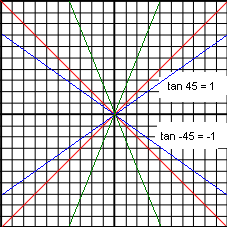
Basic Raster Graphics Algorithms for Drawing 2D Primitives

# Scan Converting Lines

A scan-conversion algorithm for lines computes the coordinates of the pixels that lie on or near an ideal, infinitely thin straight line imposed on a 2D raster grid.

**Our goal:** we would like the sequence of pixels to lie as *close to the ideal line* as possible and to be as *straight* as possible.

Let’s consider a 1-pixel-thick approximation to an ideal line.

**What properties should it** (*i.e. 1-pixel approx. to an ideal line*) **have?**

1. For lines with slopes between -1 and 1 inclusive (-45º to +45º), exactly 1 pixel should be illuminated in each *column*.
2. For lines with slopes outside this range, exactly 1 pixel should be illuminated in each *row*.
3. All lines should be drawn with constant brightness, independent of length and orientation, and as rapidly as possible.

Figure ‎3.2.1

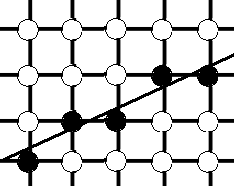
**Visualizing the geometry:**

Figure ‎3.1.2 shows a highly magnified view of a 1-pixel-thick line and of the ideal line that it approximates. The intensified pixels are shown as filled circles and the non-intensified pixels are shown as unfilled circles. On an actual screen, the diameter of the roughly circular pixel is larger than the interpixel spacing, so our symbolic representation exaggerates the discreteness of the pixels.

In the following line algorithms, we assume the following:

Figure ‎3.2.2: A scan-converted line showing intensified pixels as black circles.

1. The (*x*, *y*) coordinates of the line are integers, not floating-points.
2. Our line has slope |*m*| ≤ 1.

Lines at other slopes can be handled by suitable changes in the development that follows.

Also, the most common lines – those that are horizontal, are vertical, or have a slope of ±1 – can be handled as trivial special cases because these lines pass through only pixel centers.

## Basic Incremental Algorithm / Digital Differential Analyzer (DDA) Algorithm

**Algorithm:**

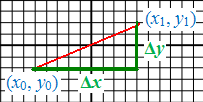
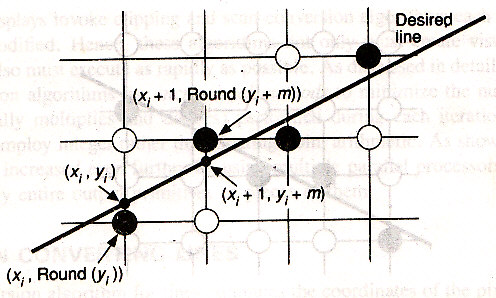
1. Let, the endpoints of the line are (*x*0, *y*0), (*x*1, *y*1).
2. Compute slope as *m* = Δ*y* / Δ*x* = (*y*1 – *y*0) / (*x*1 – *x*0).
3. For (*x* = *x*0; *x* ≤ *x*1; *x*++) {
4. *y* = *mx* + *B*
5. Intensify pixel at (*x*, Round(*y*)), where Round(*y*) = Floor(0.5 + *y*).
6. }

Figure ‎3.2.3

**Problem with the above *brute-force* algorithm:**

Inefficiency – because each iteration requires a floating-point (or binary fraction) multiply, addition, and invocation of Floor().

**Improvement – eliminating the multiplication:**

For each *xi*, we have

*yi = mxi + B*

∴ *yi* + 1 *= mxi* + 1 *+ B*

= *m*(*x*i+ Δ*x*) *+ B*

= *mx*i*+ B* + *m*Δ*x*

= *yi* + *m*Δ*x*

Figure ‎3.2.4: Incremental calculation of (*xi*, *yi*).

If Δ*x* = 1 (we’re incrementing *x* by 1 at each iteration), then *yi* + 1 = *yi* + *m*.

Thus, a unit change in *x* changes *y* by *m*, which is the slope of the line. For all points (*xi*, *yi*) on the line, we know that, if *xi* + 1 = *xi*+ 1, then *yi* + 1 = *yi* + *m*; that is, the values of *x* and *y* are defined in terms of their previous values (see Figure ‎3.1.4). This is what defines an incremental algorithm: at each step, we make incremental calculations based on the preceding step.

