J. Indian Inst. Sci., Nov.–Dec. 2006, **86**, 617–623

An alternative approach to circle drawing

BIMAL KUMAR RAY
School of Computing Sciences, Vellore Institute of Technology, Vellore 632 014, India.
email: raybk_2000@yahoo.com

Received on April 20, 2006; Revised on December 4, 2006.

Abstract

This paper presents an alternative approach to drawing circle. The algorithm proposed here draws circle using integer arithmetic only. The number of arithmetic operations required by the proposed algorithm is comparable to that required by the existing mid-point algorithm.

Keywords: Circle drawing, integer arithmetic, mid-point algorithm.

1. Introduction

One of the fundamental graphics primitives is circle. One needs to have an efficient algorithm for drawing circle, especially for animated display. In the absence of efficient algorithm the animated display will deviate from realistic view. The most famous algorithm for drawing circle is the mid-point algorithm [1–3]. The other algorithms that deal with circle drawing can be found in [4–9].

In this paper, an alternative integer-arithmetic-based circle drawing algorithm is proposed. The algorithm is based on the knowledge of the fact that semi-circular angle is 90°. At the first stage, a floating-point algorithm is developed, which is modified to derive an integer arithmetic algorithm. In Section 2, the parametric equation of a circle based on the notion of *semi-circular angle* is deduced and a trivial approach to generate circle is suggested and discussed. In Sections 3 and 4, two improvements over the trivial approach are proposed and finally in Section 5 the final integer arithmetic algorithm is presented.

2. Background

Without any loss of generality we consider a circle with center at the origin and radius r. Referring to Fig. 1, points A and B have coordinates $(-r, 0)$ and $(r, 0)$, respectively. If P be an arbitrary point on the circumference of the circle then from elementary geometry it follows that the segments AP and BP are perpendicular. Let point P has coordinates (x, y). If we take m as the slope of the segment AP then the slope of the segment BP which is perpendicular to the segment AP is $-1/m$. The equation of the segment AP is $y = mx + mr$ and that of the segment BP is $x + my = r$. Solving these equations we get the parametric equations of a circle with center at the origin and radius r as

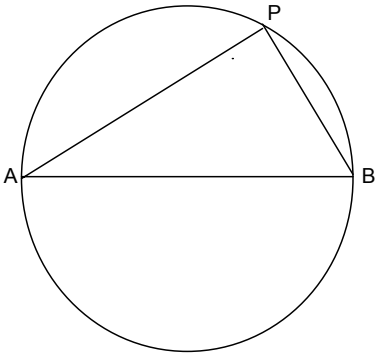

FIG. 1. Semi-circular angle is 90°. AB is along a diameter of the circle and P is an arbitrary point on the circle. The angle APB is 90°.

$$x = r\{(1 - m^2)/(1 + m^2)\} \text{ and } y = 2mr/(1 + m^2).$$

We can use these equations to draw a circle using an appropriate step size of m. We can exploit symmetry and generate the points for only one octant of the circle. For circle with center at an arbitrary point, one need to apply translation on the points generated, the translation parameters being the x and the y coordinates of the center of the circle. When one uses the parametric equation $x = r\cos\theta$ and $y = r\sin\theta$ for drawing circle, the step size of $\Delta\theta = \min(0.01, 1/(3.2r))$ is used. So the step size of Δm is equal to $\min(\tan(0.005), \tan(1/(6.4r))$. Starting with the initial value of $m = 0$ and giving increment to m by an amount of Δm, one can write down an algorithm for computing the pixel coordinates (x, y) of the circle, using the formulae for x and y stated above. But this algorithm results in a large number of overdraws of pixels.

