EUROGRAPHICS '99 / P. Brunet and R. Scopigno
(Guest Editors)

Volume 18 (1999), Number 3

Fast Lines: a Span by Span Method

V. Boyer and J.J. Bourdin

Groupe de Recherche en Infographie et Synthèse d'images
Laboratoire d'Intelligence Artificielle
Université Paris 8
2, rue de la Liberté
93256 Saint-Denis cedex 2
FRANCE
boyer, jj@ai.univ-paris8.fr
Fax : 33.1.49.40.64.10

Abstract

Straight line's scan conversion and drawing is a major field in computer graphics. Algorithm's time computation is very important. Nowadays, most of research papers suggest improvements of the DDA method that was first presented by J. Bresenham. But other approaches exist as well like combinatory analysis and linguistic methods. Both of them use multiple string copies that slow down the efficiency of the algorithms. This paper proposes a new algorithm based on a careful analysis of the line segments' properties some of them previously unused. Our algorithm is proved significantly faster than previously published ones.

1. Introduction

One of the most important functions of graphics displays is drawing straight lines. It is achieved by drawing a discrete path between two given points. Although the path is discrete, it has to be as linear as possible. The visual aspect of the path and the speed of the algorithm are the major qualities of scan-conversion. Bresenham's algorithm[1] is often used because it is both fast and easy to encode. It computes the best approximations of the true line points. The DDA method introduced by Bresenham is also used by the major improvements of his algorithm[2, 3, 4, 5, 6, 7, 8, 9].

Here are some other approaches:

- Combinatory analysis leads to significantly different and elegant as well algorithms[10, 11, 12, 13]. This method is derived from Euclid's algorithm computing the greatest common divisor.
- Linguistic methods[14, 15] lead to new presentations of lines. Due to the cost of string copies used, these algorithms are not faster than the previous ones.

It is possible to combine the above mentioned approaches to get fast algorithms[16, 17] with good visual quality results. Moreover it is possible to use hardware functionalities to draw simultaneously a limited range of pixels. For example this task is performed by the *rectwrite* function on SGI computers. The same kind of performances does not exist yet on cheap raster devices such as laser printers. But future hardware will include some of these performances in order to speed-up.

This paper summarizes different properties of lines and proposes some additional ones. These properties are chosen for the speed-up qualities (in the sense of computational effort). A new algorithm is given which uses only integer arithmetic. Benchmarks prove that this algorithm is faster than previous ones. The speed-up grows according to the lines' length, 6 times faster than the quickest previous algorithm[17] and 20 times than the original Bresenham version.

2. Properties of the Straight line

This section presents five important properties of straight lines. These properties are already well-known (see for example[18, 2, 19, 20, 13, 21, 22, 14, 10, 11]) but our formulation makes them clearer. These properties were chosen for the gain in speed or the limitation on the size of the workspace. On the discrete plane $\mathbb{N} \times \mathbb{N}$ representing the raster device, each point has two coordinates, x and y. Let $P(x_p, y_p)$ and

$Q(x_q, y_q)$ be two points of the plane. The values x_p, y_p, x_q, y_q are integers. Our purpose is to draw the line from P to Q, i.e. to draw a "linear" path from P to Q. This path will be mentioned as the line from P to Q. The continuous line will refer to the line segment of the real plane.

Let the differences of coordinates be u and v:

$$u = x_q - x_p$$
$$v = y_q - y_p$$

The line has a **slope** that is defined by the values u and v, written as the pair (u, v) or the ratio v/u (see Figure 1).

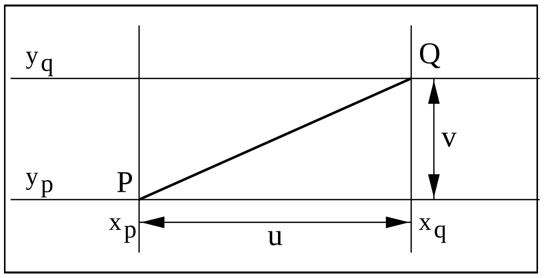

Figure 1: *a line and its slope*

Property 1 : The line from $P(x_p, y_p)$ to $Q(x_q, y_q)$ is an exact transposition of the line from $(0,0)$ to (u, v).

