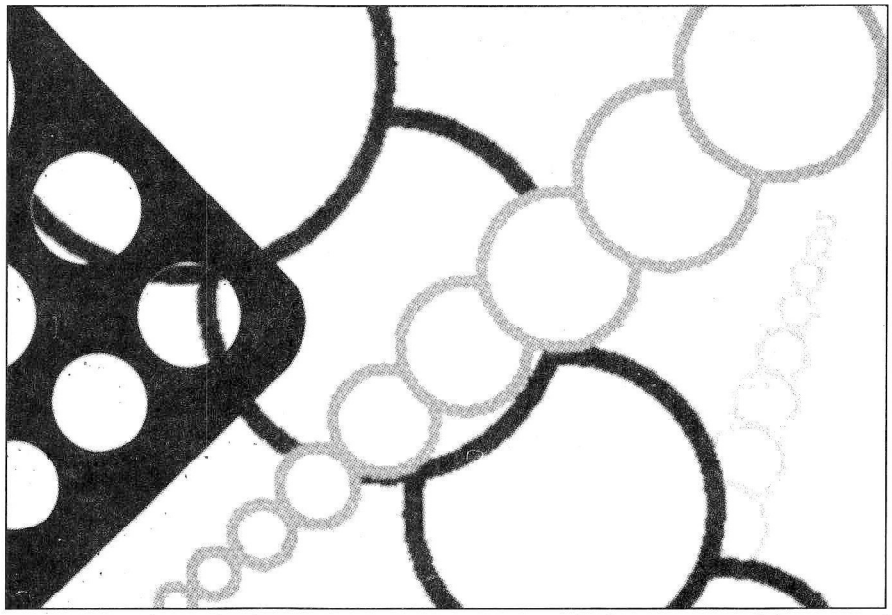
Circles and the Digital Differential Analyzer

This drawing method belongs in every graphics library

Tim Paterson

Drawing circles has long been a topic of discussion in the pages of *DDJ*. Earlier this year, Robert Zigon ("Parametric Circles," *DDJ*, January 1990) presented a mathematical analysis in which he simplified the brute force approach that would normally require either lots of sines and cosines, or lots of squares and square roots.

Unfortunately, Mr. Zigon got on the wrong track by switching to the polar coordinate system for most of his analysis. His result was an improvement, but still required four floating-point multiplications per point drawn. His answer also required the calculation of a sine and a cosine once for each circle. Another serious drawback was the fact that his equations computed points at fixed angles, which does not translate well into an unbroken curve of screen pixels.



I was sure there was a better way, so I looked into my filing cabinet and pulled out the folder labeled "Circles." In it I found the May 1983 issue of *DDJ*! Daniel Lee had written "A Fast Circle Routine" and had reached a simpler conclusion than Mr. Zigon's.

Both Messrs. Zigon and Lee were calculating the next point on the circle in relation to the last point. If you've had calculus, this sounds like a classic case of computing "the change in y, given a small change in x." In other words, a derivative is called for. The basic equations are shown in Figure 1. In other words, as Mr. Lee put it, "the change in y, given a small change in x, is simply the negative of the ratio of x to y."

The obvious way to use this fact is to compute point n+1 from point n as follows:

$$x_{n+1} = x_n + 1$$
$$y_{n+1} = y_n - x/y$$

As written, these would appear to need to be floating-point operations. However, fixed-point arithmetic would work as well. For example, x and y could be stored as 32-bit numbers, with the bi-

Tim is the original author of MS-DOS, Version 1.x, which he wrote in 1980-82 while employed by Seattle Computer Products and Microsoft. He was also founder of Falcon Technology, which became part of Phoenix Technologies, the ROM BIOS maker. Tim can be reached c/o DDJ.

(continued from page 30)
nary point in the middle: A 16-bit integer part and a 16-bit fraction part. Only the high 16 bits would be used for plotting points. The low bits are needed to keep the circle accurate. Mr. Lee's version wasn't quite this simple, as he scaled x and y by 1000, but it's the thought that counts.

