Double- and Triple-Step Incremental Generation of Lines

Phil Graham and S. Sitharama Iyengar

Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803

Abstract: A method of increasing the efficiency of line drawing algorithms by setting additional pixels during loop iterations is presented in this paper. This method adds no additional costs to the loop. It is applied here to the double-step algorithm presented in [15] and later used in [14], resulting in up to a thirty-three percent reduction in the number of iterations and a sixteen percent increase in speed. In addition, the code complexity and initialization costs of the resulting algorithm remain the same.

Keywords and Phrases: pixel, line drawing algorithms, integer logic, rasterization.

1. Introduction

Line drawing algorithms are effectively used to determine the points which lie closest to a line segment along its major axis, given the segment's starting and ending points. As one may suspect, much attention has been devoted to developing efficient line drawing algorithms in computer graphics over the years. In addition to drawing lines and modeling geometric shapes, such as squares, line drawing algorithms are used to approximate other shapes such as circles. They are also used in ray tracing [10] and polygon-filling algorithms, as well as other applications involving lines. Truly, line drawing algorithms are a fundamental topic in the area of computer graphics.

Various algorithms have been presented which eliminate or greatly reduce the need to perform multiplications and divisions. For instance, the traditional Bresenham algorithm [2] performs integer logic on the error term with respect to one coordinate as values of the other coordinate are incremented. The algorithm in [15] performs integer logic to generate line points by instead taking a double step increment with respect to one of the coordinate axes. Bresenham's run length slice algorithm [3, 4] calculates a slice of movements with respect to a particular coordinate axis in each iteration of a loop. Other approaches to speeding up the scan-conversion process are based on Euclid's algorithm [6, 7]. In addition, the properties of discrete line segments have been studied and incorporated into algorithms, thereby making them more efficient. There are also line drawing algorithms which are derivations or variations of these algorithms [5].

In this paper, we introduce a method of increasing the speed of line drawing algorithms by setting additional pixels in the loop iterations. This method is applied to the double-step algorithm proposed in [15] and later used in [14]. While the double-step algorithm always sets two pixels per loop iteration, our algorithm sets either two or three pixels per loop iteration. Also, no additional logic is needed in the loop, and the amount of the remaining logic can be considered the same as that for the double-step algorithm. Previous attempts to increase the step sizes offered no clear advantages due to the large increases in the complexity of the algorithm [1]. It should be noted that while this method can result in rather substantial increases in speed of the scan-conversion process, the improvement is not realized in practice due to the inevitable pixel write operations which dominate timewise. Nevertheless, the speed of the entire process can be improved somewhat by making each stage as fast as possible. Furthermore, as noted in [14], the bottleneck may well be in scan-conversion in future hardware. Before presenting our modified algorithm, however, we will first discuss the double-step line algorithm in greater detail.

2. The Double-Step Algorithm

The double-step algorithm for drawing curves on the raster plane is summarized as follows. Let $f(x,y) = 0$ be a two-dimensional curve having a continuous first derivative. Assume it is divided into segments which satisfy one of the following cases:

$$(a) \quad 0 \le \frac{dy}{dx} \le 1 \qquad (b) \quad 1 < \frac{dy}{dx} < \infty$$
$$(c) \quad -\infty < \frac{dy}{dx} < -1 \qquad (d) \quad -1 \le \frac{dy}{dx} \le 0$$

As in [14] and [15], the discussions in this paper are restricted to case (*a*) since the other cases can be reduced to case (*a*) by swapping x and y and/or changing the incremental direction. In addition, only lines having integer coordinate values for their starting and ending points are considered. These points are denoted as (x_0, y_0) and (x_n, y_n), respectively. The pixels set at the remaining grid locations are denoted as (x_i, y_i), $0 < i < n$. When setting the pixels of a line on a raster grid (which will have starting point (x_0, y_0) at the lower left corner) where the step size in the x direction is two, only four patterns are possible (Figure 1). It was conjectured by Freeman [8, 9] and proven by Regiori [13] that only two pattern types may occur simultaneously: either patterns 1 and 2 (3) or patterns 2 (3) and 4 (where 2 (3) is an abbreviation for pattern 2 or pattern 3). The possibility of the occurrence of a set of patterns depends on whether $0 \leq dy/dx < 1/2$ or $1/2 \leq dy/dx \leq 1$.

