Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 30 (2006) 207–212
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Technical section

# Drawing lines by uniform packing

Asif-ul Haque[a], Mohammad Saifur Rahman[a], Mehedi Bakht[a], M. Kaykobad[a,b,*]

[a]Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka 1000, Bangladesh
[b]Department of Computer Science and Engineering, North South University, Bangladesh

**Abstract**

In this paper, we introduce a new approach to line drawing that attempts to maintain a uniform packing density of horizontal segments to diagonal segments throughout the line. While the conventional line drawing algorithms perform linear time computations to find the location of the pixels, our algorithm takes logarithmic time. Also, experimental results show that the quality of line is acceptable and comparable to the well-known Bresenham's line-drawing algorithm.


*Keywords:* Algorithm; Graphics

## 1. Introduction

Line is a very important primitive in computer graphics. Sometimes designs contain hundreds of lines that are to be refreshed or redrawn in a fraction of a second. This is why it is so important to draw these lines as efficiently as possible. This importance warranted development of a number of line algorithms. The first such algorithm was devised by Bresenham [1,2] that avoided real numbers and division operations in computation for fast generation of line segments. Later on, Kappel [3] developed midpoint algorithm for drawing line segments. Parallel methods for generating lines have been discussed in Pang [4] and Wright [5]. In addition, a recursive bisection method was also developed in order to reduce computational complexity. But quality of line produced by this algorithm was inferior. This paper presents a new line-drawing algorithm for graphics systems based on the idea of uniform packing. Analysis shows that it is computationally more efficient than commonly used Bresenham's line drawing algorithm. The trade-off here is the slight degradation of the quality of lines drawn, measured in terms of quantitative measures such as the sum of squared errors of the plotted pixels, with respect to the mathematical line. Experimental results suggest that the sum of squared errors for our algorithm is quite comparable to that for Bresenham's algorithm, which produces probably minimum sum of squared errors.

## 2. Bresenham's algorithm

Bresenham developed an elegant line drawing algorithm in Ref. [1]. His algorithm uses only integer arithmetic and allows incremental calculations for finding next pixel coordinates. For lines with arbitrary real valued endpoint coordinates, a floating point version of the same algorithm can be adopted. Furthermore, Bresenham's incremental technique can be applied to integer computation of circles as well.

Pitteway [6] proposed a slightly different formulation of Bresenham's algorithm. This version, known as the *midpoint technique*, was later adapted by Van Aken

*Corresponding author.
*E-mail address:* kaykobad@cse.buet.ac.bd (M. Kaykobad).

0097-8493/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2006.01.026

[7] and other researchers. For lines and integer circles, the midpoint formulation reduces to the Bresenham formulation and therefore generates the same pixels [8].

Bresenham showed that his line and integer circle algorithms provide the best-fit approximation to the true lines and circles by minimizing the error to the true primitives [2]. We, therefore compare performance of our line drawing algorithm against Bresenham's algorithm in terms of error as well as computational time. We have used the *midpoint technique* formulation of Bresenham's algorithm, as implemented in Ref. [9].

## 3. New algorithm

If two consecutive pixels are put along the same horizontal line 1 unit apart, we define it to be a horizontal move. Similarly when two consecutive pixels are put along the diagonal of a unit square, we define it to be a diagonal move. Any straight line drawn on a computer display is a sequence of these two types of segments. The algorithm presented here depends on the idea of packing horizontal and diagonal moves uniformly.

