Run-Based Multi-Point Line Drawing

Eun Jae Lee
Larry F. Hodges

Graphics, Visualization & Usability Lab
College of Computing
Georgia Institute of Technology
(404) 952-3062, gt0054b@prism.gatech.edu

Abstract

Line drawing on discrete graphics devices such as raster video displays, plotters, and image printers is one of the fundamental operations in computer graphics. Real-time interactive applications or high speed image output (such as on a Postscript laser printer) may require line drawing speeds in the millions of pixels per second. Such demands, which are ever increasing, push the efficiency of line generation.

For nearly thirty years Bresenham's algorithm has been the standard which subsequent efforts in line drawing have sought to surpass. This work can be broadly classified into three groups: parallel algorithms which divide line generation over multiple processing units; multi-point algorithms which output a fixed number of pixels in each iteration with fewer decision tests per pixel; and structural methods, including run-length algorithms, which identify periodic patterns in raster lines to reduce the number of decision tests or even to eliminate them.

This paper describes a hybrid method which uses structural properties of raster lines, such as runs, to improve the efficiency of multi-point line generation. A quadruple-step algorithm is developed which requires fewer decision tests than other multi-point algorithms, while retaining the multi-point's advantage in pixel output efficiency, particularly when implemented in hardware. A hardware state-machine circuit is described which efficiently implements the algorithm and outputs a four pixel segment every machine cycle.

1. Introduction

In 1965, Bresenham introduced what has become the standard measure for line drawing algorithms [3]. Beginning at one of the end points, Bresenham's algorithm generates the line by deciding from its current position which of the neighboring pixels is closest to the true line and then "moving" to that new position. By dividing line space into eight octants, only two of the eight adjacent pixels need to be examined for a given line. The algorithm decides between these two pixels by sign-testing a "decision variable" (also called a "discriminator") that indicates which of the pixels is closer to the line. After each test, the decision variable is updated according to a simple recurrence relation. This process of testing and updating a decision variable to find the points of a line is commonly called "incremental" line drawing. In addition, the algorithm is said to generate “best-fit” lines, meaning the discrete representations that are closest to the actual lines. Using only simple integer arithmetic (addition, subtraction, shifting) and sign testing, Bresenham’s algorithm is both simple and efficient.

Over the last three decades there has been a steady effort to improve the efficiency of line generation beyond Bresenham's algorithm. This work can be broadly classified into three groups: parallel, multi-point and structural.

Parallel methods seek to divide the work of generating a line over multiple independent processors. In [18], Sproull presents an algorithm in which the line is generated *n* points at a time, with each point generated concurrently by one of *n* processing units. An alternative algorithm by Wright [20] divides the line into *n* segments of roughly equal length that are then generated independently by *n* processors in a MIMD environment.

Multi-point methods reduce the overhead required to output a line by generating more than one point for every decision test. Sproull [18] examined determining every *n*th point on a line and then filling the points in between with a fixed stroke. This approach does not, however, generate best-fit lines. Wu and Rokne's double-step algorithm is similar but takes advantage of the special case where the stroke length is two. Their original algorithm [21] cut in half the number of decision tests , but its output required a gray-scale display. In [17], they combined the double-step with the symmetry principle of Gardner [13] into a new algorithm whose output matched Bresenham's, but required between two and four times fewer decision tests. Bao and Rokne introduced an algorthim in [1] that used a quadruple-step size to generate best-fit lines at a cost of one to three decision tests for every four pixels, with the average being slightly less than two.

Structural methods encompass not only research in efficient line generation but also general investigations into the properties of straight lines drawn on a discrete lattice with applications in compaction, pattern recognition and image processing [6, 7, 8, 10, 11, 14, 15, 16]. For the purpose of efficiently generating lines, the most interesting finding of this research is the existence of nested periodic patterns in the output of best-fit line generators.

Brons was the first to develop an algorithm that generates a line by first determining these periodic patterns and then using them to construct the line [6]. Cederberg presented a similar algorithm in [9] that generated a specific line between two points and produced output identical to Bresenham's. He also described a simple hardware circuit that, once the shape of the line is determined, can rapidly generate points using only simple counters and comparators. Another similar approach, based upon Euclid's algorithm, has been devised by Castle and Pitteway [7, 8].

In addition to purely structural methods, an alternative is to extend incremental algorithms to recognize these periodic structures to reduce the number of decision tests. Run-length algorithms [4, 5, 12, 15, 16] recognize that the points of discrete lines are grouped into runs whose lengths are confined to two consecutive integers. Rather than a single point, each decision test determines a run of points in every iteration, reducing by at least a factor of two the number of iterations required.

This paper develops a hybrid method that uses the structural properties of discrete lines, such as runs, to improve the efficiency of multi-point line generation. A quadruple-step algorithm is developed which requires fewer decision tests than other multi-point algorithms, while retaining the multi-point's advantage in pixel output efficiency, particularly when implemented in hardware. A hardware state-machine circuit is described which efficiently implements the algorithm and outputs four pixel positions in parallel, potentially quadrupling the line drawing speed at a lower cost over other parallel methods.

2. Run-Based Line Generation

Only lines starting from the origin in the first octant (i. e. slope between 0 and 1), are considered here since other lines can be transformed to meet these conditions through simple x-y translation and reflection about the x or y axis or the lines $y = x$ or $y = -x$.

2.1 Chain Codes

A simple notation for describing a line on a raster grid is Freeman's chain coding [11]. A chain code is a string whose symbols are digits between 0 and 7. Each symbol represents a one-pixel move in the direction shown in Figure 2.1.

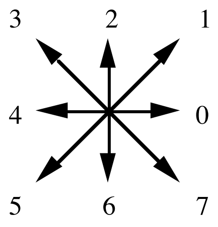


Figure 2.1. Chain code symbols and associated movements.

Freeman also identified three properties of the chain code of a line drawn on a lattice (such as lines drawn on a raster display):

1. At most two types of symbols can be present, and these can differ only by one, modulo eight.

2. One of the two symbols always occurs singly.

3. Successive occurrences of the single symbol are as uniformly spaced as possible among codes of the other value.

For lines in the first octant, only the digits 0 and 1 appear in the chain code. Some authors use the symbols *a* or *s* in place of 0 to represent an axial move and *d* in place of 1 to represent a diagonal move.

2.2 Runs

It has long been known that the chain codes of raster lines exhibit repetitive patterns. Consider a line from (0,0) to (a,b), where $a > b$. The line can be viewed as a series of "slices". The first and final slices are of height one-half. All slices in between are of height one. From the equation of the line, it is easy to determine that the length of these slices is $\frac{a}{2 \cdot b}$ for the end slices and $\frac{a}{b}$ for the rest.