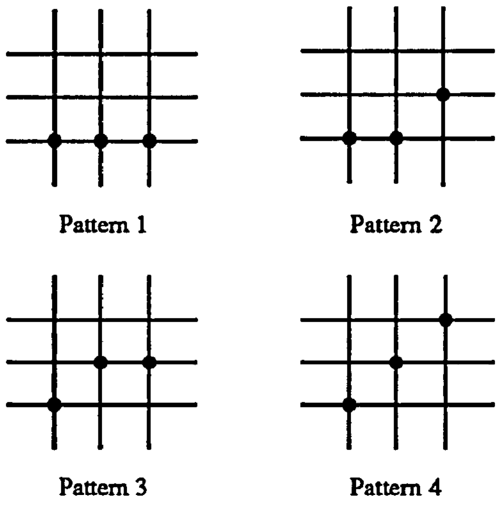

Figure 1. The four types of double-step patterns.

In the double-step algorithm, lines are classified according to their slopes, as just described. A discriminator, D_i, is then updated each iteration of a loop in which double step sizes in the x direction are taken. This discriminator determines that the patterns which should be set are as shown below:

If $dy/dx < 1/2$ then the algorithm is given by

Algorithm 1. Let $D_1 = 4dy - dx$, $\alpha_1 = 4dy$, and $\alpha_2 = 4dy - 2dx$. For $i = 1, 2, \ldots$

$$D_{i+1} = \begin{cases} D_i + \alpha_1 & \text{if } D_i < 0 \text{ (pattern 1)} \\ D_i + \alpha_2 & \text{otherwise (pattern 2 (3))} \end{cases}$$

and if $dy/dx \geq 1/2$ then the algorithm is

Algorithm 2. Let $D_1 = 4dy - 3dx$, $\beta_1 = 4dy - 2dx$, and $\beta_2 = 4(dy - dx)$. For $i = 1, 2, \ldots$

$$D_{i+1} = \begin{cases} D_i + \beta_1 & \text{if } D_i < 0 \text{ (pattern 2 (3))} \\ D_i + \beta_2 & \text{otherwise (pattern 4)} \end{cases}$$

Since the middle pixels which should be set when pattern 1 (4) occurs is known, the only work that remains is to distinguish between patterns 2 and 3. This is performed by the following test which results in the pixel corresponding to pattern 2 being set if the test is passed and pattern 3 if it is not passed:

$$D_i < \begin{cases} 2dy & \text{if } 0 \leq dy/dx < 1/2 \\ 2(dy - dx) & \text{if } 1/2 \leq dy/dx \leq 1 \end{cases}$$

The detailed double-step line algorithm is given in Figure 2. It is shown that the stopping conditions are determined by decrementing the x coordinate value of the endpoint if dx is odd. Otherwise, an additional pixel would be set because two pixels are set in each iteration of the **while** loops. In [14], it is stated that the Pascal implementation is approximately twice as fast as Bresenham's original algorithm if pixel I/O is ignored.

```
procedure LINE(a1, b1, a2, b2: integer);
var dx, dy, incr1, incr2, D, x, y, xend, c: integer;

   procedure draw(pattern: integer);
   begin
      case pattern of
         1: ++x; point(x, y); ++x; point(x, y);
         2: ++x; point(x, y); ++x; ++y; point(x, y);
         3: ++x; ++y; point(x, y); ++x; point(x, y);
         4: ++x; ++y; point(x, y); ++x; ++y; point(x, y);
      end {case}
   end {draw}

begin
   dx = a2 - a1; dy = b2 - b1;
   x = a1; y = b1;
   if dx is even then begin parity = 0;: xend = a2; end
   else begin parity = 1; xend = a2 - 1; end;
   point(x, y);
   incr2 = 4*dy - 2*dx;
   if incr2 < 0 then begin          {slope is less than 1/2}
      c = 2*dy;
      incr1 = 2*c;
      D = incr1 - dx;
      while x <> xend do
         if D < 0 then begin
            draw(1); D = D + incr1; end
         else begin
            if D < c then draw(2) else draw(3);
            D = D + incr2;
            end;
      end
   else begin              {slope is ≥ 1/2}
      c = 2*(dy - dx);
      incr1 = 2*c;
      D = incr1 + dx;
      while x <> xend do
```

```
            if D >= 0 then begin
                draw(4); D = D + incr1; end
            else begin
                if D < c then draw(2) else draw(3);
                D = D + incr2;
                end;
        end; {if}
    {plot the endpoint if dx is odd}
    if parity = 1 then point(a2, b2);
end {LINE}
```

Figure 2. The double-step line algorithm.

