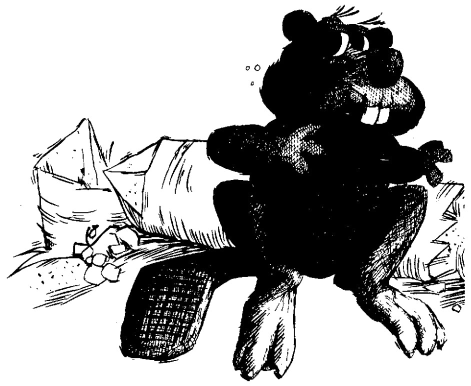

Algorithms for Drawing Anti-aliased Circles and Ellipses

Dan Field

CS-84-38

December, 1984

Algorithms for Drawing Anti-aliased Circles and Ellipses

Dan Field

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

ABSTRACT

Fast algorithms for pre-filtered anti-aliasing previously operated solely on straight line objects; this work extends the class of objects to include circles and ellipses. The new algorithms do not require floating point, square root, or division operations in the inner loops. When rendering against a constant background, a single inner loop multiplication operation may also be eliminated. We describe in detail how the technique is applied to rendering an octant of a filled circle using a square area integration filter. We briefly describe the extensions necessary for different types of anti-aliased circles and ellipses. We show how pre-computed filters stored in a lookup table may be implemented. An implementation of the algorithm for rendering anti-aliased, filled circles is included.

1. Introduction

Algorithms for drawing bi-level (not anti-aliased) circles are described in [1,2,3,9,11,12,14]. Straightforward modifications of these techniques do not seem to lend themselves to efficient anti-aliasing algorithms owing to the non-linear behavior in either the argument of the filter function or the value of the function itself. Advances for anti-aliasing along straight edges reported in [5,8,13,15] have taken advantage of the linearities inherent with straight edges to obtain efficient algorithms. Previous work on rendering anti-aliased circles and ellipses is reported in [4]. However, those algorithms use a Newton-Raphson root finding technique and suffer from the presence of a division operation within each iteration of the inner loop, and thus are expensive to implement in hardware or microcode. In this paper we report on new algorithms free of inner loop division operations.

This work was supported by grant A0183 from the Natural Sciences and Engineering Research Council of Canada.

The basic scheme of the algorithms presented here is to employ polynomial prediction of a filter function, based on previous values of the function, and to use iteration to bring the prediction within tolerance. In practice, the error in the prediction is usually quite small, so that simple linear convergence during the iteration step is sufficient to yield an efficient algorithm.

We shall assume that a raster device is modeled as an array of square, unit area pixels. For convenience, pixels are addressed by the lower left corner of the square. This amounts to a translation of 1/2 in both x and y from the standard model in which pixels are addressed by their centers. Let the color or intensity† of the circle be I. The color contribution to a pixel of a filled circle is defined to be $\alpha \cdot I$ where α is the intersection area of the pixel with the circle. For the moment, we shall focus our attention only on those pixels intersected by the circle edge since all other pixels are either wholly contained or excluded from the circle. Assuming the circle to be centered at the origin, it is sufficient to consider only those pixels lying within a single octant; symmetry of the circle allows those pixels to be reflected about the lines $x=0$, $y=0$, and $x=y$ to complete the circle.

Consider the portion of a circle with integer radius R in the second octant only. It can be shown that the edge of a circle moves no more than one pixel in the vertical direction between $x=i$ and $x=i+1$ and that there are only two ways that the edge of the circle can intersect a single column of pixels.

In the first case, the circle intersects only pixel (i,q_i) in column i. Let h_i and h_{i+1} be the distances above the line $y=q_i$ where the circle intersects $x=i$ and $x=i+1$ (See Fig. 1). A good approximation of the intersection area α is the area of the trapezoid defined by a chordal approximation of the circle; the filter value we shall use is:

$$\alpha=(h_i+h_{i+1})/2 \tag{1}$$

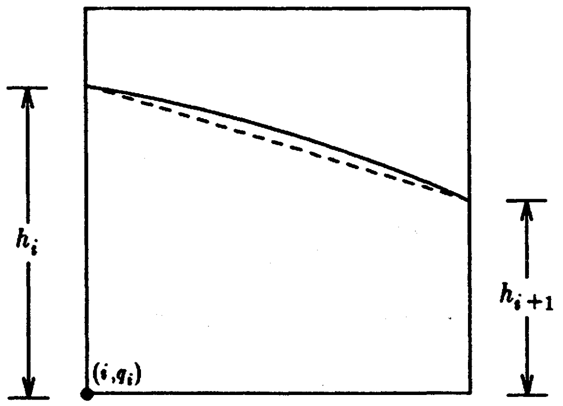

Fig. 1. The circle intersects pixel (i,q_i) in column i.

†The terms color and intensity are used interchangeably throughout. I can be thought of as a grayscale, an intensity for a single primary rgb, or a luminance channel in a color coordinate system such as CIE or HSV

In the second case, the circle intersects pixels (i,q_i) and (i,q_i+1) in column i (See Fig. 2). Let h_i be the distance above $y=q_i$ where the circle intersects $x=i$, and h_{i+1} be the distance above $y=q_i+1$ where the circle intersects $x=i+1$. An approximation of the filter at pixel (i,q_i) is:

$$\alpha=h_i/2 \tag{2}$$

and at pixel (i,q_i+1) is:

$$\alpha=(1+h_{i+1})/2 \tag{3}$$

The motivation behind choosing these approximations is that they are fast to compute, sum to the area of the chordal approximation to the circle contained in both pixels, provide smooth, monotonic intensity transitions between adjacent horizontal pixels, and do not cause disturbing visual effects.

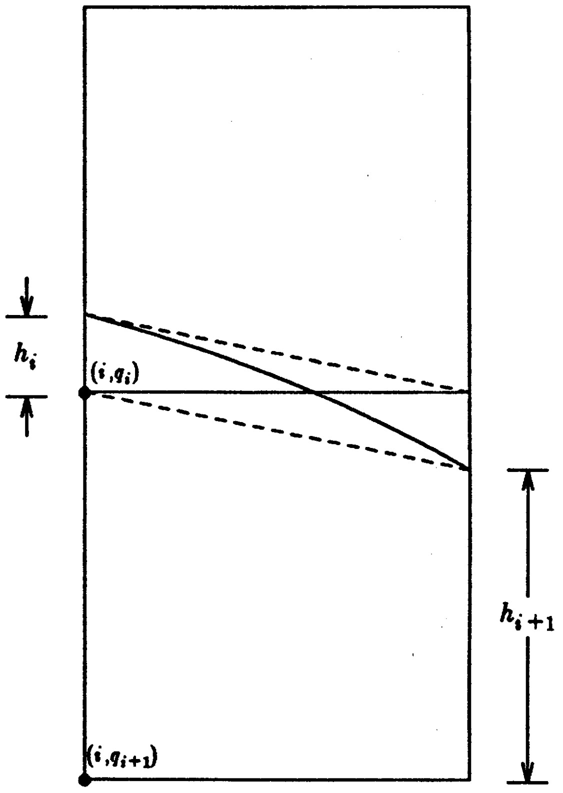

Fig. 2. The circle intersects pixels (i,q_i) and (i,q_i+1) in column i.