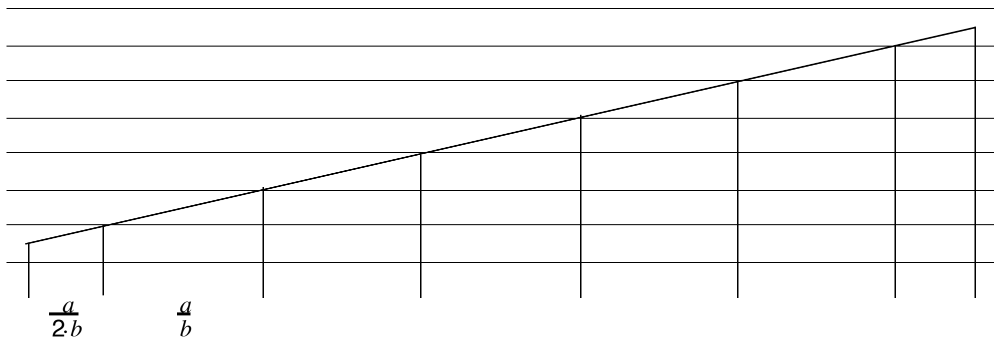


Figure 2.2 A line divided into "slices"

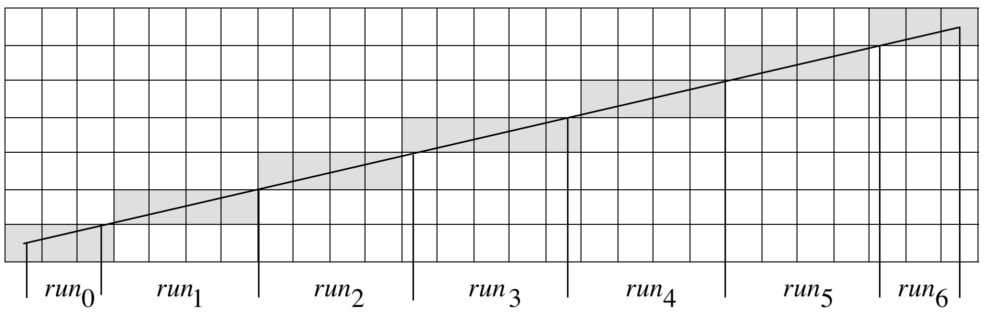
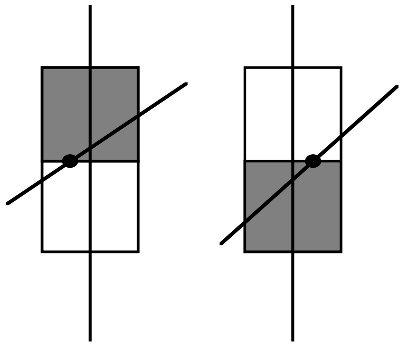
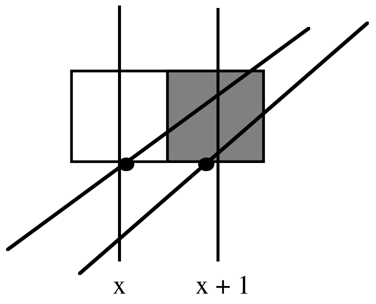


Figure 2.3 Mapping the slices onto an integer grid produces *runs*.

If the line is mapped to an integer pixel grid with each endpoint as the center of a pixel, the slices map to pixel spans, commonly called *runs*, whose lengths alternate between two consecutive integers, $\left\lfloor \frac{a}{b} \right\rfloor$ or $\left\lfloor \frac{a}{b} \right\rfloor + 1$. This is simple to prove:

1) The x-position at which the line crosses the boundary between two pixels determines which of the pixels is nearest to the line (see Figure 2.4a). Since it is assumed that a point is located in the center of a pixel, if the line crosses the lower boundary of the pixel to the left of the center, the upper pixel is closer to the line. If it crosses to the right, the lower pixel is closer.





a) The x-position at which the line crosses the boundary between two pixels determines which is closer to the real line.

b) The x coordinate of the point (x, y) at which the line crosses the pixel boundary is a rational number and when mapped to an integer grid will alway round up to the ceiling of x.

Figure 2.4 Closest pixel determination and transformation of rational coordinate onto integer grid.

2) The x position at which the line crosses a pixel boundary is a rational number. As illustrated in Figure 2.4b, when this rational coordinate is mapped to the integer grid, it will always map to $\lceil x \rceil$

3) From 1 and 2 it is clear that the length of a particular run is $\lceil x_{n+1} \rceil - \lceil x_n \rceil$, where x_{n+1} and x_n are the x coordinates where the line intersects the top and lower pixel boundaries.

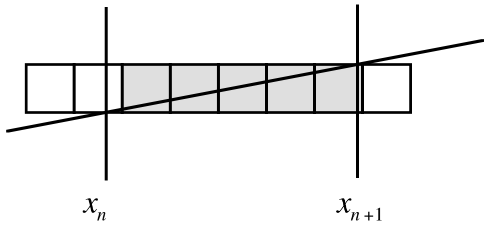


Figure 2.5 Mapping of run slices onto integer grid.

4) Thus,

$$\begin{aligned} run\_length_n &= \lceil x_{n+1} \rceil - \lceil x_n \rceil \\ &= \left\lceil x_n + \frac{a}{b} \right\rceil - \lceil x_n \rceil \\ &= \lfloor x_n \rfloor + \left\lfloor \frac{a}{b} \right\rfloor + \left\lceil \{x_n\} + \left\{\frac{a}{b}\right\} \right\rceil - \left(\lfloor x_n \rfloor + \lceil \{x_n\} \rceil \right) \\ &= \left\lfloor \frac{a}{b} \right\rfloor + \left\lceil \{x_n\} + \left\{\frac{a}{b}\right\} \right\rceil - \lceil \{x_n\} \rceil \end{aligned}$$

Here $\{x_n\}$ is the fractional part of x_n. Since x_n is a rational number with denominator *b*,

$$run\_length_n = \left\lfloor \frac{a}{b} \right\rfloor + \left\lceil \frac{e_n}{b} + \frac{r}{b} \right\rceil - \left\lceil \frac{e_n}{b} \right\rceil.$$

where $e_n = (x_n \cdot b)$ **mod** b, and $r = a$ **mod** b. And

$$\left\lceil \frac{e_n}{b} \right\rceil + 1 \geq \left\lceil \frac{e_n}{b} + \frac{r}{b} \right\rceil \geq \left\lceil \frac{e_n}{b} \right\rceil.$$

Therefore, if $e_n + r \geq b$, $run\_length_n = \left\lfloor \frac{a}{b} \right\rfloor + 1$, and the run is *long*. Otherwise, $run\_length_n = \left\lfloor \frac{a}{b} \right\rfloor$, and the run is *short*.. Using the same approach it is easy to prove that the lengths of the first and final runs are $\left\lfloor \frac{a}{2 \cdot b} \right\rfloor + 1$, unless a **mod** $(2 \cdot b) = 0$ in which case one of the end runs (whether the first or final depends on convention, usually the first) has length $\left\lfloor \frac{a}{2 \cdot b} \right\rfloor$.