For the case |*m*| > 1, a step in *x* creates a step in *y* that is greater than 1. Thus, we must reverse the roles of *x* and *y* by assigning a unit step to *y* and incrementing *x* by Δ*x* = Δ*y* / *m* = 1 / *m*.

**Error in this algorithm:**

Since real variables have limited precision, summing an inexact *m* repetitively introduces cumulative error buildup and eventually a drift away from a true Round(*yi*); for most (short) lines, this will not present a problem.

|  |
| --- |
| Listing ‎3.1.1: The incremental line scan-conversion algorithm. |
| 1 /\* Assumes -1 ≤ *m* ≤ 1, *x*0 < *x*1 \*/  2 void line(int x0, int y0,  3 int x1, int y1,  4 int value) {  5 double dy = y1 - y0;  6 double dx = x1 - x0;  7 double m = dy / dx;  8  9 int x;  10 double y = y0;  11 for (x = x0; x <= x1; x++) {  12 writePixel(x, round(y), value); // set pixel to *value*  13 y += m; // step *y* by slope *m*  14 }  15 } |

## Midpoint Line Algorithm / Bresenham’s Algorithm

**The drawbacks of incremental line algorithm:**

1. Rounding *y* to an integer takes time.
2. The variables *y* and *m* must be real or fractional binary because the slope is a fraction.

**Bresenham’s Algorithm and the Midpoint Algorithm:**

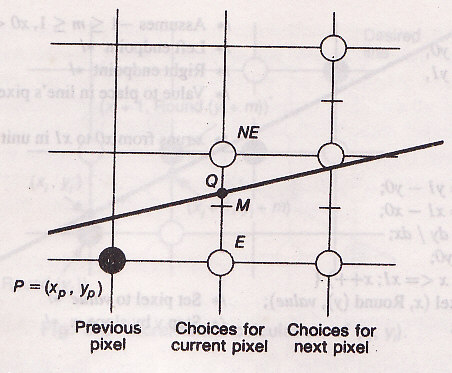
Bresenham developed a classic algorithm that is attractive because it uses only integer arithmetic, thus avoiding the Round function, and allows the calculation for (*xi* + 1, *yi* + 1) to be performed incrementally – that is, by using the calculation already done at (*xi*, *yi*).

A floating-point version of this algorithm can be applied to lines with arbitrary real-valued endpoint coordinates.

Furthermore, Bresenham’s incremental technique may be applied to the integer computation of circles as well, although it does not generalize easily to arbitrary conics.

We therefore use a slightly different formulation, the *midpoint technique*, first published by Pitteway and adapted by Van Aken and other researchers. For lines and integer circles, the midpoint formulation, as Van Aken shows, reduces to the Bresenham formulation and therefore generates the same pixels.

**The Midpoint Line Algorithm:**

We assume that the line’s slope is between 0 and 1 (0º to 45º). Other slopes can be handled by suitable reflections about the principal axes. We call the lower-left endpoint (*x*0, *y*0) and the upper-right endpoint (*x*1, *y*1).

Consider the line in Figure ‎3.1.5. Assume that:

Figure ‎3.2.5: The pixel grid for the midpoint line algorithm, showing the midpoint *M*, and the *E* and *NE* pixels to choose between.

* We’ve just selected the pixel *P*(*xp*, *yp*).
* Now we must choose between the pixels *E* and *NE*.
* Q be the intersection point of the line being scan-converted with the grid line *x = xp +* 1.

In Bresenham’s formulation,

* If *EQ – NEQ* > 0, we choose *NE*.
* If *EQ – NEQ* < 0, we choose *E*.

That is, the sign of the difference is used to select the pixel whose distance from *Q* is smaller as the best approximation to the line being scan-converted.

In Midpoint formulation, we observe on which side of the line the midpoint *M* lies.

* If *M* lies above the line, then pixel *E* is closer to the line.
* If *M* lies below the line, then pixel *NE* is closer to the line.

The line may pass between *E* and *NE* (as in this case), or both pixels may lie on one side (in the case that we’ve chosen NE and then we have to choose between the pixels on right and upper-right of *NE*), but in any case, the midpoint test chooses the closest pixel.

Also, the error – that is, the vertical distance between the chosen pixel and the actual line – is always ≤ ½.

**How can we calculate on which side of the line the midpoint lies?**

The line can be represented using the function *F*(*x*, *y*) = *ax + by + c* = 0.