3. First revision

Using the last two expressions for x and y, it is found that

$$x = r - y^2/(x + r).$$

The slope of the circle is negative and sharp ($-\infty < m \le -1$) over the region

$$\boldsymbol{R} = \{(x, y) \mid y \ge 0 \text{ and } x \ge y\}.$$

So we can increment y and compute x using

$$x = r - y^2/(x + r).$$

The C source code for the procedure of drawing eight reflections of pixels is as given below. The variables x and y hold the pixel coordinates generated by the algorithm and (*xc, yc*) hold the coordinates of the center of the circle and the variable *intensity* is used to hold the intensity/color of the circle boundary pixel. The routine named DrawMirrorPixelRounded( ) draws four reflections of pixel and DrawMirrorPixelsRounded( ) draws the eight reflections of pixels. This routine involves floating point arithmetic. The routines named DrawMirrorPixel( ) and DrawMirrorPixels( ) perform similar jobs for integer arithmetic algorithm. The system-dependent routine putpixel(x, y, *intensity*) turns on a pixel at the location (x, y) with an intensity value *intensity*.

```
void DrawMirrorPixelRounded(float xc, float yc, float x, float y, int intensity)
{
   putpixel (int(xc + x + 0.5), int(yc + y + 0.5), intensity);
   putpixel (int(xc – x + 0.5), int(yc + y + 0.5), intensity);
   putpixel (int(xc + y + 0.5), int(yc + x + 0.5), intensity);
   putpixel (int(xc + y + 0.5), int(yc – x + 0.5), intensity);
   return;
}

void DrawMirrorPixelsRounded(float xc, float yc, float x, float y, int intensity)
{
   putpixel (int(xc + x + 0.5), int(yc + y + 0.5), intensity);
   putpixel (int(xc – x + 0.5), int(yc + y + 0.5), intensity);
   putpixel (int(xc + x + 0.5), int(yc – y + 0.5), intensity);
   putpixel (int(xc – x + 0.5), int(yc – y + 0.5), intensity);
   putpixel (int(xc – y + 0.5), int(yc – x + 0.5), intensity);
   putpixel (int(xc – y + 0.5), int(yc + x + 0.5), intensity);
   putpixel (int(xc + y + 0.5), int(yc – x + 0.5), intensity);
   putpixel (int(xc + y + 0.5), int(yc + x + 0.5), intensity);
   return;
}

void DrawMirrorPixel(int xc, int yc, int x, int y)
{
   putpixel (xc + x , yc + y, intensity);
   putpixel (xc – x, yc + y, intensity);
   putpixel (xc + y, yc + x, intensity);
   putpixel (xc + y, yc – x, intensity);
   return;
}

void DrawMirrorPixels(int xc, int yc, int x, int y)
{
   putpixel (xc + x , yc + y, intensity);
   putpixel (xc – x, yc + y, intensity);
   putpixel (xc + x, yc – y, intensity);
   putpixel (xc – x, yc – y, intensity);
   putpixel (xc – y, yc – x, intensity);
   putpixel (xc – y, yc + x, intensity);
   putpixel (xc + y, yc – x, intensity);
   putpixel (xc + y, yc + x, intensity);
   return;
}
```

The C source code for drawing circle is as given below. The variable r holds the radius of the circle. The algorithm does not result in overdraw of pixels but needs to compute the full value of x for every increment of y and the quality of circle for radius $r = 1$ is poor (this has happened because x is computed without performing any test).

```
void DrawCircleOne (int xc, int yc, int r, int intensity)
{
  x = r;
  y = 0;
   DrawMirrorPixelRounded(xc, yc, x, y);
   while (x > y) {
      y = y + 1;
     x = r – y*y/(x + r);
      DrawMirrorPixelsRounded (xc, yc, x, y);
      }
   return;
}
```

4. Second revision

From $x = r - y^2/(x + r)$ it can be deduced that $x = r - (y^2/(x + r)) - (x^2 + y^2 - r^2)/(x + r)$ which reduces to $x = x - (x^2 + y^2 - r^2)/(x + r)$. If (x_k, y_k) be the pixel that had already been turned on and (x_{k+1}, y_{k+1}) be the next pixel to be turned on then

$$x_{k+1} = x_k - (x_k^2 + y_k^2 - r^2)/(x_k + r) = x_k - d_k$$

where $d_k = (x_k^2 + y_k^2 - r^2)/(x_k + r)$.