For the discussion that follows, $\left\lfloor \frac{a}{b} \right\rfloor$ is called the *base run-length* and is denoted by *q*.

It is thus clear that the length of the *n*th run depends only on the value of e_n. Computing $e_n$directly for each run would be unacceptably expensive. Instead, using a recurrence relation e_n is calculated incrementally using simple integer addition:

$$e_0 = a \ \mathbf{mod}\ (2 \cdot b)$$
$$e_{n+1} = (e_n + r) \ \mathbf{mod}\ b$$

Section 2.3 discusses how the initial modulo can be avoided when calculating e_0. Calculating e_{n+1} is simply a matter of adding r to e_n, and if the result is greater than b, subtracting b.

The runs discussed above are called "horizontal" runs in this paper, and their chain codes are strings of 0's followed by a single 1. For lines with slopes between $\frac{1}{2}$ and 1, $q = 1$, meaning that they contain runs of length one. The chain codes of such lines contain as many or more 1's as 0's. In other words 0, rather than 1, is the element that appears singly in accordance with Freeman's second property (discussed in 2.1). Thus rather than runs of 0's, these are runs of 1's followed by a single 0. For this paper, these are called "diagonal" runs.

Deriving the run-length for diagonal runs is similar to the method for horizontal runs: the line is divided into a series of slices and mapping these slices onto an integer grid produces runs. But the slices are diagonal rather than horizontal, as illustrated in Figure 2.6.

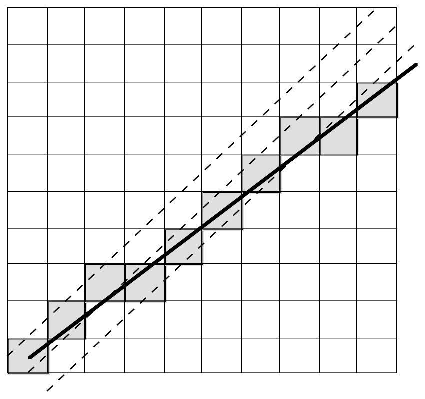


Figure 2.6 Slicing a line diagonally produces "diagonal" runs.

The equations for diagonal runs are the same as the ones for horizontal runs except that $a - b$ substitutes for b (see [5] for a more detailed discussion and proof). This means that a line with diagonal runs from (0, 0) to (a, b) can be generated in exactly the same way as a line with horizontal runs from (0, 0) to (a, a - b). The chain code of one can be transformed into the chain code of the other by simply replacing each 1 with a 0 and vice versa. Because of this equivalence property, any technique that utilizes horizontal runs can be generalized to diagonal runs as well.

2.3 Determining the Run Length

One problem with run-length algorithms is that they require a division for initialization. Thus, even though they typically require fewer decision tests than multi-point algorithms such as the ones in [1], [17] and [21], the overhead of the division is in many cases unacceptable. As an alternative, q_0, the length of the first run, can be calculated incrementally and then be used to calculate q and r with just one comparison, two shifts and one or two subtractions. This is demonstrated in the algorithm in Appendix A.

Fung, et. al. describe an equivalent approach in [12]. They also present an alternative algorithm which uses a binary search to find the run-length and the value of the decision variable. This second algorithm runs in O(log q) time and is more efficient for longer runs. However, it is not clear if this algorithm is significantly faster than a simple shift-subtract divide routine, and it is certainly slower than most hardware dividers.

For most lines the base run-length is not very long. Since run-length depends solely on the slope of a line, if all slopes are equally likely, over 80% of possible lines have base run-lengths of ten or less, as shown in Figure 2.7.

Thus, it seems that the most effective strategy for run-length determination would be one that optimizes for the most common case, in which the run-length is relatively short, without adversely affecting the speed of generation for lines where the intial run-length is long relative to the rest of the line -- when a is much larger than b. One strategy that fits well with the quadruple-step algorithm developed in the next section, is to determine the initial run-length and decision variable using a quadruple-step approach (see Appendix A).

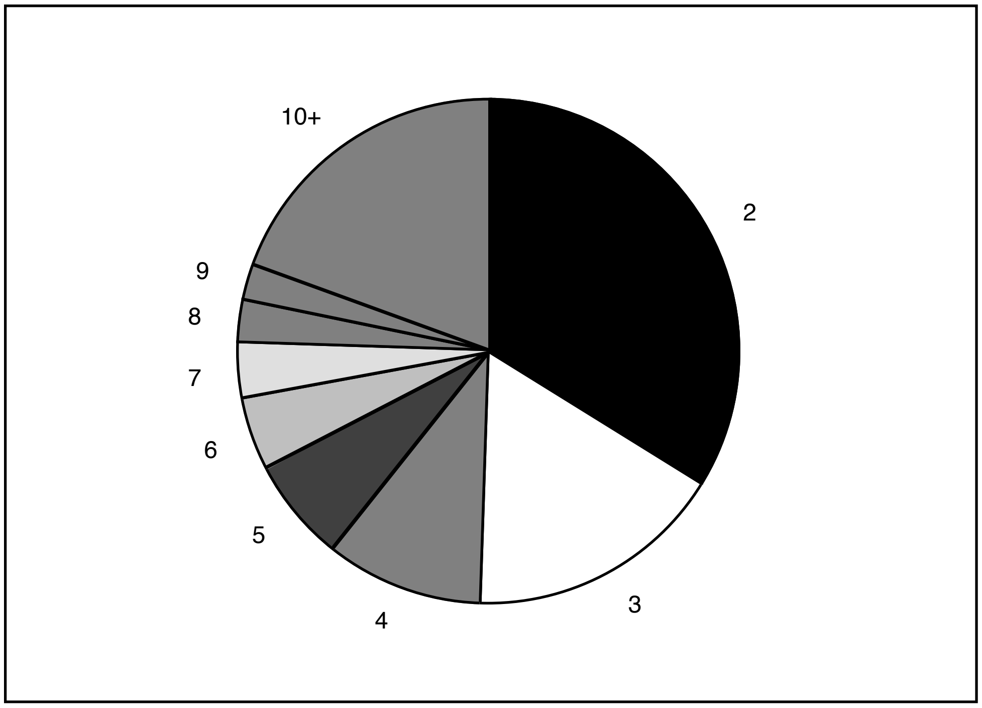


Figure 2.7 The distribution of possible base run-lengths. Note that most run-lengths are quite short -- over half are three or less.

2.4 Analysis of Run-Based Line Drawing

If properly implemented, a software run-based line algorithm can be quite efficient. Assuming that all possible lines in a 1000x1000 display are equally likely, the average run-length is between four and five, and as the display size increases, the average run-length slowly increases as well. Thus, the number of decision tests per pixel will be less than .25 on average. However, to reap the benefits of this low decision overhead, the algorithm must output runs in multi-pixel chunks.

Special case code should output short runs of length four or less directly rather than in a loop that outputs one pixel per iteration. For longer runs, a loop is necessary, but each iteration should output pixel blocks rather than single pixels.

