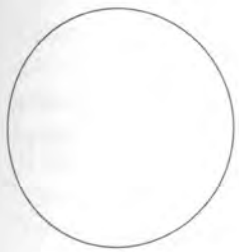
Tomado de:

Blinn, J. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. Morgan Kaufmann Publishers, Inc. San Francisco California. 1996. Capítulo 1.



CHAPTER ONE

How Many Ways Can You Draw a Circle?

AUGUST 1987

I like to collect things. When I was young I collected stamps; now I collect empty margarine tubs and algorithms for drawing circles. I will presume that the latter is of most interest to you, so in this chapter I will go through my album of circle-drawing algorithms.

It's traditional at this point in any discussion of geometry to drag in the ancient Greeks and mention how they considered the circle the most perfect shape. Even though a circle is such an apparently simple shape, it is interesting to find how many essentially different algorithms you can find for drawing the Greeks' favorite curve.

I will be very brief about some pretty obvious techniques to leave space to play with the more interesting and subtle techniques. Note that many of these algorithms might be ridiculously inefficient but are included to pad the chapter. (OK, OK, they're included for *completeness.*)

I'm not sure where I first heard of some of these. I will cite inventors where known, but let me just thank the world at large in case I've missed anybody.

A word about the programming language used: I am not using any formal algorithm display language here. These algorithms are meant to be read by human beings, not computers, so the language is a mishmash of several programming constructs that I hope will be perfectly clear to you.

The collection can be categorized by the two types of output, line endpoints or pixel coordinates. This comes from the general dichotomy of curve representation—parametric versus algebraic.

Line Drawings

First let's look at line output. All these algorithms will operate in floating point and generate a series of x, y points on a unit radius circle centered at the origin. You then play connect-the-dots.

(1) Trigonometry

Evaluate sin and cos at equally spaced angles.

```
MOVE(1,0)
FOR DEGREES = 1 to 360
    RADIANS = DEGREES * 2 * 3.14159/360.
    DRAW(COS(RADIANS),SIN(RADIANS))
```

This has to evaluate the two trig functions at each loop; ick.

(2) Polynomial Approximation

You can get a fair approximation to a circle by evaluating simple polynomial approximations to sin and cos. The first ones that come to mind are the Taylor series.

$$\cos a \approx 1 - 1/2a^2 + 1/24a^4$$
$$\sin a \approx a - 1/6a^3 + 1/120a^5$$

These require fairly high-order terms to get very close, partly because the Taylor series just fits the position and several derivatives at *one* endpoint.

A better approach is to fit lower-order polynomials to both desired endpoints and end slopes. This is effectively what is happening with various commonly used Bézier curves. For example, the four control points (1, 0), (1, .552), (.552, 1), (0, 1) describe a good approximation to the upper-right quarter of a circle. You can get the other three quadrants by rotating the control points.

When transformed to polynomial form, the first quadrant is

$$x(t) = 1 - 1.344t^2 + .344t^3$$
$$y(t) = 1.656t - .312t^2 - .344t^3$$

with the parameter t going from 0 to 1.

```
MOVE(1,0)
FOR T = 0 TO 1 BY .01
    X = 1 + T * T * (-1.344 + T * .344)
    Y = T * (1.656 - T * (.312 + T * .344)
    DRAW(X,Y)
```

This makes a pretty good circle. The maximum radius error is about .0004 at $t = .2$ and $t = .8$.

(3) Forward Differences

Polynomials can be evaluated quickly by the technique known as Forward Differences. Briefly, for the polynomial

$$f(t) = f_0 + f_1 t + f_2 t^2 + f_3 t^3$$

if you start at $t = 0$ and increment by equal steps of size δ, the forward differences are

$$\Delta f = f_1\delta + f_2\delta^2 + f_3\delta^3$$
$$\Delta\Delta f = 2f_2\delta^2 + 6f_3\delta^3$$
$$\Delta\Delta\Delta f = 6f_3\delta^3$$

Then, for our polynomials stepping in units of .01,

```
X = 1; DX = -.000134056; DDX = -.000266736; DDDX =  .000002064
Y = 0; DY =  .016528456; DDY = -.000064464; DDDY = -.000002064
MOVE(X,Y)
FOR I = 1 TO 100
    X = X + DX; DX = DX + DDX; DDX = DDX + DDDX
    Y = Y + DY; DY = DY + DDY; DDY = DDY + DDDY
    DRAW(X,Y)
```

Trust me, I'm a doctor. If you don't believe it, look up Forward Differences in Newman and Sproull's book[1]—I'm not going to do *all* the work here.