In most cases, the error in the intersection area resulting from straight-line approximations to the circle between $x=i$ and $x=i+1$ are small. If l is the length of the approximating chord between $x=i$ and $x=i+1$, the error has value $R^2\sin^{-1}\left(\frac{l}{2R}\right)-\frac{l}{2}\sqrt{R^2-l^2/4}$. An upper bound of $\frac{1}{3\sqrt{2}R}+O\left(\frac{1}{R^3}\right)$ may be obtained with a series expansion. For circles with radii greater than

10, the relative error resulting from straight-line approximation is less than 6 percent. Since the error increases as R decreases, it might be possible for small circles to be visibly distorted. Fig. 3 demonstrates that there is very little visual difference between the circles produced with and without chordal approximations for small radii.

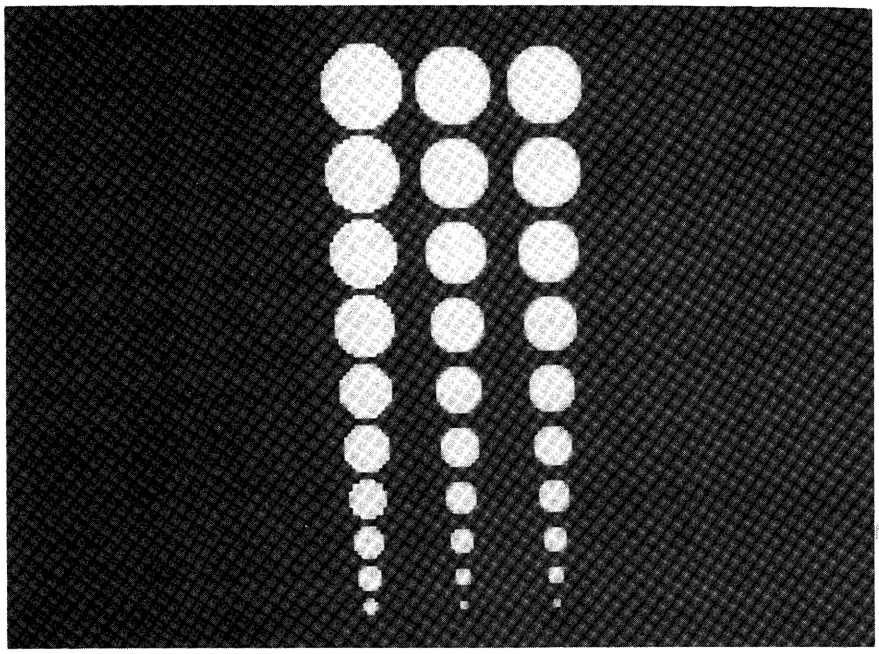

Fig. 3. A comparison of circles of radii 1 through 10 produced by 3 different algorithms. From left to right, the circles are: bilevel, anti-aliased without chordal approximations, and anti-aliased with chordal approximations. The image was produced using a 3x hardware magnification factor.

2. Mathematical Foundation

In the previous section, it was shown that the quantities i, q_i and h_i are sufficient to render anti-aliased circles. The algorithm sweeps from left to right along the x axis incrementing i from 0 until $q_i < i$. In this section, we describe how q_i and h_i are computed.

A fixed point representation for h_i is necessarily limited to some precision, say $1/Z$. Multiplying the fixed point representation by the inverse of the precision, an integral representation with equivalent accuracy is obtained. In the case of h_i, the precision becomes the accuracy to which the circle can be located between two pixels. Z is referred to as the *sub-pixel resolution*. The following equation describes the relationship between the fixed point and integral representations for h_i.

$$h_i = \frac{v_i}{Z} + \epsilon_i \tag{4}$$

where v_i is integral and ϵ_i is the error in the fixed point representation with $\frac{-1}{2Z} \le \epsilon_i < \frac{1}{2Z}$. Since $0 \le h_i < 1$, then $0 \le v_i < Z$. The integral representation for h_i is now v_i.

It may seem that the sub-pixel resolution, Z, should be as large as possible without overflowing the number of bits available for storing v_i. We shall later show that subject to accuracy constraints, Z should in fact be minimized because it contributes in a linear fashion to the number of iterations in a minimization step. For the moment Z is left as an unspecified parameter that is known at the beginning of the algorithm.

Since the circle edge is to be located to a sub-pixel resolution of Z, it is convenient to scale the equation of the circle by Z, and define the real-valued function: $f(i)=Z\sqrt{R^2-i^2}$.

Let $w_i=v_i+\lfloor 1/2+f(i+1)\rfloor-\lfloor 1/2+f(i)\rfloor$. The values of q_{i+1} and v_{i+1} can be inferred from q_i, and v_i as follows:
If $0\leq w_i$, then $(i+1,q_i)$ is on or inside the circle and

$$q_{i+1} = q_i \text{ and } v_{i+1} = w_i \tag{5}$$

Otherwise, $(i+1,q_i)$ is outside the circle and

$$q_{i+1} = q_i-1 \text{ and } v_{i+1} = w_i+Z \tag{6}$$

What remains is to find $y_{i+1}=\lfloor 1/2+f(i+1)\rfloor$. This will be accomplished by the following scheme.

1) Predict $\hat{y}_{i+1}$ as the value of $f(i+1)$
2) $\underset{\delta_{i+1}}{\text{Minimize}}\ |\hat{y}_{i+1}+\delta_{i+1}-f(i+1)|$

where $\hat{y}_{i+1}$ and δ_{i+1} are integers. That is, the prediction step yields a close approximation to $f(i+1)$, and the minimization step yields the error. The efficiency of this scheme depends on the accuracy of the prediction $\hat{y}_{i+1}$ (alternatively, the magnitude of δ_{i+1}), and the convergence rate of the minimization. We consider both prediction and minimization steps separately.