It means that if (x, y) is a point of the line from $(0,0)$ to (u, v) then $(x + x_p, y + y_p)$ is a point of the line from P to Q. Therefore the line will only be defined by its slope (u, v) and the end-points of the line will not be mentioned. The (u, v) line will be used for "the line with slope v/u". The lines will be restricted to the first octant $(u > v \geq 0)$. Other lines can be computed like simple symmetries of a line in this octant[1, 16, 2]. In other words, we can limit our studies to the first octant, knowing that the lines left are just transpositions.

Property 2 : Each line (u, v) in the first octant verifies:

$$\forall\, x \in [0, u], \exists! y \in [0, v] \ \ / \ (x, y) \in line$$

In other words for each x there is one and only one point in the path. The true ordinate $y_x \in I\!R$ associated to x is:

$$y_x = \frac{vx}{u} \tag{1}$$

As y is an integer, it is an approximation of y_x. The approximation of a real number with an integer can either be the *best* approximation or the *lower* approximation (the greatest integer less than or equal to y_x) or the *upper* approximation (the smallest integer greater than or equal to y_x). We use the usual presentation $\lfloor y_x \rfloor$, $[y_x]$ and $\lceil y_x \rceil$ (see for example[16]). Let y, y' and y'' be the corresponding approximations:

$$y = \ (lower)\, y_x \ = \left\lfloor \frac{vx}{u} \right\rfloor \tag{2}$$

$$y' = \ (best)\, y_x \ = \left[\frac{vx}{u} \right] \tag{3}$$

$$y'' = \ (upper)\, y_x \ = \left\lceil \frac{vx}{u} \right\rceil \tag{4}$$

Most algorithms[1, 16, 6, 5, 10···] compute the *best* approximation. In this case, one problem remains unsolved: when the value is half of an integer, is the rounding up more appropriate than the rounding down? For example, is 3.5 closer to 4 or to 3?

Let us recall three properties suggested by Freeman[18] and summarized by Wu[15] to describe the relationships between two neighboring points of any line. The direction between two close points is given by the Freeman code[18].

Property 3 : There are at most two basic directions and these ones can differ only by unity, modulo eight.

Property 4 : One of these values always occurs "singly"[15].

Property 5 : Successive occurrences of the principal direction occurring singly are as uniformly spaced as possible.

In the first octant the directions used are always horizontal or diagonal and are, using the Freeman's code: 0 and 1. Note that if (x, y_1) and $(x+1, y_2)$ are two neighboring points, the difference $y_2 - y_1$ is equal to the value of the Freeman code.

The line may be described by the set of points of the path or by the first point and the chain code of the successive directions.

Property 4 enables to consider a **span** as a subchain repeating the same code. Two spans are separated by one occurrence of the other code.

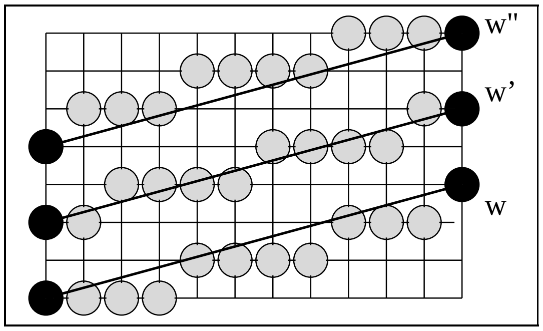

Figure 2: *Three lines for slope 11, 3*

Let a **word** be the chain code of a line. Since three different approximations exist for a slope (u, v) there are three different words. We denote $w(u, v)$ the chain code for the *lower* approximation, $w'(u, v)$ the chain code for the *best* approximation, $w''(u, v)$ the chain code for the *upper* approximation. For example, in Figure 2, the different approximations of the (11, 3) line are presented. We have:

$$w(11,3) = 00010001001$$
$$w'(11,3) = 01000100010$$
$$w''(11,3) = 10010001000$$

Note that when the slope is obvious we use the abbreviations w, w' or w''.

It has been noted[20, 15] that property 5 is "somewhat fuzzy". This fuzziness applies only for w', the *best* approximation. The words w and w'' respect property 5. In other words, the length of spans differ only by one unit. This is also true for the *best* approximation even though the first span is broken in two parts, its begin is situated at the end of the line.

