PARALLEL FIXED-POINT DIGITAL DIFFERENTIAL ANALYSER WITH
ANTIALLIASING. PFDDAA.∗

RAMÓN MOLLÁ VAYÁ† AND ROBERTO VIVO HERNANDO‡

Abstract. This paper introduces a quick, efficient and simple algorithm for drawing straight lines with anti-aliasing on discrete devices like printers or CRTs. It is based on the DDA algorithm. It uses 32 bits fixed-point arithmetic. This is a pipelined, parallelized and scalable SIMD version. It may be hardware implemented. It is suitable for MMX or Streaming SIMD PentiumIII instructions. The algorithm allows drawing true colour lines using a brush 3 pixels high and 1 pixel wide in parallel. The computing cost is very low and only integer arithmetic is used. This algorithm takes into account that the real width of the line remains constant always and it does not depend on its slope. Given that the total radiation emitted by a line is proportional to its real surface, total emitted light also increases with its slope.

Key words. Digital Differential Analyser, Line Drawing, Fixed Point Arithmetic, Parallelization, Graphic Processors, Anti-aliasing

1. Introduction. The straight line is the simplest 2D multipoint drawing primitive used in Computer Graphics. Many processes depend on this primitive’s efficient discretization and display like Page Description Languages (PDF2, PS3, PCL4,...) where the printer engine may draw text as polylines. Many CAD curve primitives are implemented also as polylines where speed is more important than fidelity, specially in real time applications. Drawing a line on a discrete device requires drawing a unit-height line segment (rectangle). The real line width increases as its slope does, so the segment surface is incremented (height by width). Supposing that the line has a uniform density of radiation (colour), the total amount of energy the line radiates is also increased when its slope does. The parallel algorithm presented in this paper can draw anti-aliased lines showing a good performance and assures the slope dependent radiation increment. It uses 32 bits fixed-point arithmetic Q15 [1] format instead of IEEE 754 floating point one. The following section surveys previous work on anti-aliasing line drawing algorithms. Section 3 presents some theoretical results that support our algorithm’s correctness. After that, we describe our algorithm in detail and provide an analysis of its computational complexity and the errors it incurs. Finally, the last section contains some concluding remarks and ideas for future work. At the end of the paper we provide the parallel software algorithm. This is a SIMD implementation. It may be implemented also using MMX, Streaming SIMD Pentium III instructions or AMD 3D-Now technology.

2. Previous work. The sequential algorithms for scan conversion of straight line segments to a frame buffer have been improved in the past to their limit. Nowadays, parallelization becomes the best solution for performance increase. Several methods to speed up Bresenham’s algorithm [2] have been tried using parallelization techniques [3][4], or by trying to take advantage of the repeated patterns that the algorithm generates [5] or by mixing both methods [6].

In the solutions given above, several hardware/software operators are proposed to work in parallel, but speed up can be constrained to three main problems in the drawing pipeline:

• The operation dependences graph avoids short steps. The operators are not simple. The pipeline is long.
• The average number of simultaneous active steps within an operator is reduced, and so, its efficiency. The pipeline has gaps.
• The ratio (amount of operators) / (speed up) is not linear. So the hardware costs can be prohibitive if speed up is high due to the amount of operators required. Many pipelines are idle in a given moment.