2.1. Prediction

An nth degree polynomial prediction $\hat{y}_{i+1}$ for $f(i+1)$ can be found with n addition operations in a forward differencing scheme. Prediction techniques based on Taylor approximations will work but require the more complex operations of multiplication and division. Define the difference operator, Δ, as:

$$\Delta y_{i+1} = y_{i+1}-y_i \tag{7}$$

Then $\hat{y}_{i+1}$ can be computed by adding the n differences in the standard manner[6].

The k^{th} order differences $\Delta^k y_{i+2}$ used to predict $\hat{y}_{i+2}$ for the next iteration are computed in the process of finding $\hat{y}_{i+1}$. However, these values will be inaccurate since this polynomial uses the predicted $\hat{y}_{i+1}$ for $f(i+1)$ instead of the the correct value, y_{i+1}. This is rectified by adding the error term $\delta_{i+1}=y_{i+1}-\hat{y}_{i+1}$ to the differences before evaluating the polynomial for the next iteration. In effect, we have set up a *sliding* nth degree polynomial prediction scheme for the function f.

The object of the prediction step is to reduce the magnitude of δ_i that determines the time taken by the minimization step. If the predictions were based on exact previous values of $f(i)$, we could decrease $|\delta_i|$ indefinitely by increasing the degree of the predicting polynomial. However, we have based the prediction on integral approximations of previous values of f, causing the roundoff errors in $\hat{y}_i$ to double in the degree of the approximating polynomial and dominate the behavior of the prediction. As will be demonstrated, a second degree polynomial is optimal over a full range of circle radii and sub-pixel resolutions. The average magnitude of δ_i has been empirically determined to be 0.7 over this range.

The quantity $\hat{y}_{i+1}-f(i+1)$ is the sum of two different errors. The first is roundoff error caused by integral representation of the $\Delta^k y_i$. We shall denote these by $t(\Delta^k y_i)=\Delta^k y_i-\Delta^k f(i)$. The total roundoff error is therefore the sum of the roundoff error in the addition cascade and has value $\sum_{k=0}^{n} t(\Delta^k y_i)$. The second type of error is caused by inaccuracies in the polynomial prediction itself. We shall denote this error by $s(\hat{y}_{i+1})$.

Roundoff errors no more than double with the degree of the polynomial. By assumption, $|t(\Delta^0 y_i)|\leq 1/2$. As differences are taken, roundoff errors may double so that $|t(\Delta^{k+1}y_i)|\leq 2|t(\Delta^k y_i)|$. Therefore, $|t(\Delta^k y_i)|\leq 2^{k-1}$. Since $\hat{y}_{i+1}=\sum_{k=0}^{n}\Delta^k y_i$, we have $|t(\hat{y}_{i+1})|\leq 2^n-1/2$ as an upper bound for the total roundoff error for nth degree polynomial prediction.

The error in nth degree polynomial prediction irrespective of roundoff is $s(\hat{y}_{i+1})=|f(i+1)-\sum_{k=0}^{n}\Delta^k f(i)|$ and is small in comparison to the roundoff errors. The remainder term in a Taylor expansion for $f(i+1)$ yields the prediction error, $s(\hat{y}_{i+1})=O(f^{(n+1)}(i+1))$.

Summing the roundoff and prediction errors over the course of drawing an octant of the circle, we obtain

$$\sum_{i=0}^{\lfloor R/\sqrt{2}\rfloor} |\delta_{i+1}| \leq \sqrt{2}R(2^{n-1}-1/4)+O\left(\frac{Z}{R^{n-1}}\right) \tag{8}$$

The asymptotic behavior of this error is governed by the $\sqrt{2}R2^{n-1}$ term, and suggests that n should be small even for large values of $\frac{Z}{R}$. Comparisons between upper and lower bounds for different values of n are difficult to obtain because of the erratic behavior in the $t(\Delta^k y_i)$ error term. Through empirical means we have determined the total error for $n=1,2,3$ and $1\leq Z,R\leq 256$. The results are presented with the aid of 3D surface graphs contained in Figs. 4-6. Each graph contains a comparison of errors between two prediction degrees: linear vs. quadratic in Fig. 4, linear vs. cubic in Fig. 5, and quadratic vs. cubic in Fig. 6. Circle radii increase in the x direction; sub-pixel resolution increases in the y direction. The height of the surface above or below the xy plane indicates the difference of the errors between the two prediction degrees being compared for a particular circle. Points where the surface intersects the xy plane represent circles that have equal error sums for the two degrees of prediction. The volume between the surface and the xy plane (positive above the plane, negative below) indicates the total error of the two prediction degrees over the range of sub-pixel resolutions and circle radii. From the figures we can deduce that quadratic is more accurate than linear, linear is more accurate than cubic, and quadratic is more accurate that cubic. Overall, quadratic prediction is most accurate.

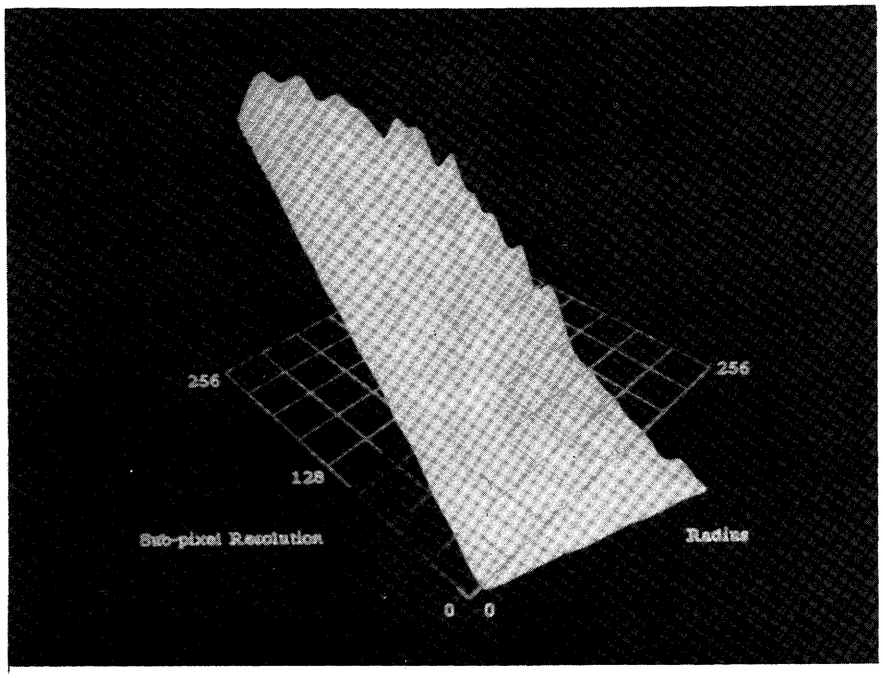

Fig. 4. Linear (larger error above the xy plane) vs. quadratic (larger error below the xy plane) polynomial prediction.