3. The Double- and Triple-Step Algorithm

It is noted in [15] that when patterns 2 and 3 are not distinguished from each other, the repetitive loop in the double-step algorithm is the same as that in Bresenham's algorithm, with the exception that two pixels are set per iteration. As a result, the double-step algorithm loops only $\lfloor dx/2 \rfloor$ times whereas the Bresenham algorithm loops dx times, and it can be roughly twice as fast. Therefore, the additions, tests, and jumps in the loop that are saved by performing fewer iterations can greatly offset the few additional initializing operations. Although the double-step algorithm is still faster, some of these benefits are reduced when patterns 2 and 3 are distinguished from each other as a result of the additional comparison(s) and jump(s) (see the "if D < c **then** ..." portions of the code in Figure 2). In this section, we present a double- and triple-step algorithm which remedies the problem of the additional computations associated with distinguishing patterns 2 and 3 by setting an additional pixel under the worst case conditions. Setting the additional pixel does not add any logic to the loop, and the remaining logic is the same as that for the double-step algorithm.

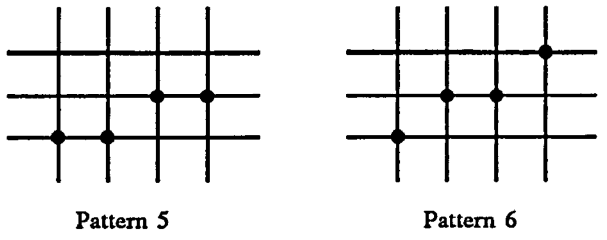

Figure 3. The two additional pixel patterns of the double- and triple-step algorithm.

3.1. Double and Triple Steps

We begin our discussion of the double- and triple-step algorithm by describing the relationship of pixels at x_i and x_{i+1}. If the pixels set at x_i and x_{i+1} have the same y coordinate value, then a *sideways movement* occurs from x_i. If the pixels set at x_i and x_{i+1} do not have the same y coordinate value, then a *diagonal movement* occurs from x_i. We now make several observations concerning lines below (Once again, only lines where $0 \leq dy/dx \leq 1$ are considered. However, additional observations for the remaining lines in the 2-D plane can easily be made):

Observation 1. If $0 \leq dy/dx < 1/2$, then it is not possible to make two successive diagonal movements.

Observation 2. If $1/2 < dy/dx \leq 1$, then it is not possible to make two successive sideways movements.

These observations are easily proven by considering the slope of any line in question and are omitted for brevity. Previously, the middle pixels in patterns 1 and 4 could be determined without any additional computation once the y coordinate value of the pixel set at x_{i+2} was determined. From Observation 1, it follows that the pixel at x_{i+3} can also be determined without any additional computation when $dy/dx < 1/2$ and pattern 2 occurs. In addition, from Observation 2, it follows that the pixel at x_{i+3} can be determined without any additional computation when $1/2 < dy/dx \leq 1$ and pattern 3 occurs. Each of these pixel patterns will be called patterns 5 and 6, respectively (Figure 3). Since a third pixel can be set without any additional computation in certain circumstances, the double- and triple-step algorithm will set three pixels whenever possible and will set two pixels in the remaining cases.

3.2. Updating the Discriminator

In the double-step algorithm, the two sets of pixel patterns possible of occurring together are patterns 1 and 2 (3) as well as patterns 2 (3) and 4. Since the same movements are made in patterns 2 and 3 (although in different order), the discriminators for each pattern are treated the same. However, in the double- and triple-step algorithm, the sets of patterns possible of occurring together are patterns 1 and 5 (3) as well as patterns 2 (6) and 4. Because the movements associated with patterns 5 and 3 (as well as 2 and 6) are different, it must now be determined how to update the discriminator when pattern 5 (or 6) occur.

Earlier, it was shown that the discriminator can be defined with respect to steps of size two. As noted in [2], the discriminator can also be defined with respect to steps of size one:

$$D'_{i+1} = \begin{cases} D'_i + 2dy & \text{if } D'_i < 0 \; (y_i = y_{i-1}) \\ D'_i + 2(dy - dx) & \text{otherwise } (y_i = y_{i-1} + 1) \end{cases}$$