In order to plot an unbroken curve, two consecutive points cannot be further apart than one pixel in either the x or y direction. In the equations above, this is always true for x, of course, but is true for y only as long as $x <= y$. This means the technique can be used on only one octant (one eighth) of the circle at a time. However, circles being so symmetrical, computing one octant is all that's needed to plot the whole thing. For each point computed in one octant, eight points are plotted: Four in all combinations of positive and negative coordinates, and the same four again but with the x and y values interchanged.

The Digital Differential Analyzer

Mr. Lee's approach still requires a division operation (albeit integer division), and I was sure there was a better way. I found in my folder some handwritten notes on the algorithm used by the run-time library of the Microsoft Basic compiler — circa 1981, when I was part of the team porting it from the 8080 to the 8086. This algorithm uses unscaled integers only, with no operations more difficult than 16-bit adding and shifting. I have since heard this technique called the "digital differential analyzer," or DDA. The DDA is the standard method for drawing both straight lines and circles for every graphics library.

Let's start trying to understand the DDA by examining the case of straight lines. Suppose we have two points that we are drawing a line between, (x_1, y_1) and (x_2, y_2). First we compute two constants, *Dx* and *Dy*:

$$Dx = x_2 - x_1$$
$$Dy = y_2 - y_1$$

Dx and *Dy* are simply how far we have to go in each direction to get from the first point to the second. If we were going to use an approach such as Mr. Lee's, we could compute points on the line like this:

$$x_{n+1} = x_n + 1$$
$$y_{n+1} = y_n + Dy/Dx$$

But *Dy/Dx* is not an integer (usually), so at least fixed-point arithmetic using scaled integers would be required.

The DDA is the standard method for drawing both straight lines and circles for every graphics library

So far, we've been looking at this in fairly "analog" (as opposed to "digital") terms. Yes, we've been incrementing x by whole pixels, but y has needed to take on noninteger values. The actual drawing operations, however, are interested only in whole numbers for pixel coordinates. Let's change our thinking so that y stays a whole number, and we'll keep another variable that will tell us when to increment it.

Basically y should get incremented when we've added *Dy/Dx* enough times to equal one more. This is the same as adding in one more *Dy* for each x pixel until the total reaches *Dx*. The C code in Example 1 shows the calculation of one additional point on the line. That code would appear in a loop repeated to draw the whole line.

Looking more closely at Example 1, *sum* would have been initialized to $-Dx/2$ (or better still, $-Dx >> 1$). For each pixel in the x direction, we add *Dy* to *sum*. When *sum* crosses over to being positive, that means we're ready to move one y pixel, and we subtract *Dx* from *sum*. *sum* gets *Dy* added for each x pixel, *Dx* to subtract for each y pixel: It almost amounts to division by repeat subtraction.

But we're really here to look at circles. In the case of lines, we were adding the constant *Dy/Dx* to y, as discussed earlier for circles, we need to add x/y. That doesn't sound much harder, and, in fact, it's not. Drawing a circle is just as easy as drawing a line. Example 2 has the C code for one pass through the circle's pixel loop, with a few added features for more accuracy.

You should have expected x to be added to *sum* each time through the loop; instead we have $2x + 1$. The reason for this is accuracy. Unlike the straight line case, the derivative we're using as the basis for our technique, y/x, is continuously varying. As we move from one x pixel to the next, which value of y/x do we use — the one before or the one after the move? The best answer is to use the value in between. That is, instead of adding x, or adding $x+1$, we can add the average: $(2x+1)/2$. But there's no reason to divide by two as long as we do it the same way for the y pixel. Note that these lines could also be coded as:

```
sum + = x;
x++;
sum + = x;
```

and

```
sum - = y;
y - -;
sum - = y;
```

By using the value of x (or y) both before and after incrementing it, we're clearly showing how we want the effect of the midpoint.

The initial conditions for the loop would be *sum* and x set to zero, and y set to the radius of the circle. But this won't work quite right. Starting *sum* at zero means y will be kept less than or equal to its perfect value on the circle for any given x coordinate. That is, we'll plot only points that fall exactly on or inside the circle. What we'd like is to plot the nearest point to the circle, whether it be inside or outside it. For the straight line case, we initialized *sum* to $-Dx/2$, which effectively shifted the y coordinate one-half a pixel. But again because the derivative for the circle is varying instead of constant, there is no value that can be used to initialize *sum* to get the same effect.