As the research in computer graphics was mainly focused on the *best* approximation, property 5 was generally ignored. Using the *lower* approximation leads to other properties of the line's code chain. These properties are presented in the following section.

3. Properties of the line word

In order to present new properties of the chain code we need some simple operators such as append or repeat. We use the following abbreviations:

append $\cdot$ will denote the append function on chains: 1010·011=1010011
repeat w^n is w n times repeated:$(001)^3 = 001001001$
reverse rev(w) is the reverse of the chain: rev(001)=100
opposite not(w) is used to replace each letter in w by its binary opposite: not(001)=110
letter w_i is the i-th letter of w: if w=001001, w_3=1 and w_1=0

Property 6 : For the line (u,v), w, w' and w'' are combined words. There exist small subchains α and β and:

$$\begin{aligned} w'\cdot\alpha &= \alpha\cdot w \; (lower - best\ commutativity) \\ \beta\cdot w' &= w''\cdot\beta \; (best - upper\ commutativity) \\ rev(\alpha) &= \beta \end{aligned}$$

For example with the line (11, 3), α =01.

Property 7 : For the line (u,v), the length n of α (see property 6) respects the condition:

$$(vn + \lfloor u/2 \rfloor)\ modulo\ u = 0$$

For the slope (11, 3), that is: $(3n+5)\ modulo\ u = 0$ and n=2 or 13 or -9...

Therefore the choice of the approximation - during line's computation - is not bound to the type of the approximation needed for the drawing. In this paper we focused on the word w. The word w' can be deduced by simple translation.

Property 8 : The (ku,kv) line is k times the (u,v) line.

$$w(ku,kv) \;=\; w^k(u,v)$$

e.g.:

$$\begin{aligned} w(22,6) &= w(11,3)\cdot w(11,3) \\ w(22,6) &= 0001000100100010001001 \end{aligned}$$

Property 9 : If $g = gcd(u,v)$ (greatest common divisor) then the line of slope (u,v) is g times the line $(u/g, v/g)$.

$$w(u,v) = w^g(u/g, v/g)$$

This property is a significant consequence of property 8. It enables to work on a smaller word and repeat its pattern. As noted by Angel and Morrison[6] the average gcd of integers in range 1 to 1024 is almost 5. The computation of only one fifth of the line clearly speeds up 5 times the algorithm.

From now on the values u and v are considered reduced:

$$gcd(u,v) = 1$$

Property 10 : **Inner symmetry**. The (u,v) line word respects:

$$\begin{aligned} w_1 &= 0 \\ w_u &= 1 \\ w_i &= w_{u+1-i} \;\; \forall i \in]1,u[\end{aligned}$$

This property is used by Rokne et al.[5] in order to speed-up their algorithm. But their formulation is not completely true because they use the *best* approximation and the rounding problem (up or down) of the numbers in $n+0.5$ remains. For the *lower* approximation the property 10 has been proved by Boyer et al.[23].

Property 11 : **Hexadecant symmetry**. The (u,v) line word respects:

$$w(u,v) = not(w''(u,u-v))$$

For example:

$$\begin{aligned} w(11,8) &= 01101110111 \\ u-v &= 11-8=3 \\ w''(11,3) &= 10010001000 \\ not(w''(11,3)) &= 01101110111 \end{aligned}$$

We give now a formal proof of this property. Let (x,y_x) be a point of the *lower* (u,v) line and (x,y''_x) be a point of the *upper* $(u,u-v)$ line. We get:

$$\begin{aligned} y_x &= \left\lfloor \frac{vx}{u} \right\rfloor \\ y''_x &= \left\lceil \frac{(u-v)x}{u} \right\rceil \end{aligned}$$