It turns out that the discriminator for the double-step algorithm is defined and updated in a manner very similar to that for the discriminator above. Therefore, letting $D''_1 = D_1$, the new value of the discriminator in the double- and triple-step algorithm is defined as:

$$D''_{i+1} = D''_i + (4dy - 2dx) + 2dy$$

when pattern 5 occurs. When pattern 6 occurs, the new value of the discriminator is defined as:

$$D''_{i+1} = D''_i + (4dy - 2dx) + 2(dy - dx).$$

Of course, the discriminator values are updated as before when patterns 1, 2, 3, and 4 occur. Therefore, the steps taken by the double- and triple-step algorithm are summarized as follows:

If $0 \leq dy/dx \leq 1/2$ then the algorithm is given by

Algorithm 3. Let $D''_1 = 4dy - dx$, $\alpha_1 = 4dy$, $\alpha_2 = 4dy - 2dx$, and $\alpha_3 = 2dy$. For $i = 1, 2, \ldots$

$$D''_{i+1} = \begin{cases} D''_i + \alpha_1 & \text{if } D''_i < 0 \text{ (pattern 1)} \\ D''_i + \alpha_2 + \alpha_3 & \text{if } 0 \leq D''_i < \alpha_3 \text{ (pattern 5)} \\ D''_i + \alpha_2 & \text{if } \alpha_3 \leq D''_i \text{ (pattern 3)} \end{cases}$$

and if $1/2 < dy/dx \leq 1$ then the algorithm is

Algorithm 4. Let $D''_1 = 4dy - 3dx$, $\beta_1 = 4dy - 2dx$, $\beta_2 = 4(dy - dx)$, and $\beta_3 = 2(dy - dx)$. For $i = 1, 2, \ldots$

$$D''_{i+1} = \begin{cases} D''_i + \beta_1 & \text{if } D''_i < \beta_3 \text{ (pattern 2)} \\ D''_i + \beta_1 + \beta_3 & \text{if } \beta_3 \leq D''_i < 0 \text{ (pattern 6)} \\ D''_i + \beta_2 & \text{if } 0 \leq D''_i \text{ (pattern 4)} \end{cases}$$

3.3. Termination

Since either two or three pixels will be set in each iteration of a loop, the stopping conditions of the loops must be determined in a different manner from the double-step algorithm. In this section, the termination conditions of the double- and triple-step algorithm for the case where $dy/dx < 1/2$ will be discussed first. Then the cases where dy/dx is greater than 1/2 and dy/dx equals 1/2 will be considered.

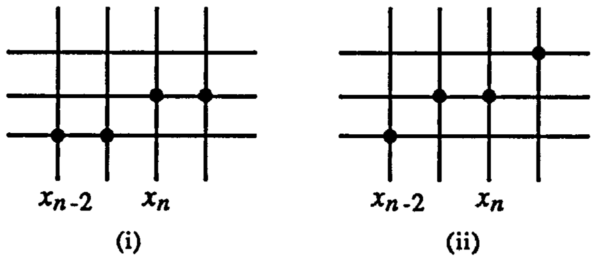

Figure 4. Cases where additional pixels are set by the double- and triple-step algorithm.

When $dy/dx < 1/2$, the termination conditions are easily determined by constructing a loop which stops iterating once the pixel at x_{n-1} is set. It is followed by an **if** clause which sets the pixel corresponding to the endpoint (provided it has not already been set) since the loop either sets the correct number of pixels or all the pixels except the endpoint. The fact that the loop sets the pixels described above is proven by considering all possible cases. For instance, if the first pixel to be set in the current iteration of the loop is at location x_{n-1}, then the correct number of pixels will be set when pattern 1 (3) occurs. On the other hand, if pattern 5 occurs, an "extra" pixel will be set (Figure 4 (i)). However, pattern 5 cannot occur in such a situation; since the slope is less than 1/2, a sideways movement must always be made from x_{n-1}. In the remaining cases, either the correct number of pixels are set, or more iterations must be performed.

