Bresenham Lines and Circles

Major points of these notes:

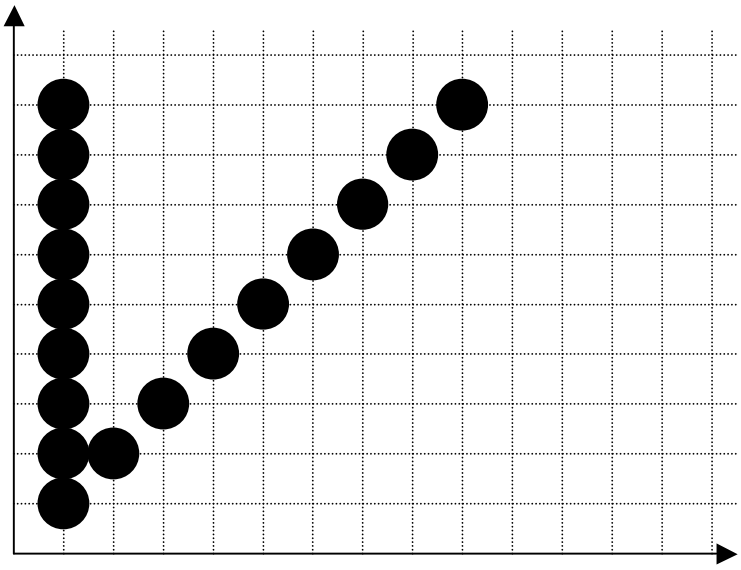
1. Goals for Line Drawing Algorithms

We're assuming that we want our line drawing algorithm to be able to draw a line between two user-specified pixels on the screen, (x_0, y_0) and (x_1, y_1). Note that we aren't specifying the line as being between two 3D points in the image, but are assuming that any line has already been projected into screen coordinates and clipped to the screen, so that (x_0, y_0) and (x_1, y_1) are integer pairs that are pixel addresses that are presumably projections from 3D floating point points.

There are a number of goals that we would like to have for any line drawing algorithm, not all of which can be satisfied at the same time. Some of these goals are:

1. The line displayed should include the end pixels (x_0, y_0) and (x_1, y_1).
2. The line intensity should be constant, and shouldn't depend on the angle of the line.
3. The algorithm should be very fast, and easy to implement in a hardware graphics accelerator.
4. The line generated should look straight, without jaggies.
5. The line shouldn't have any gaps which would let color escape in a flood fill operation. It should be 8-connected.

Unfortunately these goals contradict each other. The only way, for example, to fully satisfy 2 and 4 is to use anti-aliased lines, and that process is computationally very expensive. E.g., consider goal 2, and look at the figure below, where I've drawn two lines, one from (1, 1) to (1, 9), and the other from (1, 1) to (9, 9), with both drawn in the obvious "best next pixel" way:



It should be obvious that the line at 45° is much less intense than the vertical line, because its length is $\sqrt{2}$ longer, but it contains the same number of pixels. However any fast line drawing algorithm will have this problem, and graphics systems just have to hope that it won't be noticed. (And the chances are that you have never noticed, even though you have seen a very large number of graphics images that have this problem.)

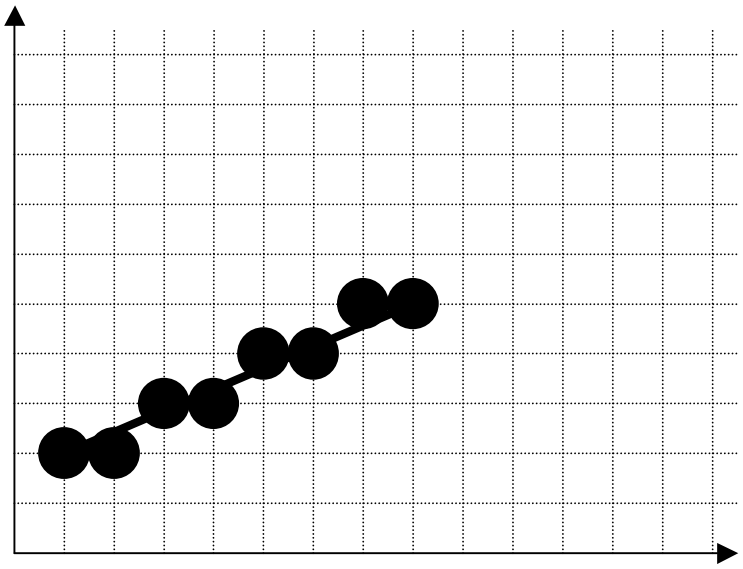
Getting back to our original goals, we can easily meet goals 1 and 3, and goal 2 we can approximate by saying that the intensity difference between lines will never exceed a factor of 1.414 ($\sqrt{2}$). Jaggies will exist in most lines, but as screen resolutions have improved they have become much less obvious. They were much more of a problem when the typical screen resolution was 640 × 512.