The slope-intercept form of the function can be written as:

*y = x + B*, where *dy = y*1 – *y0* and *dx = x*1 – *x0*

∴ *F*(*x*, *y*) = *dy* . *x – dx* . *y + B* . *dx* = 0.

It can easily be verified that *F*(*x*, *y*) is

* = 0 for points *on* the line
* > 0 for points *below* the line
* < 0 for points *above* the line

To apply the midpoint criterion, we need only to compute *F*(*M*) = *F*(*xp* + 1, *yp* + ½) and to test its sign:

* If *d*[[1]](#footnote-1) = *F*(*xp* + 1, *yp* + ½) = *a*(*xp* + 1) *+ b*(*yp* + ½) *+ c* > 0, we choose *NE*
* If *d* < 0, we choose *E*
* If *d* = 0, we can choose either *E* or *NE*, so we pick *E* (arbitrarily).

**Now, what happens to the location of *M* and therefore to the value of *d* for the *next grid line*?**

Both depend, of course, on whether we chose *E* or *NE*.

If *E* is chosen,

*M* is incremented by one step in the *x* direction. Then,

*dnew* = *F*(*xp* + 2, *yp* + ½)

= *a*(*xp* + 2) *+ b*(*yp* + ½) *+ c*

But, *dold* = *a*(*xp* + 1) *+ b*(*yp* + ½) *+ c*

Subtracting *dold* from *dnew* to get the incremental difference, we write *dnew* = *dold* + *a*.

If *NE* is chosen,

*M* is incremented by one step each in both the *x* and *y* directions. Then,

*dnew* = *F*(*xp* + 2, *yp* + )

= *a*(*xp* + 2) *+ b*(*yp* + ) *+ c*

Subtracting *dold* from *dnew* to get the incremental difference, we write *dnew* = *dold* + *a* + *b*.

**Calculating the initial value of *d*:**

Since the first pixel is simply the first endpoint (*x*0, *y*0), we can directly calculate the initial value of *d* for choosing between *E* and *NE*. The first midpoint is at (*x*0 + 1, *y*0 + ½), and

*dstart* = *F*(*x*0 + 1, *y*0 + ½)

= *a*(*x*0 + 1) *+ b*(*y*0 + ½) *+ c*

= *ax*0 + *by*0 + *c* + *a* + *b* / 2

= *F*(*x*0, *y*0) + *a* + *b* / 2

= 0 + *a* + *b* / 2 [∵ (*x*0, *y*0) is a point *on* the line]

= *a* + *b* / 2

= *dy – dx* / 2

To eliminate the fraction in *dstart*, we redefine our original *F* by multiplying it by 2; *F*(*x*, *y*) = 2(*ax + by + c*). This multiplies each constant (*a*, *b* and *c*) and the decision variable (*d*) by 2, but does not affect the sign of the decision variable, which is all that matters for the midpoint test.

|  |
| --- |
| Listing ‎3.2.2: The midpoint line scan-conversion algorithm. |
| 1 /\* Assumes 0 <= m <= 1, x0 < x1 \*/  2 void midpointLine(int x0, int y0, int x1, int y1, int value) {  3 int dx = x1 - x0;  4 int dy = y1 - y0;  5 int d = 2 \* dy - dx; // initial value of d  6 int incrE = 2 \* dy; // increment used for move to E  7 int incrNE = 2 \* (dy - dx); // increment used for move to NE  8  9 int x = x0;  10 int y = y0;  11 writePixel(x, y, value); // the start pixel  12  13 while (x < x1) {  14 if (d <= 0) { // choose E  15 d += incrE;  16 x++;  17 } else { // choose NE  18 d += incrNE;  19 x++;  20 y++;  21 }  22 writePixel(x, y, value); // the selected pixel closest to the line  23 }  24 } |

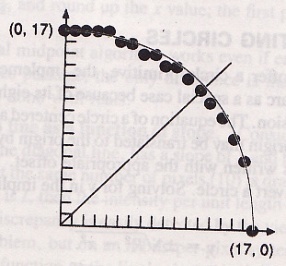
# Scan Converting Circles

## An Easy (and Inefficient) Algorithm

The equation of a circle centered at the origin (0, 0) is *x*2 + *y*2 = *R*2.

Solving for *y*, we get *y* = ±.

To draw a quarter circle (the other quarters are drawn by symmetry), we can increment *x* from 0 to *R* in unit steps, solving for +*y* at each step.

