**Bresenham Algorithm - Optimized Line Drawing Code**

We want to draw a line from (0,0) to (x1,y1), where 0<=y1<=x1, by setting one pixel per column. For example, if x=10, y=7, we get this:

Here are several versions, ending with Bresenham's algorithm. The point of this is to use the simplest possible operations.

1. Simple program:
   1. m=y1/x1;
   2. pixel(0,0);
   3. for(x=1;x<=x1;x++)
   4. { y=round(m\*x);
   5. pixel(x,y);
   6. }
2. Track the deviation of the current y from the perfect line. When the deviation gets too large, increment y, and decrement the deviation.
   1. m=y1/x1;
   2. d=0;
   3. pixel(0,0);
   4. y=0;
   5. for(x=1;x<=x1;x++)
   6. { d+= m;
   7. if (d>= 1/2)
   8. { d -= 1;
   9. y++; }
   10. pixel(x,y);
   11. }
3. Scale up m and d by 2x to convert fractions to integers. This is ok since they are used only in the *if*; and it still branches at the same times as before.
   1. m=2\*y1;
   2. d=0;
   3. pixel(0,0);
   4. y=0;
   5. for(x=1;x<=x1;x++)
   6. { d+= m;
   7. if (d>=x1)
   8. { d-= 2\*x1;
   9. y++;}
   10. pixel(x,y);
   11. }
4. Shift d by -x1 to remove a subtraction.
   1. m=2\*y1;
   2. d= -x1;
   3. pixel(0,0);
   4. y=0;
   5. for(x=1;x<=x1;x++)
   6. { d+= m;
   7. if (d>=0)
   8. { d-= 2\*x1;
   9. y++;}
   10. pixel(x,y);
   11. }
5. Assume that the pixels are stored in a framebuffer, and that pixelx y is stored at location fb+x\*maxy+y where fb is the address of the start of the framebuffer, and maxy is the max value of y in the framebuffer, i.e., 0<= y<= y1<= maxy. Then we get:
   1. m=2\*y1;
   2. d= -x1;
   3. \*fb=1;
   4. y=0;
   5. for(x=1;x<=x1;x++)
   6. { d+= m;
   7. if (d>=0)
   8. { d-= 2\*x1;
   9. y++;}
   10. \*(fb+x\*maxy+y)=1;
   11. }
6. Letting p be the current address, we can optimize.
   1. m=2\*y1;
   2. d= -x1;
   3. p=fb;
   4. \*p=1;
   5. for(x=1;x<=x1;x++)
   6. { d+= m;
   7. p+=maxy;
   8. if (d>=0)
   9. { d-= 2\*x1;
   10. p++;}
   11. \*p=1;
   12. }
7. The next problem is that modern computers use a pipelined architecture, where different stages of several instructions are processed in parallel. However, when there is a conditional, the computer can't tell what the next instructions are, and so is forced to flush the pipeline. Therefore we want to eliminate the conditional in the inner loop. The following assumes that the right shift is arithmetic, which it is on the Sun.

Mask is all ones when d>=0, all zeroes otherwise.

We also pulled 2\*x1 out of the loop.

m=2\*y1;

d= -x1;

p=fb;

\*p=1;

doublex1= x1<1; for(x="1;x<=x1;x++)" { d+="m;" p+="maxy;" mask="(~(d">>31));

d -= (doublex1&mask);

p += (1&mask);

\*p=1;

}

1. We can also offset x by x1 so that the termination condition is a test against 0.

This is all and well for lines originating at (0,0) but what about real lines. Here is an implementation of Bresenhams that works for slopes in the range 0 < m < 1:

{

        int xa = 10, ya = 7, xb = 200, yb = 150;

        int dx = abs(xa - xb), dy = abs(ya - yb);

        int p = 2 \* dy - dx;

        int twoDy = 2 \* dy, twoDyDx = 2 \* (dy - dx);

        int x, y, xEnd;

        if (xa > xb) {

                x = xb;

                y = yb;

                xEnd = xa;

        }

        else {

                x = xa;

                y = ya;

                xEnd = xb;

        }

        cgw->DrawPoint(x,y);

        while (x < xEnd) {

                x++;

                if (p < 0)

                        p += twoDy;

                else {

                        y++;

                        p += twoDyDx;

                }

                cgw->DrawPoint(x,y);

        }

}

**Bresenham Circle Algorithm**

Here is a formula to calculate the points on a acircle. 

y = yc(+/-)sqrt(r^2 - (xc - x)^2)

This is of course a very complex formula and not something we would use.

1. Taylor series for sqrt(1-x2), in the 2nd octant: 0<= x<= y, where the curve doesn't get too vertical. The following is expanded about the origin. The error at x=.71 is 0.0003.

y(x) = 1 - (1/2) x2 - (1/8) x4 - (1/16) x6 - (5/128) x8 - (7/256) x10 - (21/1024) x 12+ ...

This is popular but usually the wrong way.

1. If you must use Taylor, at least expand about the center of the interval:

z = x- 0.3536

y(z) = ( 0.9353967287- 0.3780214203 z- 0.6109173568 z2 - 0.2468897312 z3 - 0.2992736736 z4 - 0.2821915805 z5 - 0.3420828869 z6 - 0.4015387039 z7 - 0.5080487872 z8 - 0.6481416197 z9 - 0.8517407025 z10 - 1.133286086 z11 - 1.531581071 z12 - 2.091410948 z13+...

The error at x=.71 is 0.000 004. This is better, but there are more nonzero coefficients, which means that it is slower.

