TWO ALGORITHMS FOR DRAWING ANTI-ALIASED LINES

Dan Field

Computer Graphics Laboratory
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
(519) 886-1351

ABSTRACT

Two new algorithms are presented for drawing anti-aliased lines. The algorithms do not require floating point or table lookup operations within their inner loops. When rendering against a constant background, inner loop multiplications can also be eliminated. These qualities make the algorithms attractive for hardware and firmware implementations. Inner loop operation counts show the algorithms to be faster than others reported in the literature. Anti-aliasing is accomplished using a square area integration filter; one algorithm approximates filter response, the other computes response values exactly.

On présente deux algorithmes pour le tracé de courbes décrénelées. Les boucles itératives de ces algorithmes ne font appel ni à la virgule flottante ni à des opérations de référence a des tables de conversion. Les cas dans lesquels l'arrière-plan est uniforme peuvent être traités en éliminant les multiplications l'intérieur des boucles d'itération. Ces qualités sont particulièrement attrayantes pour l'incorporation de ces algorithmes à des matériels ou des microprogrammes. Le comptage du nombre d'exécution des boucles d'itération montre que ces algorithmes sont plus rapides que ceux décrits dans la litérature. Le dérénelage (anti-aliasing) fait intervenir un filtre d'intégration par surfaces carrées; un algorithme donne une approximation de la réponse du filtre et l'autre calcule les valeurs exactes.

Key Words: Aliasing, anti-aliasing, filtering, algorithms

1. Introduction

Anti-aliasing algorithms operate by replacing previous pixel values according to the *blending function*

$$pixel(i,j) \leftarrow \alpha I + (1-\alpha) I_{back} \qquad (1)$$

where I is the color† of the object being drawn, I_{back} is the previous color of pixel (i,j), and α is the response of a low pass filter to the object at (i,j). Normally, $0 \leq \alpha \leq 1$. Typically, the filter function is computationally expensive and is stored in a lookup table indexed by the distance from the object to (i,j). Thus, for each pixel modified by the algorithm, table lookup, multiplication, and normalization operations are required within the inner loop steps.

Values for α can be determined directly when the filter function is computationally inexpensive. Pitteway and Watkinson [Pitt80] show how the remainder value of Bresenham's algorithm [Bres65] may be used to obtain the response of a square area integration filter. However, their algorithm uses floating point addition, subtraction, and multiplication operations within the inner loop steps, and makes poor approximations of α in certain cases.

This work was supported in part by the NSF under grant MCS81-14207.

†To simplify presentation of the algorithms, we shall assume that we are using a grayscale raster device. Later, we describe how to generalize to full *rgb* color.

In this paper, we present two algorithms for drawing anti-aliased lines that compute αI directly without lookup tables or floating point operations. The first algorithm uses approximation techniques to compute filter response values. In some applications, it is important that the filter be computed exactly. The second algorithm achieves this using results from elementary number theory. With a constant background, inner loop multiplications can be eliminated entirely resulting in algorithms that are especially attractive for hardware or firmware implementations. We show that the approximation algorithm is faster than the table lookup algorithms of Gupta and Sproull [Gupt81] and Turkowski [Turk82]. With non-uniform backgrounds, we resort to multiplication within the inner loop, while remaining faster than [Gupt81] and [Turk82]. The algorithms presented here are an extension of the ideas contained in [Pitt80].

The remainder of this section describes the fundamentals of computing the filter values α. The next two sections describe approximation and exact computation schemes for α. Section four contains implementation details about accuracy, conversion from rational to integral arithmetic, and word length characteristics. In Section five we describe some extensions to the algorithms. We conclude in Section six with an analysis of the algorithms.

1.1. Fundamentals

We shall assume that a raster device is modeled by a square grid of integral points (i,j) called *pixels*. We define an *el* (i,j) in the grid to be the unit area region (x,y) where $i \le x \le i+1$ and $j \le y \le j+1$. The filter response α for pixel (i,j) is the area of intersection between the line and the el (i,j).

Consider the problem of drawing a line segment of color I between the points (0,0) and (dx,dy). It is assumed here that the segment lies within the first octant, implying $dy \le dx$. We represent the segment by a parallelogram defined by the points $(0,1/2)$, $(dx,dy+1/2)$, $(0,-1/2)$, and $(dx,dy-1/2)$.

Drawing a parallelogram using our anti-aliasing technique involves tracing the *upper bounding segment* (UBS), $(0,1/2)$, $(dx,dy+1/2)$; tracing the *lower bounding segment* (LBS), $(0,-1/2)$, $(dx,dy-1/2)$; and computing the parallelogram-el intersection areas. It is enough to trace only UBS since LBS always lies one vertical grid unit below. This is achieved by a variant of Bresenham's algorithm. Computing intersection areas is achieved by exploiting the linear increase in area between UBS and the line $y=0$ as we move from left to right along the x axis.