---

∗THIS WORK IS PARTIALLY SUPPORTED BY THE CICYT PROJECT TIC1999-0510-C02-01
†Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia. Camino de Vera, s/n. 46022 Valencia (SPAIN) Phone: ++34-6-387 7000 ext. 3549 Fax: ++34-6-387 7359 (rmolla@dsic.upv.es)
‡Phone: ++34-6-387 7795 Fax: ++34-6-387 7359 (rvivo@dsic.upv.es)
1Pentium is a trade mark from Intel Corporation
2PDF is a trademark from Adobe Systems Inc.
3PostScript is a trademark from Adobe Systems Inc.
4PCL is a trademark from Hewlett Packard Inc.
An incremental algorithm such as Digital Differential Analyzer (D.D.A.) has been traditionally inadequate for hardware implementation because it uses floating point arithmetic. Using fixed point instead of floating point arithmetic [7], diminishes the hardware and timing cost. A parallel SIMD version [8], increments performance greatly. This paper adds antialiasing capabilities to this algorithm. Two aliasing problems arise when lines are drawn on a screen, one is vertical and the other one is horizontal. Assuming without losing generality, that the slope of the line belongs to $[0,1]$ interval, the most evident antialiasing effect is seen in the horizontal stepping of the lines. However, vertical aliasing also occurs, especially at the ends of the line. Given that this defect occurs in an insignificant number of points with respect to the overall length of the line, its effect passes unnoticed and is dealt with in the literature on very few occasions [9] [10]. Gupta and Sproull [9] used an antialiasing technique based on pre-computation of the sub-volumes of the filtering function in a table, which is later used to smooth the edges of each pixel during the drawing phase. An area of two pixels diameter is used. This is translated as a three pixels vertical brush that sweeps the whole line. Typically a line tends to intersect a three pixels brush (north, centre, and south). The brush may be divided into 16 subareas (4x4) with any distribution of intensities (hemispheric, hyperbolic, non-weighted,...) [11]. Although the results are very good, the spatial and computing costs can increase greatly. Blinn’s article [12] is especially useful in this sense where he presents an approximation of sub-sampling using integer scaled numbers where 5 and 16 bits represent the decimal part. In other cases, antialiasing is obtained by estimating the amount of real image covered by each pixel, the intensity of the pixel being linearly dependent on the amount of image intersectioned [13]. Another version [14] starts with an algorithm to represent generic conics and ends with a line primitive very similar to that of Gupta and Sproull [9]. When working with reduced pixel colour depth screens, other improved algorithms appear [15] at the cost of loss of generality. However, as pixel colour depth increases this algorithm loses efficiency, making differences from reality perceptible. When working with conics and cubic curves, the distance from the point to the curve is used as an antialiasing weighted value. The approach is not novel except that it uses a very efficient algorithm to determine the proximity of the curve incrementally [16]. There are more general solutions for arbitrary curves and filters using pre-filtering [17]. Others are neither so versatile nor so efficient [18] and they can only work with circles and ellipses parallel to the axes and with non-weighted filtering. Other approximations attempt to draw the curves as sequences of very short segments and apply already known efficient antialiasing polygon methods [19]. Knowing previously that the image to filter is a straight line, it may be used an edge detector and a simple convolution filter algorithm to reduce jagging. These algorithms tend to be used in situations where response time must be extremely quick (video-games, high-speed laser printers, virtual reality, multimedia, etc.), where speed is more valued than accuracy [20]. Other attempts [21], draw the line in successive approximations moving the line vertically in different consecutive sweeps. The filtering of each segment is carried out during the drawing of each straight line. If all possible lines in a given NxN screen are grouped into a N entry table, the line can be drawn using antialiasing techniques, multiple run-lengths and repeated patterns precomputed and stored in that table [22]. These repeated patterns allow the algorithm to be parallelised easily. Oversampling is another way of avoiding high frequency components in images as it can be understood as a low-pass filter. Depending on how the distribution of this oversampling is made, different results are obtained [23].

The aim of this work is to obtain an efficient parallel algorithm for drawing straight lines using antialiasing techniques, following the real energetic model, which can efficiently be implemented by hardware. It uses decimal numbers and incremental techniques to represent the primitives in an efficient and photo-realistic way. This new algorithm has been called PFDDAA. Basically it is an energetic and pipelined parallelized SIMD modification of FPDDA algorithm [7] suitable for hardware implementation.

3. Theoretical bases. The surface of a 2D object (lines, squares, circles, etc.) can be divided into small pieces called unitary surfaces (u.s.). In this paper, a u.s. has the same surface area and shape that a pixel on a discrete peripheral. Given a screen pixel $P$, its intensity of radiation depends on the sum of all objects radiation projected onto this pixel including the intensity of the background by default. When an object is projected on a discrete screen, if a pixel is completely fulfilled by the
object, the intensity it radiates is the maximum that the object can give, $I_{r_{\text{max}}}$. When a straight line is drawn on a discrete screen, the u.s. of the line do not coincide exactly with the pixels of the screen, except when the line has a zero or infinite slope. The intensity of each pixel will depend on the $I_{r_{\text{max}}}$ of the line to be drawn and the pixel surface intersected by the line.