A no-division run-based line drawing algorithm implementing the ideas discussed in this section has been developed by the authors and is found in Appendix A.

3. Multi-point Line Generation: A Run-Based Quadruple-Step Algorithm

As mentioned in the introduction, a multi-point line algorithm generates a fixed length *segment* of consecutive pixels with every iteration. As Sproull points out in [18], methods which generate a fixed number of pixels per iteration have a significant advantage in efficiency over variable length methods (such as run-length algorithms) when more than one pixel can be output per memory access. Specialized frame buffer memories can reap great performance benefits using this method, since memory access speed can be a limiting factor in hardware line drawing. One such design is the 8x8 Display described in [19]. Even in a software implementation, because the output of a multi-point algorithm comes in segments of fixed length and shape, the software can pre-calculate the address offset between pixels and take advantage of pointer offset addressing modes found on most modern processors so that no address incrementing is required between the points of a segment.

In this section an algorithm is developed which outputs a four pixel segment at a cost of slightly more than one decision test per segment on the average. Algorithms which use other segment lengths (*step sizes*) are easily derived using the same methodology, but a segment length of four seems to strike the best balance of speed versus size.

3.1 Valid Segment Derivation

A segment of length *n* has an *n* element chain code, and the elements of a line's chain code have only two values; they are binary. Thus, it is clear that there are 2^n possible segments of length *n*. Thus, for a step size of 4, there are 2^4 or sixteen possible segments. This is illustrated in Figure 3.1. For convenience, references to a segment will use its chain code as a label (e.g., segment 0000).

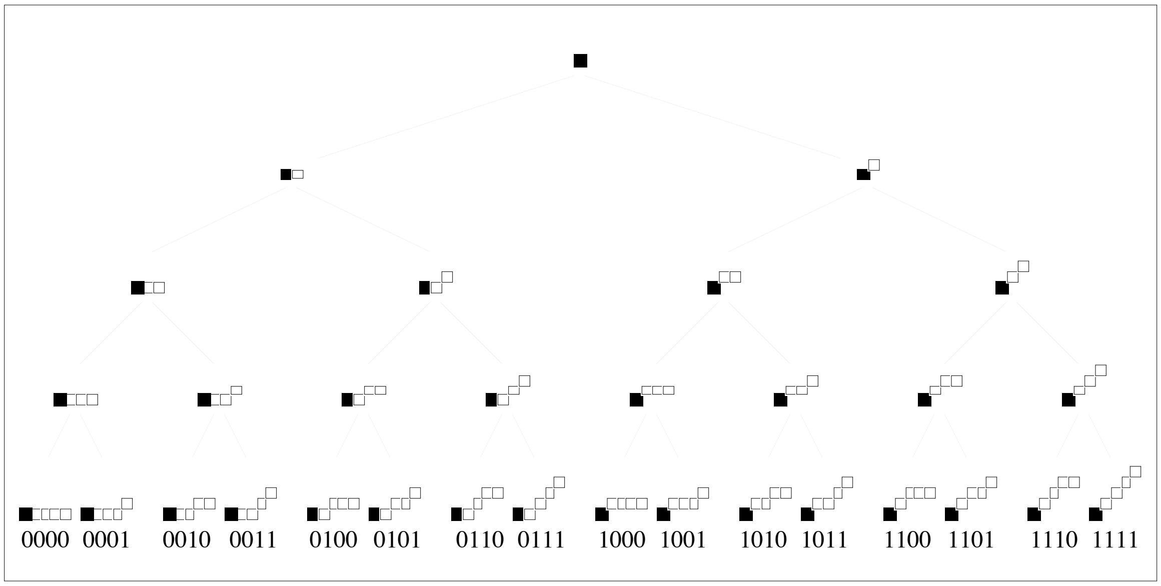


Figure 3.1. Segment derivation for step length 4. The black square represents the last pixel of the previous segment. Beneath each segment is its chain code.

Having to decide between sixteen segments on every iteration would require four decision tests. This would obviously be no improvement over Bresenham. However, the number of possible segments, and thereby the number of decision tests, can be reduced by applying two of the properties of lines discussed in Section 2 as constraints on segment choice:

1. A line's chain code consists of at most two symbols. One of these symbols, if there are two, always appears singly.

2. The runs of any particular line are limited to one of two consecutive integers.

Constraint 1 eliminates two segments, 0011 and 1100, and divides the remaining fourteen segments into three groups, as shown in Figure 3.2. The six segments marked H in the figure have consecutive 0's in their chain code; these segments can only appear in lines with horizontal runs. Similarly, the six segments marked D contain consecutive 1's in their chain codes; these segments can only appear in lines with diagonal runs. Finally, two segments are marked X, indicating that they may appear in either type of line. So for each of the two line types there are eight possible segments.

0000	0001	0010	0100	0101	0110	0111	1000	1001	1010	1011	1101	1110	1111
H	H	H	H	X	D	D	H	H	X	D	D	D	D

Figure 3.2. Segments divided by line type.

In the first octant, horizontal runs occur in lines of slope less than $\frac{1}{2}$ and diagonal runs in lines of slope greater than $\frac{1}{2}$. Lines with slope exactly equal to one half can be viewed either way. The following discussion assumes horizontal run lines, but can be applied to diagonal run lines as well since they can be derived using the transformation of Section 2.2.

Dividing lines into three cases based on base run-length further reduces the number of possible segments to consider.

Case I. Lines with a base run-length of two

Case II. Lines with a base run-length of three

Case III. Lines with a base run-length of four or above.

As Figure 3.3 illustrates, applying constraint 2 reduces the number of possible segments for each of the three cases to five. So any particular line will contain at most five of fourteen possible segment types.

Our algorithm implements three state machines, one for each of the three cases. The state machines for the first two cases each have five states, and each state has two to four transitions. Case III is handled differently from the other two and is explained separately in Section 3.4.

0000	0001	0010	0100	0101	1000	1001	1010
2	2	▨	▨	▨	2	▨	▨

Case I. q = 2

0000	0001	0010	0100	0101	1000	1001	1010
2	▨	▨	▨	2	▨	▨	2

Case II. q = 3

0000	0001	0010	0100	0101	1000	1001	1010
▨	▨	▨	▨	2	▨	2	2

Case III. q >= 4

Figure 3.3. Valid segments for each of the three cases. A shaded box indicates that the segment is valid. A number represents the constraint that is violated by that segment.

3.2 Segment Transitions For Cases I and II

The constraints restrict not only which segments can appear in a given line but also the transitions from segment to segment.