Notice the number of significant digits in the constants. It might seem that that many digits would require double precision, but, in practice, the accumulated roundoff error using single precision is less than the error due to the polynomial approximation.

(4) Incremental Rotation

Let's back off from the approximation route and try another approach. Start with the vector (1, 0) and multiply it by a one-degree rotation matrix each time through the loop.

```
DELTA = 2 * 3.14159/360.
SINA = SIN(DELTA)
```

1 William M. Newman and Robert F. Sproull, *Principles of Interactive Computer Graphics*, 2nd ed. (New York: McGraw-Hill, 1979), page 328.

```
COSA = COS(DELTA)
X = 1;  Y = 0
MOVE(X,Y)
FOR I = 1 TO 360
    XNEW = X * COSA - Y * SINA
    Y    = X * SINA + Y * COSA
    X    = XNEW
    DRAW(X,Y)
```

(5) Extreme Approximation

If the incremental angle is small enough, we can approximate $\cos a = 1$ and $\sin a = a$. The number of times through the loop is $n = 2\pi/a$ or, contrariwise, the angle is $a = 2\pi/n$, depending on which you want to use as input.

```
A = .015;  N = 2 * 3.14159/A
X = 1; Y = 0
MOVE(X,Y)
FOR I = 1 TO N
    XNEW = X - Y * A
    Y    = X * A + Y
    X    = XNEW
    DRAW(X,Y)
```

But there's a problem. Each time through the loop we are forming the product

$$[x_{new}, y_{new}] = [x_{old}, y_{old}] \begin{bmatrix} 1 & a \\ -a & 1 \end{bmatrix}$$

The matrix is almost a rotation matrix, but its determinant equals $1 + a^2$. This is bad. It means that the running $[x, y]$ can be magnified by this amount on each iteration, so what we get is a spiral that gets bigger and bigger. How to fix this? Introduce a bug into the algorithm.

(6) Unskewing the Approximation

Since vector multiplication and assignment don't occur in one statement, we had to calculate y carefully, using the old value for x. Suppose we were dumb and did it the naive way.

```
A = .015;  N = 2 * 3.14159/A
X = 1;     Y = 0
MOVE(X,Y)
FOR I = 1 TO N
```

```
X = X - Y * A
Y = X * A + Y
DRAW(X,Y)
```

Now, what is the effect of this? Really what we get is

$$x_{new} = x_{old} - y_{old}a$$
$$y_{new} = x_{new}a + y_{old} = x_{old}a + y_{old}(1-a^2)$$

In other words,

$$[x_{new}, y_{new}] = [x_{old}, y_{old}]\begin{bmatrix} 1 & a \\ -a & 1-a^2 \end{bmatrix}$$

This matrix has a determinant of 1, and there is no net spiraling effect. What you get is actually an ellipse that is stretched slightly in the northeast-southwest direction and squeezed slightly in the northwest-southeast direction. The maximum radius error in these directions is approximately $a/4$.

Now comes the interesting part. Since you can start out with any vector, let's try (1000, 0). Now, cleverly select a to be an inverse power of 2 and the multiplication becomes just a shift. For example, a value of $a = 1/64$ is just a right shift by 6 bits. This generates the circle in about 402 steps. So, you can do this all with just integer arithmetic and no multiplication. This, children, is how we used to draw circles quickly—and in fact do rotation incrementally—before the age of hardware floating point and even hardware multiplication. (I understand that this was invented by Marvin Minsky.)

(7) Rational Polynomials

Another polynomial tack can be taken by looking in our hat and pulling out the following rabbit:

$$\begin{aligned} &\text{if} && x = (1-t^2)/(1+t^2) \\ &\text{and} && y = 2t/(1+t^2) \\ &\text{then} && x^2 + y^2 = 1 \end{aligned}$$

no matter what t is (or *identically*, as the mathematicians would say). Running t from 0 to 1 gives the upper-right quadrant of the circle. We can again evaluate these polynomials by forward differences, stepping t in increments of .01, and get

```
X = 1;   DX = -.0001;   DDX = -.0002
Y = 0;   DY =  .02
W = 1;   DW =  .0001;   DDW = .0002
```

```
MOVE(X,Y)
FOR I = 1 TO 100
    X = X + DX;   DX = DX + DDX
    Y = Y + DY
    W = W + DW;   DW = DW + DDW
    DRAW(X/W,Y/W)
```

Note that this is *not* an approximation like the last few tries. It is exact—except for roundoff error. Even roundoff error can be removed, either by calculating the polynomials directly or by scaling all numbers by 10000 and doing it with integers. (The division x/w must still be done in floating point.)