The vertical position of UBS at $x_i = i$ is given by the equation

$$1/2 + \frac{dy}{dx} i = q_i + \frac{r_i}{2dx} \tag{2}$$

Here, q_i and r_i are integral and $0 \le r_i < 2dx$. The quantity q_i represents the integral height of UBS above the x axis, and the quantity $\frac{r_i}{2dx}$ is the fractional height above $y = q_i$. When $i=0$, we have $q_0 = 0$ and $r_0 = dx$. Simple recurrence formulas for q_i and r_i follow directly.

$$\begin{aligned} q_{i+1} &= q_i & r_i &< 2dx - 2dy \\ q_{i+1} &= q_i + 1 & 2dx - 2dy &\le r_i \end{aligned} \tag{3}$$

and

$$\begin{aligned} r_{i+1} &= r_i + 2dy & r_i &< 2dx - 2dy \\ r_{i+1} &= r_i + 2dy - 2dx & 2dx - 2dy &\le r_i \end{aligned} \tag{4}$$

Thus, tracing the parallelogram is accomplished by generating the values q_i and r_i for the upper segment. The lower segment, LBS, has remainder values r_i identical with the upper segment; its integral vertical heights are $q_i - 1$. The two remaining edges of the parallelogram, $(0,1/2)$, $(0,-1/2)$ and $(dx,dy+1/2)$, $(dx,dy-1/2)$, having unit length and being vertical, are traced implicitly by starting the upper segment trace at $x=0$ and finishing at $x=dx$.

Intersection areas for els are determined on a column-by-column basis as UBS is traced. Since $dy \le dx$, and UBS is separated from LBS by a vertical distance of 1 grid unit, the parallelogram intersects either two or three els in any column (See Fig. 1). The two cases are distinguished by the value of r_i. If $r_i \le 2dx - 2dy$, then two els are intersected by the parallelogram in column i. Otherwise, $2dx - 2dy < r_i$, and the parallelogram intersects three els in column i. In both cases, the total intersection area in column i is 1.

A counter A_i is used to keep track of the areas for the upper one or two els intersected by the parallelogram in column i. When the parallelogram intersects two els (i,q_i) and (i,q_i-1), A_i is the intersection area contained in (i,q_i). The intersection area contained in (i,q_i-1) is therefore $1-A_i$ (See Fig. 2).

When the parallelogram intersects three els, (i,q_i+1), (i,q_i), and (i,q_i-1), A_i is defined to be the trapezoidal area bounded by the lines UBS, $x=i$, $x=i+1$, and $y=q_i$. If B_i is the intersection area contained in (i,q_i+1), then $A_i - 2B_i$ is the intersection area contained in (i,q_i), and $1-A_i+B_i$ is the intersection area contained in (i,q_i-1) (See Fig. 3).

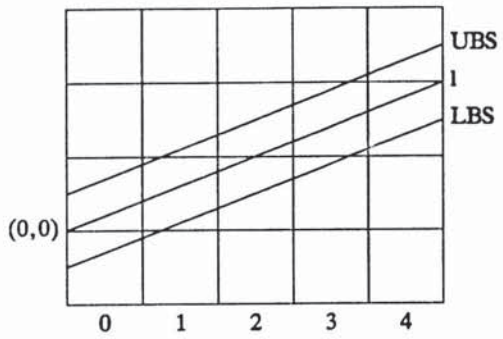

Fig. 1. The parallelogram crosses two els in columns 0, 2, and 4, and three els in columns 1 and 3.

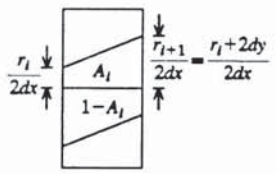

Fig. 2. Intersection areas for two els crossed by the parallelogram in column i.

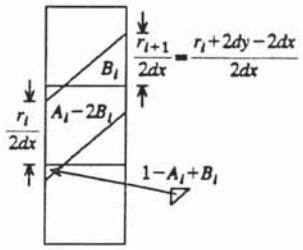

Fig. 3. Intersection areas for three els crossed by the parallelogram in column i.

We are left with specifying the computation of A_i and B_i. It can be seen from the previous discussion that the values for A_i and B_i are determined solely by the remainder r_i and are defined by the following equations.

$$A_i = \frac{r_i + dy}{2dx} \tag{5}$$

and

$$B_i = \frac{(r_i + 2dy - 2dx)^2}{8dxdy} \tag{6}$$

Simple recurrence formulas can be employed to iteratively compute A_i. Initially,

$$A_0 = \frac{dx + dy}{2dx} \tag{7}$$

The recurrence formula for A_i is

$$\begin{aligned} A_{i+1} &= A_i + \frac{dy}{dx} && r_i < 2dx - 2dy \\ A_{i+1} &= A_i + \frac{dy - dx}{dx} && 2dx - 2dy \le r_i \end{aligned} \tag{8}$$

Unfortunately, B_i is a quadratic function of the non-monotonic parameter r_i, and does not have a simple recurrence formula. In the next two sections we investigate an approximation scheme for B_i and an alternate approach in which exact values for B_i may be computed.

2. Approximating B_i

There are three properties that we would like of an approximation B_i' for B_i.

1) B_i' can be computed without multiplication or division.
2) B_i' is as accurate as possible.
3) $B_i' = B_i = 0$ when $r_i = 2dx - 2dy$.

The reasoning behind the first two properties is straightforward. The third property allows the combination of some special cases. We have seen that q_{i+1} and r_{i+1} have different iterative calculations when $r_i < 2dx - 2dy$ and $2dx - 2dy \le r_i$. The parallelogram intersects three els only when $2dx - 2dy$ is strictly less than r_i. Therefore, three different actions are necessary when r_i is less than, equal to, or greater than, $2dx - 2dy$. We can consider the parallelogram as crossing three els instead of two at $r_i = 2dx - 2dy$. The intersection areas for the two cases will be equivalent only if $B_i' = 0$ at this point. The third property allows the combination of the cases $r_i = 2dx - 2dy$ and $2dx - 2dy < r_i$, saving extra conditional and branch operations.