Next State

Current State

	0010	0100	0101	1001	1010
0010	2	▨	▨	▨	▨
0100	2	2	2	▨	▨
0101	▨	▨	▨	1	1
1001	▨	▨	▨	1	1
1010	2	▨	▨	▨	▨

Case I. q = 2

Next State

Current State

	0001	0010	0100	1000	1001
0001	▨	▨	2	1	1
0010	2	▨	▨	2	2
0100	2	2	▨	▨	▨
1000	2	2	2	▨	▨
1001	▨	▨	2	1	1

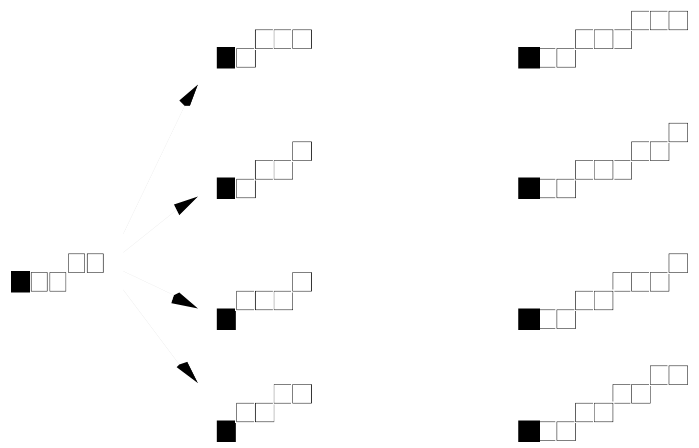
Case II. q = 3

Figure 3.4 Valid transitions for Cases I and II. A shaded box indicates that the segment is valid. A number represents the constraint that is violated by that segment.

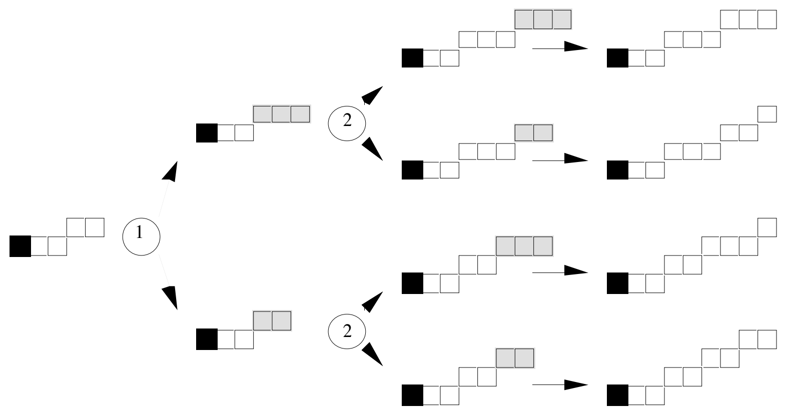
Figure 3.4 shows all possible transitions for Cases I and II. Two segments have four possible transitions, three segments have three, and four segments have two.

3.3 Next Segment Determination For Cases I and II

The algorithm selects the next segment from the set of possible transitions for the current segment (Figure 3.4) with one or two decision tests. Each transition is uniquely identified by the length of one or two of the runs in the current-next segment pair. Thus, one or two tests to determine the length of the identifying run or runs will decide the next segment. This process is illustrated in Figure 3.5.



a) Segment 0010 and its four transitions b) The four possible segment pairs



c) The decision process for segment 0010.

Figure 3.5 Next segment determination.

The tables in Figure 3.6 give the identifying run or runs for each transition in Cases I and II, and indicate their positions in the chain code of the transition segment pairs. Notice that the chain code position after the decision tests (the number after the two-letter code for each transition) is the same for all transitions in the same column, and that this number matches the "In" value (the position in the chain code where the decision tests begin) of the segment in that column. This means that, no matter which transition occurs, the decision tests for the next state will always begin at the correct place, guaranteeing correctness.

Next State

Current State

	In	0010	0100	0101	1001	1010
0010	2		LL/4	LS/3	SL/3	SS/2
0100	4				L/3	S/2
0101	3	L/2	LS/4	LS/3		
1001	3	L/2	LS/4	LS/3		
1010	2		LL/4	LS/3	SL/3	SS/2

Case I. q = 2

Two-letter codes indicate two tests yeilding four possibilities:

LL	- two succesive long runs
LS	- long run followed by short
SL	- short run followed by long
SS	- two succesive long runs

The numbers after the slash indicate the chain code position after the decision tests.

Next State

Current State

	In	0001	0010	0100	1000	1001
0001	3	L/4	S/2			
0010	2		L/2	S/1		
0100	1			L/1	SL/4	SS/3
1000	4				L/4	S/3
1001	3	L/4	S/2			

Case II. q = 3

Figure 3.6 Transition tests for Cases I and II. The column marked "In" indicates the position (from 0 to 3) in the segment's chain code where the first decision test begins. Values of 4 mean that testing begins at position 0 of the next segment. An empty box indicates that the segment is not valid.

Every run is tested exactly one time, and the decision variable and testing are exactly the same as in the algorithm of Section 2.5.

3.4 Case III Lines

There are two ways of handling Case III. The first is a software-only method; the second is necessary when implementing a table-based state machine that outputs exactly four pixels at a time (as in the hardware circuit of Section 5).

Next State

Current State

	0000	0001	0010	0100	1000
0000	▨	▨	▨	▨	▨
0001	▨	▨	2	2	1
0010	▨	▨	▨	2	2
0100	▨	▨	▨	▨	2
1000	▨	▨	▨	▨	▨

Case III. q >= 4

Figure 3.7 Valid transitions for Case III. A shaded box indicates that the segment is valid. A number represents the constraint that is violated by that segment. Note the large number of possible transitions.

3.4.1 Software Handling of Case III

In a software implementation which does not require that every iteration produces exactly four pixels, the most efficient method for rendering Case III lines is to simply plot the runs directly. For example, a line with base run-length of five has runs of length five or six. For each of these runs, the algorithm would output a segment of length five or six with chain codes 10000 or 100000, respectively.

Obviously, the algorithm cannot handle every run-length as a special case. The segment 10000000 represents a run of length 8. This segment can be seen as two segments, 1000 followed by 0000. Similarly the segment 100000000000 can be seen as a1000 segment followed by two 0000 segments.

Any run with length $i \geq 4$ can be represented by (1000 + 10000 + 100000 + 1000000) $(0000)^n$, where the number of 0000 segments is $n = (i$ **div** $4) - 1$. Thus, for run-lengths greater than or equal to four, the software version of the algorithm outputs one of the four base segments (selected by i **mod** 4) followed by n 0000 segments.

3.4.2 Four-Step Transitions for Case III

For certain implementations (such as the hardware circuit of Section 5), it is necessary for every segment to be exactly four pixels. The method of 3.4.1 cannot be used in such circumstances because of the variable length of the base segment, but the method can be adapted utilizing a state machine similar to the ones for Case I and II lines.

As in 3.4.1, Case III lines are further subdivided into four groups. A line belongs to Group *g* where *g* = *q* **mod** 4. All the lines in the same group share the same set of possible transitions (shown in Figure 3.8) and differ only in the number of 0000 segments which is*n* = (*q* **div** 4) - 1 or *n* + 1.

