An efficient circle drawing algorithm

This is a documentation of a lecture of mine, which I have given several times since 1997 to motivate the use of mathematics in programming. I thought it was about time I wrote something down. The problem of drawing a straight line is treated in most computer graphics textbooks, mostly by presenting the rightfully famous Bresenham line algorithm, but the circle variant is not often seen in the literature. I make absolutely no claim of having invented this algorithm, but it seems to have fallen into oblivion. The only place I have seen this kind of stuff lately is in the nerdy but highly entertaining book "A trip down the graphics pipeline" by Jim Blinn, where he presents the Bresenham circle algorithm along with a plethora of other circle algorithms.

What I present here is a slightly different algorithm, avoiding a quirk in the Bresenham derivation which I find hard to follow. The two algorithms are almost identical. In fact, they plot exactly the same pixels except for a few rare cases, and where they differ, both can be said to be correct.[1]

Stefan Gustavson (stegu@itn.liu.se) 2003-08-20

The problem

We want to design a highly efficient algorithm to draw a circle outline on a pixel-based computer display, using only the primitive function of setting a single pixel.

Moving into pixel space

First, we need to make the mental leap of looking at the problem from the perspective of a computer program. The problem of which pixels to plot boils down to three basic steps:

1. Pick a good pixel to start the drawing.
2. Decide which pixel to plot next.
3. Repeat from step 2 until the circle is done.

The algorithm is, not surprisingly, a loop. Let the circle radius be R, and let's assume we are plotting the circle with its midpoint at (0,0). Good starting points are (R, 0) or (0, R). Let's pick (0, R).

We then observe that a circle is a highly symmetric shape. If we plot all pixels in the second octant, from the y-axis going right to the line $x = y$, we can plot the rest of the circle by mirroring the pixel coordinates, like so:

```
plot(x,y)   // Second octant
plot(x,-y)  // Seventh octant
plot(-x,y)  // Third octant
plot(-x,-y) // Sixth octant
plot(y,x)   // First octant
plot(y,-x)  // Eighth octant
plot(-y,x)  // Fourth octant
plot(-y,-x) // Fifth octant
```

The criterion for when we leave the second octant and step into the first octant is conveniently expressed as "stop when $x > y$".

So, that takes care of step 1 and 3 of the loop, and the actual plotting. Now for step 2, which is the difficult part.

---

1. Actually, both are very slightly wrong, but in different ways. The small errors in both algorithms can be attributed to an approximation of the distance from a position on the pixel grid to the circle outline. Finding the true Euclidean distance involves taking the square root of a second order polynomial in x and y. Both algorithms instead use the value of a second order polynomial directly, without the square root, and the actual polynomials used for the two algorithms are ever so slightly different.

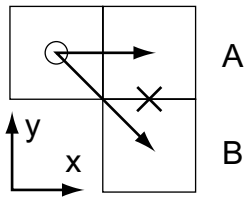
Tracing the circle

To trace the outline of the circle, we observe that the circle equation $x^2 + y^2 = R^2$ gives us a convenient way of knowing whether we are inside or outside of the circle. The function $f(x, y) = x^2 + y^2 - R^2$ is negative inside the circle, positive outside, and zero on the circle.

In the second octant, the slope of the curve is always in the range 0 to -1, so a pixel train approximating the curve should be built from only two elementary steps:

A) Take one step to the right: $x_{i+1} = x_i + 1$, $y_{i+1} = y_i$

B) Take one step to the right and one step down:
$x_{i+1} = x_i + 1$, $y_{i+1} = y_i - 1$



To decide which of the two candidates for the next pixel is the closest, A or B, we check whether the midpoint between the two pixels, $(x_i + 1, y_i - 1/2)$ is outside or inside the circle, which means that we want to evaluate the function $f(x, y)$ at that point. We will use this value a lot henceforth, so we give it a name of its own, d_i:

$$d_i = f(x_i + 1, y_i - 1/2)$$

This trick gives our algorithm its name: "the midpoint algorithm". If $d_i > 0$, the midpoint is outside the circle and pixel B is closest, and if $d_i < 0$, the midpoint is inside the circle and pixel A is closest. If $d_i = 0$, the midpoint is precisely on the circle, and either case could be picked.

At this stage, we can write down an algorithm that works, even if it is still inefficient:

1. Set $x_0 = 0$, $y_0 = R$, $i = 0$.
2. Plot pixels (x_i, y_i), $(x_i, -y_i)$, $(-x_i, y_i)$, $(-x_i, -y_i)$, (y_i, x_i), $(y_i, -x_i)$, $(-y_i, x_i)$, $(-y_i, -x_i)$.
3. If $(x_i + 1)^2 + (y_i - 1/2)^2 - R^2 < 0$, set $y_{i+1} = y_i$, else set $y_{i+1} = y_i - 1$
4. Set $x_{i+1} = x_i + 1$
5. If $x_{i+1} < y_{i+1}$, increment i and repeat from 2, else stop.