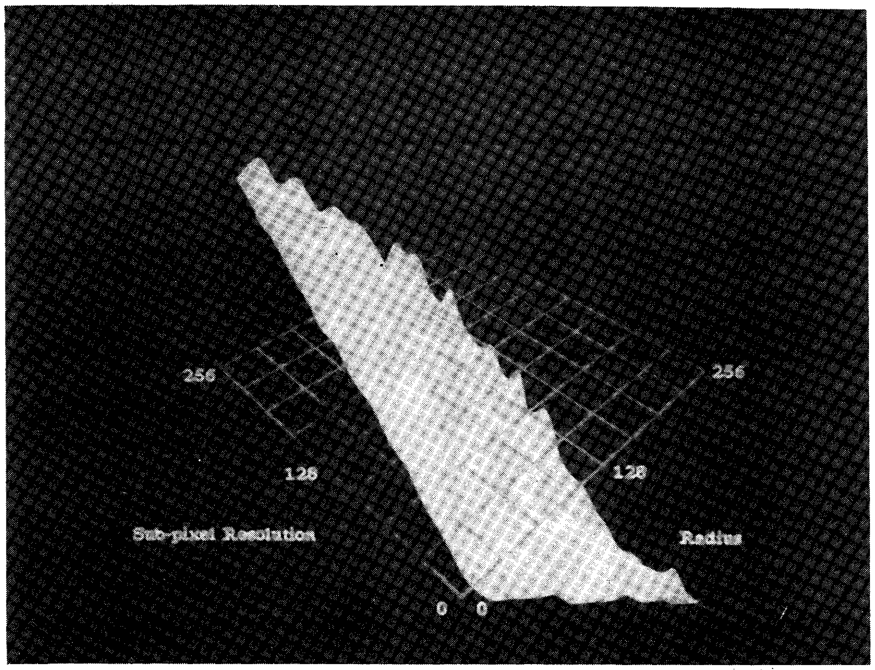

Fig. 5. Linear (larger error above the xy plane) vs. cubic (larger error below the xy plane) polynomial prediction.

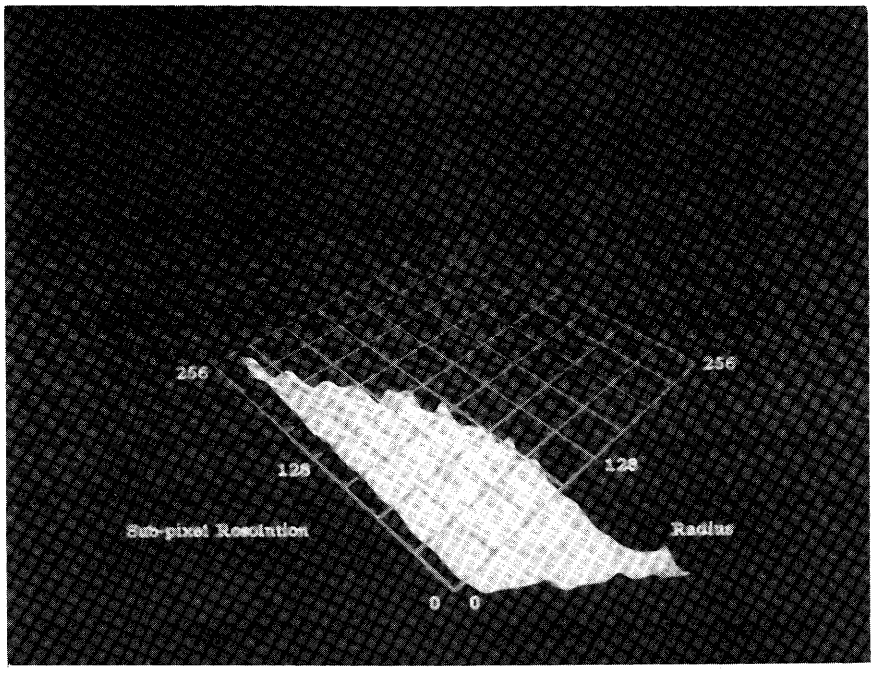

Fig. 6. Quadratic (larger error above the xy plane) vs. cubic (larger error below the xy plane) polynomial prediction.

2.2. Minimization

We next turn to the minimization step. The problem of finding the δ_{i+1} that minimizes $|\hat{y}_{i+1}+\delta_{i+1}-f(i+1)|$ can be transformed to the equivalent problem of finding the δ_{i+1} that minimizes the difference of squares $|(\hat{y}_{i+1}+\delta_{i+1})^2-f^2(i+1)|$. The terms in the latter minimization are free of square roots and may be computed with less difficulty. The $f^2(i+1)$ values are easily found using forward difference equations for $Z^2R^2-Z^2i^2$.

The quantity δ_{i+1} may be found using any root finding method. However, if we are to avoid multiplication and division operations, techniques such as Newton-Raphson that are based on extrapolating the function using its first derivative may be eliminated from consideration. Obvious candidates are linear and logarithmic (subdivision) searches. The number of operations taken in one iteration for linear search is less than that for logarithmic search. Initialization for linear search is also faster than initialization for logarithmic search. Because the expected value of $|\delta_{i+1}|=0.7$, the search will take very few iterations, and the constants in the asymptotic behavior of the algorithm become significant. Therefore, we shall implement linear search. This choice must be re-examined if other types of curves are to be rendered by the techniques presented here.

The search for δ_{i+1} is begun at 0. The search proceeds, updating the tentative value of δ_{i+1} by $d=\pm 1$. At step k of the minimization, our tentative value is kd. If $\hat{y}_{i+1}-f^2(i+1)>0$, then the prediction overshot the actual value and $d=-1$. Otherwise, $d=1$. Define

$$e_k = d[(\hat{y}_{i+1}+kd)^2-f^2(i+1)] \tag{9}$$

Thus, $e_0\geq 0$ and subsequent e_k decrease toward 0. The minimization is complete when e_k becomes negative. The value of δ_{i+1} that minimizes Eq. (9) is $(k-1)d$ if $e_{k-1}+e_k>0$, or kd otherwise.

