few points on it are actually displayed. This is illustrated by the procedure *DisplayStroke*, which accesses an array *Stroke*[i, x] to find the y coordinate of a pixel given the stroke rise i and the x coordinate relative to the beginning of the stroke.

```
procedure DisplayStroke(originX, originY, rise, maxX: integer);
    var x: integer;
  for x := 0 to maxX do parbegin
    display(originX + x, originY + Stroke[rise, x])
  parend;
```

Note that the individual pixels of the stroke are written in parallel. In the 8 × 8 display [14], the *DisplayStroke* function requires only two memory cycles (n = 8), one to read the stroke pattern and one to write it, possibly truncated, at an arbitrary position in the frame buffer.

This procedure can be incorporated into N5 to yield the complete line-drawing algorithm Q (the algorithm is shown without the prologue N5p):

Q

```
var dx, dy, xi, ysi, s, t, r: integer;
* Insert N5p here to compute s and t *
r := t − dx;
```

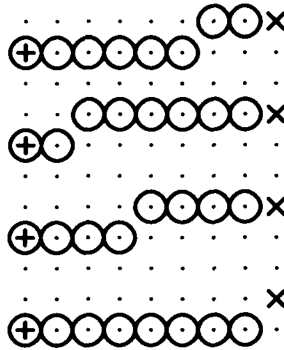

Fig. 4. Four of the eight different strokes with $n = 8$ and a rise of 1.

```
ysi := 0;
for xi := 0 to dx by n do begin
  if r ≥ 0 then begin
    DisplayStroke(xi, ysi, s + 1, min(n - 1, dx - xi))
    ysi := ysi + [s + 1];
    r := r - [2* dx - t]
  end else begin
    DisplayStroke(xi, ysi, s, min(n - 1, dx - xi))
    ysi := ysi + s;
    r := r + t
  end
end
```

While algorithm Q is substantially simpler than algorithm P5, it does not generate optimal lines. Although the stroke origins lie within 1/2 unit of the true line, the other points along the stroke may err by as much as 1 unit. This property arises because the y coordinate of a pixel is the sum of two independent computations, the position of the stroke origin and the position of the pixel within the stroke, each of which may make an error of 1/2. An example of a vertical error of 0.913 occurs at $x = 18$ in the line with $dx = 23$, $dy = 18$. Another way to see the nonoptimality of Q is to observe that although only a single stroke is displayed for each distinct rise in y, there are actually several different strokes with the same rise (Figure 4).

While it might be tempting to devise an algorithm that chooses the optimum stroke based on the value of r, this proves difficult in practice. The optimum stroke depends on the distance from the stroke origin to the true line, a distance related to the value of r. But the scaling of r that makes it convenient for the line-drawing calculation makes it inconvenient to index a table of strokes; r would have to be divided by the scale factor, a computation whose expense is not in keeping with our performance expectations. The topic is discussed in more detail by Gupta [9]

Even though algorithm Q produces nonoptimal lines, the endpoint of the line is always exact. The appendix contains a proof of this fact.

Before leaving the subject of stroke selection, we should mention that it is essential to have algorithm Q choose one of two strokes, rather than merely position the origin of a single stroke with a rise of s. If a single stroke is placed

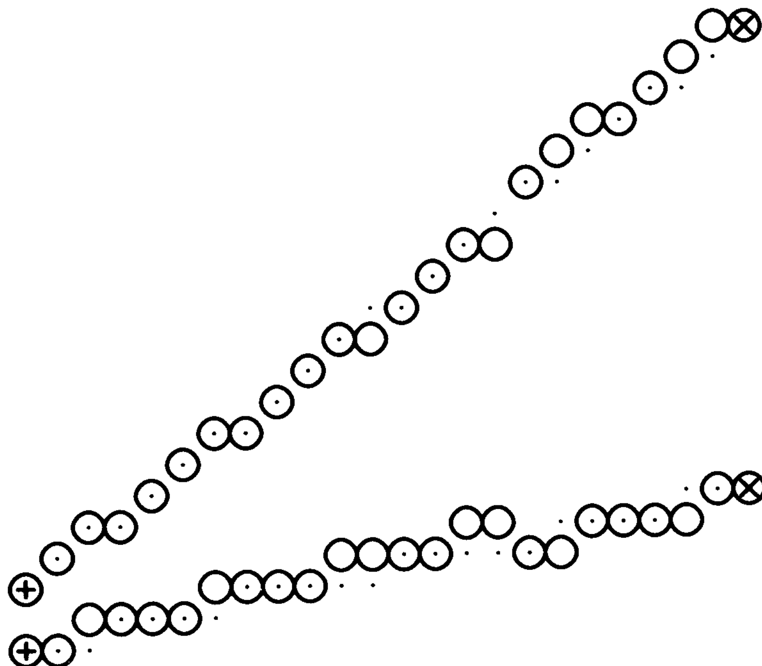

Fig. 5. Lines illustrating gaps and nonmonotonicities that occur when only one kind of stroke is used to draw a line (n =8). The top line (dx = 23, dy = 18) is drawn with 3 strokes with a rise of 6, which leave a gap. The small dots show the optimal line. The bottom line (dx = 23, dy = 5) shows a nonmonotonicity when 3 strokes with a rise of 2 are used.

repeatedly to display an entire line, the maximum deviation from the optimal line may be greater than 1 or the line may have gaps or nonmonotonicities, as illustrated in Figure 5.