Due to the line’s slope, the line end pixels are skewed in relation to the screen. This rotation causes the formation of triangles "$S_i$" beyond the pixel borders, which produce similar holes within the line end pixel. See Fig. 3.1. Thus, $S_i = S_j \ i,j \in [1..4]$, for the u.s. $A$ and $C$. Clearly the surface to be irradiated beyond the pixel boundaries is equal to the non-irradiated surface within it. Trigonometrically it can be demonstrated that any $S_i$ surface has the following value, depending on the line angle $S_i = \frac{1}{2} \frac{(\sin(a) + \cos(a) - 1)^2}{\sin(a) \cos(a)}$. The maximum value is achieved at 45 degrees ($S_i = 4\%$ S). From 45 degrees to 90 degrees, the line’s behaviour is symmetrical. Without loss of generality, the starting line point is the left end and its neighbouring pixels (North, West and South) have a total illuminated area of $S_1$, $S_2$ and $S_3$ respectively. $I_s = I_r S_i$. See Fig. 3.1. The radiant value of the central pixel will be that of the line at that point minus the holes left by $S_1$, $S_2$ and $S_3$. Thus, $I_s = I_r S = I_r (1 - S_1 - S_2 - S_3) = I_r (1 - 3S_i)$. In the worst case, $I_s = 87.5\%$ of $I_r$. If the line length is greater than one pixel, the pixel East forms part of the next pixel to be drawn. Furthermore, the hole $H_4$ missing from the first pixel due to the triangle $S_4$, is also filled by the next pixel East. Thus $I_s = I_r (1 - 3S) \not= I_r (1 - 4S)$. Moreover, between $H_4$ and $S_1$ exists the hole $H_5$ which is also fulfilled by the next pixel. Clearly the surface value of the initial pixel and its neighbours is at least $H_5$ greater than one unit. Going on the line to the next step towards the east, the middle points appear. Every middle point is drawn using a brush of only three pixels: North, Centre and South.

Let us assume a continuous line segment that starts on $P_0 = (X_0, Y_0)$ and ends on $P_e = (X_e, Y_e)$, where the segment width is $A_x = X_e - X_0$ and the segment height is $A_y = Y_e - Y_0$. Let us suppose, without loss of generality, that this line has a slope $0.0 \leq m = A_y / A_x \leq 1.0$. The actual amount of the line’s u.s. ($L$) is
In practice, the amount of pixels (P) used to draw it on a discretised screen can be significantly less than L. The number of pixels actually used will never be greater than \( P = \max(A_x, A_y) = A_x \leq L \) assuming that \( |m| \leq 1 \). The proportionality constant \( K = L/P = A_x \sqrt{1 + m^2}/A_x = \frac{1}{\cos(a)} \) determines the relationship between the real line length (L) and the discrete line length drawn on the bitmap (P). Thus, on average, each pixel of the discrete line must represent at least K real line u.s. For example, in the illustration 3.2, \( P = \max(A_x, A_y) = A_x = 20 \leq L = \sqrt{100^2 + 200^2} = 22.36 \) being K = L/P = 22.36/20 = 1.12. Since P cannot be physically increased, the increment of radiating line surface must be compensated by an increment of K units in line’s thickness. That is to say, as the line slope increases, the vertical brush radiance is also increased by K times the line’s \( I_{\max} \).

The height of the vertical brush H used to sweep the line, increases proportionally to the inverse cosine of the line slope. That is to say, \( H = \frac{G}{\cos(a)} = G.K \). The algorithm maintains the width of the line G perpendicular to its longitudinal axis independently of its slope. See 3.3. For this reason, the line brightness remains constant although its slope changes.