1. A Chebyshev series, which tries to minimize the maximum error over an interval beats a Taylor series:

y(x) = + 1.000000093 - 0.00002511816061 x - 0.4988901592 x2 - 0.01882886918 x3 + 0.03498610076 x4 - 0.7655182976 x5 + 2.101697099 x6 - 3.603814683 x7 + 3.259950635 x8 - 1.315059748 x9

This has a max error of 1.5\*10-7, which is much better than above, expecially as there are only 8 coefficients. We could get as good as either Taylor with many fewer terms than they.

1. Rotate the point (R,0) by Dt repeatedly (t is the angle). What should Dt be?
2. Approx the rotation matrix:

|  |  |
| --- | --- |
| cos t | sin t |
| - sin t | cos t |

1. approximately =

|  |  |
| --- | --- |
| 1 | t |
| t | 1 |

1. with everything multiplied by R. But this circle may grow since the determinant >1.
2. Improve the approx matrix so the determinant =1:

|  |  |
| --- | --- |
| 1 | t |
| t | 1-t2 |

1. Binomial theorem on sqrt(1-x2). Does this do the same as Taylor expanded about 0?
2. Here are two parametric ways to draw circles.
   1. x = R cos t  
      y = R sin t

What should Dt be?

* 1. x = R (2t)/(t2+1}   
     y = R (t2-1)/(t2+1) ; -infinity<t<infinity

Unfortunately, the speed of the curve wrt t varies with t, so the stepsize must also.

* 1. There is no parametric polynomial, of any degree, that exactly represents a circle. However you can get good approximations.

1. ???? (You suggest)
2. Now let's optimize C code. Do the second octant only.
   1. pixel(0,r);
   2. for(x=1;x<r/sqrt(2);x++)
   3. { y=sqrt(r2-x2);
   4. pixel(x,y);
   5. }
3. Assume that the last point was (x-1,y). Find whether (x,y-½) is above or below the circle, and move SE to (x,y-1) or E to (x,y), respectively.
   1. y=r;
   2. pixel(0,r);
   3. for(x=1;x<r/sqrt(2);x++)
   4. { d=x2+(y-1/2)2-r2;
   5. if (d>=0) y--;
   6. pixel(x,y);
   7. }
4. Optimize: d=x2+(y-1/2)2-r2 becomes d=x2+y2-y+1/4-r2. How does the d for a given x differ from d for x-1? If y is unchanged, then dx = dx-1+2x-1. If y decreases, add -2y+2, where that is the old y. Also the only time that 1/4 affects the test d>0 is when d=1/4, so ignore it, but make the test d>=0.
   1. y=r;
   2. d= -r;
   3. pixel(0,r);
   4. for(x=1;x<r/sqrt(2);x++)
   5. { d+= 2x-1;
   6. if (d>=0)
   7. { y--;
   8. d -= 2y; /\* Must do this AFTER y-- \*/
   9. }
   10. pixel(x,y);
   11. }

This is Bresenham's circle algorithm. Note that you can store 2x-1 and -2y and update them by adding or subtracting 2.

1. Lo-level optimizations can also be done thus:
2. y=r;
3. d= -r;
4. x2m1= -1;
5. pixel(0,r);
6. for(x=1;x<r/sqrt(2);x++)
7. { x2m1 += 2;
8. d+= x2m1;
9. if (d>=0)
10. { y--;

d -= (y<<1); /\* Must do this AFTER y-- \*/ } pixel(x,y); }

1. You can also do the same pipelining techniques used for the line algorithm.
2. Finally, on an Intel processor, there are ways to use the available instructions efficiently.
3. Here is the MidPoint Circle drawing algorithm.   
   1. void circlePlotPoints(CGWindow \*cgw, int xCenter, int yCenter, int x, int y)
   2. {
   3. cgw->DrawPoint(xCenter + x, yCenter + y);
   4. cgw->DrawPoint(xCenter - x, yCenter + y);
   5. cgw->DrawPoint(xCenter + x, yCenter - y);
   6. cgw->DrawPoint(xCenter - x, yCenter - y);
   7. cgw->DrawPoint(xCenter + y, yCenter + x);
   8. cgw->DrawPoint(xCenter - y, yCenter + x);
4. cgw->DrawPoint(xCenter + y, yCenter - x);
5. cgw->DrawPoint(xCenter - y, yCenter - x);
6. }
7. void circleMidPoint(CGWindow \*cgw, void \*arg)
8. {
9. int xCenter = 100, yCenter = 100, radius = 50;
10. int x = 0;
11. int y = radius;
12. int p = 1 - radius;
14. cgw->SetColor(0, RGB\_RANGE, 0);
15. circlePlotPoints(cgw, xCenter, yCenter, x, y);
17. while (x < y) {
18. x++;
19. if (p < 0)
20. p += 2 \* x + 1;
21. else {
22. y--;
23. p += 2 \* (x - y) + 1;
24. }
25. circlePlotPoints(cgw, xCenter, yCenter, x, y);
26. }

}

**Two Filling Algorithms**

Here are two filling algorithms, since each handles things the other doesn't.

**Polygon Fill**

The input is a polygon defined by a list of vertices and edges. *Hearn & Baker* describe this in section 3.11 on page 117, altho they goes into more detail than we need.

If a scan line goes exactly thru a vertex, then pretend that it really goes slightly below. The effect is that the code to test whether a scan line crosses an edge is thus:

if yscan<= yhi and ylo scanthen*We have an intersection*

**Seed (Boundary) Fill**

The input is a region defined by setting the pixels around the border. You mark one inside pixel, then the color spreads like a fire. More general regions can be filled.*Hearn & Baker*mention this on page 127.

The inside must be connected, else only one component is filled. This is harder than you'd think. If the polygon is thin, then the inside may not be connected, so the color stops too soon. This figure shows a long thin triangle, zoomed, and filled with green by touching a point in the center and flooding. Note that two red regions are missed.

Polygon Not Properly Filled