When $1/2 < dy/dx \leq 1$, a loop and an **if** clause are constructed as before. Once again, it is proven that no "extra" pixels are set by considering the different cases which are possible. For instance, if the first pixel to be set in the current iteration is at location x_{n-1}, the correct number of pixels will be set when pattern 4 (2) occurs. On the other hand, if pattern 6 occurs, an "extra" pixel will be set (Figure 4 (ii)). However, pattern 6 cannot occur in such a situation; since the slope is greater than 1/2, a diagonal movement must always be made from location x_{n-1}. The remaining cases are proven using similar arguments. Lastly, lines having a slope equal to 1/2 are now handled by Algorithm 3 since an additional pixel may be set when Algorithm 4 processes these lines. The calculations performed when determining the termination conditions as well as the rest of the steps of the double- and triple-step algorithm are shown in detail in Figure 5.

```
procedure LINE(a1, b1, a2, b2: integer);
var dx, dy, incr1, incr2, incr3, D, x, y, xend, c: integer;

   procedure draw(pattern: integer);
   begin
      case pattern of
         1: ++x; point(x, y); ++x; point(x, y);
         2: ++x; point(x, y); ++x; ++y; point(x, y);
         3: ++x; ++y; point(x, y); ++x; point(x, y);
         4: ++x; ++y; point(x, y); ++x; ++y; point(x, y);
         5: ++x; point(x, y); ++x; ++y; point(x, y);
            ++x; point(x, y);
         6: ++x; ++y; point(x, y); ++x; point(x, y);
            ++x; ++y; point(x, y);
         end {case}
   end {draw}

begin
   dx = a2 - a1; dy = b2 - b1;
   x = a1; y = b1;
   xend = a2 - 1;
   point(x, y);
   incr2 = 4*dy - 2*dx;
   if incr2 <= 0 then begin          {slope is <= 1/2}
      c = 2*dy;
      incr1 = 2*c;
```

```
        incr3 = incr2 + c;
        D = incr1 - dx;
        while x < xend do
            if D < 0 then begin
                draw(1); D = D + incr1; end
            else if D >= c then begin
                draw(3); D = D + incr2; end
            else begin
                draw(5); D = D + incr3; end;
        end
    else begin                          {slope is > 1/2}
        c = 2*(dy - dx);
        incr1 = 2*c;
        incr3 = incr2 + c;
        D = incr1 + dx;
        while x < xend do
            if D >= 0 then begin
                draw(4); D = D + incr1; end
            else if D < c then begin
                draw(2); D = D + incr2; end
            else begin
                draw(6); D = D + incr3; end;
        end; {if}
    {plot endpoint if it is not set}
    if x < a2 then point(a2, b2);
end {LINE}
```

Figure 5. The double- and triple-step algorithm.

Table 1. Run times (in μs) of the double- and triple-step (DTS) and the double-step (DS) algorithms for lines 100 units in length.

Endpoint	DTS	DS	% Reduction
(100, 0)	50.9	50.9	0.0
(99, 4)	51.1	51.7	1.2
(99, 8)	52.8	53.5	1.3
(99, 13)	54.5	55.9	2.5
(98, 17)	56.1	57.6	2.6
(97, 21)	54.1	58.6	7.7
(96, 25)	59.1	60.4	2.2
(95, 30)	52.8	61.6	14.3
(93, 34)	53.8	62.5	13.9
(92, 38)	58.5	64.4	9.2
(90, 42)	63.1	65.4	3.5
(88, 46)	63.4	64.6	1.9
(86, 49)	57.9	62.4	7.2
(84, 53)	51.6	59.9	13.9
(81, 57)	51.0	56.1	9.1
(79, 60)	53.8	54.4	1.1
(76, 64)	50.4	51.7	2.5
(73, 67)	48.2	48.2	0.0
(70, 70)	45.8	45.8	0.0

3.4. Complexity Analysis

In this section, the double-step and the double- and triple-step algorithms are compared under best, average, and worst case conditions. The run-time performances of each algorithm are then given.

Under best case conditions, it is apparent that the number of iterations is reduced by 33% by considering a line where $3dy = dx$ and dx is some large integer. Once rasterized by the double- and triple-step algorithm, this line will correspond to a sequence of pattern 5 settings, and three pixels are set each iteration. However, when rasterized by the double-step algorithm, only two pixels are set each iteration. Therefore, the ratio of the number of iterations performed by each algorithm equals $\frac{dx/3}{dx/2} = 2/3$. For an average case analysis, we will only consider lines such that $0 \leq dy/dx \leq 1/2$. The remaining cases are proven in a similar manner. We begin by noting that the relative speed of the double- and triple-step algorithm is dependent on the value of dy. In other words, when dy equals zero, two steps are taken every iteration. When dy equals one, there can be at most one iteration where three steps are taken, and so on. Assuming that the number of steps of size three is $dy/2$ on average and the average value of dy is $dx/4$, the average number of iterations having steps of size three equals $dx/8$, and the average number of iterations is $7dx/16$. Since the double-step algorithm always iterates $dx/2$ times, the number of iterations is reduced by 12.5% on average. It is obvious that there are no differences under worst case conditions.

