CMU-CS-81-117

Using Program Transformations to Derive Line-Drawing Algorithms

Robert F. Sproull
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

April 1981

Abstract

A wide variety of line-drawing algorithms can be derived by applying program transformations to a simple, obviously correct algorithm. The transformations increase the algorithm's performance and eliminate the need for floating-point computations. Two familiar algorithms are derived in this way: Bresenham's algorithm and the digital differential analyzer (DDA). The transformations are then used to derive several highly parallel 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. Moreover, the transformations derive Bresenham's algorithm without recourse to complex geometric arguments.

Keywords: program transformation, line-drawing, computer graphics

CR Categories: 5.24, 5.25, 8.2

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 document are those of the authors 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.

1. Introduction

Many computer graphics devices use "line-drawing algorithms" to approximate the appearance of straight lines on devices that can only produce dots on a discrete grid. Incremental pen plotters that move a pen in small steps are common devices that require such a line-generation algorithm. Point-plotting CRT displays use the algorithms to approximate straight lines on interactive graphics displays. 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 the 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) generates 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 [1] dominates the DDA: it generates the optimal line, in a sense of optimal described below; 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, we need algorithms that can determine the location of several points on a line in parallel. None of the current line-drawing techniques is suitable, as they trace out the line sequentially, one point at a time. Parallel algorithms have several applications, chiefly in raster-scanned systems that can write more than one pixel at a time into the image. The "8x8 frame-buffer display" [2], which can in one memory cycle write a square region 8 pixels on a side located anywhere on the screen, motivated the investigation of parallel algorithms.

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.

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 closest 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 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.

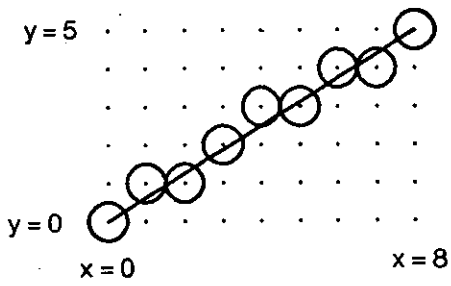

Figure 1. The line from (0, 0) to (8, 5). Small dots represent pixel centers. The solid line represents the "true" line. Circles show the pixels that are illuminated to display the optimal line.

The *optimal* line will illuminate exactly one pixel in each vertical column. This assumption minimizes variations in pixel spacing that make lines appear to vary in width or brightness. The assumption depends on the fact that the line's extent in x exceeds its extent in y.

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 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): $e_p = (dx/\sqrt{(dx^2+dy^2)})e_v$. 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: $y_i = round(y_t)$, or equivalently, $y_i = trunc(y_t + 1/2) = \lfloor y_t + 1/2 \rfloor$. (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 $-1/2 < e_v \leq 1/2$. Lines with this error behavior are said to be *optimal*, in the sense that each pixel illuminated is within one-half unit of the true line. Optimality thus requires that a single pixel be illuminated in each column and that the pixel be the one closest to the true line

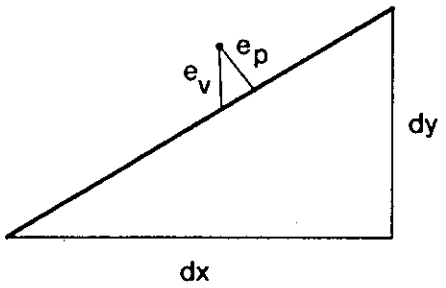

Figure 2. Illustration of the relationship between the vertical distance e_v and the perpendicular distance e_p.

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, 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 is 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.

1. *Incremental transformation.* 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.

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
```

2. *Substitution of variable (simple).* 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
```

3. *Substitution of variable (complex).* 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
```

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

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

The objectives of this transformation are (1) to change the comparison in the inner loop to a sign check (i.e., a comparison with 0), and (2) to eliminate division operations by scaling by $2dx$. 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

$$2dy-2dx \leq r < 2dy \tag{5}$$

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

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

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, because the same sequence of points is generated. When *display* is called, $e_v = y_{si}-y_t$. Using (1) to substitute for y_t, and then (2) to substitute for y_s, we have

$$e_v = y_{si}-(y_s-1/2) = y_{si}-(y_{si}+y_{sf}-1/2)$$
$$e_v = 1/2-y_{sf}$$

Applying transformation (4) yields

$e_y = -(r+dx-2dy)/(2dx)$ or
$r/(2dx) = -e_y+(dy/dx-1/2)$

The value r is thus linearly related to e_y, but is offset by 1/2 due to the loop's initial conditions, and is moreover scaled by $2dx$ 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.

4. The digital differential analyzer (DDA)

The digital differential analyzer numerically integrates the line equation, obtaining $x = \int dx$ and $y = \int dy$. The conventional DDA treats both coordinates symmetrically. The numerical integration requires choosing the number of integration steps, as shown in the following algorithm:

```
A6:   var x,y: exactreal; dx, dy, nsteps: integer;
      x := 0; y := 0;
      nsteps := some number ≥ dx and ≥ dy;
      for i := 0 to nsteps do begin
          display(trunc(x+[1/2]),trunc(y+[1/2]));
          x := x+[dx/nsteps];
          y := y+[dy/nsteps]
      end
```

This algorithm generates exact values for x and y in the loop because "exact" real arithmetic is used. This algorithm may not produce the optimal line, in the sense defined in Section 2, because of the separate rounding in x and y. A trivial example arises if *nsteps* = 1; only the pixels at the two endpoints of the line will be displayed. A more interesting example arises when $dx = 10$, $dy = 8$, and *nsteps* is chosen to be 40: the point (2, 1) is displayed, even though its vertical error is -0.6.

An important problem with the DDA is the choice of *nsteps*. A common approach is to choose *nsteps* to be a power of two so that the divisions may be performed simply by shifting dx and dy: thus, $nsteps = 2^n$, where $2^n \geq dx$. Unfortunately, this causes x to be incremented by a quantity less than unity, so that more than one pixel in a column may be illuminated. Although the second pixel in a column can be omitted by a suitable test in the loop, it is harder to guarantee that the pixel that is displayed is the one closest to the line.

Another approach is the "unit increment" DDA, in which we choose $nsteps = dx$ so that x will always be incremented by unity, and algorithm A6 becomes identical to A2, which generates the

optimal line. By this route, the DDA transforms into the Bresenham algorithm.

It is important to note that a common approximation to the unit increment DDA, often used in hardware implementations, does *not* generate optimal lines. The approximation is obtained from A4 by substituting y_{sd} for $(2^n)y_{sf}$ in order to introduce an integer variable y_{sd} that has "sufficient" precision to represent the fractional part of the y coordinate.

```
A7:   var dx, dy, xi, ysi, ysd: integer;
      ysi := 0; ysd := 2^(n-1);
      for xi := 0 to dx do begin
          display(xi,ysi);
          if ysd+[2^n*(dy/dx)-ε] ≥ 2^n then begin
              ysi := ysi+1;
              ysd := ysd+[2^n*(dy/dx)-ε-2^n]
          end else begin
              ysd := ysd+[2^n*(dy/dx)-ε]
          end
      end
```

The above algorithm is precise only if $\varepsilon=0$. In practice, $2^n(dy/dx)-\varepsilon$ is chosen to be integral, and ε is the error: $-1/2 \leq \varepsilon < 1/2$. The value of n is chosen to be sufficiently large that errors introduced by the approximation are acceptably small. If n is chosen to be the smallest integer such that $2^n \geq dx$, the line is guaranteed to begin and end at the proper coordinates, but not to be optimal. To illustrate the non-optimality, consider $dx = 120$, $dy = 1$, $n = 7$. At $xi = 62$, the algorithm will call *display*(62, 0). However, the point (62, 1) is closer to the true line. There is a value of n, $2^n >> dx$, for which the generated line will be optimal. However, this value may be quite high.

The hardware implementation of the unit-increment DDA is simpler than algorithm A7 implies (Figure 3). On each iteration, which corresponds to a clock cycle in the circuit, a new value for y_{sd} is computed by adding $2^n(dy/dx)-\varepsilon$, y_{si} will be incremented if the sum equals or exceeds 2^n, and x_i will always increment. The same idea can be used to build a fixed-point software implementation.

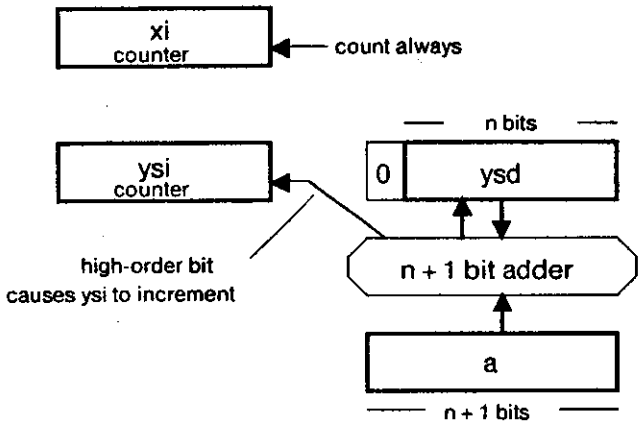

Figure 3. Hardware implementation of the unit-increment DDA. Counters compute x_i and y_{si}. An adder sums y_{sd} and $a = 2^n(dy/dx)-\epsilon$. On each cycle, y_{sd} is loaded with the new sum, x_i is incremented, and y_{si} is incremented if the high-order bit of the adder result is 1.

5. An n-step algorithm

Before exploring algorithms that exploit parallelism, we shall illustrate the transformation techniques developed in the previous sections 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 do 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:

```
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 $0 \leq n(dy/dx) - s \leq 1$; 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 the transformation that makes a "Bresenham-like" algorithm: $r = 2ndy + 2(y_{sf} - 1 - s)dx$.

```
N5:   var ysf: exactreal; dx, dy, xi, ysi, n, s, t: integer;
      (* assume s and t = 2ndy-2sdx 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 $0 \leq n(dy/dx) - s \leq 1$ is not violated. The other possibility for $dy = dx$, $s = 1$, generates the same points, although the algorithm is then not identical to A5.

A minor difficulty with N5 is the need to compute s and $t = 2ndy - 2sdx$. Although this could be done with multiply and divide operations, a small incremental algorithm can be used to compute s, developed using the same principles shown in A1-A5:

```
var rm: exactreal; s, i, n: integer;
s := 0; rm := 0;
for i := 0 to n-1 do begin
    if rm+[dy/dx] ≥ 1 then begin
        s := s+1;
        rm := rm+[dy/dx-1]
    end else rm := rm+[dy/dx]
end
```

This program is transformed by substituting $rp = (rm-1)dx+dy$ and including obvious calculations for t into the following prologue to algorithm N5:

```
N5p: var dx, dy, s, t, rp, i, n: integer;
     begin "N5prologue"
     s := 0; t := 0;
     rp := dy-dx;
     for i := 0 to n-1 do begin
         (* assert i(dy/dx) = s+(rp+dx-dy)/dx *)
         t := t+dy;
         if rp ≥ 0 then begin
             s := s+1;
             t := t-dx;
             rp := rp-[dx-dy]
         end else rp := rp+dy
     end;
     t := t+t;
     end "N5prologue"
```

It is important to remember that the n-step algorithm generates the same optimal points as the Bresenham algorithm.

6. Parallel algorithms

This section develops line-drawing algorithms that are capable of generating several points on a line in parallel. These algorithms are useful if a frame buffer can update several pixels in one cycle, or if lines must be approximated with special "characters" [3, 4]. The transformations illustrated in the preceding sections are extremely useful in designing 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.

6.1. The (n,n) algorithm

The n-step algorithm developed in Section 5 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, $2n$, . . .; the copy with $phase=1$ generates points at $x=1$, $n+1$, $2n+1$, . . .; and so on. This technique is simply expressed as (c.f. A1):

```
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 3-6. The inner loop is transformed into one almost identical to the inner loop of N5; only the iteration of *xi* is different. The initial computation for *ys* in P2 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 P2:

```
P2:   var dx, dy, s, t, n, rp, i, phase: integer;
      begin "N5prologue"  (* Prologue is identical to N5p, above *)
      s := 0; t := 0;
      rp := dy-dx;
      for i := 0 to n-1 do begin
          (* assert i(dy/dx) = s+(rp+dx-dy)/dx *)
          t := t+dy;
          if rp ≥ 0 then begin
              s := s+1;
              t := t-dx;
              rp := rp-[dx-dy]
          end else rp := rp+dy
      end;
      t := t+t;
      end "N5prologue"

      for phase := 0 to n-1 do parbegin
          var xi, ysi, r, i: integer; (* These variables are duplicated for each phase. *)
          r := t-dx;
          ysi := 0;
          begin "P2init"
              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 "P2init"

          for xi := 0+phase to dx by n do begin "P2loop"
              (* Exactly the same code as 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 "P2loop"
      parend
```

Notice that the loop P2init corresponding to the initial computations for *ys* bears a strong resemblance to the 1-step Bresenham algorithm, A5; the difference arises because of the slightly different expressions for *r*. There is no particular virtue to executing the loops in parallel—one loop can be used to compute initial values of *ysi* and *r* for all phases.