We have considered only lines with slope from 0 to 1. The algorithm can be easily extended for other lines. A mathematical line segment when plotted turns into a collection of consecutive line segments with common endpoints. Each of these segment has a horizontal projection of 1 unit. Suppose a line segment with slope between 0 and 1 has horizontal projection $dx$ and vertical projection $dy$. Since $dy/dx \leqslant 1$, we have $dx \geqslant dy$. So to render the line, $dx+1$ pixels are needed to be plotted – one for each column from 0 to $dx$. This results in $dx$ line segments where $dy$ segments have slope 1 (diagonal move) and $dx$–$dy$ line segments have slope 0 (horizontal move). Any permutation of $dx$–$dy$ horizontal segments and $dy$ diagonal segments results in a representation of the actual mathematical line through the two endpoints. We show a method of choosing a particular permutation efficiently which results in a line of acceptable quality. The main idea is to pack the two kinds of segments as uniformly as possible. If we have $h$ horizontal segments and $d$ diagonal segments with say, $h<d$, we must have at least $t = \lfloor d/h \rfloor$ diagonal segments for each horizontal segment. In particular, we must have $t$ diagonal segments for each of $h$-($d$ mod $h$) horizontal segments and $t+1$ diagonal segments for each of $d$ mod $h$ horizontal segments. If $h = 0$ then we must make all the diagonal moves. On the other hand, if $h = d$ then the uniform packing is 1 horizontal segment for each diagonal segment. To put 1 horizontal segment in $t$ diagonal segments, we put $\lceil t/2 \rceil$ agonal segments before the horizontal segment and then the remaining $t-\lceil t/2 \rceil$ diagonal segments. So, we have two kinds of collection of unit length segments—some with $t$ diagonal segments and some with $t+1$ diagonal segments per horizontal segment. These two kinds of composite segments can be thought of as two new basic segments like the horizontal and diagonal segments. The procedure described above can be performed again to pack the new composite segments to obtain uniformity. Doing so results in two kinds of even larger composite segments. The procedure is repeated until a uniform packing of the two composite segments can be done trivially—the number of one composite segment becoming either zero or equaling the other one. In the above description, we have assumed $h<d$. The case where $h>d$ is symmetric to this case.

As an example, suppose $H$ represents a horizontal move and $D$ represents a diagonal move. Suppose the line has a horizontal projection of length 10 and vertical projection of length 7. So we need to plot 11 pixels in total to render the line—one for each of the columns. Thus there are 7 diagonal moves ($D$) and $10–7 = 3$ horizontal moves ($H$). Distributing the 7 $D$s for the 3 $H$'s result in $DHD$, $DHD$ and $DDHD$ collections. The composite segments for the next iterations are $DHD$ and $DDHD$. We must pack 2 $DHD$ and 1 $DDHD$ segments in the next iteration. So, the final packing will be $DDHD$ in the middle with the $DHD$s leading and trailing it, i.e. $DHDDDHDDHD$. To render the line we must first draw a pixel at the starting point of the line segment. Then for each of the letters $D$ and $H$ in the resultant string, we must draw a pixel according to the following rule: if the letter is an $H$, we must draw the next pixel on the same horizontal line and if the letter is a $D$, we must draw the next pixel 1 unit higher along the $y$-axis. The following enlarged diagrams show the plotted pixels for this line segment using our algorithm in red and Bresenham's algorithm in blue. In this example, both algorithms choose the same pixels Fig. 1.

As another example we show the line segment defined by the endpoints (0,0) and (20,10). Here, there are 21 pixels to be plotted and the two algorithms choose different sets of pixels for rendering the line. As before, our line is the red one while the blue one is of Bresenham's algorithm Fig. 2.

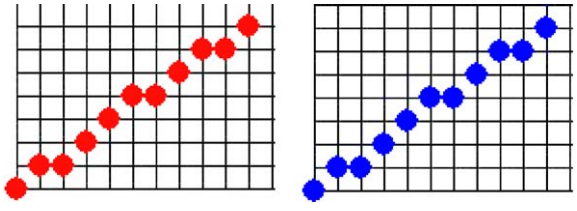

Fig. 1. Our line vs. Bresenham's line.