Eq. (9) can be evaluated with addition and subtraction operations using forward differences. Simplifying to obtain

$$e_k = 2k\hat{y}_{i+1}+k^2d+d(\hat{y}_{i+1}^2-f^2(i+1)) \tag{10}$$

Initial values for the differences are:

$$\begin{aligned} \Delta^2 e_0 &= 2 \\ \Delta e_0 &= 2d\hat{y}_{i+1}-1 \\ e_0 &= \hat{y}_{i+1}^2-f^2(i+1) \end{aligned} \tag{11}$$

Except for the $\hat{y}_{i+1}^2$ term, the computations involved in Eq. (11) may all be implemented with addition and subtraction operations. Since $\hat{y}_{i+1}^2$ is a product of two second degree polynomials, it may be computed via four forward difference equations. However, just as the differences for $\hat{y}_{i+1}$ were updated by the value of δ_{i+1}, so must the differences for $\hat{y}_{i+1}^2$. It can be shown that two additional forward differences are sufficient for this task. For brevity we shall leave the reader to verify the details. As a practical concern, when a multiplication operation is part of the underlying machine instruction set, it will be more efficient to compute $\hat{y}_{i+1}^2=\hat{y}_{i+1}\cdot\hat{y}_{i+1}$ than to implement the forward difference formulation. The tradeoff is 1 multiplication versus $6+4\delta_i$ additions in step i of the algorithm.

2.3. Summary

Let us summarize what has been accomplished. We have outlined the steps to find the integral values of a function $f(i)$ from $i=0$ to $i=m$. An efficient algorithm was obtained using an accurate prediction for $f(i)$, followed by a minimization of the error between the prediction and $f(i)$. The original minimization problem was transformed, so that the error could be easily computed. The result of the minimization was used as an approximation for $f(i)$ and as feedback for the prediction of $f(i+1)$ in the next step. In principle, difference equations can eliminate multiplication and division operations for many functions $y=f(i)$ and transforms T when $T(f(i))$ is a polynomial in i and $T(y)$ is a polynomial in y. An interesting and difficult problem is to find functions and transforms for which this technique is more efficient than direct computation of the function itself.

3. Putting It All Together

The basic outline of the algorithm is contained in Fig. 7. This section contains the details necessary to produce an implementation from the information presented in the previous section.

```
1) initialize
2) set $i=0$, $q=R$
3) while i<q do steps 4 through 8
4)    predict
5)    minimize
6)    correct forward differences
7)    use computed values to fill in perimeter and interior pixels
8)    $i=i+1$
9) fill in four remaining pixels on the diagonals
```

Fig. 7. Outline of the algorithm for drawing anti-aliased, filled circles.

3.1. Initialization

Initialization of the forward differences for y involves square roots to compute $\Delta^2 y_0$ and $\Delta^1 y_0$. These may be eliminated by estimating $ZR = y_0 = y_1 = y_2$ to obtain $0 = \Delta^2 y_0 = \Delta^1 y_0$. The estimation will affect performance only during the first two iterations of the algorithm because predictions for y_3 and beyond are independent of y_0, y_1, and y_2.

3.2. Completing the Circle

A pixel location and intensity for the second octant is reflected about the lines $x=0$, $y=0$, and $x=y$ to fill all eight octants of the perimeter. Members of the eight reflection-induced locations with common x coordinates are paired and vertical lines are generated connecting members of the same pair. These lines serve to fill the interior of the circle. Therefore, the circle is drawn in three distinct portions: the *left* portion (LP) that is filled toward the right, the center portion (CP) that is filled both toward the left and right, and the right portion (RP) that is filled toward the left (See Fig. 8).

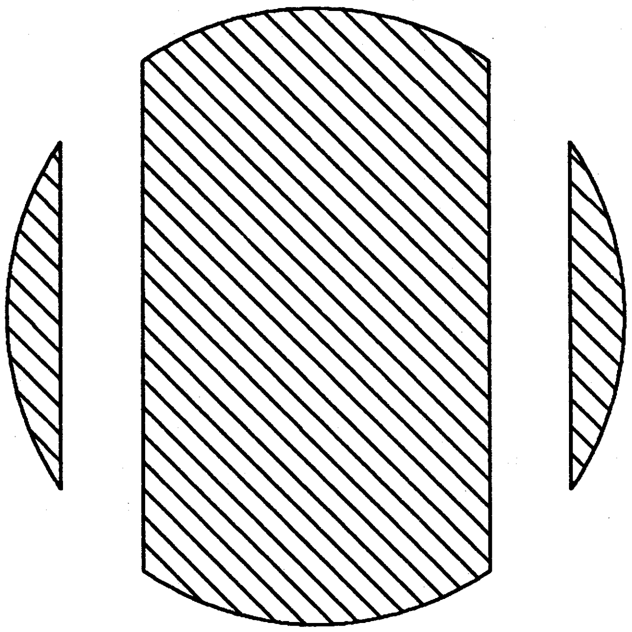

Fig. 8. The left (LP), center (CP), and right portions (RP) of a circle as it is being rendered.

Since pixels are addressed by their lower left corners and not by their centers, reflections about the x and y axes are achieved by adding 1 to the appropriate coordinate before changing the sign. For example, the reflection of the pixel (x,y) about the x axis is $(x,-y-1)$. We use the notation

eight_plot(x,y,I)

to represent the operation of setting the second octant pixel (x,y) as well as the pixels related by symmetry in the other 7 octants to intensity I.

Two vertical lines are drawn each step of the algorithm and increase the area of CP. Vertical lines to increase the area of LP and RP are not generated every step of the algorithm. Only when the circle edge crosses into a new column of pixels in LP and RP is it necessary to fill the interior pixels in that column. In the second octant, this is precisely the case when the circle edge crosses into a new row of pixels in CP. That is, only when the edge intersects the two pixels (i,q_i) and (i,q_i+1) in the second octant. In this case, lines $(q_i+1,i-1),(q_i+1,-i)$ and $(-q_i+1-1,i-1),(-q_i+1-1,-i)$ are also drawn as long as $i>0$. We use the notation

fill(x,ymax,ymin,I)

to represent the operation of drawing the vertical line of intensity I from pixel $(x,ymax)$ through $(x,ymin)$. If $ymax<ymin$, no line will be drawn.

The pixels on the perimeter that are drawn last are $(\lfloor R/\sqrt{2}\rfloor, \lfloor R/\sqrt{2}\rfloor)$ and the 3 others related by symmetry. These have the potential for being filled more than once if we are not careful about the stopping criteria for the algorithm. The strategy will be to run the main loop from $x=0$ to $x=\lfloor R/\sqrt{2}\rfloor-1$. A final step outside the loop is required to fill these pixels.