	0001	0010	0100	1000
0001	▨			■
0010	▨	▨		
0100		▨	▨	
1000			▨	▨

Group 0

	0001	0010	0100	1000
0001			■	■
0010	▨			■
0100	▨	▨		
1000		▨	▨	

Group 1

	0001	0010	0100	1000
0001		■	■	
0010			■	■
0100	▨			■
1000	▨	▨		

Group 2

	0001	0010	0100	1000
0001	■	■		
0010		■	■	
0100			■	■
1000	▨			■

Group 3

Figure 3.8 Possible transitions for each of the groups in Case III. Shaded boxes indicate valid transitions with *n* 0000 segments. Black boxes indicate valid transitions with *n* + 1 0000 segments.

Unlike in Cases I and II, where each segment required one or two decisions, each decision in Case III determines multiple segments. From the current segment, the state machine outputs *n* or *n* + 1 0000 segments then uses the decision to decide the next segment. Each state has two possible transitions for the next segment as shown in the transition tables in Figure 3.8.

3.5 State Machine Implementation

Putting everything together results in the state machine tables of Figure 3.9. These tables can be used to produce implementations in either software or hardware. In software, each state must be coded directly for efficiency. A hardware state machine circuit is presented in Section 5.

Every state has two, three or four transitions. The Boolean variables d_0 and d_1 determine the transition for a given state. For $q = 2$ and $q = 3$, d_0 and d_1 represent the results of testing the lengths of the current run-pair that determine the next state. These tests are as described in Section 2. A zero means that the run is short, a one that the run is long. An “x”indicates a “don't care” condition -- it does not matter what the value of that Boolean variable is. For $q \geq 4$, d_0 indicates whether the current run is long or short, and d_1 is the signal from a counter, which counts $n = a \textbf{ div } b$ iterations. While the counter is counting, d_1 is zero and the state machine outputs a 0000 segment. When the counter reaches the end of its count, d_1 becomes one and the transition is made to the next state using d_0.

The adv signal is not required for a software implementation, since each state would code its decision tests directly, and it can make whatever number of decision tests are needed. In hardware, adv is a signal to the decision circuit that tells whether one or two new decisions are needed by the next state. When adv = 0, d_1 becomes the previous d_0, and d_0 is a new decision. When adv = 1, both d_0 and d_1 are new decisions.

INPUT			OUTPUT			
current state	d_1	d_0	adv	next state	offset	chain code
0	0	0	1	4	0011	0010
0	0	1	1	3	0011	0010
0	1	0	1	2	0011	0010
0	1	1	0	1	0011	0010
1	x	0	1	4	0111	0100
1	x	1	1	3	0111	0100
2	0	0	1	2	0112	0101
2	0	1	0	1	0112	0101
2	1	x	0	0	0112	0101
3	0	0	1	2	1112	1001
3	0	1	0	1	1112	1001
3	1	x	0	0	1112	1001
4	0	0	1	4	1122	1010
4	0	1	1	3	1122	1010
4	1	0	1	2	1122	1010
4	1	1	0	1	1122	1010

a) State machine for q = 2

INPUT			OUTPUT			
current state	d_1	d_0	adv	next state	offset	chain code
0	x	0	0	1	0001	0001
0	x	1	0	0	0001	0001
1	x	0	1	2	0011	0010
1	x	1	0	1	0011	0010
2	0	0	0	4	0111	0100
2	0	1	0	3	0111	0100
2	1	x	0	2	0111	0100
3	x	0	0	4	1111	1000
3	x	1	0	3	1111	1000
4	x	0	0	1	1112	1001
4	x	1	0	0	1112	1001

b) State machine for q = 3

INPUT				OUTPUT			
group	current state	d_1	d_0	adv	next state	offset	chain code
0	0	0	x	0	0	1111	1000
	0	1	0	0	4	1111	1000
	0	1	1	0	3	1111	1000
	1	0	x	0	1	0000	0000
	1	1	0	0	1	0001	0001
	1	1	1	1	4	0001	0001
	2	0	x	0	2	0000	0000
	2	1	0	0	2	0011	0010
	2	1	1	0	1	0011	0010
	3	0	x	0	3	0000	0000
	3	1	0	0	3	0111	0100
	3	1	1	0	2	0111	0100
	4	0	x	0	4	0000	0000
	4	1	0	0	4	1111	1000
	4	1	1	0	3	1111	1000
1	0	0	x	0	0	1111	1000
	0	1	0	0	3	1111	1000
	0	1	1	0	2	1111	1000
	1	0	x	0	1	0000	0000
	1	1	0	1	4	0001	0001
	1	1	1	1	3	0001	0001
	2	0	x	0	2	0000	0000
	2	1	0	0	1	0011	0010
	2	1	1	1	4	0011	0010
	3	0	x	0	3	0000	0000
	3	1	0	0	2	0111	0100
	3	1	1	0	1	0111	0100
	4	0	x	0	4	0000	0000
	4	1	0	0	3	1111	1000
	4	1	0	0	2	1111	1000
2	0	0	x	0	0	1111	1000
	0	1	0	0	2	1111	1000
	0	1	1	0	1	1111	1000
	1	0	x	0	1	0000	0000
	1	1	0	1	3	0001	0001
	1	1	1	1	2	0001	0001
	2	0	x	0	2	0000	0000
	2	1	0	1	4	0011	0010
	2	1	1	1	3	0011	0010
	3	0	x	0	3	0000	0000
	3	1	0	0	1	0111	0100
	3	1	1	1	4	0111	0100
	4	0	x	0	4	0000	0000
	4	1	0	0	2	1111	1000
	4	1	0	0	1	1111	1000
3	0	0	x	0	0	1111	1000
	0	1	0	0	1	1111	1000
	0	1	1	1	4	1111	1000
	1	0	x	0	1	0000	0000
	1	1	0	1	2	0001	0001
	1	1	1	1	1	0001	0001
	2	0	x	0	2	0000	0000
	2	1	0	1	3	0011	0010
	2	1	1	1	2	0011	0010
	3	0	x	0	3	0000	0000
	3	1	0	1	4	0111	0100
	3	1	1	1	3	0111	0100
	4	0	x	0	4	0000	0000
	4	1	0	0	1	1111	1000
	4	1	0	1	4	1111	1000

c) State machine for q >= 4

Figure 3.9 State Machines for a) $q = 2$, b) $q = 3$, and c) $q \geq 4$.

In software, each state would directly code the instructions needed to output a segment to the frame buffer. For a hardware state machine, there are two options for output: chain code or offsets -- where each pixel of the segment is represented by a number indicating its displacement relative to the start position of the segment. For example, the offset code for the segment 0101 is 0112. Regardless of which method is used, the output code is passed to a circuit which transforms this code into one or more memory writes to the frame buffer. The nature of this circuit will vary with the nature of the frame buffer that it addresses.