The sum $y_x + y''_x$ depends only on x:

$$\begin{aligned} y_x + y''_x &= \left\lfloor \frac{vx}{u} \right\rfloor + \left\lceil \frac{(u-v)x}{u} \right\rceil \\ y_x + y''_x &= \left\lfloor \frac{vx}{u} \right\rfloor + \left\lceil \frac{ux}{u} + \frac{-vx}{u} \right\rceil \\ y_x + y''_x &= x + \left\lfloor \frac{vx}{u} \right\rfloor + \left\lceil \frac{-vx}{u} \right\rceil \end{aligned}$$

but

$$\left\lfloor \frac{vx}{u} \right\rfloor = -\left\lceil \frac{-vx}{u} \right\rceil$$

so we get

$$y_x + y''_x = x$$

and

$$y_{x+1} + y''_{x+1} = x+1$$

Since y_x, y_{x+1}, y''_{x+1} and y''_x are integers and since the path is made of neighboring points, we have:

$$\begin{aligned} \textbf{if } y_x = y_{x+1} \quad & \textbf{then } y''_{x+1} = 1 + y''_x \\ \textbf{if } y_x = 1 + y_{x+1} \quad & \textbf{then } y''_{x+1} = y''_x \end{aligned}$$

Therefore the associated code values differ.□

This property is important because it enables to divide by 2 the workspace. Furthermore the next property shows that it is possible to work only with the *lower* approximation.

Property 12 : If the slope is (u,v), then we have:

$$\begin{aligned} w_1 = 0, &\qquad w''_1 = 1, \\ w_u = 1, &\qquad w''_u = 0, \\ w_i = w''_i &\qquad \forall\, i \in \,]1,u[\end{aligned}$$

We give now the proof of this property. A diagonal move occurs each time the continuous line crosses the grid. This is valid for both *lower* and *upper* approximations. The exceptions occur when the division $(vx)/u$ has no fractional part (is an integer). But since $gcd(u,v) = 1$, and $x \in [0,u]$, there are only two such exceptions: $x = 0$ and $x = u$. The corresponding letters are treated separately.

Therefore if $u < 2v$, the word $w(u,v)$ can be computed as:

$$\begin{aligned} w_1(u,v) &= 0, \\ w_u(u,v) &= 1, \\ w_i(u,v) &= not(w_i(u,u-v)) \;\; \forall\, i \;\in \,]1,u[\end{aligned}$$

The workspace is reduced to the first hexadecant:

$$0 \le v < 2v \le u$$

In this hexadecant, the spans are always horizontal and separated by diagonal moves (see figure 3). Let us focus now on consequences of property 5. As indicated above, the length of spans differ only by a single unit. As the spans are separated by diagonal moves (occurring singly) and as the first move is the beginning of a span and the last move is a diagonal move, there are as much spans as there are diagonal moves:

$$nb\_spans = v$$

Property 13 : The average length of spans is:

$$av\_lgt = \frac{u-v}{v}$$

That is the ratio of horizontal moves divided by the number of spans. As this value is not an integer (since $gcd(u,v) = 1$) there are two different lengths: $\lfloor (u-v)/v \rfloor$ and $\lceil (u-v)/v \rceil$. We define the *short spans* as the spans whose length is the lower approximation and the *long spans* as the upper one. Figure 3 presents short and long spans along the line (97, 31).

$$\begin{aligned} short\_lgt &= \left\lfloor \frac{u-v}{v} \right\rfloor \\ long\_lgt &= \left\lceil \frac{u-v}{v} \right\rceil \end{aligned}$$

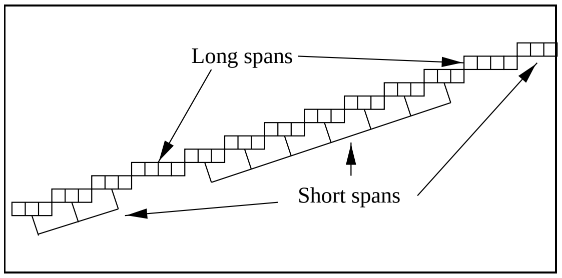

Figure 3: *a part of the (97, 31) line*

Property 14 : The numbers of short (nb_s) and long (nb_l) spans are precisely defined:

$$\begin{aligned} nb\_l &= (u-v)\ modulo\ v = u\%v \\ nb\_s &= v - (u\%v) \end{aligned}$$

As in C, we use the abbreviation % for the modulo.

To understand these equations one can imagine filling each span as if it was a short span. The number of horizontal moves left unused *nb_hml* is:

$$nb\_hml = u - v - v\left\lfloor \frac{u-v}{v} \right\rfloor$$

so,

$$nb\_hml = (u-v)\ modulo\ v = u\%v$$

As the length of spans differ only by a single unit, the horizontal moves left are distributed between long spans. The number of horizontal moves left is therefore equal to the number of long spans. Now we can dispose these spans along the line.