We note that $y_{k+1} = y_k + 1$ but $x_{k+1} = x_k + 1$ if $d_k \geq 0.5$, otherwise $x_{k+1} = x_k$. This amounts to the test

$$2(x_k^2 + y_k^2 - r^2) - (x_k + r) \geq 0$$

which does not involve division and the computation can be done in integer arithmetic only. The C source code for the algorithm is presented below.

```
void DrawCircleTwo (int xc, int yc, int r, int intensity)
{
x = r;
  y = 0;
   DrawMirrorPixel(xc, yc, x, y);
   while (x > y) {
     y = y + 1;
      if (2*(x*x + y*y - r*r) – (x + r) ≥ 0)
  x = x – 1;
      DrawMirrorPixels(xc, yc, x, y);
   }
return;
}
```

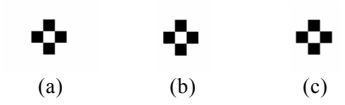

FIG. 2. Circle with radius $r = 1$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

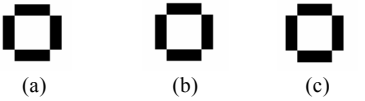

FIG. 3. Circle with radius $r = 2$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

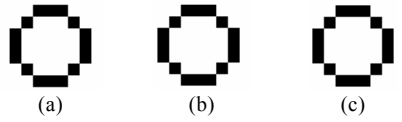

FIG. 4. Circle with radius $r = 3$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

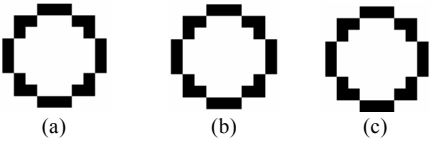

FIG. 5. Circle with radius $r = 4$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

5. Final algorithm

Let us take

$$g_k = 2(x_k^2 + y_k^2 - r^2) - (x_k + r)$$

so that

$$g_{k+1} = 2(x_{k+1}^2 + y_{k+1}^2 - r^2) - (x_{k+1} + r).$$

This leads to

$$g_{k+1} = g_k - 4x_k + 4y_k + 5 \text{ if } g_k \geq 0$$

and

$$g_{k+1} = g_k + 4y_k + 2 \text{ if } g_k < 0.$$

As y is incremented by 1, the increment in x will be decided by the decision parameter g. The C source code of the procedure is presented below. This code is more efficient than those presented in the last sections. It does not overdraw pixels and makes use of integer arithmetic only. The zoomed-in output of the algorithm for different values of radius is shown in Figs 2–10 along with the output of the mid-point algorithm.

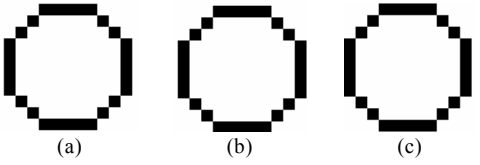

FIG. 6. Circle with radius $r = 5$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

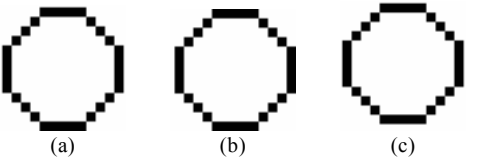

FIG. 7. Circle with radius $r = 6$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

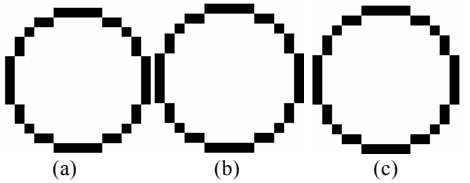

FIG. 8. Circle with radius $r = 7$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

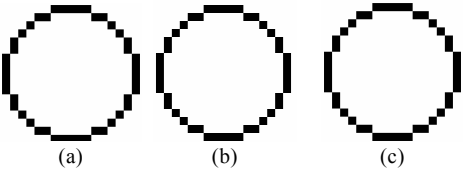

FIG. 9. Circle with radius $r = 8$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

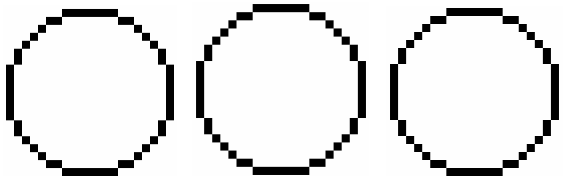

FIG. 10. Circle with radius $r = 10$. Output of (a) DrawCircleTwo, (b) DrawCircle, and (c) Mid-point algorithm.

```
void DrawCircle(int xc, int yc, int r, int intensity)
{
     x = r;
     y = 0;
     g = 2 – 2*r;
      DrawMirrorPixel(xc, yc, x, y);
      while(x > y) {
        y = y + 1;
          if(g < 0) g = g + 4*y + 2;
          else {
            x = x – 1;
            g = g + 4*(y – x) + 5;
          }
          DrawMirrorPixels(xc, yc, x, y);
      }
   return;
}
```