The solution is to explicitly shift the

$$r^2 = x^2 + y^2, \text{ or}$$
$$y = \sqrt{r^2 - x^2}$$
$$\frac{dy}{dx} = -\frac{x}{\sqrt{r^2 - x^2}} = -\frac{x}{y}$$

Figure 1: *Equation for a circle and ts derivative*

```
sum += Dy;
x++;
if (sum > 0)
{
   sum -= Dx;
   y++;
}
```

Example 1: *The calculation of one additional point on the line*

```
sum += 2*x + 1;
x++;
if (sum > 0)
{
   sum -= 2*y - 1;
   y--;
}
```

Example 2: *C code for one pass through the circle's pixel loop*

(continued from page 32)
y coordinate by half a pixel as we plot the points. That is, instead of having the error from the perfect value for *y* range from 0 to −1 pixel, we'll add one half, and the error will be 1/2 to −1/2 pixel. To do this with integers is simple: Double the radius, so that adding one is equivalent to half a pixel. Then we'll plot only even values of *x*, using the points *x*/2 and (*y*+1)/2.

To add aspect ratio to our circle drawing algorithm requires a little bit of work up front

Example 3 shows a working C function that plots a circle. It calls an additional function *plot8( )*, which actually plots the point generated in all eight octants. Note that all multiplications and divisions by two have been replaced with shift operations << and >>, respectively.

Ellipses and Aspect Ratio

Circles plotted with any of these techniques are round only when the dimensions of the graphics controller, in pixels, correspond to the aspect ratio of the video monitor. Standard video monitors have a 4:3 aspect ratio, so VGA 640 × 480 and super VGA 800 × 600 resolutions produce a "square" pixel (and a round circle). Other typical graph-

```
void circle(int radius)
{
   int x,y,sum;

   x = 0;
   y = radius << 1;
   sum = 0;

   while (x <= y)
   {
      if ( !(x & 1) )
           /* plot if x is even */
        plot8( x >> 1, (y+1) >> 1);
      sum += (x << 1) + 1;
      x++;
      if (sum > 0)
      {
         sum -= (y << 1) - 1;
         y--;
      }
   }
}
```

Example 3: *C function that plots a circle*

ics resolutions (such as CGA, EGA, and Hercules) do not match the 4:3 screen aspect ratio, and will display an ellipse instead of a circle.

Ellipses have their place, too. If we could have arbitrary control over aspect ratio of the circle itself, then we could draw circles or ellipses for every graphics system. And, in fact, any graphics library will have this capability, although it is often expressed in different ways. The CIRCLE statement of Microsoft Basic, for example, requires you to specify the center, the radius, and the aspect ratio. On the other hand, Microsoft Windows and the Microsoft C graphics library want you to specify a "bounding rectangle": You get the biggest ellipse (or circle) that will fit into the box.

To add aspect ratio to our circle drawing algorithm requires a little bit of work up front, plus a single integer multiplication for each point plotted. The simple-minded approach would be to multiply the y coordinate value by the aspect ratio. However, if the aspect ratio is greater than one, this could cause gaps in the arc. In that case, we will multiply the x coordinate by the reciprocal of the aspect ratio. Thus we will always be multiplying by less than one, compressing points on the arc and keeping it unbroken.

Always multiplying by a number less than one has the further advantage of giving us a limit on the range of the number. We can scale the number by up to 16 bits and still store it as a 16-bit number. This is done by *SetAspect( )* in the complete circle drawing program shown in Listing One, page 96. Then as points are plotted, the scaled aspect ratio is multiplied by the x or y coordinate, as appropriate, with the result shifted right 16 bits. Individual points are plotted using the *_setpixel( )* function of the Microsoft C graphics library.

So we now have a complete algorithm for generating circles or ellipses of any aspect ratio. Of course, in a real graphics library, assembly language would be used for maximum speed. And one feature still missing is the ability to draw only an arc, rather than the complete ellipse. The algorithm for a partial arc is no different from what we have so far, but adds a check to see which points we actually want to plot, and which will be discarded.

DDJ

(Listing begins on page 96.)