Property 15 : The short and long spans define precisely the (v, nb_s) line.

Note that $v = nb\_s + nb\_l$.

For example:

$$\begin{aligned} w(11,3) &= 00010001001 \\ short\ spans &= 00 \\ long\ spans &= 000 \\ nb\_s &= 3 - (11\%3) = 1 \\ w(3,1) &= 001 \end{aligned}$$

Let $S_m(w)$ be the function that turns each letter of w into a span with the separator:

$$\begin{aligned} S_m(0) &= 0^{m+1}1 \\ S_m(1) &= 0^m 1 \end{aligned}$$

Property 15 may be reformulated as:

$$\begin{aligned} w(u,v) &= S_m(w(v, v-(u\%v))) \\ m &= \left\lfloor \frac{u-v}{v} \right\rfloor \end{aligned}$$

4. Previous algorithms

Most of the line drawing algorithms take advantage of one or more of these properties. This section summarizes the relations between different algorithms and these properties.

First of all, Bresenham's algorithm[1] is obviously based on properties 1 and 2. In his paper Bresenham presented property 3.

Brons[14], Wu[15], Berstel[11], Troesch[13] and Reveilles[12] worked on the properties of the chain code. Therefore their algorithms use most of line word's properties presented above. Both multiple recursions and multiple string copies slow down the computing. That is why compared to Bresenham's, these algorithms are not faster.

Castle and Pitteway's algorithm[10] uses Euclid's algorithm and property 15. In their algorithm subtractions are used for the dividing operations.

Angel and Morrison[6] presented and used property 9. They estimated a speedup factor of 5 when long lines are drawn and evaluated it to 3 by software simulation. But the cost of gcd calculation remains predominant.

The run length slice algorithms, presented by Bresenham[16] and Pitteway et al.[24], mainly use properties 5 and 14. Another algorithm[17] uses the spans called "steps". Since some hardware presents the possibility to draw a limited range of pixels with a single subroutine, these algorithms are particularly efficient. The first part of[17] uses property 15, while the second part uses Bresenham's algorithm to compute spans' positions.

In recent algorithms, the double-step[5], triple-step[8] and N-step[7] are Sproull's derivation algorithm's adaptations[3]. Rokne et al. divided the octant in two parts. Two different formulas are used according to the hexadecant involved. The multiple step principle is to compute y_{x+n} and deduce from its value the values of intermediate y_{x+i}. For example, in the first hexadecant, $y_{x+2} = y_x$ implies that $y_{x+1} = y_x$. This property is useful to speed up the Bresenham's algorithm but even if a symmetry (property 10) is used (Rokne et al. algorithm[5]), it doesn't reduce computation time significantly.

Since properties 11 and 12 are new ones, our algorithm presented hereafter is the first one to take advantage of them.

5. The new algorithm

In this section we present our new algorithm. Its advantage comparing to previous ones is the extremely short time computation and display as well of drawing straight lines. The *lower* approximation was chosen for its rapidity. This choice implies no further consequences as the drawing keeps good visual properties. The only exception is the case where v is very small relative to u (for example $u = 100$, $v = 1$). But in this case, all known algorithms lead to partially degenerated lines: without anti-aliasing process anyone could see the discontinuity. Moreover, as noted in property 6, the *best* approximation line is a translation of the *lower* one.

Furthermore a simple anti-aliasing consists in the drawing of two pixels with attenuated colors for each point of the path[25]. The choice of the pixels to draw is easy while using *lower* approximation: the pixels drawn are the current one and its vertical up neighbor. With the *best* approximation, sometimes the second pixel would be up and some other times it would be down. Our algorithm uses also an adaptation of the double-step algorithm[5] to the *lower* approximation. Since $u > 2v$, there are only three possibilities for the path following a point (x,y) (see Figure 4). These possibilities are:

A horizontal move, horizontal move
B horizontal move, diagonal move
C diagonal move, horizontal move