6.2. 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 4 for $n=8$. Note that each stroke has only n points ($x=0, 1, \ldots n-1$), but that the rise is that of the $(n+1)$st 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.

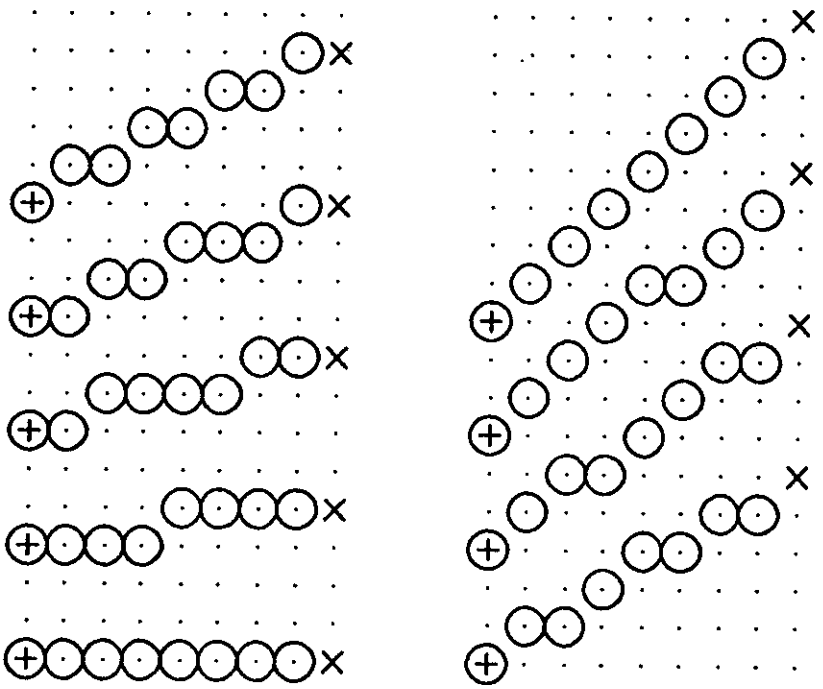

Figure 4. The nine different strokes for $n=8$. The left column shows rises of 0 (bottom), 1, 2, 3, and 4 (top). The right column shows rises of 5 (bottom), 6, 7, and 8 (top). The origin of a stroke is marked with a + and the origin of the next stroke with an x.

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.