This algoritm has a serious flaw: it requires too much calculation for the test in the inner loop. Taking two squares for each pixel is quite costly. Fortunately for us, there is a remedy.

Forward differences

We can get rid of most of the calculations in the inner loop quite easily by a simple observation: the loop calculates a value d_i for each midpoint, and then moves on to an adjacent pixel. For case A above, the next midpoint to test is $(x_i + 2, y_i - 1/2)$, and for case B, the next midpoint is $(x_i + 2, y_i - 3/2)$. For the two cases, we can easily express d_{i+1} in terms of d_i:

$$d_i = (x_i + 1)^2 + (y_i - 1/2)^2 - R^2 = x_i^2 + 2x_i + 1 + y_i^2 - y_i + \frac{1}{4} - R^2$$

For case A, we have:

$$d_{i+1} = (x_i + 2)^2 + (y_i - 1/2)^2 - R^2 = x_i^2 + 4x_i + 4 + y_i^2 - y_i + \frac{1}{4} - R^2 = d_i + 2x_i + 3$$

and similarly for case B, we have:

$$d_{i+1} = (x_i + 2)^2 + (y_i - 3/2)^2 - R^2 = x_i^2 + 4x_i + 4 + y_i^2 - 3y_i + \frac{9}{4} - R^2 = d_i + 2x_i - 2y_i + 5$$

Thus, we can now express our algorithm in a more efficient manner as follows:

1. Set $x_0 = 0$, $y_0 = R$, $d_0 = R^2 - 5/4$, $i = 0$.
2. Plot the pixels at (x_i, y_i) etc.
3. If $d_i < 0$, set $y_{i+1} = y_i$ and $d_{i+1} = d_i + 2x_i + 3$,
   else set $y_{i+1} = y_i - 1$ and $d_{i+1} = d_i + 2x_i - 2y_i + 5$
4. Set $x_{i+1} = x_i + 1$.
5. If $x_{i+1} < y_{i+1}$, increment i and repeat from 2, else stop.

The only operations left in the inner loop is now a few additions. (The multiplications by 2 are easily implemented as bit shift operations, so they don't count as multiplications.) This is a lot faster than before, and we could implement this in some programming language and expect a fast performance. We can, however, take our simplifications one step further, and for the sake of completeness, we will.

Second order forward differences

In each step of the loop above, we add either $2x_i + 3$ or $2x_i - 2y_i + 5$. Observing that we in the immediately preceding step added $2x_{i-1} + 3$ or $2x_{i-1} - 2y_{i-1} + 5$, and that we know that $x_i = x_{i-1} + 1$ and either $y_i = y_{i-1}$ or $y_i = y_{i-1} - 1$, we can save our increments to d_i in two additional variables and save some work at the cost of a minimal amount of extra storage.

Introduce two more variables, dA_i and dB_i, and we can rewrite the algorithm as follows:

1. Set $x_0 = 0$, $y_0 = R$, $d_0 = R^2 - 5/4$, $dA_0 = 3$, $dB_0 = 5 - 2R$, $i = 0$.
2. Plot the pixels at (x_i, y_i) etc.
3. If $d_i < 0$, set $y_{i+1} = y_i$, $d_{i+1} = d_i + dA_i$, $dB_{i+1} = dB_i + 2$
   else set $y_{i+1} = y_i - 1$ and $d_{i+1} = d_i + dB_i$, $dB_{i+1} = dB_i + 4$
4. Set $x_{i+1} = x_i + 1$ and $dA_{i+1} = dA_i + 2$.
5. If $x_{i+1} < y_{i+1}$, increment i and repeat from 2, else stop.

The final algorithm is conspicuously devoid of complicated arithmetic, despite the fact that it draws a very acccurate rendition of a quadratic curve. Neat, isn't it?

One last speedup: to get rid of the fraction $5/4$ and use ony integer arithmetic, we can multiply everything by 4. Some pseudo-code to help you follows. Now go ahead and implement this in your programming language of choice and see that it really works. Good luck!

```
x=0; y=R; d=5-4*R;
dA=12; dB=20-8*R;
while (x<y)
  plot(x,y);
  if(d<0)
    d=d+dA; dB=dB+8;
  else
    y=y-1;
    d=d+dB; dB=dB+16;
  end if
  x=x+1; dA=dA+8;
end while
```