Fig. 3.4 shows this increase in the length of the line (L) depending on its slope. This graphic represents the real line length increment when its width (A_x) remains constant and its height (A_y) is increased until its slope is one. Since the computation of the function is complex, an approximation based on the McLaurin series has been used with acceptable results. The upper line in Fig. 3.4 corresponds to the function as calculated using the McLaurin series, \( L = 1 + \frac{m^2}{2} \), and taking into account only two terms; while the lower curve is the real function. For any brush on the line, the average total intensity of the pixels South, Centre and North must always equal \( I_r.(1 + m^2/2) \). This is called Total Brush Radiance (TBR). The greatest error, 8% compared to the real function, was deemed acceptable. Notice that no point on the discrete line may have its intensity greater than the maximum intensity of the real line (MaxColor).

4. Software algorithm. Since a very precise calculation is computationally costly, some approximations have been used to generate very similar results at a lower cost. The brush is rectangular,
vertical, one pixel wide and 3 pixels high. See Fig. 3.3. The function $\sqrt{1 + m^2}$ is calculated using McLaurin series developed until their second term. If more precision is required, the series can be developed on. The highest intensity contrast between South or West pixels and Centre achieves a proportion of 1 to 25 (4%). These pixels cannot be differentiated by the human eye from those achieved using purist algorithms based on brute force, specially in a rich graphic real time environment. In practice, these pixels are not calculated (vertical aliasing) in order to save computational effort. A fixed-point numerical format has been used with 16 bits for the decimal part and 16 for the integer part, including the sign bit. This numeric format will be referred from now on as Q15 format [1].

5. Implementation. Without loss of generality the following facts are assumed: the line slope is always in the range $[0, 1]$, the line starts at bottom left ($X_0, Y_0$) and ends at top right ($X_f, Y_f$), a grey-toned raster is used and the origin of coordinates is placed at the bottom left corner of the screen. A block diagram of this algorithm is given in the Fig. 5.2.

5.1. Initialization phase. Suppose that we have sampled a line point $P_i(x_i, y_i)$ and we want to draw the next $n$ consecutive line points. Assuming without loss of generality that line slope belongs to $[-1, 1]$, the next $n$ line points can be calculated by this way $P_{i+1} = (X_i + 1, Y_i + m), P_{i+2} = (X_i + 2, Y_i + 2 * m) ... P_{i+n} = (X_i + n, Y_i + n * m)$

The starting position $(X, Y)$ of all the brushes is calculated in parallel using $n$ adders. The algorithm’s first task is to calculate the line’s slope. This slope could be calculated during the previous clipping phase. This phase calculates the line illumination increment for every consecutive pixel (ColorInc). The algorithm works with $N$ parallel F.D.D.A.A. operators. Each operator has its own private three pixels brush ($N[i], C[i]$ and $S[i]$). During this phase, the algorithm has to calculate the X position ($X_p[i]$), the Y position ($Y_N[i], Y_C[i]$ and $Y_S[i]$) and the initial intensity of each pixel of every brush. The calculation of all the $i*m (m[i])$ products may be implemented by a set of adders and wired shifters. This circuit is called AMC (Accelerated Multiplication Circuit) [7]. If $k = log_2(n)$, this multiplier can be accomplished using less than $2^{k-1}$ adders: $n/2$ adders in the worst case. Temporal cost is $k-2$ integer additions if $k \geq 4$, $k-1$ for $3 \geq k \geq 2$, and 0 for $k = 1$. An example for $n=8$ operators is given in the following figure, where the $k = log_2(n) = log_2(8) = 3$, the temporal cost is $k-1 = 3-1 = 2$ additions and the hardware cost $C$ is $C = 3 \leq 2^{k-1} = 2^2 = 4$ adders. Without hardware restrictions and assuming a screen maximum line length, this phase could draw a whole line with a timing cost $O(\log)$.