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 = 2ndy-2sdx *)
      r := t-dx;
      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
```

Algorithm Q has several advantages over P. The setup is substantially simpler, as are the computations performed in parallel. The scheme is very similar to that of a character generator in which pre-computed patterns are displayed: the strokes play the role of characters. It differs from many character generators in that a character may have an arbitrary origin on the screen and may be partially truncated.

The chief disadvantage of algorithm Q is that 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 is shown in the top line of Figure 6, at $x = 18$. Another way to see the non-optimality 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 5). In practice, the error is hardly noticeable.

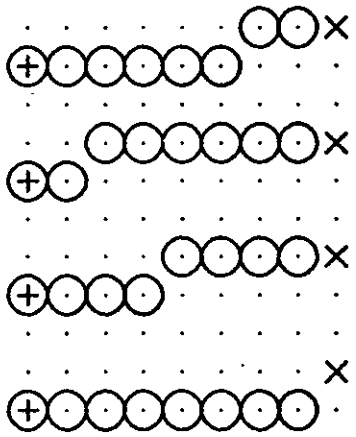

Figure 5. Four of the 8 different strokes with $n=8$ and a rise of 1.

Before leaving the subject of stroke selection, we should mention that it is essential to have the algorithm P2 choose from two strokes, rather than merely position the origin of a single stroke. If a single stroke is used for an entire line, the maximum deviation from the optimal line may be greater than 1 or the line may have gaps or non-monotonicities, illustrated in Figure 6.

Even though algorithm Q produces non-optimal lines, it turns out that the endpoint of the line is always exact. Appendix A contains a proof of this fact. Exhaustive simulations of algorithm Q for all lines of length 1024 or less have verified that endpoints are always computed correctly.

7. Conclusion

This paper began by showing how simple mathematical and program transformations could 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 complex geometric constructions such as those in [1]. The techniques are examples of routine program transformations that should be a commonplace activity in program design and implementation.

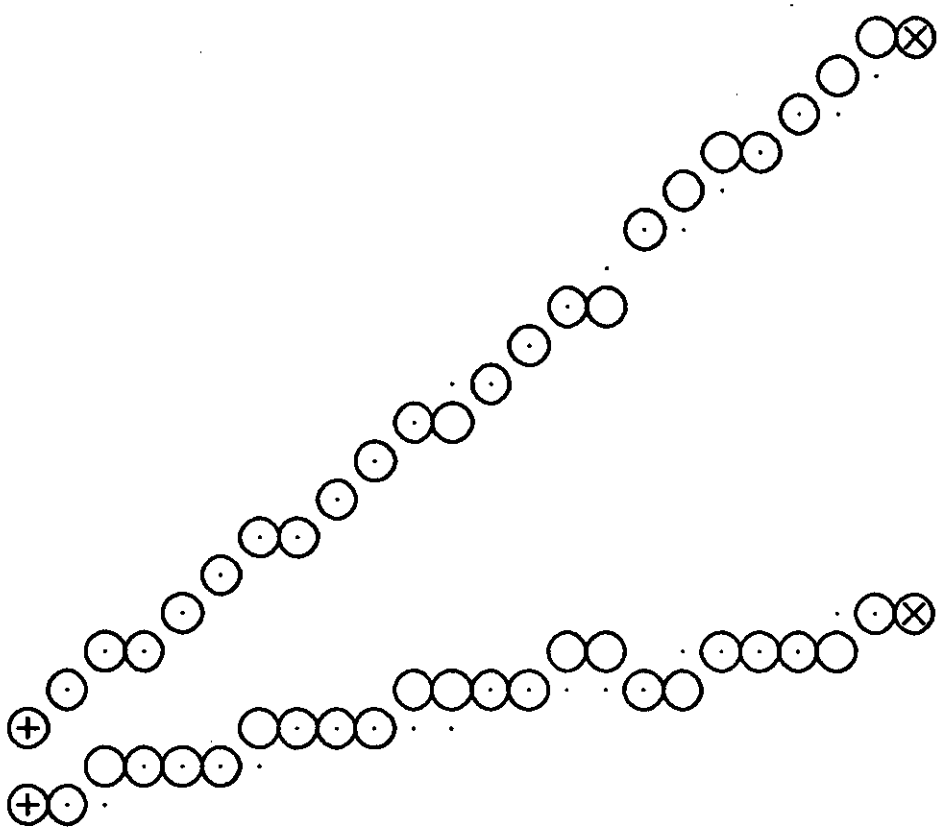

Figure 6. Lines illustrating gaps and non-monotonicities. The top line ($dx=23$, $dy=18$) is drawn with three strokes with $n=8$, which leave a gap. The small dots show the optimal line. The bottom line ($dx=23$, $dy=5$) shows a non-monotonicity.

The main reason for applying these techniques is to extend line-drawing algorithms to exploit parallel activities. Although only two parallel schemes are explored in Section 6, one can imagine many more. The difficulty of developing such algorithms is subtantially reduced by using the program transformations.

Acknowledgements

This paper grew from attempts to write very fast line-drawing microcode for the "8x8 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 Appendix A is due to Mike Spreitzer of Caltech.

References

[1] J.E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Syst. J.*, 4(1):25–30, 1965.

[2] R.F. Sproull, I.E. Sutherland, A. Thompson, S. Gupta, and C. Minter, "The 8x8 Display," in preparation.

[3] B.W. Jordan, Jr., and R.C. Barrett, "A Cell Organized Raster Display for Line Drawings," *CACM*, **17**(2):70, February 1974.

[4] C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, and D.R. Boggs, "Alto: A personal computer," in D.P. Siewiorek, A. Newell, and C.G. Bell, *Computer Structures: Principles and Examples*, second edition, McGraw-Hill, 1981.

Appendix A

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 $a = 2dy$, $b = -2dx$. 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 give $|z_{ni+j}| < -b$ 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 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.