This one has always amazed me: you get to effectively evaluate two transcendental functions *exactly* with only a few additions. What's the catch? It's an application of the No-Free-Lunch Theorem—you don't get to pick the angles. If you watch the points, you see that they are not equally spaced around the circle. In fact, as t goes to infinity, the point keeps going counterclockwise but slows down, finally running out of juice at (–1, 0). If you go backwards to minus infinity, the point goes clockwise, finally stopping again at (–1, 0). (Yet more evidence that $-\infty = +\infty$.) To draw a complete circle, you are best advised to run t from –1 to +1, which draws the whole right half, and then mirror it to get the left half.

(8) Differential Equation

An entirely different technique is to describe the motion of $[x, y]$ dynamically. Imagine the point rotating about the center as a function of time t. The position, velocity, and acceleration of the point will be

$$[x, y] = [\cos t, \sin t]$$
$$[x', y'] = [-\sin t, \cos t] = [-y, x]$$
$$[x'', y''] = [-\cos t, -\sin t] = [-x, -y]$$

You can cast these into differential equations and use any of several numerical integration techniques to solve them.

The dumbest one, Euler integration, is just

$$x_{new} = x_{old} + x'_{old}\Delta t = x_{old} - y_{old}\Delta t$$
$$y_{new} = y_{old} + y'_{old}\Delta t = y_{old} + x_{old}\Delta t$$

This looks a lot like Algorithm 5, and it has the same spiraling-out problem. You can generate better circles by using better integration techniques. My two favorites are the leapfrog technique and the Runge-Kutta technique.

Leapfrog calculates the position and acceleration at times

$$t,\quad t+\Delta t,\quad t+2\Delta t,\quad \ldots$$

but calculates the velocity at times halfway between them:

$$t+\tfrac{1}{2}\Delta t,\quad t+\tfrac{3}{2}\Delta t,\quad \ldots$$

Advancing time one step then looks similar to Euler, with just the evaluation times offset:

$$x_{t+\Delta t} = x_t + x'_{t+\frac{1}{2}\Delta t}\Delta t$$
$$x'_{t+\frac{3}{2}\Delta t} = x'_{t+\frac{1}{2}\Delta t} + x''_{t+\Delta t}\Delta t$$

(with similar equations for y). The position and velocity "leapfrog" over each other on even/odd half-time steps, so you have to keep separate variables for the velocities, x' and y'. The code has a lot in common with Algorithm 6, and probably for good reason.

```
X  = 1;              Y = 0
VX = -SIN(DT/2);     VY = COS(DT/2)
MOVE(X,Y)
FOR I  = 1 TO N
    X  = X  + VX * DT   "update posn"
    Y  = Y  + VY * DT
    VX = VX - X  * DT   "update veloc, AX = -X"
    VY = VY - Y  * DT                  "AY = -Y"
    DRAW(X,Y)
```

Runge-Kutta is a slightly involved process that takes a fractional Euler step, reevaluates the derivatives there, applies the derivative at the original point, steps off in this new direction, generally screws around, and finally takes some average between all these to get the new time step. Plugging our differential equation into the formulas and simplifying requires about a page of algebra. You can look up the actual equations;[2] they're not *incredibly* complicated but their *derivation* is "beyond the scope" of almost all numerical analysis textbooks I have seen.

One advantage of Runge-Kutta is that it finds the position and velocity at the same time step, so for circles you can generate x and y with the same computation. Another advantage is that it comes in second-order, third-order, fourth-order, etc., versions for higher orders of precision than leapfrog.

2 Runge-Kutta equations can be found in any numerical analysis text; e.g., Francis Scheid, *Schaum's Outline Series, Theory and Problems of Numerical Analysis* (New York: McGraw-Hill, 1968).