The algorithm calculates the brush intensity distribution for the first operator and the increment of radiance for the line depending on its slope assigning the maximum brightness to the South pixel, the radiance increment to the Center pixel and null to the North pixel. The intensity distribution of the other brushes may be calculated in parallel although they have been obtained sequentially in the program listing. Once all the brushes are positioned and calculated, they are sent to the raster, one by one. During this phase, the algorithm calculates the increment of intensity for the brushes (ColorIncLoop) when the line is shifted up in the loop phase in each iteration.
5.2. Loop phase. In this phase, \( n \) F.D.D.A.A. operators work in parallel to obtain the next \( n \) consecutive line pixels. So the operator \( j \) would have to add \( n \) to \( X_i \), multiply \( n \) by \( m \) and add \( n*m \) to \( Y_i \). When the last point sent to video memory is detected, the Loop Phase is finished and another line drawing can be achieved. This phase is composed of two main concurrent blocks that work in a pipelined fashion: the F.P.D.D.A. operators and the Queue Manager.

**F.D.D.A.A. Operators (F.O.).** If there are \( j \) operators, everyone has to add \( j \) to \( X_i \) and \( j*m \) to \( Y_i \) in order to obtain the south (S) pixel screen coordinates for the \( i \)-th brush, redistribute illumination within the \( i \)-th brush through the \( S \), \( C \) and \( N \) pixels and draw the brush in its correct raster position \((X_i,Y_i)\). This \( i \)-th phase has three input data: the line slope increment \((\text{mIncLoop})\) and the intensity increment \((\text{ColorIncLoop})\), the brush \( X_{i-1} \) (integer) and \( Y_{i-1} \) (decimal) coordinates and the brush intensity distribution: \( S_{i-1}, C_{i-1} \) and \( N_{i-1} \). For each iteration, each brush must move \( \text{mIncLoop} \) units up and \( j \) units to the right. For this reason, the brilliance of each pixel increases or decreases \( \text{ColorIncLoop} \) units.

Every F.D.D.A.A. operator is a compound of two suboperators called X-operator and Y-operator. The former calculates the pixel \( X \) coordinates and the latter the \( Y \) ones. The \( i \)-th X-operator uses an adder to increase \( X \) in \( j \) pixels. Once this new coordinate has been obtained, it is immediately compared to the last point \( X \) coordinate \((X_e)\). The X-operator performs one addition of \( k \) bits, one \( k \) bits comparison and a register load, where \( k = 16 \) in a typical application. The \( i \)-th Y-operator adds \( \text{mIncLoop} \) to \( Y \) to obtain \( Y + j*m \). This operation costs one addition of \( 2k+1 \) bits. Only the integer part is used to provide the \( Y \) pixel coordinate on the discrete screen. For each iteration and for each operator, the increment of intensity \((\text{ColorIncLoop})\) is subtracted from the South pixel of the brush. If the result is negative, an underflow has to be corrected adding \( \text{MaxColor} \). In order to maintain the TBR constant for the brush, the South pixel radiance is subtracted from TBR and assigned to the Center pixel. If this amount is higher than the maximum intensity allowed for a pixel \((\text{MaxColor})\), an overflow has to be corrected. The light excess, if any, is assigned to the North pixel, otherwise this point is not sent to the raster. If there is an excess, the Center pixel is saturated to \( \text{MaxColor} \). If one of the status bits is set, then the FDDAA Operators phase is over and if there is available data, another line starts to be drawn while the Queue Manager finishes to extract the last line final points to the video memory.

5.2.1. **Queue Manager (Q.M.).** When the control circuits detect that the new coordinates have been calculated, a register load signal is activated. An \( n \) state machine begins to extract these coordinates to video memory. When the status bit of an X-register is set, it means this is the last point to draw. So this point will be sent to video memory and the queue manager loop will be finished. When the new calculations are finished and the previous line pixels have been sent to memory, a load of intermediate registers is ordered again and both blocks can go on. In the program listing, this serialisation is performed implicitly by the \texttt{DrawBrushSC} and \texttt{DrawBrushSCN} functions.