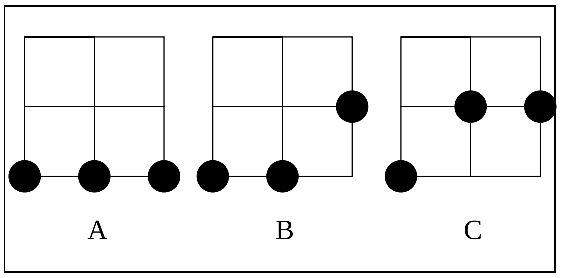

Figure 4: *The three possible moves*

The error made when choosing a point P for the line is estimated by the following error function:

$$error(x,y) = vx - u(y+1)$$

if $error(x,y) > 0$ then the upper point $(x, y+1)$ is a point of the line and (x,y) is not. This leads to:

```
if $error(x+2,y) < 0$        (1)
  then case A
  else if $error(x+1,y) < 0$        (2)
    then case B
    else case C
```

During the loop, for a current point (x,y) the value Δ of the error computed is:

$$\Delta = error(x+2,y)$$

The sign of Δ is tested at line (1) and the sign of $(\Delta - v)$ at line (2).

Our algorithm expects an existing array *clr* of value 1. This array is the second argument of the *rectwrite* function.

Our algorithm builds an array *slope* with the lengths of the successive spans. For example if the line to draw is (11,3), the long spans are made of three 0 and short spans are made of two 0, the array slope will be {3,3,2} when completed. As in the C language the first value of the array is *slope[0]*.

The new algorithm is given by a function *quickline* (see figure 5). It is composed of three parts. The first part computes intermediate values used by the *line* functions. This is an application of properties 9 and 15. The second part of the algorithm (the *line* and *line2* functions calls) fills the slope array. This is close to Rokne and al.[5] algorithm with an adaptation to the *lower* approximation and to properties 11 and 12. It fills simultaneously the beginning and the end of the array. The third part effectively draws the line.

The *rectwrite* function, used for the drawing, performs the writing of a limited range of pixels. The first parameter N is the number of pixels to draw, the second parameter is an array of entries of the color Look-Up-Table. The i-th pixel will be drawn with the i-th color in the array, for all i from 0 to $N-1$. Here the array will be filled with 1s.

The *rectwrite* function does not yet permit the drawing of vertical or diagonal sets of pixels. Adding these two possibilities should improve our algorithm.

```
void quickline(xp, yp, u, v) {
  g = gcd (u, v);
  u = u / g;
  v = v / g;
  long = u / v; /*length of long*/
  short = long - 1; /*and of short spans*/
  nb_l = u % v; /*number of long spans*/
  if (v > 2*nb_l)
    line2(v, nb_l, long, short, slope);
  else
    line (v, v-nb_l, long, short, slope);
  for ( ; g != 0 ; g--) {
    for (step = 0; step < v; step++) {
      cmov2i (xp, yp);
      rectwrite (slope [step], clr);
      xp += slope [step];
      yp++;
    }
  }
}
```

Figure 5: *the quickline function*

The algorithm works for the $0 \leq 2v \leq u$ case. Properties 1, 2 and 12 show how to adapt it to any line of the plane.

Both algorithms are given in a C-like language.

In practice the code of functions *line* and *line2* (see figures 6 and 7) is slightly different: to avoid multiple copies at the same location, the *cpt*≥ 0 tests (lines (1) and (2)) becomes *cpt*>0 and the case where *cpt*=0 is treated separately. In this case, the number of spans to write depends on the parity of *cpt*.

```
void line (u, v, long, short, slope) {
  incH = v * 2;
  incD = incH - u;
  delta = incD + v;
  slope [0] = long;
  slope [u - 1] = short;
  x = 1;
  cpt = (u - 2) / 4;
  for ( ; cpt≥0; cpt--) {      /* (1) */
    if (delta < 0) { /* case A */
      slope [x] = long;
      slope [u - ++x] = long;
      slope [x] = long;
      slope [u - ++x] = long;
      delta += incH;
    }
    else {
      if (delta < v) { /* case B */
        slope [x] = long;
        slope [u - ++x] = long;
        slope [x] = short;
        slope [u - ++x] = short;
      }
      else { /* case C */
        slope [x] = short;
        slope [u - ++x] = short;
        slope [x] = long;
        slope [u - ++x] = long;
      }
      delta += incD;
    }
  }
}
```

Figure 6: *first hexadecant*

6. Benchmarks

A software simulation was used to test the speed of this algorithm. As previously noted[6, 5] the simple software simulation is not particularly useful because in practice these functions would be realized at the chip level. Moreover the results depend on the quality of the code generated by the compiler. However the ratio between the different algorithms remains constant. As in other benchmark simulations[6, 5] the assumption of equal likelihood of all line segments within a large frame buffer is unrealistic but also inevitable.