Before getting into Bresenham's algorithm we'll first look at the more obvious algorithm called DDA, which adds in the slope and rounds to the nearest y for each new pixel displayed. Bresenham will produce exactly the same output as DDA, but dramatically faster since it only uses integer addition without the float add and round that occurs in DDA.

For both the DDA and Bresenham algorithms we will assume that the line is being drawn from left to right (i.e., $x_0 < x_1$), and that the line points increase gently, and so its slope satisfies $0 < slope < 1$. Then in a subsequent section we will show how to generalize the results to any line. We ignore lines with slopes of 0 or ∞, since they can best be handled as special cases.

2. DDA Line Drawing Algorithm

The DDA (Digital Differential Analyzer) algorithm is, despite its long and impressive name, the obvious way to draw a line. Since we are looking at lines whose increase in *y* is less than their increase in *x*, the simple solution is to start with a pixel at (*x0*, *y0*) and then loop increasing *x* by one each time and *y* by the slope, and then rounding the *y* to find the pixel to be displayed. E.g., say we want to draw the line from (1, 2) to (8, 5), then we want to draw the line as shown below:



The actual line is shown for reference. In the DDA line there is a pixel drawn for each *x* between the starting and ending values, and then the *y* value closest to the line is selected for the second coordinate.

The calculations that will be performed are shown in the table below:

x	*y*	round(*y*)
1	2	2
2	$2\frac{3}{7}$	2
3	$2\frac{6}{7}$	3
4	$3\frac{2}{7}$	3
5	$3\frac{5}{7}$	4
6	$4\frac{1}{7}$	4
7	$4\frac{4}{7}$	5
8	5	5

The algorithm which we are performing here can be written as:

```
procedure DDAline(in int x0, y0, x1, y1; in colorval color) {
assert:  (x0 < x1) and ((y1 – y0) < (x1 – x0))
        declare:  int x;  float y, slope;

        slope = (y1 – y0) / (x1 – x0);
        x ← x0;  y ← y0;
        printpixel(x, y, color);
        while (x < x1) {
                x++;
                y += slope;
                printpixel(x, round(y), color);
        } end while
} end procedure DDAline
```

where printpixel will display the pixel in the desired color.

The expensive parts of this algorithm are *y += slope;* which is a float add and round(*y*), which converts from float to integer. Bresenham, which we'll look at next, only uses integer additions and comparisons.

3. Bresenham Line Algorithm for $x_0 < x_1$ and $0 < slope < 1$

Bresenham's algorithm displays exactly the same pixels as the DDA algorithm, given the same end points. The difference is that it does it much more efficiently. The DDA algorithm has to maintain y as a float, and in each loop does a float add and a float to integer round. We also have a single float divide to compute the slope. By comparison, Bresenham's algorithm only uses integer variables, and only uses integer compares and additions. This not only makes it much faster, but also makes it much easier to put into hardware.

In these notes I won't prove why Bresenham's algorithm works, but will just list the code and give an example. In CS 525 I'll prove why it works. So first, here is the algorithm:

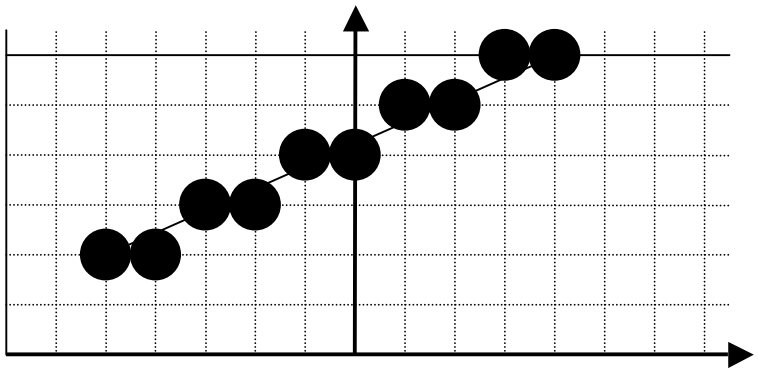
```
procedure BresenhamLine(in int x0, y0, x1, y1; colorval color) {
assert: (x0 < x1) and ((y1 – y0) < (x1 – x0))
    declare: int x, y, δy, δy, d, incrE, incrNE;

    δx = x1 – x0;  δy = y1 – y0;
    incrE = 2 * δy;
    incrNE = 2 * (δy - δx);
    d = 2 * δy – δx;
    x ← x0;  y ← y0;
    printpixel(x, y, color);

    while (x < x1) {
        x++;
        if (d < 0) {
            d += incrE;  /* go East */
        } else {
            y++;
            d += incrNE;  /* go NorthEast */
        } end if
        printpixel(x, y, color);
    } end while
} end procedure BresenhamLine
```