We shall use the approximation

$$B_i' = 1/2\left(A_i + \frac{dy}{2dx} - 1\right) \tag{9}$$

Since this can be evaluated with an addition, subtraction, and shift, the first property is satisfied. When $r_i = 2dx - 2dy$, the value of A_i is $1 - \frac{dy}{2dx}$. Therefore, the third property is satisfied.

What about other choices for B_i'? Combining Eqs. (4), (5), and (9), we see that $B_i' = 1/2\frac{r_{i+1}}{2dx}$. The third property forces any approximation for B_i to take the form $c\frac{r_{i+1}}{2dx}$ for some constant c. Since the first property precludes the use of multiplication and division, we are forced to use shifts and adds to obtain a suitable value for c. As $B_i' \ge B_i$, the next logical choice for c is $1/4$. However, $\max_i |B_i - B_i'| \le \max_i |B_i - 1/2B_i'|$, that is, the maximum error for $c = 1/2$ is less than that when $c = 1/4$. While it is possible to obtain constants between $1/4$ and $1/2$, extra shifts and adds are required. Therefore, we use $c = 1/2$ for the approximation.

The error in the approximation is

$$B_i - B_i' = \frac{r_i^2 + 2r_i(dy - 2dx) + 4dx^2 - 4dxdy}{8dxdy} \le 0 \tag{10}$$

and is greatest when $r_i = 2dx - dy$. The error is small enough to ensure a smooth transition in the intersection areas along horizontal rows of els. In practice, the lines drawn with B_i' appear as good as those drawn with the exact value of B_i. See Plates 1 through 5 for examples.

3. Exact Calculation of B_i

We have seen that B_i is a quadratic function of the non-monotone parameter r_i. We shall cause the remainder values to increase monotonically by permuting the order along the x axis in which UBS is traced. This allows exact calculation of B_i through forward differencing techniques.

The different trace order invalidates the formulas developed for q_i, A_i, and B_i. In what follows, we give new formulas for A_i and B_i and redefine the trace parameters x_i' and q_i'.

Assume that $gcd(dx, dy) = 1$ (this restriction will be removed later). If dx is even, the remainders r_i form a permutation of the integers $0,2,4,\ldots,2dx-2$. If dx is odd, the remainders form a permutation of the integers $1,3,5,\ldots,2dx-1$. This implies that there exists an ordering x_i' along the x axis such that the remainders r'_i increase monotonically. That is,

$$r_i' = 2i + \rho_{dx} \tag{11}$$

where $\rho_{dx} = 0$ if dx is even, 1 otherwise.

Substituting r_i' for r_i in Eq. (5), we obtain

$$A_i = \frac{2i + dy + \rho_{dx}}{2dx} \tag{12}$$

The monotonicity of r'_i simplifies the iterative formula for A_i. Initially,

$$A_0 = \frac{dy + \rho_{dx}}{2dx} \tag{13}$$

The new recurrence for A_i becomes

$$A_{i+1} = A_i + \frac{1}{dx} \tag{14}$$

As long as $r_i' \le 2dx - 2dy$, the parallelogram intersects only two els in column x_i', and the value of A_i suffices to compute both intersection areas. The parallelogram intersects three els in

column x_i' when $2dx-2dy<r_i'$, which is equivalent to the condition $M \le i$ where

$$M = dx-dy-\rho_{dx}+1 \tag{15}$$

Substituting r_i' for r_i in Eq. (6), we obtain

$$B_{M+j} = \frac{(2j+2-\rho_{dx})^2}{8dxdy} \tag{16}$$

for $0 \le j$. B_{M+j} is now a quadratic function of the monotone parameter j. Forward differencing techniques reduce its computation to two additions. Initially,

$$\begin{aligned} B_M &= \frac{4-3\rho_{dx}}{8dxdy} \\ \Delta B_M &= \frac{12-4\rho_{dx}}{8dxdy} \\ \Delta^2 B_M &= \frac{8}{8dxdy} \end{aligned} \tag{17}$$

Forward difference formulas for B_{M+j} are

$$\begin{aligned} B_{M+j+1} &= B_{M+j}+\Delta B_{M+j} \\ \Delta B_{M+j+1} &= \Delta B_{M+j}+\Delta^2 B_{M+j} \\ \Delta^2 B_{M+j+1} &= \Delta^2 B_{M+j} \end{aligned} \tag{18}$$

for $0 \le j$. We are left with determining the current coordinates (x_i',q_i') for the trace of UBS. From Eqs. (2) and (11) the equation of the line containing UBS is

$$1/2+\frac{dy}{dx}x_i' = q_i'+\frac{2i+\rho_{dx}}{2dx} \tag{19}$$

Congruence relations for x_i' and q_i' may be obtained from Eq. (19).