Plugging in for second-order Runge-Kutta, the ultimate result is

$$x_{new} = x_{old}(1 - \tfrac{1}{2}\Delta t^2) + y_{old}(-\Delta t)$$
$$y_{new} = x_{old}(\Delta t) + y_{old}(1 - \tfrac{1}{2}\Delta t^2)$$

Does this look familiar? It's just the Taylor series approximation to sin and cos again. The third-order Runge-Kutta and another page of algebra leads to

$$x_{new} = x_{old}(1 - \tfrac{1}{2}\Delta t^2) + y_{old}(-\Delta t + \tfrac{1}{6}\Delta t^3)$$
$$y_{new} = x_{old}(\Delta t - \tfrac{1}{6}\Delta t^3) + y_{old}(1 - \tfrac{1}{2}\Delta t^2)$$

Guess what fourth-order Runge-Kutta gives. . . . You're right. I won't even bore you with the code.

(9) Half Interval

Another idea is the half interval method suggested by Jim Kajiya. This assumes you have two endpoints of an arc and wish to fill in the middle with points on the circle. At each step you insert a new point between two others. Assuming a circle centered at origin, the new point will be approximately halfway between the surrounding ones:

$$[x_m, y_m] = \left[\frac{x_1 + x_2}{2}, \quad \frac{y_1 + y_2}{2}\right]$$

It just needs to be moved outwards to lie on the circle. This involves scaling the above to length 1. If the original points are at unit distance from the origin, this means dividing by $\sqrt{1 + x_1x_2 + y_1y_2} / \sqrt{2}$.

By doing this recursively, you can keep splitting until some error tolerance is met. The code is something like

```
X1 = 1; Y1 = 0
X2 = 0; Y2 = 1
MOVE(X1,Y1)
SPLIT(X1,Y1,  X2,Y2)
```

where we define `SPLIT(X1,Y1, X2,Y2)` to be

```
D  = SQRT(.5 * (1 + X1 * X2 + Y1 * Y2))
XM = (X1 + X2)/D
YM = (Y1 + Y2)/D
IF error tolerance ok
    DRAW(XM,YM)
    DRAW(X2,Y2)
```

```
ELSE
    SPLIT(X1,Y1,    XM,YM)
    SPLIT(XM,YM,    X2,Y2)
```

The error tolerance could be just a recursion depth counter, stopping at a fixed recursion depth. This is nice because, for a given pair of initial points, the value of D is just a function of recursion depth and can be precomputed and placed in a table.

Pixel-Based Techniques

The other major category of algorithms involves output more directly suited to raster displays. Here the question is not where to move the "pen" next, but which of the grid of pixels to light up. The above algorithms can of course be applied to pixels by generating coordinates and feeding them to a line-to-pixel drawing routine, but we won't pursue those. Let's just look at ways to generate the desired pixels directly. For simplicity we will assume we are drawing a 100-pixel-radius circle with pixels addressed so that (0, 0) is in the center and negative coordinates are OK. The algorithms operate in integer pixel space, assuming square pixels. Note that the variables below start with I, indicating that they are integers.

(10) Fill Disk

Perhaps the dumbest algorithm is just to see how far each pixel is from the center and color it in if it's inside the circle:

```
FOR IY = -100 TO 100
FOR IX = -100 TO 100
    IF (IX * IX + IY * IY < 10000) SETPXL(IX,IY)
```

This of course fills in the entire disk instead of just drawing lines, but who's being picky?

You would be correct in assuming that this might be a bit slow. Some quick speedups: calculate the value of x^2 by forward differences; calculate the allowable range of x^2 outside the x loop (forward differences probably aren't worth the trouble for this latter calculation).

```
FOR IY = -100 TO 100
    IX2MAX = 10000 - IY * IY
    IX2 = 10000;  IDX2 = -199;  IDDX2 = 2
    FOR IX = -100 TO 100
        IF (IX2 < IX2MAX) SETPXL(IX,IY)
        IX2 = IX2 + IDX2;  IDX2 = IDX2 + IDDX2
```

(11) Solve for *x* Range Covered

The above still examines every pixel on the screen. We can skip some of this by explicitly solving for the range in x.

```
FOR IY = 100 TO -100 BY -1
    IXRNG = SQRT(10000 - IY * IY)
    FOR IX = -IXRNG TO IXRNG
        SETPXL(IX,IY)
```

Or just plot the endpoints instead of filling in the whole disk.

```
FOR IY = 100 TO -100 BY -1
     IX = SQRT(10000 - IY * IY)
     SETPXL(-IX,IY)
     SETPXL(IX,IY)
```

This leaves unsightly gaps near the top and bottom.