The most important thing to note about this algorithm is that everything is integer. So it avoids all floating point calculations. Also the only operations

are integer additions and comparisons since multiplication by 2 will be implemented as a single addition. This is why Bresenham is much faster than DDA, and also why it is easy to put into graphics hardware accelerators.

Example: The algorithm looks a bit of a mess, so we'll use it to generate the line from (-5, 2) to (4, 6). First we need the initial values of the variables δx, δy, d, *incrE*, and *incrNE*, which will be 9, 4, -1, 8, and -10, respectively. We then initialize x and y to the starting point, -5 and 2, display this pixel in the required color, and enter the while loop. This increments x by 1 each time through the loop, and so it will execute nine times, with x starting at -4 and ending at 4. Inside the loop one of two things happens, based on the value of d. If d is negative, it draws the pixel to the right (East) of the previous pixel, since it doesn't change the value of y, and adds incrE (for increment East) to the current value of d. If d isn't negative it moves northeast, by also adding 1 to y, and adds *incrNE* to d. In either case it prints out the new (x, y) pixel. A table and graph of the values are shown below.

x	y	d
-5	2	-1
-4	2	7
-3	3	-3
-2	3	5
-1	4	-5
0	4	3
1	5	-7
2	5	1
3	6	-4
4	6	4



If you compare this against the DDA algorithm you will see that both algorithms have generated exactly the same pixels. The difference is that Bresenham has only used integer arithmetic because of its use of the decision variable, *d*, which is used to determine whether the next pixel should be east or northeast of the previous pixel. Note that because of the constraints on the slope these are the only possible positions for the next pixel.

4. Bresenham Algorithm for General Lines

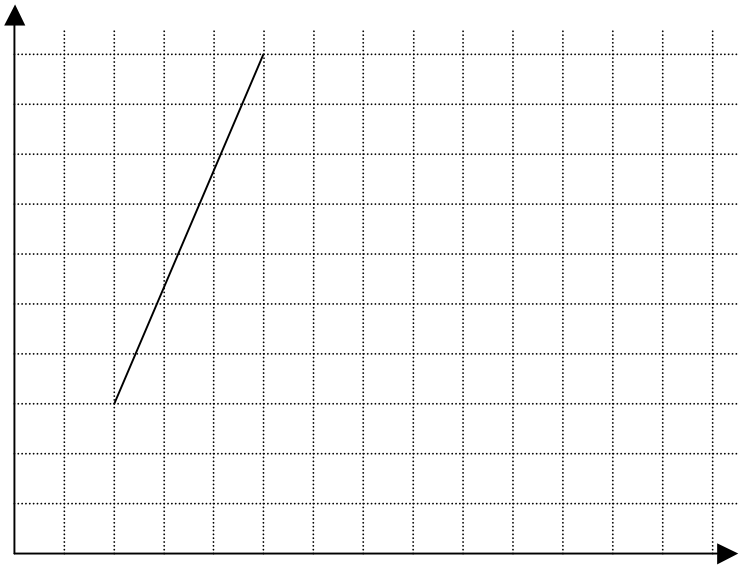
We have assumed that ($x_0 < x_1$) **and** (($y_1 - y_0$) < ($x_1 - x_0$)). I.e., that the line is going from left to right and that *slope* satisfies 0 < *slope* < 1. Clearly it is trivial to draw vertical and horizontal lines without calling on Bresenham, which means that we can assume that $x_0 \neq x_1$ and $x_0 \neq x_1$.