3.3. Full Color and Non-black Backgrounds

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

$$pixel(i,j) \leftarrow \alpha I+(1-\alpha)I_{back} \tag{12}$$

where α is the filter response at pixel (i,j), I is the intensity of the object being rendered, and I_{back} is the previous color of pixel (i,j). The blending function computes the linear "blend" between the object and the background. Note that this appears to be the best we can do without knowledge of how the background was actually created. We now demonstrate how to combine the results of the previous two sections for efficient and accurate computation of the blending function.

By rearranging the terms of Eq. (12), it is easy to see how one of the multiplications can be eliminated.

$$\alpha I+(1-\alpha)I_{back} = \alpha(I-I_{back})+I_{back} \tag{13}$$

One can think of renaming the original intensity I of the object as $I-I_{back}$. Once the product is computed, it is added to the old pixel value, I_{back}. If $I-I_{back}<0$, then we can rewrite the blending function as

$$I_{back}-\alpha(I_{back}-I) \tag{14}$$

and think of renaming the original intensity I as $I_{back}-I$.

When rendering against a constant background, the product $\alpha(I-I_{back})$ may be computed without any multiplications. First, if $0=I-I_{back}$, then the circle we are rendering will not be visible against the background and it may be ignored. Otherwise, let $Z=|I-I_{back}|$. Three analogs to Eqs. (1)-(3) are obtained by using Eq. (13) or (14) and substituting with the renamed intensity. When the circle edge crosses a single pixel in column i we have:

$$\begin{aligned}\alpha|I-I_{back}| &= (h_i+h_{i+1})|I-I_{back}|/2 \\ &= Z\left(\frac{v_i}{Z}+\epsilon_i+\frac{v_{i+1}}{Z}+\epsilon_{i+1}\right)/2 \\ &= (v_i+v_{i+1})/2+(\epsilon_i+\epsilon_{i+1})Z/2 \\ &\approx \left\lfloor (v_i+v_{i+1})/2 \right\rfloor\end{aligned} \tag{15}$$

When the circle edge crosses two pixels in column i we have the equation for pixel (i,q_i):

$$\begin{aligned}\alpha|I-I_{back}| &= h_i|I-I_{back}|/2 \\ &= v_i/2+\epsilon_i Z/2 \\ &\approx \left\lfloor v_i/2 \right\rfloor\end{aligned} \tag{16}$$

and the equation for pixel (i,q_{i+1}):

$$\begin{aligned}\alpha|I-I_{back}| &= (h_i+1)|I-I_{back}|/2 \\ &= (v_i+Z)/2+\epsilon_i Z/2 \\ &\approx \left\lfloor (v_i+Z)/2 \right\rfloor\end{aligned} \tag{17}$$

The sub-pixel resolution, Z, was chosen to eliminate multiplication operations and retain accuracy in the resulting pixel intensity. The errors caused by taking the floor after dividing the v terms lie in the range $[-1/2,0]$. From Eq. (4), it can be shown that the errors caused by ignoring the ϵ terms lie in the range $[-1/2,1/2]$. Therefore, the total error in the final pixel intensity resulting from truncation is in the range $[-1,1/2]$.

When rendering against a non-constant background, each pixel on the perimeter must be read from the frame buffer before computing the blending function. In general, there is no a priori knowledge of the background intensity, and the blending function must be computed directly. Therefore, the sub-pixel resolution, Z, is set to the smallest power of two larger than the maximum possible intensity. If v is the filter response at some pixel, v is multiplied by $|I-I_{back}|$ and subsequently divided by Z. While the multiplication is necessary, the choice for Z allows the division to be implemented by a shift operation.

There are three alternatives when rendering full rgb color. The first is to set Z to a sufficiently large power of two as before, and perform a separate multiplication for v with each primary color. For the red component, this amounts to computing $v(I_{red}-I_{back_{red}})$ and shifting the result $\log(Z)$ bits to the right. This requires an extra 3 multiplications per pixel.

If the background is constant, three distinct copies of the algorithm could be run in parallel (one for each primary color). As in the constant background case described above, multiplications are avoided by setting Z to the appropriate rgb component. For example, the red algorithm would set $Z=(I_{red}-I_{back_{red}})$. This approach would be feasible only for hardware implementations.

Guangnan et. al.[7] propose a color representation for anti-aliasing consisting of intensity and two normalized color components. The intensity component is used as the value of I for computing the blending function. This approach requires the addition of special hardware to transform the alternative color system into rgb values at the video output stage.

The question of computing the blending function for non-constant backgrounds without multiplication operations is open.

3.4. Machine Word Size

The maximum magnitude of $\hat{y}_{i+1}^2$ and $f^2(i+1)$ is Z^2R^2. It follows that $ZR < 2^{15}\sqrt{2}-1$ for 32-bit signed arithmetic. It is still possible to draw very large circles by scaling the sub-pixel resolution Z down 2^k before entering the algorithm, and scaling final pixel intensities back up 2^k before writing them to the frame buffer. The rightmost k bits of precision for pixel intensities are lost.

3.5. An Implementation

An implementation of the algorithm in the C programming language[10] is contained in Figs. 9 through 16 demonstrating that the mathematics of the previous sections lead to an efficient, straightforward algorithm, and providing ready-to-use code for readers not interested in the mathematics behind the algorithm. The program assumes a constant, 0 intensity background to simplify the presentation.

```
/*  Global Variables  */
static int i, q;              /* Current pixel address */
static int R;                 /* The circle radius */
static int x_center, y_center; /* Circle center */
static int I;                 /* Intensity of circle */
static int Z;                 /* Sub-pixel resolution */
static int y, y_hat;          /* Exact and predicted values of f(i) */
static int delta_y, delta2_y;  /* 1st and 2nd order differences of y */
static int f2;                /* The value of f(i)*f(i) */
static int delta_f2, delta2_f2; /* 1st and 2nd order differences of f(i)*f(i) */
static int v, v_old;  /* v/Z = percentage of pixel intersected by circle */

circle( ) {

int c;

initialize( );

q = R;
i = 0;
while( i<q ) {
   predict( );
   minimize( );
   correct( );
   v_old = v;
   v = v + delta_y;
   if( v>=0 ) { /* Single pixel on perimeter */
      eight_pixel( i, q, (v+v_old)>>1 );
      fill( i, q-1, -q, I );
      fill( -i-1, q-1, -q, I ); }
   else { /* Two pixels on perimeter */
      v = v + Z;
      eight_pixel( i, q, v_old>>1 );
      q = q - 1;
      fill( i, q-1, -q, I );
      fill( -i-1, q-1, -q, I );
      if( i<q ) { /* Haven't gone below the diagonal */
         eight_pixel( i, q, (v+Z)>>1 );
         fill( q, i-1, -i, I );
         fill( -q-1, i-1, -i, I ); }
      else /* Went below the diagonal, fix v for final pixels */
         v = v + Z; }
   i = i + 1; }

/* Fill in 4 remaining pixels */
c = v >> 1;
pixel( q, q, c );
pixel( -q-1, q, c );
pixel( -q-1, -q-1, c );
pixel( q, -q-1, c );

return; }
```