(12) Various Approximations to SQRT

Make a polynomial or rational polynomial approximation to $\sqrt{10000 - y^2}$ that is good for the range –100 . . . 100. Evaluate it with forward differences.

(13) Driving *x* Away

Let's do only the upper-right quarter of the circle and follow the point [0, 100]. For each downwards step in y, we move to the right some distance in x. Start at the x that's left over from last time and step it to the right until it hits the circle, leaving a trail of pixels behind.

```
IX = 0
FOR IY = 100 TO 0 BY -1
     IX2MAX = 10000 - IY * IY
     DO UNTIL (IX * IX) > IX2MAX
          SETPXL(IX,IY)
          IX = IX + 1
```

Calculation of IX2MAX and IX^2 can be done by forward differences.

```
IX = 0
IX2 = 0;      IDX2 = 1;        IDDX2 = 2
IX2MAX = 0;  IDX2MAX = 199;  IDDX2MAX = -2
FOR IY = 100 TO 0 BY -1
     DO UNTIL IX2 > IX2MAX
          SETPXL(IX,IY)
          IX = IX + 1
```

```
        IX2 = IX2 + IDX2;  IDX2 = IDX2 + IDDX2
    IX2MAX  = IX2MAX + IDX2MAX
    IDX2MAX = IDX2MAX + IDDX2MAX
```

This still has a few problems, but we won't pursue them because the next two algorithms are so much better.

(14) Bresenham

The above begins to look like Bresenham's algorithm—this is the top of the line in pixel-oriented circle algorithms. It endeavors to generate the best possible placement of pixels describing the circle with the smallest amount of (integer) code in the inner loop. It operates with two basic concepts.

First, the curve is defined by an "error" function. For our circle, this is $E = 10000 - x^2 - y^2$. For points exactly on the circle, $E = 0$. Inside the circle $E > 0$; outside the circle $E < 0$.

Second, the current point is nudged by one pixel in a direction that moves "forward" and in a direction that minimizes E. We will consider just the octant of the circle from (0, 100), moving to the right by 45 degrees. At each iteration we will choose to move either to the right (R), $x = x + 1$, or diagonally (D), $x = x + 1$ and $y = y - 1$.

The nice thing about this is that the value of E can be tracked incrementally. If the error at the current $[x, y]$ is

$$E_{cur} = 10000 - x^2 - y^2$$

then an R step will make

$$\begin{aligned} E_{new} &= 10000 - (x+1)^2 - y^2 \\ &= E_{cur} - (2x+1) \end{aligned}$$

and a D step will make

$$\begin{aligned} E_{cur} &= 10000 - (x+1)^2 - (y-1)^2 \\ &= E_{cur} - (2x+1) + (2y-1) \end{aligned}$$

Now, for the octant in question,

$$\begin{aligned} x &\le y \\ x &\ge 0 \\ y &> 0 \end{aligned}$$

So an R step subtracts something from E, and a D step adds something to E. The naive version of the algorithm determines which way to go by looking at the current sign of E, always striving to drive it towards its opposite sign.

```
IX = 0;  IY = -100
IE = 0
WHILE IX <= IY
     IF (IE < 0)
          IE = IE + IY + IY - 1
          IY = IY - 1
     IE = IE - IX - IX - 1
     IX = IX + 1
     SETPXL(IX,IY)
```

(15) Improved Bresenham

We can do better. What we want to do at each step is actually to pick the direction that generates the smallest size error, $|E|$. We want to look ahead at the two possible new error values:

$$E_R = E - (2x+1)$$
$$E_D = E - (2x+1) + (2y-1)$$

and test the sign of $|E_D| - |E_R|$. The trick is to avoid calculating absolute values. Look at the possibilities in Table 1.1.

Now comes the tricky part. We can define a "biased" error from the (+ –) case

$$G = E_D + E_R$$

and use this as the test for *all three* cases. This works for the following reason. In the (+ +) case, $|E_D| - |E_R| = 2y - 1$ is positive, but so is G. In the (– –) case, $|E_D| - |E_R| = -(2y - 1)$ is negative, but so is G.