6. CONCLUSION

The early sections of this paper show how simple mathematical and program transformations can be used to transform an obvious line-drawing method based on analytic geometry into an efficient and exact algorithm that requires only integer arithmetic. These methods help persuade us that the algorithm is correct without recourse to geometric constructions used by Bresenham [4]. The techniques are examples of routine program transformations that should be a commonplace activity in program design and implementation.

The main reason for applying these techniques is to extend line-drawing algorithms to write several points on a line at once. Although only two multipoint schemes are explored in Section 5, one can imagine many more. The difficulty of developing such algorithms is substantially reduced by using program transformations.

APPENDIX

We demonstrate that the (1, n) algorithm terminates at the proper endpoint (dx,dy). Assume $dx \geq dy \geq 0$, let $z = ax + by$ be the measure of distance from the

line, and further let $a = 2\ dy$, $b = -2\ dx$. Define y_i to be the y coordinate of the pixel displayed at $x = i$; z_i is the distance from this pixel to the true line.

The Bresenham algorithm that generates the origin for a stroke will guarantee that $|z_{ni}| \leq -b/2$. The points $x = ni + j$ for $j = 1$ to n are members of one of the two strokes that represent a line of slope s/n, where s is the vertical distance from the origin of the stroke to the origin of the next stroke (i.e., $s = y_{ni+n} - y_{ni}$). If these pixels are generated by a Bresenham algorithm aiming at a line of slope s/n, then we will have $|z_{ni+j} - (j/n)(z_{ni+n} - z_{ni})| \leq -b/2$, for $0 \leq j \leq n - 1$. We consider two cases: first, that the expression is positive, and second, that it is negative.

1. We have $z_{ni+j} \leq (j/n)(z_{ni+n} - z_{ni}) - b/2$. From the triangle inequality, we also have $|z_{ni+n} - z_{ni}| \leq -b$. However, the equality case never occurs—if it did, the slope of the line would be an integer multiple of $1/n$ and the z_{ni} would be zero for all i. So we now have $|z_{ni+n} - z_{ni}| < -b$. For $1 \leq j \leq n/2$, this yields $z_{ni+j} < -b$.

2. The negative case, by similar argument, gives $z_{ni+j} > b$ for $1 \leq j \leq n/2$.

Both cases together $|z_{ni+j}| < -b$ for $1 \leq j \leq n/2$. By a similar argument approaching from the other side (i.e., $x = ni + n, ni + n - 1, \ldots$), we obtain $|z_{ni-j}| < -b$ for $1 \leq j \leq n/2$. Both forward and backward approaches together give $|z_{ni+j}| < -b$ for $1 \leq j \leq n$.

When $x = dx$, at the endpoint of the line, we must have $|z_{dx}| < -b$, which, together with the fact that z must be a multiple of b, forces $z_{dx} = 0$. Therefore, the last point lies exactly on the line.

ACKNOWLEDGMENTS

This paper grew from attempts to write fast line-drawing microcode for the "8 × 8 display," designed by Ivan Sutherland and the author. Satish Gupta devoted considerable coding effort to this display and to simulations of the (1, n) method. The proof in the appendix is due to Mike Spreitzer. Mary Shaw and referees offered several helpful comments on the manuscript.

REFERENCES

1. AHO, A.V. AND ULLMAN, J.D. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., 1977.
2. AHO, A.V. AND ULLMAN, J.D. *The Theory of Parsing, Translation, and Compiling*. Vol. 2, *Compiling*. Prentice-Hall, Englewood Cliffs, N.J., 1973.
3. BENTLEY, J.L. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, N.J. 1982.
4. BRESENHAM, J.E. Algorithm for computer control of a digital plotter. *IBM Syst. J. 4*, 1 (Jan. 1965), 25–30.
5. BRESENHAM, J.E. Incremental line compaction. *Comput. J. 25*, 1 (Feb. 1982), 116–120.
6. CEDERBERG, R.L.T. A new method for vector generation. *Comput. Gr. Image Proc. 9*, 2 (Feb. 1979), 183–195.
7. DARLINGTON, J. AND BURSTALL, R.M. A system which automatically improves programs. *Acta Inf. 6*, 1 (1976), 41–60.
8. EARNSHAW, R.A. Line tracking for incremental plotters. *Comput. J. 23*, 1 (Feb. 1980), 46–52.
9. GUPTA, S. Architectures and algorithms for parallel updates of raster-scan displays. Tech. Rep. CMU-CS-82-111, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa., Dec. 1981.

10. JORDAN, B.W., JR., AND BARRETT, R.C. A cell organized raster display for line drawings. *Commun. ACM 17*, 2 (Feb. 1974), 70–77.
11. KNUTH, D.E. Structured programming with go to statements. *Comput. Surv. 6*, 4 (Dec. 1974), 261–301.
12. PITTEWAY, M.L.V. AND GREEN, A.J.R. Bresenham's algorithm with run line coding shortcut. *Comput J. 25*, 1 (Feb. 1982), 114–115.
13. REGGIORI, G.B. Digital Computer Transformations for Irregular Line Drawings," Dept. of Electrical Engineering, New York Univ., Bronx, N.Y., April 1972, pp. 46–61. Available as U.S. Dept. of Commerce AD-745-015.
14. SPROULL, R.F., SUTHERLAND, I.E., THOMPSON, A., GUPTA, S., AND MINTER, C. The 8 × 8 display. Tech. Rep. CMU-CS-82-105, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa., Dec. 1981.

Received April 1981; revised April 1982; accepted September 1982