3.6 Initialization and Start State Determination

Prior to rendering, the algorithm calculates q, q_0 and e_0, either directly or incrementally as described in 2.3. It then determines the start state based upon q_0 as shown in Figure 3.10. Case I lines require may require one decision test to obtain the start state. Case II lines always start in state 2. For Case III lines, the initial state and the intial value of the counter are calculated directly using q_0.

q0	d1	d0	initial state
1	x	0	2
1	x	1	1
2	x	x	0

a) Initial state table for q = 2

q0	d1	d0	initial state
2	x	x	1

b) Initial state table for q = 3

initial state = 4 - (q0 **mod** 4)
initial num = q0 **div** 4

c) Initial values for state and num for Case III lines

Figure 3.10 State machine initialization values.

It is important to note here an ambiguity that arises from the use of chain codes to represent segments. Since chain codes represent discrete movements rather than actual pixel positions, the number of elements in the chain code will always be one less than the number of pixels drawn, because a chain code representation assumes an implicit starting point. Typically, single-step algorithms such as Bresenham's draw a pixel at this implicit starting point prior to making any movements. For a quadruple-step algorithm which is optimized to output pixels four at a time, plotting a single start point before rendering the rest of the line is not optimal. The alternative used here is to set the start point one pixel to the left -- a simple sleight of hand that allows use of chain code representation for segments.

3.7 Termination

Termination occurs when fewer than four pixels of the line remain. The remaining zero to three pixels are drawn by truncating the next segment or through special case code.

4. Analysis

For the majority of segment transitions, the run-based quadruple-step algorithm requires only one decision test. In the worst case it requires two. In fact, the algorithm requires exactly the same number of decision tests as the run-length algorithm of Section 2. Restricting attention to lines with slopes between 0 and $\frac{1}{2}$ (with angles of degree 0 to 26.6) and assuming that the distribution of lines is even through this range, the median slope is approximately .236, and thus, the median run length is between 4 and 5. This seems to indicate that the average number of decision tests per segment will be slightly less than one. Experimental measurements support this. However, these measurements do not take into consideration that for lines with $q \geq 4$ each run generally spans more than one segment, requiring a loop and thus more overhead. If the cost of each loop iteration is assumed equal to the cost of a state transition, the average number of decision tests measured in experiments rises to slightly above one.

This is still better than the quadruple-step algorithm of Bao and Rokne[1]. Also, the decision tests remain consistent from line to line and segment to segment, unlike in [1] where different segments require different decision tests. The algorithm presented in this paper requires the computation of the base run-length and the scan-conversion of the initial run, but as discussed in 2.3, for most lines this initial overhead is not too great. Moreover, Bao and Rokne's method carries considerable initial overhead as well. For longer lines, the overhead decreases in significance, and the run-based quadruple-step algorithm is more efficient. And as discussed below, it is possible to improve upon the algorithm presented here so that substantially fewer decisions are made on average for most segments at the cost of increased complexity.

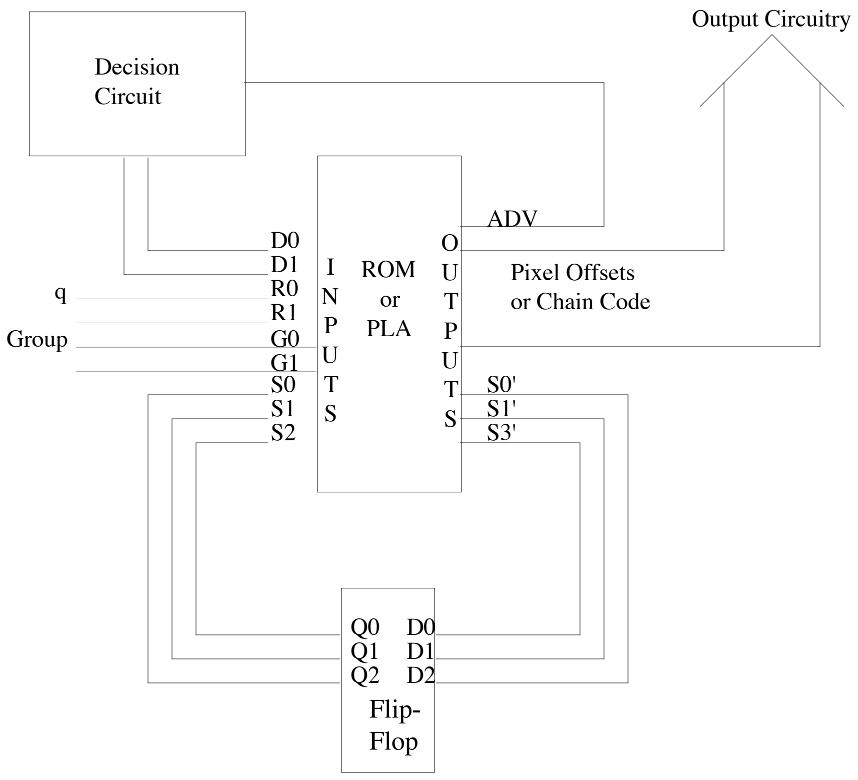


Figure 5.1 Hardware state machine circuit that outputs a four pixel segment every machine cycle. The Decision Circuit calculates the two decision tests, D0 and D1 in parallel.

5. Hardware Implementation

Another advantage of the run-based quadruple-step algorithm is that it can be implemented as a very efficient hardware circuit (shown in Figure 5.1). By performing the decision tests in parallel, a circuit can be built which will output four pixels every machine cycle. The hardware cost to implement this circuit should be lower than for other parallel methods, since a separate processing unit is not necessary for each pixel. The PLA or ROM is quite small, and the complexity lies in the decision circuit, which is roughly twice as complex as a circuit implementing a Bresenham-type algorithm.

The Decision Circuit determines the values of D0 and D1. The ROM or PLA implements the state machines, using the information from the tables in Figure 3.9. R0 and R1 contain a two-bit binary value representing *q*: R1R0 = 00 for $q = 2$, 01 for $q = 3$ and 10 for $q \geq 4$; 11 is unused.

Similarly, for lines with $q \geq 4$, G0 and G1 contain the two-bit binary value of the line's group number. S0 - S2 hold the current state. The state machine outputs the next state onto S0' - S2'.

6. Conclusion

This paper has developed an efficient algorithm for generating lines four pixels at a time, which reduces overhead over other multi-point method by exploiting structural properties of discrete lines to reduce decision overhead. The method used in this paper can be applied to step sizes other than four. Two- and three-step algorithms have successfully been developed. Step sizes greater than four are more problematic, since the number of states and transitions grow quickly as the step size increases. But if size is not a concern, extending the step size does in general lead to increased efficiency for sufficiently long lines.

Another way to improve efficiency is to use more constraints. For instance, in a given line one of its two run-lengths will appear singly [5, 6, 9]. If applied as a constraint, this property reduces the number of transitions for Case I so that the maximum number of transitions for any state is three rather than four. Alternatively, the state machine can be made "deeper", with more states, so that constraints can be applied across more than one transition. Both of these techniques improve efficiency but increase size.