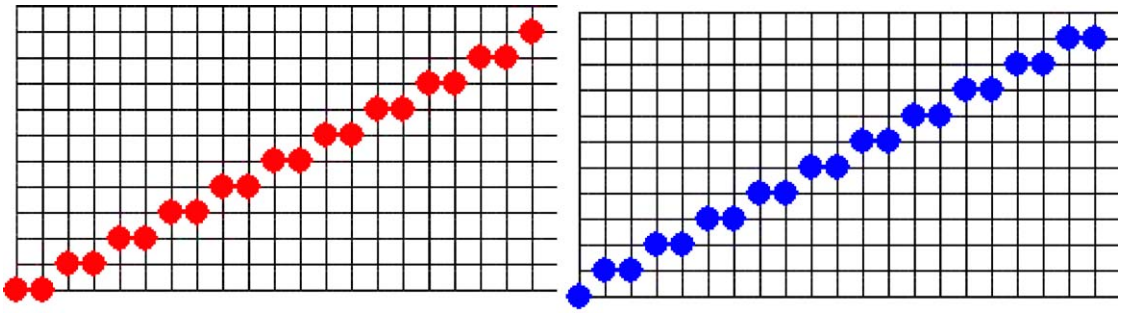

Fig. 2. Our line vs. Bresenham's line.

The pseudo-code for the algorithm is given below:
Draw-Line (horizontal-moves, diagonal-moves)

```
1.    h←horizontal-moves, d←diagonal-moves
2.    H←H'', D←''D''
3.    repeat
4.      if h>d then
5.        swap h,d
6.        swap H,D
7.      if h = 0 then
8.      print D d times
9.      terminate loop
10.   else if h = d then
11.     print H+D d times
12.     terminate loop
13.   else
14.     t←⌊d/h⌋
15.     tH←⌈t/2⌉ D+H +⌊t/2⌋ D
16.     t←t+1
17.     tD←⌈t/2⌉ D+H+⌊t/2⌋ D
18.     td←mod h
19.     th← h-td
20.   h←th, d←td, H←td, D←tD
```

The '+' operator in lines 11, 15, 17 means concatenation of two strings. Integers preceding strings in lines 15, 17 represent the number of times a string has to be concatenated with itself. For example, if $D$ holds the string "$HDH$", then $3D$ means "$HDH\ HDH\ HDH$".

## 4. Time complexity analysis

To determine the number of iterations of the loop at line 3, we compare our algorithm with Euclid's algorithm for finding greatest common divisor(gcd). The recurrence for finding gcd of two numbers $a$ and $b$ where $a \geqslant b$ is $\gcd(a,b) = \gcd(b, a \bmod b)$.

It is known (see pp. 852–853 in Ref. [10]) that number of iterations to find the gcd using this recurrence is O(log $b$). In our algorithm, we have used the following recurrence to obtain our iterative implementation: line$(h,d)$ = line$(h - d \bmod h,\ d \bmod h)$.

Observing the correspondence between the two recurrences, we conclude that asymptotically the number of iterations in our algorithm is O(log min$(h,d)$). If the line consists of $n$ segments ($n = h + d$), Bresenham's algorithm uses O($n$) additions and comparisons while our algorithm uses O(log min$(h,d)$) additions, subtractions, comparisons, divisions and modular operations. Suppose $k_1$ and $k_2$ are two constants that are the weighted sums of the number of operations in each pass of the two algorithms, respectively, here, the weights are the costs of performing various operations. For example, for $l$ bit integers addition can be given weight O($l$), multiplication O($l^2$),etc. So, $k_1$O($n$) will easily surpass $k_2$O(log min$(h,d)$), for lines involving large number of segments.

In our algorithm, we perform 1 addition, 1 subtraction and 6 divisions per iteration. 2 divisions in line 15 and 17 can be eliminated by storing the prior division results in those lines. The floor operator does not introduce any additional cost. The ceiling operator can be implemented by adding 1, when division operation produces remainder. So, at most 3 additions will be incurred.

So the total computational effort

$$= (3O(l) + 1O(l) + 4O(l^2))O(\log\min(h, d))$$
$$= (4O(l) + O(l^2) + O(\log\min(h, d))$$
$$= O(l^2\ \log\min(h, d)).$$

If we assume fixed length numbers as the coordinates of the lines to be drawn, then $l$ is constant. Therefore, the computational complexity of our algorithm is O(log min$(h,d)$). Tables 1 and 2 confirm the time complexity analysis just presented.

## 5. Experimental results

Much of our experiments are dedicated to the comparison of our algorithm to the widely used Bresenham's algorithm both in terms of efficiency and quality of rendered lines.