Table 2. Run times (in μs) of the double- and triple-step (DTS) and the double-step (DS) algorithms for lines 1000 units in length.

Endpoint	DTS	DS	% Reduction
(1000, 0)	487	487	0.0
(999, 43)	493	505	2.4
(996, 87)	506	524	3.4
(991, 130)	523	541	3.3
(984, 173)	538	558	3.6
(976, 216)	531	574	7.5
(965, 258)	563	587	4.1
(953, 300)	508	600	15.3
(939, 342)	516	614	16.0
(923, 382)	563	623	9.6
(906, 422)	611	634	3.6
(887, 461)	618	630	1.9
(866, 499)	553	606	8.7
(843, 537)	486	581	16.4
(819, 573)	487	552	11.8
(793, 608)	504	524	3.8
(766, 642)	487	497	2.0
(737, 675)	463	465	0.4
(707, 707)	434	434	0.0

Comparisons were also made by implementing each algorithm using compiled C on a DECstation without graphics output. The lines which were tested form the spokes in the first octant of wheels having radii which are 10, 100, and 1000 units in length. These wheels are centered at the origin so the lines can be specified by simply giving the endpoint. As one may suspect, there are no noticable differences in the initialization costs. If anything, a case could be made that the initialization costs are reduced by an instruction or two. The differences are also negligible for lines which are 10 units in length since there are only a few instructions that can be eliminated. Therefore, these results are not given. However, as shown in Tables 1 and 2, the speed is reduced significantly for some of the longer lines. Again, it is acknowledged that these savings will not be realized in practice due to the more time-consuming pixel write operations. We also note that the time reductions should not approach the values by which the iterations are reduced since some work, such as the initialization costs and incrementing x and y, must always be performed.

4. Final Remarks

A method of increasing the speed of one of the fastest line drawing algorithms is presented and analyzed. Our investigation shows that the speed of the algorithm can be improved while keeping the code complexity and initialization costs the same. The speed of the resulting algorithm could be improved further by also exploiting the symmetry of lines as noted in [11]. Under these conditions, either four or six pixels will be set each iteration of the loop. Perhaps similar results could be obtained by applying this method to various other line drawing algorithms that exist.

5. References

[1] P. Bao and J. Rokne. Quadruple-step line generation, *Computers & Graphics 13*(4), 461-469 (1989).

[2] J.E. Bresenham. Algorithm for computer control of a digital plotter, *IBM Systems Journal 4*(1), 25-30 (1965).

[3] J.E. Bresenham. Incremental line compaction, *Computer Journal 25*(1), 116-120 (1982).

[4] J.E. Bresenham. Run length slice algorithm for incremental lines, in *Fundamental Algorithms for Computer Graphics* (R.A. Earnshaw, Ed.), NATO ASI Series, Springer-Verlag: New York, 59-104 (1985).

[5] G. Casciola. Basic concepts to accelerate line algorithms, *Computers & Graphics 12*(3/4), 489-502 (1988).

[6] C.M.A. Castle and M.L.V. Pitteway. 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, 135-139 (1985).

[7] C.M.A. Castle and M.L.V. Pitteway. An efficient structural technique for encoding 'best-fit' straight lines, *Computer Journal 30*(2), 168-175 (1987).

[8] H. Freeman. Boundary encoding and processing, in *Picture Processing and Psychopictorics* (B.S. Lipkin and A. Rosenfeld, Eds.), Academic Press: New York, 241-266 (1970).

[9] H. Freeman. On the encoding of arbitrary geometric configurations, *IRE Trans. EC-102*, 260-268 (1961).

[10] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated Ray-Tracing System, *IEEE CG&A 6*(4), 16-26 (1986).

[11] P.L. Gardner. Modifications of Bresenham's algorithm for displays, *IBM Tech. Disclosure Bull. 18*(5), 1595-1596 (1975).

[12] M.L.V. Pitteway and A.J.R. Green. Bresenham's algorithm with run line coding shortcut, *Computer Journal 25*(1), 114-115 (1982).

[13] G.B. Regiori. Digital computer transformations for irregular line drawings, Tech. Rep. 403-22, Department of Electrical Engineering and Computer Science, New York Univ., April 1972.

[14] J.G. Rokne, B. Wyvill, and X. Wu. Fast line scan-conversion, *ACM Transactions on Graphics 9*(4), 376-388 (1990).

[15] X. Wu and J.G. Rokne. Double-step incremental generation of lines and circles, *Computer Vision, Graphics, and Image Processing 37*(3), 331-344 (1987).