Another improvement that can be made on this algorithm is to introduce two new variables namely, u and v, defined by $u = 4*x$ and $v = 4*y$. The initial value of u and v are $2r$ and zero, respectively. The values of u and v can be updated by the formula $u = u - 4$ and $v = v + 4$. This results in avoidance of multiplication inside the main loop albeit at the cost of two additional variables u and v.

The number of arithmetic operations performed inside the loop of the last algorithm DrawCircle is equal to that required by the mid-point algorithm. This algorithm also draws each pixel once and once only (there is no overdraw of pixel by this algorithm).

Mean square error and the maximum error are computed to compare the quality of output between the mid-point algorithm and the last algorithm. The mean square error is the sum of squares of differences between the radius of the circle and the distance between the center and the pixel locations. The maximum error is the maximum of the absolute difference between the radius of the circle and the distance between the center and the pixel locations. The mean square error and the maximum error produced by the algorithm and those produced by the mid-point algorithm are comparable. Table I shows the results of error analysis for different values of radius. The runtime of mid-point algorithm is also compared with that of the present algorithm (Table II).

Table I
Error analysis

Radius	Mid-point algorithm		Present algorithm	
	Mean squares error	Maximum error	Mean squares error	Maximum error
20	0.048398	0.396078	0.048398	0.396078
15	0.046823	0.45978	0.046823	0.45978
10	0.049403	0.440307	0.049403	0.440307
9	0.06695	0.455996	0.070075	0.455997
8	0.042001	0.384227	0.042001	0.384227
7	0.0426441	0.291796	0.0426441	0.291796
6	0.035191	0.324555	0.035191	0.324555
5	0.039539	0.385165	0.039539	0.385165
4	0.056915	0.394449	0.056915	0.394449
3	0.013167	0.162278	0.013167	0.162278
2	0.027864	0.236068	0.027864	0.236068
1	0	0	0	0

Table II
Run time (in seconds)

Radius	Mid-point algorithm	Present algorithm
200	0.0100	0.0100
100	0.0050	0.0050
50	0.0025	0.0025
25	0.0010	0.0010

6. Conclusion

An alternative approach to draw circle is suggested. This approach makes use of the fact that the semi-circular angle is 90°. The algorithm involves integer arithmetic only. The computational load as well as the quality of circle drawn by this algorithm is comparable to that of the mid-point algorithm.

References

1. J. E. Bresenham, A linear algorithm for incremental digital display of circular arcs, *Commun. ACM*, **20**, 100–106 (1977).
2. J. D. Foley, A. van Dam, S. K. Feiner, and J. H. Hughes, *Computer graphics: Principles and practice*, Second edition, Addison-Wesley (1990).
3. Donald Hearn, and M. Pauline Baker, *Computer graphics*, *C Version*, Second edition, Prentice-Hall (1996).
4. Eric Andres, Discrete circles, rings and spheres, *Computers Graphics*, **18**, 695–706 (1994).
5. Xiaolin Wu, and Jon G. Rokne, Double-step incremental generation of lines and circles, *Computer Vision, Graphics, Image Processing*, **37**, 331–344 (1987).
6. Dan Field, Algorithms for drawing anti-aliased circles and ellipses, *Computer Vision, Graphics, Image Processing*, **33**, 1–15 (1986).
7. Akira Nakamura, and Kunio Aizawa, Digital circles, *Computer Vision, Graphics, Image Processing*, **26**, 242–255 (1984).
8. Marek Doros, Algorithms for generation of discrete circles, rings, and disks, *Computer Graphics Image Processing*, **10**, 366–371 (1979).
9. B. K. P. Horn, Circle generators for display devices, *Computer Graphics Image Processing*, **5**, 280–288 (1976).