Table 1
Run time statistics for the new algorithm, taken over all possible lines of a particular number of segments

| Number of segments | Worst case statistics | | Mode statistics | | Average iteration |
|---|---|---|---|---|---|
| | No. of iteration | No. of cases | No. of iterations | No. of cases | |
| 500 | 6 | 8 | 3 | 184 | 3.224000 |
| 1000 | 7 | 2 | 4 | 318 | 3.592000 |
| 1500 | 7 | 2 | 4 | 474 | 3.673333 |
| 5000 | 9 | 2 | 4 | 1362 | 4.531200 |
| 20,000 | 10 | 2 | 6 | 5128 | 5.303800 |
| 50,000 | 10 | 154 | 6 | 13,292 | 5.853800 |
| 100,000 | 12 | 4 | 6 | 24,962 | 6.247360 |
| 500,000 | 14 | 2 | 7 | 118,554 | 7.187836 |
| 1,000,000 | 14 | 14 | 8 | 228,430 | 7.587130 |

Table 2
Run time statistics for Euclid's gcd algorithm

| Sum of the two integers | Worst case statistics | | Mode statistics | | Average iteration |
|---|---|---|---|---|---|
| | No. of iterations | No. of cases | No. of iteration | No. of cases | |
| 500 | 11 | 2 | 4 | 111 | 4.904000 |
| 1000 | 11 | 4 | 5 | 197 | 5.422000 |
| 1500 | 11 | 8 | 6 | 300 | 5.546667 |
| 5000 | 15 | 4 | 7 | 930 | 6.788400 |
| 20,000 | 15 | 52 | 8 | 3,340 | 7.908300 |
| 50,000 | 18 | 6 | 9 | 8,035 | 8.695160 |
| 100,000 | 19 | 14 | 9 | 15,821 | 9.264460 |
| 500,000 | 23 | 8 | 11 | 73,589 | 10.621212 |
| 1,000,000 | 24 | 2 | 11 | 144,766 | 11.196298 |

### 5.1. Number of arithmetic operations

Bresenham's algorithm and our algorithm were run with the same data to compute the number of arithmetic operations performed in making decisions whether to make a horizontal or diagonal move to obtain the next pixel. The cost of putting a pixel and the involved arithmetic was excluded, as in both cases the cost would be identical. For lines with a particular number of segments, we used 1000 randomly chosen lines for each of the segment lengths and averaged the operation counts. It is evident from the pseudo code that depending on the slope of the line, number of operations varies for our algorithm. However, for Bresenham's algorithm, the slope does not have any effect on the number of operations (Table 3).

### 5.2. Running time

The running time of the two algorithms, excluding the time to actually put the pixels, were also compared Table 4. As before, the results were averaged over 1000 random lines for each segment length.

The data clearly show that our algorithm works faster, even though the algorithm encounters considerable amount of memory access, compared to Bresenham's algorithm. The memory operations do not hamper the speed of the algorithm, since the memory operation and computations can take place in parallel.

### 5.3. Normalized mean square error

For each plotted pixel, the distance from true line was calculated and sum of squares of these errors was obtained. The result was normalized by dividing by the total number of pixels, so that the error for different lines can be compared. Again, 1000 random lines were used for each particular line size, i.e. number of segments and average was taken see Table 5.

Table 3
Number of operations

| No. of segments | Addition | | Subtraction | | Multiplication | | Division | |
|---|---|---|---|---|---|---|---|---|
| | New | Bresen-ham's | New | Bresen-ham's | New | Bresen-ham's | New | Bresen-ham's |
| 20,000 | 16.27 | 20,000 | 5.42 | 4 | 0 | 3 | 21.69 | 0 |
| 50,000 | 17.95 | 50,000 | 5.99 | 4 | 0 | 3 | 23.94 | 0 |
| 100,00 | 18.07 | 100,000 | 6.03 | 4 | 0 | 3 | 24.10 | 0 |
| 500,000 | 17.97 | 500,000 | 5.99 | 4 | 0 | 3 | 23.96 | 0 |
| 1,000,000 | 17.97 | 1,000,000 | 5.99 | 4 | 0 | 3 | 23.96 | 0 |