5.3. **Computational cost.** This section will analyze how many operations must be carried out to draw a straight line using fixed-point and antialiasing. The cost will be compared with that of other classic algorithms such as Gupta-Sproull. Note also that the drawing loop control calculations
have also been included in the computational cost. The table below shows a comparison of the exact computational cost of the initialization phase and main loop of PFDDAA as well as Gupta-Sproull where

- \( D = \text{Unitary displacement} \) // \( C = \text{Comparison} \)
- \( \text{Di} = \text{Integer division} \) // \( P = \text{Integer product} \)
- \( \text{Pr} = \text{Floating point product} \) // \( S = \text{Integer addition} \)
- \( \text{Sr} = \text{Floating point addition} \) // \( R = \text{Square root} \)
- \( \text{Ab} = \text{Absolute value} \) // \( V = \text{Array access} \)

Assuming that the cost of carrying out a sum of 16 or 32 bits is equivalent with current technology, the assignments are virtually cost-free, a comparison between two non-zero values has a cost of one addition, the two 16 bits words that make a register of 32 can be independently directed, or that the CPU used supports Q15 numerical format and the average line’s slope is 0.5, then the computational cost would be the following:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Initialisation</th>
<th>Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFDDAA</td>
<td>( 2D + 1\text{Di} + 3P + (n+8.5)S )</td>
<td>( 5.5S/n )</td>
</tr>
<tr>
<td>Gupta-Sproull</td>
<td>( 4S + 2D + 2\text{Sr} + 2I + 6\text{Pr} )</td>
<td>( 1\frac{3}{2}I + 5S + 3\text{Pr} + 3\text{Ab} )</td>
</tr>
<tr>
<td>( (G_S) )</td>
<td>+ 1R + \text{Ab}</td>
<td>+ 5\text{Sr}</td>
</tr>
<tr>
<td>( G_S - \text{PFDDAA} )</td>
<td>( (n+4.5)S - \text{Di} - 3P )</td>
<td>( 3.5S + 1.5I )</td>
</tr>
</tbody>
</table>

In the previous table, it is assumed that the PFDDAA algorithm has 4 parallel operators without loss of generality. Notice that this is a temporary cost assuming that all operations without dependencies between them can be done in parallel, as shown in the program listing. This could be the case of an ASIC, where many operations would be calculated at the same time incurring in a lower time response. Notice that the line slope could be calculated in a previous clipping phase.

In this case, the algorithm initialisation cost would be even lower. Avoiding the pixels scan writing timing costs, the FDDAA sequential version speeds up the Gupta-Sproull algorithm 666% both on Pentium II and III CPUs and more than 1000% on AMD K6 CPU, no matter the clock frequency or model. This algorithm uses a SIMD architecture, thus it may be easily implemented using MMX, Streaming SIMD or 3D Now instructions. Many parallel operations have to be done sequentially in the software simulations, since actually they are run on one single processor. The main loop may be improved calculating up to four pixels at a time when using MMX technology. Several tests have to be done in order to avoid line end overide or colour under or overflow. Even though, speed up improvements higher than 25% can be achieved using MMX technologies.

6. Conclusion. We have presented an algorithm that maintains the line’s radiance independently of its slope. It does not require complex computational operations such as square roots. All the operations use integer or fixed point arithmetic. The algorithm has a low computational cost in both initialization and loop phase. It is implementable in hardware. This algorithm mixes a quick and efficient algorithm like DDA with fixed-point arithmetic. It provides a smooth antialiasing result not visually distinguishable from traditional algorithms like Gupta-Sproull. The simplifications taken produce no noticeable effects on the screen. It optimizes the amount of pixels sent to the raster because it only writes 2 or 3 pixels when required. Other algorithms, like Gupta-Sproull, always write 3 pixels. This algorithm uses separated phases, so it can be pipelined easily. It may support decimal subpixel or multipixel line thickness in the interval \([0, \sqrt{2}]\) with very little source code modifications.