Let dx^{-1} represent the multiplicative inverse of dx modulo dy, and dy^{-1} represent the multiplicative inverse of dy modulo dx. Then

$$x_i' \equiv i dy^{-1}+c_{dx} (\mathrm{mod}\ dx) \tag{20}$$

where c_{dx} is a constant with value $\left(dy^{-1}\left\lfloor\frac{dx+\rho_{dx}}{2}\right\rfloor\right)$ mod dx. Since the x_i' lie in the half open interval $[0,dx)$, this congruence can be used to derive a simple recurrence formula for computing x_i. Initially,

$$x_0' = c_{dx} \tag{21}$$

Then

$$\begin{aligned} x_{i+1}' &= x_i'+dy^{-1} && x_i'+dy^{-1}<dx \\ x_{i+1}' &= x_i'+dy^{-1}-dx && dx \le x_i'+dy^{-1} \end{aligned} \tag{22}$$

A congruence for q_i' may be obtained similar to Eq. (20).

$$q_i' \equiv -i dy^{-1}+c_{dy} (\mathrm{mod}\ dy) \tag{23}$$

where $c_{dy}=\left(dx^{-1}\left\lfloor\frac{dx-\rho_{dx}}{2}\right\rfloor\right)$ mod dy. This congruence can be used to obtain a recurrence formula for q_i'. We must be careful, however, because the q_i' lie in the closed interval $[0,dy]$, while the congruence distinguishes values only for the half open interval $[0,dy)$.

This is handled by a special case. We need to determine when $q_i'=dy$. This occurs only near the segment endpoint (dx,dy). It is not too hard to see that $q_i'=dy$ only when $dx-\left\lfloor\frac{\lfloor 1/2dx\rfloor}{dy}\right\rfloor \le x_i'$. The following recurrence formulas iteratively compute q_i. Initially,

$$q_0' = c_{dy} \tag{24}$$

Then

$$\begin{aligned} q_{i+1}' &= dy && dx-\left\lfloor\frac{\lfloor 1/2dx\rfloor}{dy}\right\rfloor \le x_i' \\ q_{i+1}' &= q_i'-dx^{-1} && 0 \le q_i'-dx^{-1} \\ q_{i+1}' &= q_i'-dx^{-1}+dy && q_i'-dx^{-1}<0 \end{aligned} \tag{25}$$

Suppose now that $gcd(dx,dy)=g$. Let $dx'=\frac{dx}{g}$ and $dy'=\frac{dy}{g}$. The parallelogram can now be divided into g identical pieces modulo an integral translation. Therefore, intersection areas for els $(j+ldx',k+ldy')$, where $0 \le j<dx', 0 \le k<dy'$, and $0 \le l<g$, are equivalent. The quantities g, dx', dy', dy'^{-1}, and dx'^{-1} can be obtained using Knuth's [Knut81] extended gcd algorithm. The algorithm runs in time $\Theta(\log dx)$. A tighter analysis reveals that the loop in the gcd algorithm is never executed more than $4.79\log_{10}dx+1.68$ times. For $dx \le 1024$, this is less than 16 iterations. We shall see that this small preprocessing time is dominated by the time to draw a line.

4. Algorithm Implementation

The key term in the computation of the blending function is the product αI (See Eq. 1). The el intersection areas lie in the range $\alpha \in [0,1]$. This range can be mapped to $[0,I]$ by considering the area of an el to be I instead of 1. With the mapped range $[0,I]$, pixel values for anti-aliasing are identical to el intersection areas. Note that range mapping takes place once in the initialization steps; no inner loop multiplications need be performed.

The two algorithms *APPROXIMATE_B_i* and *EXACT_B_i* are depicted in Figs. 4 and 5. As defined in the figures, they render grayscale lines against a constant background of black. Later, we shall show how to extend the technique to full color and non-uniform backgrounds. We now describe some implementation details that have been used to obtain the algorithms in the figures from the earlier results.

4.1. Rational Representation

Thus far, we have defined A_i and B_i to be rational quantities and operated on them as such. Through the representation of rational numbers as integral and fractional portions with an implied denominator, perfect accuracy of A_i and B_i can be retained using only integer arithmetic. In the following, we will use the suffix *err* appended to variable names to represent fractional numbers with implied denominator *den*.

Algorithm *APPROXIMATE_B_i*

$A \leftarrow \left\lfloor \frac{dx + I(dx+dy)}{2dx} \right\rfloor$;

$Aerr \leftarrow (dx + I(dx+dy)) \bmod 2dx$;

$Tri \leftarrow \left\lfloor \frac{dx + Idy}{2dx} \right\rfloor$;

$IlessTri \leftarrow I - Tri$;

$Sq \leftarrow \left\lfloor \frac{2Idy}{2dx} \right\rfloor$;

$Sqerr \leftarrow 2Idy \bmod 2dx$;

$over \leftarrow 2dy - 2dx$;

$under \leftarrow 2dy$;

$r \leftarrow 2dy - dx$;

$q \leftarrow 0$;

for $x = 0$ **to** $dx - 1$ **by** 1 **begin**

  if $r \geq 0$ **then begin**

    $A \leftarrow A - I$;

    $L \leftarrow \lfloor 1/2(Tri - A) \rfloor$;

    pixel$(x, q+1) \leftarrow L + A$;

    pixel$(x, q) \leftarrow IlessTri$;

    pixel$(x, q-1) \leftarrow L$;

    $r \leftarrow r + over$;

    $q \leftarrow q + 1$; **end**;

  else begin

    pixel$(x, q) \leftarrow A$;

    pixel$(x, q-1) \leftarrow I - A$;

    $r \leftarrow r + under$; **end**;

  $ADDF(2dx, A, Aerr, A, Aerr, Sq, Sqerr)$; **end**;

Fig. 4. Algorithm for drawing anti-aliased lines approximating the value of B_i.