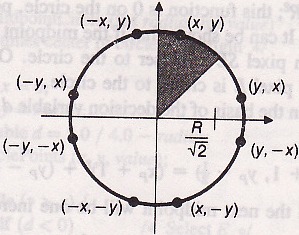
Circles not centered at the origin may be translated to the origin by integer amounts and then scan converted, with pixels written with the appropriate offset.

**Reasons why this algorithm is inefficient:**

1. Contains multiplication and square-root operations.
2. The circle will have large gaps for values of *x* close to *R*, because the slope of the circle becomes infinite there (see Figure ‎3.3.1).

## Eight-Way Symmetry

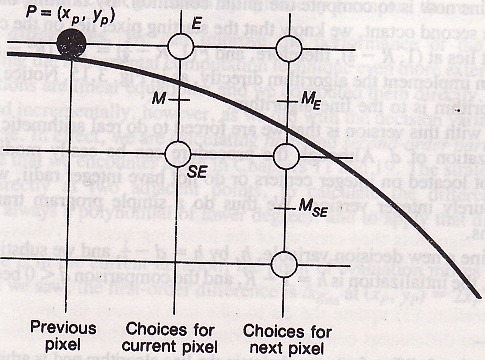
Figure ‎3.3.1

We can improve the drawing process of the previous section by taking greater advantage of the symmetry in a circle.

Consider first a circle centered at the origin. If the point (*x*, *y*) is on the circle, then we can trivially compute seven other points on the circle, as shown in Figure ‎3.3.2. Therefore, we need to compute only one 45º segment to determine the circle completely. For a circle centered at the origin, the eight symmetrical points can be displayed with procedure circlePoints():

Figure ‎3.3.2

|  |
| --- |
| Listing ‎3.3.1: The incremental line scan-conversion algorithm. |
| 1 void circlePoints(int x, int y, int value) {  2 writePixel(x, y, value);  3 writePixel(y, x, value);  4 writePixel(y, -x, value);  5 writePixel(x, -y, value);  6 writePixel(-x, -y, value);  7 writePixel(-y, -x, value);  8 writePixel(-y, x, value);  9 writePixel(-x, y, value);  10 } |

We do not want to call circlePoints() when *x* = *y*, because each of four pixels would be set twice; the code is easily modified to handle that boundary condition.

## Midpoint Circle Algorithm

We consider only 45º of a circle, the second octant from *x* = 0 to *x* = *y* = *R* / √2, and use the circlePoints procedure to display points on the entire circle.

The goal: to select which of two pixels is closer to the circle by evaluating a function at the midpoint between the two pixels. In the second octant, if pixel *P* at (*xp*, *yp*) has been previously chosen as closest to the circle, the choice of the next pixel is between pixel *E* and *SE* (see Figure ‎3.3.3).

Figure ‎3.3.3

Let, *F*(*x*, *y*) = *x*2 + *y*2 – *R*2. This function is 0 on the circle, +ve outside the circle, and –ve inside the circle. It can be shown that if the midpoint between the pixels *E* and *SE* is outside the circle, then pixel *SE* is closer to the circle.

As for the lines, we choose on the basis of the decision variable *d*, which is the value of the function at the midpoint,

*dold* = *F*(*xP* + 1, *yP –* ½) = (*xP* + 1)2 + (*yP* – ½)2 – *R*2

If *dold* < 0, *E* is chosen, and the next midpoint will be one increment over in *x*. Then,

*dnew* = *F*(*xP* + 2, *yP –* ½) = (*xP* + 2)2 + (*yP* – ½)2 – *R*2

= *dold* + (2*xP* + 3)

∴ The increment, Δ*E* = 2*xP* + 3

If *dold* ≥ 0, *SE* is chosen, and the next midpoint will be one increment over in *x* and one increment down in *y*. Then,

*dnew* = *F*(*xP* + 2, *yP –* ) = (*xP* + 2)2 + (*yP* – )2 – *R*2

= *dold* + (2*xP* – 2*yP* + 5)

∴ The increment, Δ*SE* = 2*xP* – 2*yP* + 5

**Calculating the initial value of *d*:**

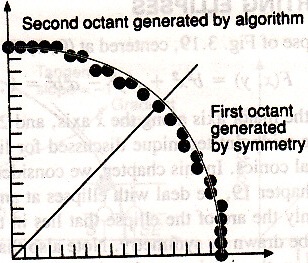
By limiting the algorithm to integer radii in the second octant, we know that the starting pixel lies on the circle at (0, *R*). The next midpoint lies at (1, *R* – ½), therefore, *dstart* = *F*(1, *R* – ½) = 1 + (*R*2 – *R* + ¼) – *R*2 = 5/4 – *R*.