Every line in the corresponding range is computed. For example with size=500, for all u belonging to [1, 500] and for all v in $[1, v/2]$, the (u, v) line is computed. Where the

```
void line2 (u, v, long, short, slope) {
  incH = v * 2;
  incD = incH - u;
  delta = incD + v;
  slope [0] = long;
  slope [u - 1] = short;
  x = 1;
  cpt = (u - 2) / 4;
  for ( ; cpt≥0; cpt--) {      /* (2) */
    if (delta < 0) { /* case A */
      slope [x] = short;
      slope [u - ++x] = short;
      slope [x] = short;
      slope [u - ++x] = short;
      delta += incH;
    }
    else {
      if (delta < v) { /* case B */
        slope [x] = short;
        slope [u - ++x] = short;
        slope [x] = long;
        slope [u - ++x] = long;
      }
      else { /* case C */
        slope [x] = long;
        slope [u - ++x] = long;
        slope [x] = short;
        slope [u - ++x] = short;
      }
      delta += incD;
    }
  }
}
```

Figure 7: *second hexadecant*

rectwrite function could not draw the range of pixels (diagonal or vertical spans), it was simulated. The total of computing time for all the lines of the range has been measured.

The benchmarks were realized on three computers: SGI Elan, SGI O2 and Digital Alpha 433. The values given in table 1 are the average of the three computer times obtained.

Finally the % value represents the ratio between the current algorithm and Bresenham's algorithm.

A large range of sizes has been tested. Since the drawing of lines does not occur only on CRT devices (where the maximum length remains 4096) but also on laser printers or slide plotters (where the number of dots is significantly greater) it was important to test extremely long lines.

Discussion: The gcd algorithm is not as good as expected. In fact, the time to compute the gcd is long and slows down the entire process. To solve this problem it would be easy to build an array of gcd values. If the lines are within a limited range the array size is also limited. Unfortunately, even within poor CRT device range (≤ 1024) the memory used would be 4 MB and would exceed this value for larger ranges. Therefore this solution can not be used. The slow down is mainly due to the fact that, even if the average value of the gcd is 5, there are few numbers with $gcd \neq 1$. Our tests showed that more than 60% of pairs of numbers (u, v) in a given range have a *gcd* of 1. Moreover even with an adaptation of Euclid's algorithm, the computing of the *gcd* value is very long when the *gcd* is 1. An array where the values would be the answer of the test "$gcd = 1$" would be a good compromise. We intend to make tests with a solution based on the concept of memo-functions for the *gcd*.

The span algorithm (see[17] or[16]) gives an average speed-up of 4.

The double-step technique[5] leads to an average speed up of 30%. Recall that in our concept, the double-step applies easily: each hexadecant is treated as the first hexadecant via property 12.

Finally, the resulting algorithm is proved faster than the previous ones. The speed-up increases proportionally to the length of lines to approximately 20 for very long lines.

This result may be compared to the global result of double-step and symmetry algorithm by Rokne et al.[5]. The authors get a speedup factor of roughly 3 over the original Bresenham version. Here this result is more than 6 times superseded.

7. Conclusion

A new algorithm for the scan-conversion of straight lines has been presented. It is proved to be at least 6 times faster than previous algorithms. We think that such an improvement should be implemented on hardware for any computer graphics device. Moreover the algorithm and the choices presented lead to a very fast anti-aliasing of lines. Two other improvements would speed-up slightly the results: the N-steps[7] or at least triple-step[8] techniques and the use of a Boolean array of "$gcd = 1$" values.