It is convenient to define an operator *ADDF* which adds two rational numbers v and w represented in this fashion.

$$ADDF(den, u, uerr, v, verr, w, werr)$$

represents the following operations:

$u \leftarrow v + w$;
$uerr \leftarrow verr + werr$;
if $uerr \geq den$ **then begin** $u \leftarrow u + 1$; $uerr \leftarrow uerr - den$; **end**;

Note that fractional portions always lie in the half open interval $[0, den)$.

All rational numbers in the algorithms must be split into integral and fractional portions. This process occurs once at the beginning of the algorithm as a preprocessing step.

4.2. Round Off Errors

We have shown that final pixel colors are obtained by sums and differences of the A_i and B_i. Several of these operations can be saved by clever combinations of the operations. We will explain the case when the parallelogram crosses three els in a column with algorithm *APPROXIMATE_B_i* the other cases are left to the reader to verify.

Recall that here we want

$$\begin{aligned} pixel(i, q_i + 1) &\leftarrow B_i' \\ pixel(i, q_i) &\leftarrow A_i - 2B_i' \\ pixel(i, q_i - 1) &\leftarrow I - A_i - B_i' \end{aligned} \tag{26}$$

Note that these values have been mapped to the range $[0, I]$.

Let $Tri = \frac{Idy}{2dx}$. Substituting with Eq. (9), the formulas in Eq. (26) can be rewritten as

$$\begin{aligned} pixel(i, q_i + 1) &\leftarrow 1/2(Tri + A_i - I) \\ pixel(i, q_i) &\leftarrow I - Tri \\ pixel(i, q_i - 1) &\leftarrow 1/2(Tri - A_i + I) \end{aligned} \tag{27}$$

Since $I - Tri$ is constant, this value is pre-computed and stored in variable *IlessTri*. Using a temporary variable L, we obtain a faster, equivalent series of assignments.

$$\begin{aligned} A_i &\leftarrow A_i - I \\ L &\leftarrow 1/2(Tri - A_i) \\ pixel(i, q_i + 1) &\leftarrow L + A_i \\ pixel(i, q_i) &\leftarrow IlessTri \\ pixel(i, q_i - 1) &\leftarrow L \end{aligned} \tag{28}$$

Since values on the right hand sides of the above assignments are rational numbers represented as two integers, additions and subtractions should be performed on both integral and fractional portions of the representation. In the interest of speed and at the expense of some accuracy, we ignore fractional portions in the above assignments. We can do better than full truncation of the values by performing a *pre-rounding* operation. If we assume that A_i and Tri are artificially increased by $1/2$, the maximum error can be cut in half over full truncation. Pre-rounding is achieved by increasing fractional portions by half the implied denominator before entering the inner loop of the algorithm.

We leave the reader to verify the following:

Fact 1: The maximum truncation error with pre-rounding of any pixel for algorithm *APPROXIMATE_B_i* lies in the range $[-1, +1]$.

Fact 2: The maximum truncation error with pre-rounding of any pixel for algorithm *EXACT_B_i* lies in the range $[-1.5, +1.5]$.

Fact 3: Every pixel is given a color in the range $[0, I]$.

For applications that require full accuracy, rounding must be performed.

Algorithm $EXACT\_B_i$

if $dy = 0$ **then begin**

draw the horizontal line from (0,0) to $(dx,0)$; **return**; **end**;

find g; dx'; dy'; dx'^{-1}; and dy'^{-1};

$A \leftarrow \left\lfloor \frac{4dx'dy' + 4I(dy'^2 + \rho_{dx'})}{8dx'dy'} \right\rfloor$;

$Aerr \leftarrow (4dx'dy' + 4I(dy'^2 + \rho_{dx'})) \bmod 32dx'dy'$;

$B \leftarrow \left\lfloor \frac{4dx'dy' + I(4 - 3\rho_{dx'})}{8dx'dy'} \right\rfloor$;

$Berr \leftarrow (4dx'dy' + I(4 - 3\rho_{dx'})) \bmod 8dx'dy'$;

$\Delta B \leftarrow \left\lfloor \frac{I(12 - 4\rho_{dx'})}{8dx'dy'} \right\rfloor$;

$\Delta Berr \leftarrow I(12 - 4\rho_{dx'}) \bmod 8dx'dy'$;

$\Delta^2 B \leftarrow \left\lfloor \frac{8I}{8dx'dy'} \right\rfloor$;

$\Delta^2 Berr \leftarrow 8I \bmod 8dx'dy'$;

$Rect \leftarrow \left\lfloor \frac{8Idy}{32dx'dy'} \right\rfloor$;

$Recterr \leftarrow 8Idy \bmod 32dx'dy'$;

$x \leftarrow c_{dx'}$;

$q \leftarrow c_{dy'}$;

$C1 \leftarrow dx' - \left\lfloor \frac{\lfloor 1/2dx' \rfloor}{dy'} \right\rfloor$;