|  |
| --- |
| Listing ‎3.3.2: The midpoint circle scan-conversion algorithm. |
| 1 /\* Assumes center of circle is at origin \*/  2 void midpointCircle(int radius, int value) {  3 int x = 0;  4 int y = radius;  5 double d = 5 / 4 - radius;  6  7 circlePoints(x, y, value);  8  9 while (y > x) {  10 if (d < 0) { // select E  11 d += 2 \* x + 3;  12 } else { // select SE  13 d += 2 \* (x - y) + 5;  14 y--;  15 }  16 x++;  17 circlePoints(x, y, value);  18 }  19 } |

**Problem with this algorithm:**

We’re forced to do real arithmetic because of the fractional initialization of *d*.

**Optimizing the program by doing a simple program transformation to eliminate fractions:**

Let, *h* = *d* – ¼

We substitute *h* + ¼ for *d* in the code.

Now, the initialization is *h* = 1 – *R*, and the comparison *d* < 0 becomes *h* < –¼.

However, since *h* starts out with an integer value (i.e. 1 – *R*) and is incremented by integer values (Δ*E* and Δ*SE*), we can change the comparison to just *h* < 0.

Figure ‎3.3.4 shows the second octant of a circle of radius 17 generated with the algorithm, and the first octant generated by symmetry.

Figure ‎3.3.4: Second octant of circle generated with midpoint algorithm, and first octant generated by symmetry.

## Midpoint Circle Algorithm

We can improve the performance of the midpoint circle algorithm by using the incremental computation technique even more extensively.

Any polynomial can be computed incrementally. In effect, we are calculating *first-* and *second-order partial differences*.

**The strategy is:**

1. Evaluate the function directly at two adjacent points.
2. Calculate the difference (which, for polynomials, is always a polynomial of *lower* degree).
3. To apply the difference in each iteration.

If we choose *E* in the current iteration, the point of evaluation[[2]](#footnote-2) moves from (*xP*, *yP*) to (*xP* + 1, *yP*).

∴ The first-order difference, at (*xP*, *yP*) = 2*xP* + 3

∴ at (*xP* + 1, *yP*) = 2(*xP* + 1) + 3

∴ The second-order difference is – = 2

Similarly, at (*xP*, *yP*) = 2*xP* – 2*yP* + 5

∴ at (*xP* + 1, *yP*) = 2(*xP* + 1) – 2*yP* + 3

∴ The second-order difference is – = 2

If we choose *SE* in the current iteration, the point of evaluation moves from (*xP*, *yP*) to (*xP* + 1, *yP* – 1).

∴ at (*xP* + 1, *yP* – 1) = 2(*xP* + 1) + 3

∴ The second-order difference is – = 2

Similarly, at (*xP* + 1, *yP* – 1) = 2(*xP* + 1) – 2(*yP* – 1) + 5

∴ The second-order difference is – = 4

The revised procedure using this technique is shown below (Δ*E* and Δ*SE* are initialized using the start pixel (0, *R*)).

|  |
| --- |
| Listing ‎3.3.3: The midpoint circle scan-conversion algorithm using second-order differences. |
| 1 /\* Assumes center of circle is at origin \*/  2 void midpointCircle(int radius, int value) {  3 int x = 0;  4 int y = radius;  5 int d = 1 - radius;  6 int deltaE = 3;  7 int deltaSE = -2 \* radius + 5;  8  9 circlePoints(x, y, value);  10  11 while (y > x) {  12 if (d < 0) { //select E  13 d += deltaE;  14 deltaE += 2;  15 deltaSE += 2;  16 } else { //select SE  17 d += deltaSE;  18 deltaE += 2;  19 deltaSE += 4;  20 y--;  21 }  22 x++;  23 circlePoints(x, y, value);  24 }  25 } |

1. *d* is called the *decision variable*. [↑](#footnote-ref-1)
2. **Point of evaluation:** Because the functions Δ*E* (= 2*xP* + 3) and Δ*SE* (= 2*xP* – 2*yP* + 5) are expressed in terms of (*xP*, *yP*), we call *P* the point of evaluation. [↑](#footnote-ref-2)