References

1. J.E. Bresenham. Algorithm for computer control of a digital plotter. *IBM System Journal*, 4(1):25–30, 1965.
2. H. Freeman. Computer processing of line drawing images. *ACM Computing Surveys*, 6(1):57–97, 1974.
3. R.F. Sproull. Using program transformations to derive line-drawing algorithms. *ACM Transactions on Graphics*, 1:259–273, 1982.
4. M.L.P. van Lierop, C.W.A.M. van Overveld, and H.M.M. van de Wetering. Line rasterization algorithms that satisfy the subset line property. *CVGIP*, 41:210–228, 1988.
5. J.G. Rokne, B. Wyvill, and X. Wu. Fast line scan-conversion. *ACM Transactions on Graphics*, 9(4):376–388, October 1990.
6. E. Angel and D. Morrison. Speeding up Bresenham's algorithm. *IEEE Computer Graphics & Applications*, 11:16–17, November 1991.

Sizes	Bresenham's	by Spans	gcd + Spans	id + double step	new algorithm
500	16,46	4,06	2,30	2,36	1,81
1000	132	31	17	15	10
2000	1053	248	135	107	66
4000	8433	1966	1058	787	451
6000	28458	6629	3575	2576	1432
10000	131923	30576	16394	11631	6279
%	100	23,2	12,4	8,8	4,8

Table 1: *Results of the computation time*

7. G. Gill. N-Step Incremental Straight-Line Algorithms. *IEEE Computer Graphics and Applications*, pages 66–72, May 1994.

8. P. Graham and S.S. Iyengar. Double and triple step incremental generation of lines. In *IEEE Computer Graphics & Application*, pages 49–53, May 1994.

9. Y.P. Kuzmin. Bresenham's line generation algorithm with built-in clipping. *Computer Graphics forum*, 14(5):275–280, 1995.

10. C.M.A. Castle and M.L.V. Pitteway. An application of Euclid's algorithm to drawing straight lines. In *Fundamental Algorithms in Computer Graphics*, pages 135–139. Springer-Verlag, 1985.

11. J. Berstel. *Mots, Mélanges offerts à MP. Schützenberger*, chapter Tracés de droites, fractions continues et morphismes itérés. Hermès, 1990.

12. J.P. Reveilles. Droites discrètes et fractions continues. Technical Report R90/01, ULP Département d'Informatique, Strasbourg, France, Janvier 1990.

13. A. Troesch. Interprétation géométrique de l'algorithme d'Euclide et reconnaissance de segments. *Theoretical Computer Science*, 115:291–319, 1993.

14. R. Brons. Linguistic methods for description of a straight line on a grid. *CGIP*, 3:48–62, 1974.

15. L. Wu. On the chain code of a line. *IEEE Transactions on Patterns analysis and machine intelligence*, PAMI-4(3):347–353, 1982.

16. J.E. Bresenham. Run length slice algorithm for incremental lines. In *Fundamental Algorithms in Computer Graphics*, pages 59–104. Springer-Verlag, 1985.

17. F.P. Chalopin and J.J. Bourdin. Straight lines: a step by step method. In *Winter School in Computer Graphics*, Plzen, feb 1996.

18. H. Freeman. Boundary encoding and processing. In *Picture Processing and Psychopictorics*, pages 241–266. B.S. Lipkin and A. Rosenfeld, Eds, New-York: Academic, 1970.

19. M.L.V. Pitteway. The relationship between Euclid's algorithm and run-length encoding. In *Fundamental Algorithms in Computer Graphics*, pages 105–112. Springer-Verlag, 1985.

20. T. Pavlidis. *Structural Pattern Recognition*. New-York: Springer-Verlag, 1977.

21. P.L. Gardner. Modifications of Bresenham's algorithm for displays. *IBM Tech. Disclosure Bull*, 18:1595–1596, 1975.

22. M. Luby. Grid geometries which preserve properties of the euclidean geometry: A study of graphics line drawing algorithms. In R.A. Earnshaw editor, editor, *Theoretical Foundations of Computer Graphics*, pages 397–432. Springer-Verlag, 1987.

23. V. Boyer, J. Tayeb, and J.-J. Bourdin. Une accélération du tracé de droites. In *I.A.C : Intelligence Artificielle et Complexité*, pages 142–149, February 1997.

24. M.L.V. Pitteway and A.J.R. Green. Bresenham's algorithm with run line coding shortcut. *The Computer Journal*, 25(1):114–115, 1982.

25. J.D. Foley, A. Van Dam, S. Feiner, and J. Hughes. *Computer Graphics, Principles and Practice*. Addison Wesley, 1990.