$M \leftarrow dx' - dy' - \rho_{dx'} + 1$;

for $i = 0$ **to** $M - 1$ **by** 1 **begin**

if $x \geq C1$ **then** $q \leftarrow dy'$;

for $0 \leq k < g$ pixel$(x + kdx', q + kdy') \leftarrow A$;

for $0 \leq k < g$ pixel$(x + kdx', q + kdy' - 1) \leftarrow I - A$;

$x \leftarrow (x + dy'^{-1}) \bmod dx'$;

$q \leftarrow (q - dx'^{-1}) \bmod dy'$;

$ADDF(8dx'dy', A, Aerr, A, Aerr, Rect, Recterr)$; **end**;

for $i = M$ **to** $dx' - 1$ **by** 1 **begin**

$Tmp \leftarrow A - B$;

for $0 \leq k < g$ pixel$(x + kdx', q + kdy' + 1) \leftarrow B$;

for $0 \leq k < g$ pixel$(x + kdx', q + kdy') \leftarrow Tmp - B$;

for $0 \leq k < g$ pixel$(x + kdx', q + kdy' - 1) \leftarrow I - Tmp$;

$x \leftarrow (x + dy'^{-1}) \bmod dx'$;

$q \leftarrow (q - dx'^{-1}) \bmod dy'$;

$ADDF(8dx'dy', A, Aerr, A, Aerr, Rect, Recterr)$;

$ADDF(8dx'dy', B, Berr, B, Berr, \Delta B, \Delta Berr)$;

$ADDF(8dx'dy', \Delta B, \Delta Berr, \Delta B, \Delta Berr, \Delta^2 B, \Delta^2 Berr)$; **end**;

Fig. 5. Algorithm for drawing anti-aliased lines using the exact value of B_i.

4.3. Parallelogram Width

The width of a parallelogram determines the response of area integration anti-aliasing. The perpendicular width of the parallelogram varies according to its slope. When horizontal, it has a width of one grid unit. When it has slope 1, it has a width of $\frac{\sqrt{2}}{2}$ grid units. Hence, parallelograms with slopes near 1 appear thinner than those with slopes near 0.

This effect can be eliminated with an initial intensity compensation dependent on the slope. We scale the intensity I to obtain a new intensity I' that is used throughout the algorithm.

$$I' = \frac{I^2}{I_{\max}} \tag{29}$$

where $I_{\max}$ has value

$$I_{\max} = \frac{(4dx - dy)I}{4dx} \tag{30}$$

since $\frac{4dx - dy}{4dx}$ is the maximum area that the parallelogram intersects a single el.

4.4. Word Size Requirements

Many of the constants and variables used in both algorithms can have magnitudes that are large. For example, in algorithm $EXACT\_B_i$ the implied denominator is $8dxdy$. When the display device has a resolution of $2^{10} \times 2^{10}$, this denominator can be as large as 2^{25}. We look at the magnitude of these numbers and sketch a technique for reducing the word size requirements. We shall make the assumptions that the display device has a spatial resolution of $2^{10} \times 2^{10}$ and intensity resolution of 2^9.

First, we examine $APPROXIMATE\_B_i$ This algorithm contains no quantities whose magnitude exceeds $2dx$. The maximum value for this quantity is 2^{11} and easily fits into a 16 bit word length common to many microcomputers.

While no final value that is stored in a variable exceeds 11 bits, intermediate values may. In particular, the initial computation that splits rational quantities into integral and fractional portions have intermediate values that are large. For example, the numerator of $\frac{Idy}{2dx}$ involved in the computation for the initial values of A and $Aerr$ may take 19 bits to represent. The whole term may be computed without using more than 15 bits using the following normalization scheme.

$$\frac{Idy}{2dx} = \frac{I}{2^7} \times \frac{dy}{2^7} \times \frac{2^{13}}{dx} \tag{31}$$

The first step is to express each term on the right hand side as a sum of integral and fractional portions. The first two terms are then multiplied to obtain another sum of integral and fractional portions, this time with denominator 2^{14}. The final multiplication is carried out in a similar fashion. Being careful to reduce remainders to within allowable ranges (in this case 2^7 and 2^{14}), all the intermediate terms will have a magnitude less than $2^{15} - 1$. This type of normalization procedure works for the initialization of all quantities in the algorithms.

We have seen that the implied denominators in $EXACT\_B_i$ will not fit within 16 bits. This is not a problem if the algorithm is to be implemented in hardware or most bit-slice firmware processors. However, many mini and micro computers do not support the word sizes required by this algorithm. We note that the magnitudes of numbers in $EXACT\_B_i$ may be reduced to 2^{23} by being careful in the computation of B_i. As technology progresses, we expect the next generation of mini and micro processors to support 24 bit word lengths; there is already a 32 bit microprocessor in production. Algorithm $EXACT\_B_i$ will then be efficiently implementable in more host CPU's.

5. Extensions

5.1. Full Color and Non-black Backgrounds

It has been assumed throughout that the line rendering algorithms have been implemented on a grayscale raster device and that the lines are drawn against a black background. For most applications, these assumptions are prohibitively restrictive. It is simple to generalize the algorithms to draw full color lines against constant colored and non-uniform backgrounds.

There are two different techniques for drawing lines in full color. The first is to run three identical algorithms on each of the *rgb* primary colors. Three grayscale values are computed for each pixel and are mixed to obtain color. At first glance, it might seem that this method would triple the time over drawing a single grayscale line. However, the time only doubles since none of the logic that determines pixel locations along the line need be repeated. This technique is an obvious candidate for hardware implementation where parallelism may be exploited.