G can be calculated incrementally, just like E was. The new values due to R and D steps are

$$G_R = G - 4x - 6$$
$$G_D = G - 4x + 4y - 10$$

Table 1.1 *Possible signs of E_D and E_R*

E_D	E_R	$\lvert E_D\rvert - \lvert E_R\rvert$
+	+	$E_D - E_R = 2y - 1$; always positive
+	–	$E_D + E_R$
–	+	Can never happen
–	–	$-E_D + E_R = -(2y - 1)$; always negative

Further, the increments to G can be calculated incrementally. You get the idea by now . . .

```
IR = 100
IX = 0; IY = IR
IG = 2 * IR - 3
IDGR = -6; IDGD = 4 * IR - 10
WHILE IX <= IY
    IF IG < 0
        IG = IG + IDGD    "go diagonally"
        IDGD = IDGD - 8
        IY = IY - 1
    ELSE
        IG = IG + IDGR    "go right"
        IDGD = IDGD - 4
    IDGR = IDGR - 4
    IX = IX + 1
    SETPXL(IX,IY)
```

Whew!

Why Bother?

Why is all this interesting—aside from the pack rat joy of collecting things?

Well, you can certainly use the algorithms to optimize your circle-drawing programs, if you're into circles. Each algorithm has its own little niche in the speed/accuracy/complexity trade-off space. Sometimes economy is misleading—the `SETPXL` routine often gobbles up any time you saved being clever with Bresenham's algorithm. Let's face it: unless there's something very time-critical, I usually use Algorithm 1 because it's easiest to remember.

The really interesting thing about all these is the directions they lead you when you try to generalize them. Algorithms 2 and 7 lead to general polynomial curves. Algorithm 4 leads to iterated function theory. Algorithm 5 leads to the CORDIC method of function evaluation.[3] Algorithm 11 has to do with rendering spheres. (I wonder what happens to Algorithm 15 if you use some other simple functions of x and y for G, G_R, and G_D.) In fact, although many of these algorithms look quite similar when applied to circles, their generalizations lead to very different things.

3 Kenneth Turkowski, Anti-aliasing through the use of coordinate transformations, *ACM Transactions on Graphics* 1(3):215–234, July 1982.

It sort of shows the underlying unity of the universe. Maybe the Greeks had something there.

That's all for now. If you know of any essentially different circle-drawing algorithms, let me know.

Ellipses

A reader named D. Turnbull wrote and asked about the generalization of Bresenham's algorithm to ellipses. Here's my reply.

The problem of ellipses is in fact also applicable to the problem of circles if you have non-square pixels. The Bresenham algorithm can be extended to any conic section by roughly the following process. The error function for an arbitrary conic section in an arbitrary orientation is

$$E = ax^2 + bxy + cy^2 + dx + ey + f$$

As usual, at each step you advance one pixel in the x or y direction or in both directions. If, for example, you step to the right one unit in x, the error becomes

$$\begin{aligned} E_R &= a(x+1)^2 + b(x+1)y + cy^2 + d(x+1) + ey + f \\ &= E + (2ax + by + d) + a \end{aligned}$$

Likewise, after an upwards vertical step, the error becomes

$$E_U = E + (bx + 2cy + e) + c$$

Let's give names to the values in parentheses.

$$F = 2ax + by + d$$
$$G = bx + 2cy + e$$

A negative step in x or y requires subtraction of F or G.

These increments can themselves be calculated incrementally. A step to the right changes F and G by

$$F_R = F + 2a$$
$$G_R = G + b$$

A step upwards changes F and G by

$$F_U = F + b$$
$$G_U = G + 2c$$

The actual loop then incrementally keeps track of E, F, and G as the point crawls around the curve. The decision of whether to increment or

decrement x and/or y comes from the signs of F and G. Note that all arithmetic is still integer arithmetic as long as the coefficients a through f are integers. If they aren't, you can make them integers by scaling them all by the same factor.

A simple version of this technique that steps only horizontally or vertically is described in Cohen's paper.[4] A somewhat better version that steps diagonally and optimizes the magnitude of E is described in Jordan et al.'s paper.[5] Ideally, we would like to reduce the inner loop calculation to an absolute minimum. The Jordan algorithm requires explicit testing of absolute values of the error in the three potential step directions (horizontal, vertical, and diagonal). It was possible to get rid of most of this for the final circle algorithm in this chapter. I don't know if it's possible for the general case here, though.

4 E. Cohen, A method for plotting curves defined by implicit equations, *SIGGRAPH '76 Confere.* (New York: ACM), pages 263–265.

5 B. W. Jordan et al., An improved algorithm for generation of nonparametric curves, *IEEE Transacti puters,* C-22(12):1052–1060, December 1973.