7. Algorithm. The following code introduces the fixed point arithmetic data type, the variables and arrays used by the SIMD operators, some functions and the algorithm function PFDDAA.

typedef union
  {
    long V;          //32 bits integer
    struct {
      unsigned int Dec; //Two 16 bits integers.
      int Int;          //The 16 msbs represent the integer part
    } FP;             //The 16 lsbs represent the decimal part
  };
#define TOTAL_OPERATORS 8
inline void DrawBrushSCN (int Operator)
{
    putpixel (X(Operator), YN(Operator), N(Operator).Int);
    DrawBrushSC (Operator);
}

inline void DrawBrushSC (int Operator)
{
    putpixel (X(Operator), YC(Operator), C(Operator).Int);
    putpixel (X(Operator), YS(Operator).Int, S(Operator).Int);
}

void AMC () //Designed for TOTAL_OPERATORS = 8
{
    m[1].V = slope.V ; //1.slope
    m[2].V = slope.V <<1; //2.slope
    m[4].V = m[2].V <<1; //4.slope
    m[5].V = m[4].V + slope.V ; //5.slope
    m[6].V = m[3].V <<1; //6.slope
    m[7].V = m[6].V + slope.V ; //7.slope
    mIncLoop.V = m[4].V <<1; //8.slope
}

void PFDDAA (int X0, int Y0, int Xf, int Yf, unsigned char Gray)
{
    int Ax;
    FP Ay, MaxColor, TBR, //Total Brush Radiance
        ColorInc, ColorIncLoop; //Color Increment during Init. and loop phase
    IN_PARALLEL //General variables initialisation
    {
        Ax = Xf - X0; //Line width
        Ay.Int= Yf - Y0; //Line heigth
        Ay.Dec= 0;
    }
    slope.V = Ay.V / Ax;
    AMC (); //Advanced Multiplication Circuit. Slope array initialisation
    IN_PARALLEL
    {
        MaxColor.Int= Gray;
        MaxColor.Dec= 0;
        ColorInc.V = slope.V * Gray;
        ColorIncLoop.V = mIncLoop.Dec * Gray;
    }
    //First operator initialisation
    C[0].V = slope.V >>1; //Up = Gray*(slope*slope) / 2.0 in FP format
    C[0].V *= C[0].V ; //It is assumed that the slope is always positive
    C[0].V = (C[0].V <<1) + CERO_FIVE;
    C[0].Dec= C[0].Int; //Equivalent to Center[0].V >>16
    C[0].Int= 0;
    C[0].V *= Gray;
    IN_PARALLEL
    {
        S[0].V = MaxColor.V; N[0].V = 0;
        TBR.V = MaxColor.V + C[0].V ; //S + C + N
        Xp[0]= X0; YS[0].Int= Y0;
        YS[0].Dec= 0; YC[0]= Y0 + 1;
    }
    IN_PARALLEL
    {
        DrawBrushSC (0);
        Parfor(i=0;i<TOTAL_OPERATORS;i++)
        {
            IN_PARALLEL { Xp[i] = X0 + i; YS[i].V = YS[0].V +m[i].V;}
        }
    }
IN_PARALLEL { YC[i] = YS[i].Int + 1; YN[i] = YS[i].Int + 2; }

Parfor(i=1;i<TOTAL_OPERATORS;i++)
{
    //Brush colour calculation
    S[i].V = S[i-1].V - ColorInc.V;
    if (S[i].V <= 0) S[i].V += MaxColor.V;
    C[i].V = TBR.V - S[i].V;
    N[i].V = C[i].V - MaxColor.V;
    if (N[i].V <= 0) DrawBrushSC (i);
    else { C[i].V = MaxColor.V; DrawBrushSCN (i); }
}

do
{
    Parfor(i=0;i<TOTAL_OPERATORS;i++)
    {
        IN_PARALLEL { Xp[i] += TOTAL_OPERATORS; }
        IN_PARALLEL { YS[i].V += mIncLoop.V; S[i].V -= ColorIncLoop.V; }
        if(S[i].V <= 0) S[i].V += MaxColor.V;
        C[i].V = TBR.V - S[i].V;
        N[i].V = C[i].V - MaxColor.V;
        if (N[i].V <= 0) DrawBrushSC (i);
        else { C[i].V = MaxColor.V; DrawBrushSCN (i); }
    }
}
while (true);

}//Function end

REFERENCES


Edited by: Marta Fairén and Xavier Pueyo
Received: October 5th, 1999
Accepted: May 20th, 2000