A potential drawback to the algorithm is its relatively large size, particularly when implemented in software. But the size of the code seems large only in comparison to more compact line algorithms. An assembly language implementation takes up just a few kilobytes. Although there are circumstances where this may be an unreasonable size, the rapid decline in hardware prices makes such circumstances increasingly rare, especially when it is considered that no matter how quickly the speed of hardware advances, the demand for ever faster systems never abates. In such a light, the benefits of a small tradeoff in size for increased speed seems worthwhile.

Appendix A: Run-Length Line Algorithm

The following is a no-division run-length line drawing algorithm developed using the concepts discussed in Section 2.

```
Draw a line from (x1, y1) to (x2, y2) in the first octant.

All variables are integers.

   a = x2 - x1;
   b = y2 - y1;
   x = x1;
   y = y1;

   /* determine if diagonal or horizontal runs */
   if(a < 2*b) {
      /* diagonal */
      b = a - b;              /* perform half-octant transformation
                              /* for diagonal runs */
      d = 1;
   }
   else {
      /* horizontal */
      d = 0;
   }

   /* Scan convert initial run -- the quadruple step version below can be */
   /*                             substituted here */
   e = a;
   diff = 2*b;
   base = 2*a;
   overflow = base - diff;

   if(d == 1)                 /* to handle equality case for */
      overflow += 1;          /* diagonal runs */

   if(d == 0)
      while( x <= x2) {
         if(e >= overflow) {
            Point(x,y);
            e = e + diff - base;
            break;
         }
         else {
            Point(x,y);
            e = e + diff;
         }
      }
   else
      for(x = x1; x <= x2; x++) {
         Point(x,y);
         if(e >= overflow) {
            e = e + diff - base;
            break;
         }
         else {
            e = e + diff;
            y++;
         }
      }

   q0 = x - x1;

   /* calculate run-length */
   if(e >= diff/2) {
      q = 2*(q0 - 1);
      r = 2*e - diff;
   }
   else {
      q = 2*q0 - 1;
      r = 2*e;
   }

   base = diff;

   if(r == 0) {
      q++;
   }
   else
      diff = r;

   overflow = base - diff;

   if(d == 1)                  /* to handle equality case for */
      overflow += 1;           /* diagonal runs */

   /* do run-based scan conversion */
   while(y < y2) {
      if(e >= overflow) {
         /* short run */
         e = e + diff - base;
         Run(x,y,q,d);         /* draw a run of length q */
      }
      else {
         /* long run */
         e = e + diff;
         Run(x,y,q+1,d);       /* draw a run of length q+1 */
      }
   }

   Run(x,y,x2-x+1,d);          /* draw final run */
```

A more efficient way to determine the length of the initial run and the initial value of the decision variable is with a quadruple step approach:

```
e = a
overflow = 2*a
diff = 2*b
for(x = x1; x <= x2; x = x + 4) {
   if(e + 4*diff >= overflow) {
      if(e + 2*diff >= overflow) {
         if(e + diff >= overflow) {
            e = e + diff - base
         }
         else {
            e = e + 2*diff - base
            x = x + 1
         }
      }
      else {
         if(e + 3*diff >= overflow) {
            e = e + 3*diff - base
            x = x + 2
         }
         else {
            e = e + 4*diff - base
            x = x + 3
         }
      }
      break;
   }
   else {
      e = 4*diff;
   }
}
```

References

1. Bao, P. G. and Rokne, J. G. Quadruple-step line generation. *Comput. & Graphics 4* (1989), 461-469.

2. Boothroyd, J., and Hamilton, P. A. Exactly reversible plotter paths. *Australian Comput. J. 2* (1970), 20-21.

3. Bresenham, J. E. Algorithm for computer control of digital plotter. *IBM Syst. J. 4* (1965), 25-30.

4. Bresenham, J. E. Incremental line compaction. *Comput. J. 25* (Jan. 1982), 116-120.

5. Bresenham, J. E. Run length slice algorithms for incremental lines. In *Fundamental Algorithms for Computer Graphics,* R. A. Earnshaw, Ed. NATO ASI Series, Springer Verlag, New York, 1985, 59-104.

6. Brons, R. Linguistic methods for the description of a straight line on a grid. *Comput. Gr. Image Process. 9* (1979), 183-195.

7. Castle, C. M. A., and Pitteway, M. L. V. An application of Euclid's algorithm to drawing straight lines. In *Fundamental Algorithms for Computer Graphics*, R. A. Earnshaw, Ed. NATO ASI Series, Springer Verlag, New York, 1985, 135-139.

8. Castle, C. M. A., and Pitteway, M. L. V. An efficient structural technique for encoding 'best-fit' straight lines. *Comput. J. 30* (1987), 168-175.

9. Cederberg, R. L. T. A new method for vector generation. *Comput. Gr. Image Proc. 9*, 2(Feb. 1979), 183-195.

10. Earnshaw, R. A. Line tracking for incremental plotters. *Comput. J. 23* (1980), 46-52.

11. Freeman, H. Boundary encoding and processing . In *Picture Processing and Pyshopictorics.* B. S. Lipkin and A. Rosenfeld, Eds., Academic Press, New York 1970, 214-266.

12. Fung, K. Y., Nicholl, T. M. and Dewdney, A. K. A run-length slice line drawing algorithm without division operations. *Comput. Gr. Forum 3* (1992), 267-277.

13. Gardner, P. L. Modifications of Bresenham's algorithm for displays. *IBM Tech. Disclosure* Bull. 18 (1975), 1595-1586.

14. McIlroy, M. D. A note on discrete representation of lines. *AT&T Tech. J. 64* (Feb. 1985), 481-490.

15. Pitteway, M. L. V., and Green, A. J. R. Bresenham's algorithm with run line coding shortcut. *Comput. J. 25* (1982), 114-115.

16. Regiori, G. B. Digital computer transformations of irregular line drawings. Tech. Rep. 403-22, Department of Electrical Engineering and Computer Science. New York Univ., Apr. 1972.

17. Rokne, J. G., Wyvill, B. and Wu, X. Fast line scan-conversion. *ACM Trans Gr. 4* (1990), 376-388.

18. Sproull, R. F. Using program transformations to derive line-drawing algorithms. *ACM Trans. Gr. 1* (Oct. 1982), 259-273.

19. Sproull, R. F., Sutherland, I.E., Thompson, A., Gupta, S., and Minter C. The 8 by 8 display. *ACM Trans Gr. 1* (1983), 32-56.

20. Wright , W. E. Parallelization of Bresenham's Line and Circle Algorithms. *IEEE Comput. Gr. Appl. 10* (Sept. 1990), 60-67.

21. Wu, X. and Rokne, J. G. Double-step incremental generation of lines and circles. *Comput. Vision, Gr. Image Process. 37* (Mar. 1987), 331-344.