The second option for drawing full color lines is to map the α range from [0,1] to $[0,2^k]$ where 2^k is the maximum resolution of each *rgb* channel. Pixel colors are computed by multiplying α in the new range by each of the line primary colors and normalizing by 2^k. For example, if the line has a red component of I_r and α is the intersection area of el (i,j), then pixel (i,j) will have a red component of

$$\frac{\alpha I_r}{2^k} \tag{32}$$

where the division is implemented as a right shift of k bits. Unfortunately, this introduces six to nine multiplications per inner loop step since each pixel color computed will require three multiplications.

By rearranging terms of the blending function, it is easy to see how our algorithms can draw against a constant, non-black background with little extra work.

$$\alpha I-(1-\alpha)I_{back} = \alpha(I-I_{back})+I_{back} \tag{33}$$

Instead of mapping the range of α from [0,1] to $[0,I]$, we now map to the range $[0,I-I_{back}]$. The only extra work over the original algorithm is to add the value I_{back} to each of the computed pixel colors. This is takes two or three more operations per inner loop step for grayscale implementations, or six or nine more operations per inner loop step for full color implementations. When $I-I_{back}$ is negative, the blending function can be rewritten as

$$-(\alpha(I_{back}-I)-I_{back}) \tag{34}$$

Here we map α to the range $[0,I_{back}-I]$, subtract I_{back} from computed pixel colors, and change the sign.

When drawing against a non-uniform background, each pixel must be read from the frame buffer before computing the blending function. In general, there is no a priori knowledge of the background color and the blending function must be computed directly. Here we map α to the range $[0,2^k]$ as before, and do the necessary multiplications.

5.2. Different Filter Functions

We have been careful throughout to distinguish els from pixels. By changing the functional correspondence between the two, different filter functions may be used for anti-aliasing. For example, instead of a direct one-to-one correspondence, we can allow the final value of each pixel (i,j) to be the weighted sum of els $(i,j),(i+1,j),(i,j+1)$, and $(i+1,j+1)$. By halving the size of an el in comparison to a pixel, endpoints of lines can be specified at twice the resolution available on the raster device. With an appropriate selection of el resolution and weights, virtually any filter can be simulated. Of course, computation time degrades as the functional mapping from els to pixels becomes more complex. In [Fiel83] we give some examples and analyze the performance of the algorithms under different filters.

5.3. Application to Polygons

It is not hard to see how the algorithms can be modified to anti-alias polygon edges. In [Fiel83] we report an anomaly that arises with long, thin portions of polygons when the filter response is not computed with full accuracy. We show how the filter accuracy available using $EXACT\_B_i$ is enough to eliminate the anomaly.

6. Analysis

Ordered from top to bottom in Plates 1 through 5 are lines rendered by algorithms [Bres65], [Gupt81], [Turk82], $APPROXIMATE\_B_i$ and $EXACT\_B_i$ Note that lines produced by approximating B_i are indistinguishable from those produced with exact values of B_i. Dissimilarity between the lines of a particular slope are caused by different filter functions.

To obtain performance estimates of the algorithms we total the number of operations performed within the inner loops (those that are executed $\Omega(dx)$ number of times). Addition, subtraction, multiplication, and branch instructions are all considered to require equal execution times. In [Fiel83], we give a more detailed description and justification of the underlying machine model used for algorithm timing statistics.

Table 1 contains rendering speeds for drawing grayscale lines from (0,0) to (dx,dy) on a constant black background. Algorithms [Gupt81] and [Turk82] each need three or four multiplications per inner loop step. Since our machine model does not distinguish between different types of operations, the time figures for [Gupt81] and [Turk82] are slightly skewed in their favor. We see that $APPROXIMATE\_B_i$ is the fastest of all the anti-aliasing algorithms and runs at least two, but less than three times slower than

the bilevel lines produced by [Bres65]. The large time required for [Turk82] is owing to the bitwise cordic rotation operations performed within each inner loop that are more suitably implemented in hardware.

Algorithm	T
[Bres65]	$8dx+8+dy$
[Gupt81]	$26dx+26+dy$
[Turk82]	$\geq 192dx+94dy$
$APPROXIMATE\_B_i$	$16dx+8dy$
$EXACT\_B_i$	$\leq 27dx+13dy-13$

Table 1. Rendering times for various line segment algorithms.

Modifications of [Gupt81] and [Turk82] to cope with full color and non-black backgrounds add extra inner loop multiplication and addition operations. Even when our algorithms require some inner loop multiplications, $APPROXIMATE\_B_i$ is always faster and does not use lookup tables. It is true that the square area integration filter we use does not always reduce the artifacts of aliasing as well as the other algorithms. Our algorithms lie somewhere on a "time-beauty tradeoff curve" between the bilevel lines produced by [Bres65] and the anti-aliased lines produced by [Gupt81] and [Turk82].

Because they are well suited to hardware implementation, our algorithms should see wide use in the future.

References

[Bres65] BRESENHAM, J.E. Algorithm for computer control of a digital plotter. *IBM Systems J. 4*, 1 (1965), 25-30.

[Fiel83] FIELD, D. *Algorithms for drawing simple geometric objects on raster devices.* Ph.D. Thesis, Princeton University Technical Report #314 Jun. 1983.