Fig. 9. Variable declarations and central portion of the algorithm.

```
initialize( ) {

v = 0;
/* The sub-pixel resolution Z would be different for
   non-zero, non-constant, or full rgb backgrounds. */
Z = I;

delta2_y = 0;
delta_y = 0;
y = Z * R;

delta_f2 = Z * Z;
delta2_f2 = -delta_f2 - delta_f2;
f2 = y * y;

return; }
```

Fig. 10. Initialize the forward differences and the sub-pixel resolution.

```
predict( ) {

delta_y = delta_y + delta2_y;
y_hat = y + delta_y; /* y_hat is predicted value of f(i) */

return; }
```

Fig. 11. Prediction step.

```
minimize( ) {

int e, e_old, d;

/* Initialize the minimization */
delta_f2 = delta_f2 + delta2_f2;
f2 = f2 + delta_f2;

e = y_hat * y_hat - f2;
d = 1;
y = y_hat;

/* Force e negative */
if( e>0 ) {
   e = -e;
   d = -1; }

/* Minimize */
while( e<0 ) {
   y = y + d;
   e_old = e;
   e = e + y + y - d; }

/* e or e_old is minimum squared error */
if( e+e_old>0 ) y = y - d;

return; }
```

Fig. 12. Minimize the difference of the squares, while computing y.

```
correct( ) {

int error;

error = y - y_hat;
delta2_y = delta2_y + error;
delta_y = delta_y + error;

return; }
```

Fig. 13. Correct the y forward differences for the next iteration.

```
eight_pixel( x, y, c )
int x, y, c; {

pixel( x, y, c );
pixel( x, -y-1, c );
pixel( -x-1, -y-1, c );
pixel( -x-1, y, c );
pixel( y, x, c );
pixel( y, -x-1, c );
pixel( -y-1, -x-1, c );
pixel( -y-1, x, c );

return; }
```

Fig. 14. Fill in all 8 pixels given a second octant pixel location.

```
fill( x, ymax, ymin, c )
int x, ymax, ymin, c; {

while( ymin<=ymax ) {
   pixel( x, ymin, c );
   ymin = ymin + 1; }

return; }
```

Fig. 15. Draw a vertical line between pixels (x,ymin) and (x,ymax).

```
pixel( x, y, c )
int x, y, c; {

/* Set pixel (x+y_center,y+y_center) to intensity c.  Note that
   we might have to obtain the previous contents of this pixel
   if we are rendering against a non-constant background.
   Further work must be done for non-zero backgrounds and full
   color implementations.

return; }
```

Fig. 16. Set pixel (x,y) to intensity c.

4. Extensions

In this section techniques are presented for drawing different types of circles. We briefly sketch the modifications to the algorithm necessary to draw ellipses. Finally, we sketch how a pre-computed filter stored in a look-up table can be implemented.

The algorithm we have presented draws a filled circle or *disk* of radius R_1 (See Fig. 17). By removing a concentric portion of radius R_2 from the disk, a *ring* is formed (See Fig. 18). Rings may be rendered directly by running two disk algorithms in parallel and composing the results. Often a ring with $R_1=R_2+1$ is desired. We can use the strategy for rendering rings above, but there is an alternative approximation approach that is almost twice as fast. Consider the circle with radius R_1 centered at (0,0). Now consider another circle of radius R_1 centered at $(0,-1)$. In the second octant the resulting figure closely approximates the ring with $R_1=R_2+1$. A slight widening of the circle occurs near the diagonals, but may be acceptable for many applications. The width distortion ranges from 0 to $(2-\sqrt{2})/2 \approx 0.3$. Only one set of v_i values need be generated since both circles have the same radius.

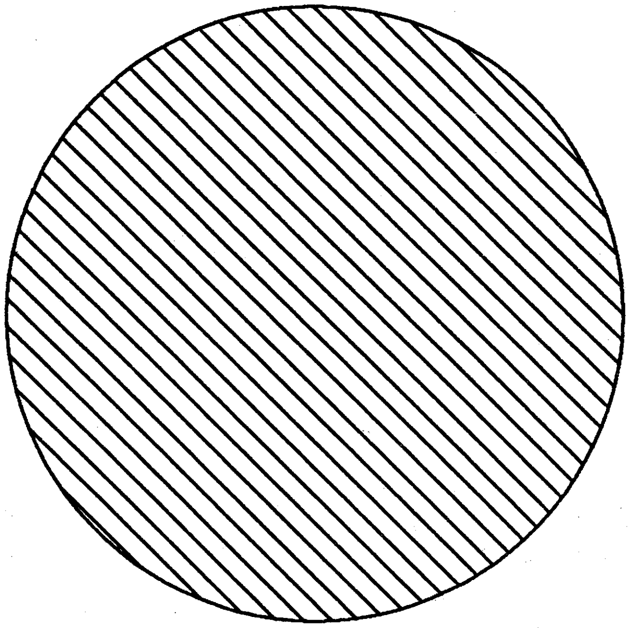

Fig. 17. A circular disk.

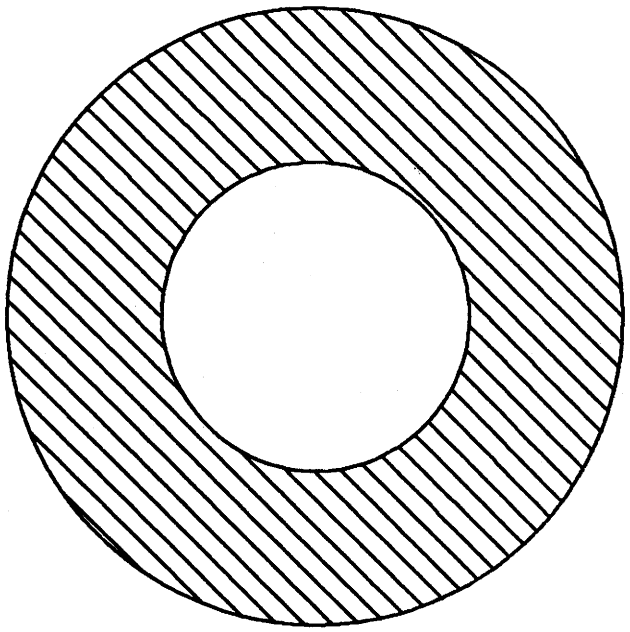

Fig. 18. A circular ring.