Table 4
Running time (in milli seconds)

| No. of segments | New algorithm | Bresenham's algorithm |
|---|---|---|
| 500 | 0.010 | 0.010 |
| 1000 | 0.010 | 0.020 |
| 1500 | 0.010 | 0.030 |
| 50,00 | 0.030 | 0.100 |
| 20,000 | 0.110 | 0.400 |
| 50,000 | 0.260 | 0.972 |
| 100,000 | 0.541 | 1.893 |
| 500,000 | 4.847 | 8.722 |
| 1,000,000 | 11.907 | 17.015 |

Table 5
Normalized mean square error

| No. of segments | Our algorithm | Bresenham's algorithm |
|---|---|---|
| 20,000 | 0.093627 | 0.083373 |
| 50,000 | 0.094293 | 0.083332 |
| 100,000 | 0.086688 | 0.083333 |
| 500,000 | 0.083468 | 0.083333 |
| 1,000,000 | 0.083365 | 0.083333 |

### 5.4. Variance of segment slope

To calculate the slope of different parts of a line of a particular length and to measure variance, 20 random lines were selected. From each line, 1000 different segments, identified by start and end point pair, were chosen at random. The slope of each segment was computed. These slopes can differ from true slope of the mathematical line, as both the algorithms consider pixels to be put only on integer coordinates. The variance of these slopes from the mathematical slope was measured. Finally, average was taken over the 20 selected lines.

From the data given in Table 6, it is clear that the new algorithm performs comparably with Bresenham's algorithm, with respect to variance of segment slope.

Table 6
Variance of segment slope

| No. of segments | New algorithm | Bresenham's algorithm |
|---|---|---|
| 20,000 | 0.000007 | 0.000007 |
| 50,000 | 0.000014 | 0.000014 |
| 100,000 | 0.000002 | 0.000002 |
| 500,000 | 0.000004 | 0.000004 |
| 1,000,000 | 0.000013 | 0.000013 |

## 6. Conclusion

We have introduced a new approach for line drawing that attempts to maintain a uniform packing density of horizontal segments and diagonal segments throughout the line. Experimental results show that the quality of lines drawn is comparable to the ones drawn by Bresenham's algorithm. Also, the computational effort in making decision as to the location of the pixels is logarithmic in number of segments in the line, rather than linear as in Bresenham's algorithm. However, the memory requirement in the new algorithm is linear rather than constant. Future studies can be done in attempt to reduce the memory requirements. Also, the algorithm is not incremental in its current implementation. Works can be done to find an incremental and simpler implementation of the algorithm.

## References

[1] Bresenham JE. Algorithm for computer control of a digital plotter. IBM Systems Journal 1965;4(1):25–30.
[2] Bresenham JE. A Linear algorithm for incremental digital display of circular arcs. CACM 1977;20(2):100–6.
[3] Kappel M.R. An ellipse-drawing algorithm for faster displays. in: Fundamental algorithms for computer graphics, Springer, Berlin, pp. 257–280, 1985.
[4] Pang AT. Line-drawing algorithms for parallel machines. IEEE Computer Graphics and Applications 1990;10(5): 54–9.

[5] Wright WE. Parallelization of bresenham's line and circle algorithms. IEEE Computer Graphics and Applications 1990;10(5):60–7.
[6] Pitteway MLV. Algorithm for drawing ellipses or hyperbolae with a digital plotter. Computer Journal 1967;10(3): 282–9.
[7] Van Aken JR. An efficient ellipse-drawing algorithm. CG&A 1984;4(9):24–35.
[8] Van Aken JR, Novak M. Curve-drawing algorithms for raster displays. ACM TOG 1985;4(2):147–69.
[9] Foley, JD, Andries van F, Steven K, Hughes JF. Computer graphics principles and practice, 2nd ed. in C, Fourth Indian Reprint, 2000, Addison Wesley Longman, Singapore.
[10] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms, 2nd ed. Cambridge, MA: MIT Press; 2002–2003.