[Gup81] GUPTA, S. AND SPROULL, R.F. Filtering edges for gray-scale displays. *Computer Gr. 15*, 3 (Aug. 1981), 1-5.

[Knut81] KNUTH, D.E. *The Art of Computer Programming Vol. 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 1981.

[Pitt80] PITTEWAY, M.L.V. AND WATKINSON, D.J. Bresenham's algorithm with grey scale. *Commun. ACM 23*, 11 (Nov. 1980), 625-626.

[Turk82] TURKOWSKI, K. Anti-aliasing through the use of coordinate transformations. *ACM Trans. on Graphics 1*, 3 (Jul. 1982), 215-234.

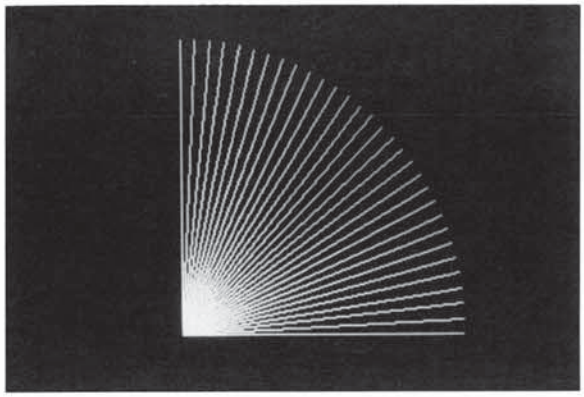

Plate 1. [Bres65]

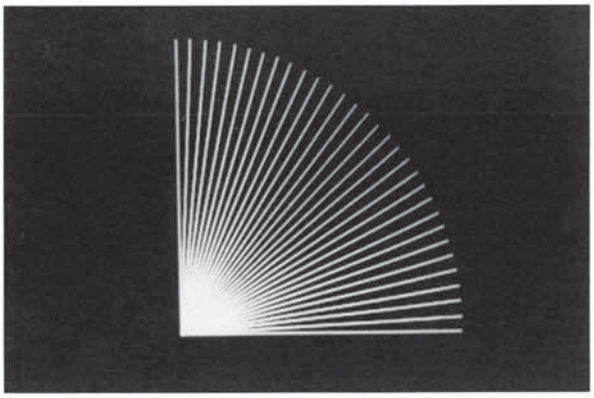

Plate 2. [Gupt81]

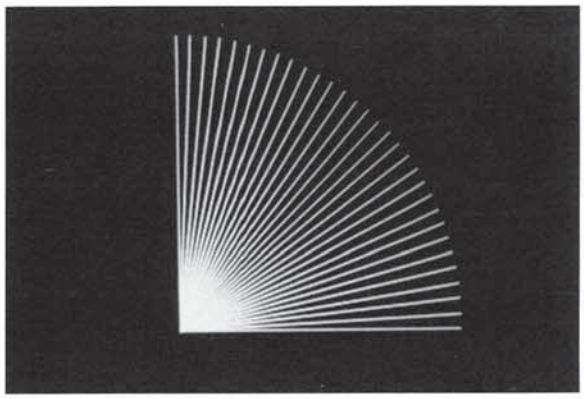

Plate 3. [Turk82]

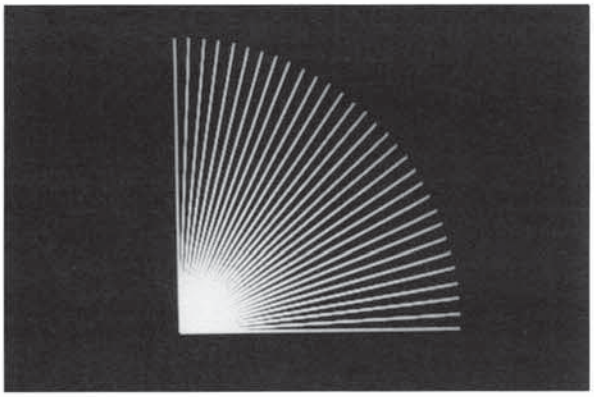

Plate 4. $APPROXIMATE\_B_i$

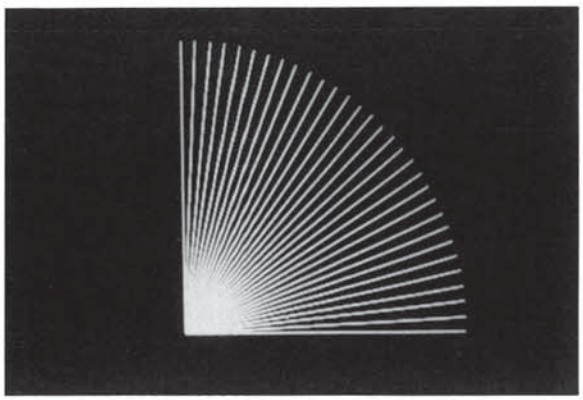

Plate 5. $EXACT\_B_i$

Line angles run from 0 to 90 degrees in 3 degree increments. To accentuate the differences between various rendering algorithms, the lines were photographed at 2x hardware magnification. All lines are 200 pixels long measured before magnification resulting in lengths of 400 pixels after magnification. Gamma compensation was performed for the display monitor only and does not include the photographic and printing processes.