4.1. Ellipses

Anti-aliased ellipses may also be rendered using the techniques we have developed for drawing circles. The octant symmetry for circles has an analog for ellipses. The portion of an ellipse in the first quadrant may be divided into two pieces: the piece where the tangent ranges from 0 to -1, and the piece where the tangent ranges from -1 to $-\infty$. Combined, both pieces can be reflected about the lines $x=0$ and $y=0$ to completely fill the ellipse. Each have similar defining equations. If the equation of the ellipse is

$$\frac{x^2}{a^2}+\frac{y^2}{b^2} = 1 \tag{18}$$

one piece may be obtained from the other by interchanging a and b and swapping the roles of the x and y axes. We will sketch an algorithm for rendering only the portion of the ellipse in the first quadrant where $0 \le x \le \frac{a^2}{\sqrt{a^2+b^2}}$. This range of x values is defined by the inequalities $0 \le b^2 i < a^2 q_i$ for pixel locations (i, q_i) along the ellipse. As was the case for circles, it can be shown that the ellipse crosses no more than two pixels per column within this range.

The function necessary to obtain w_i for an ellipse is:

$$f(i) = \frac{Z}{a}\sqrt{a^2b^2-b^2i^2} \tag{19}$$

Prediction proceeds as with the case for the circle; the transformed minimization becomes

$$|\, a^2(\hat{y}_{i+1}+\delta_{i+1})^2 - a^2 f^2(i+1)\,| \tag{20}$$

Except for $a^2\hat{y}_{i+1}^2$, all of the quantities in Eq. (20) including $b^2 i$ and $a^2 q_i$ needed to test termination of the algorithm may be obtained using forward difference schemes. As in the previous section, $a^2\hat{y}_{i+1}^2$ may also be computed with forward differences but it is more efficient to use a single multiplication.

There is a direct extension of the algorithm from the disk form of ellipses to the ring form.

4.2. Pre-computed Filters

Square area integration filters assume a crude model of a physical raster device. As a result, aliasing effects are not always ameliorated as well as by a filter obtained with a more realistic model. Because the filters for these models are often expensive to compute on the fly, they are pre-computed and stored in a look-up table that is accessed at scan conversion time. These tables may also be determined experimentally and tailored for a specific environment. Two examples of algorithms that use look-up tables are described in [8,15].

We now sketch how to apply look-up table anti-aliasing to the problem of rendering circles. The desired filter function stored in the table will be indexed by the distance from the center of each pixel (i,q_i) to the nearest point on the circle. In the second octant, this distance may be approximated by the perpendicular distance p_i from (i,q_i) to the tangent of the circle at $x=i$. The perpendicular distance is a product of the vertical distance to the tangent, v_i with a scale factor c_i. Unlike the case for lines in [8], c_i does not remain constant throughout the course of the algorithm. However, both v_i and c_i have equations reminiscent of the function $f(i)$ of the previous sections.

The strategy is to compute an accurate value for c_i by prediction and minimization. Similar triangles can be used to show that

$$c_i = \frac{1}{R}\sqrt{R^2-i_2} \tag{21}$$

Let Z be an appropriately large power of 2 so that the function $g(i)=Zc_i$ can be represented accurately as an integer. The function $g(i)$ is now in a form suitable for computation by the technique described in section 2.

We need the vertical distance v_i to the tangent of the circle at $x=i$. This was computed in section 2 using the function $f(i)=Z\sqrt{R_2-i^2}$. But $f(i)=Rg(i)$. Thus, q_i and v_i are computable from $g(i)$ using a multiplication operation and the formulas of Eqs. (5) and (6).

The perpendicular distance is therefore:

$$p_i = v_i g(i) \tag{22}$$

Perpendicular distances from pixels $(i,q_i\pm m)$ to the tangent of the circle at $x=i$ are obtained by adding positive and negative multiples of $g(i)$ to p_i. Note that all distances have been scaled by Z implying that the look-up table has a sub-unit resolution of Z.

5. Conclusion

A general technique for incremental function evaluation has been described and applied resulting in efficient rendering algorithms for anti-aliased circles and ellipses. Incremental function evaluation plays a large role in scan conversion. We hope that the technique may find other uses, particularly for texture and surface lighting calculations, and edge tracking for curved surface patches. It would be especially useful if this technique could be made efficient for trigonometric and exponential functions.

I thank David Dobkin for many helpful conversations. Figs. 4-6 were created by ray tracing spline surfaces with software developed by Mike Sweeney and aided by members of the Computer Graphics Laboratory at Waterloo.

References

[1] BADLER, N.I. Disk generators for a raster display device. *Comp. Gr. and Im. Proc. 6,* 6 (Dec. 1977), 589-593.

[2] BRESENHAM, J.E. A linear algorithm for incremental digital display of circular arcs. *Commun. ACM 20,* 2 (Feb. 1977), 100-106.

[3] DOROS, M. Algorithms for generation of descrete circles, rings, and disks. *Comp. Gr. and Im. Proc. 10,* 4 (Aug. 1979), 366-371.

[4] FIELD, D. *Algorithms for drawing simple geometric objects on raster devices* . Ph.D. Thesis, Princeton University 1983.

[5] FIELD, D. Two algorithms for drawing anti-aliased lines. *Proc. Graphics Interface '84,* (May, 1984), 87-95.

[6] FOLEY, J.D. AND VAN DAM, A. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass., 1982.

[7] GUANGNAN, N., TANNER, P., WEIN, M., AND BECHTHOLD, G. An algorithm for generating anti-aliased polygons for 3-d applications. *Proc. Graphics Interface '83,* (May, 1983), 23-32.

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

[9] HORN, B.K.P. Circle generators for display devices. *Comp. Gr. and Im. Proc. 5,* 2 (Jun. 1976), 280-288.

[10] KERNIGHAN, B.W. AND RITCHIE, D.M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[11] MCILROY, M.D. Best approximate circles on integer grids. *ACM Trans. on Gr. 2,* 4 (Oct. 1983), 237-263.

[12] PITTEWAY, M.L.V. Algorithm for drawing ellipses or hyperbolae with a digital plotter. *Computer J. 10,* 3 (1967), 282-289.

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

[14] SUENAGA, Y., KANAE, T., AND KOBAYASHI, T. A high-speed algorithm for the generation of straight lines and circular arcs. *IEEE Trans. Comput. TC-28,* 10 (Oct. 1979), 728-736.

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