We need to be able to handle three other cases:

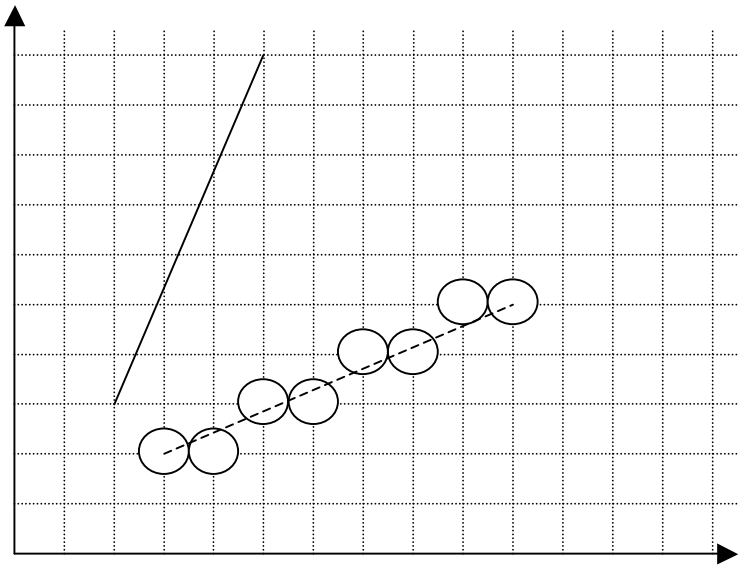
1. $x_0 > x_1$
2. slope > 1
3. slope < 0

We'll discuss each of these cases independently.

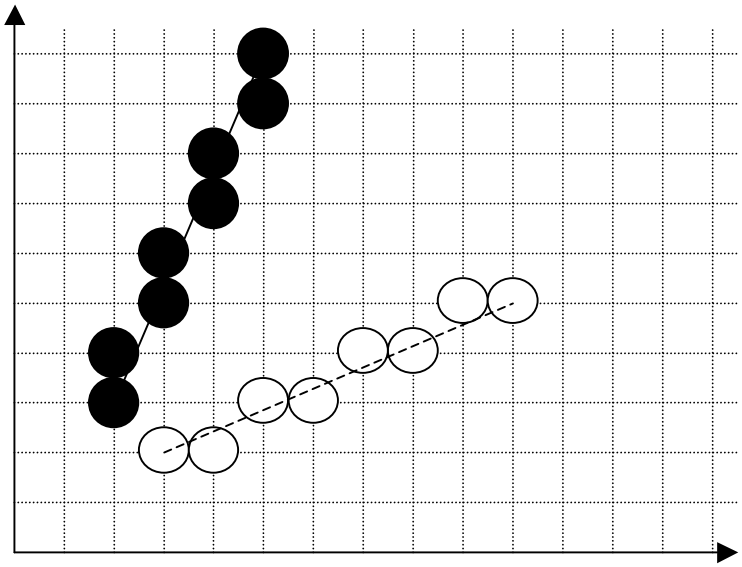
1. $x_0 > x_1$: This is trivial to handle. If the line should be drawn from right to left instead of left to right, we just draw it backwards and everything will be all right. I.e., we just exchange (x0, y0) and (x1, y1). E.g., if the algorithm is asked to draw a line from (3, 5) to (1, 4), we just draw the line from (1, 4) to (3, 5).
2. slope > 1: Say that we are drawing a line from (2, 3) to (5, 10) as shown in the figure below.



We can exchange x and y values and use Bresenham to calculate the dashed line from (3, 2) to (10, 5), which will give the pixel locations shown below.

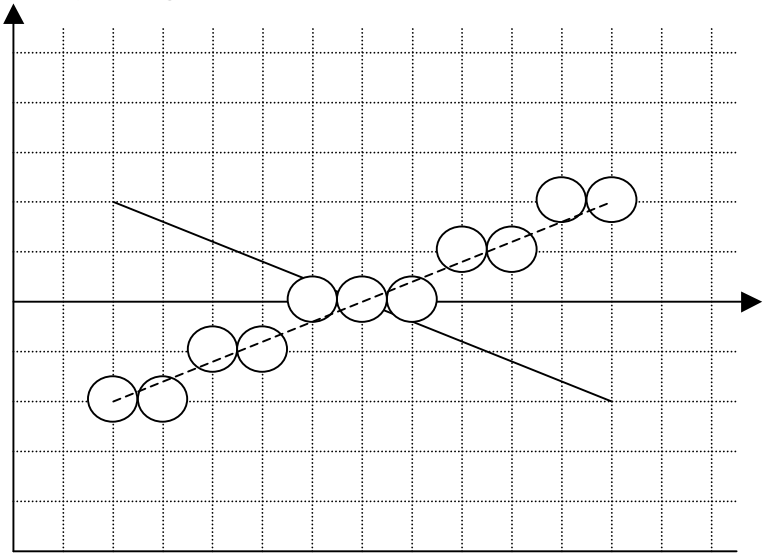


Now we can exchange their x and y values to get the pixels displayed on the line.

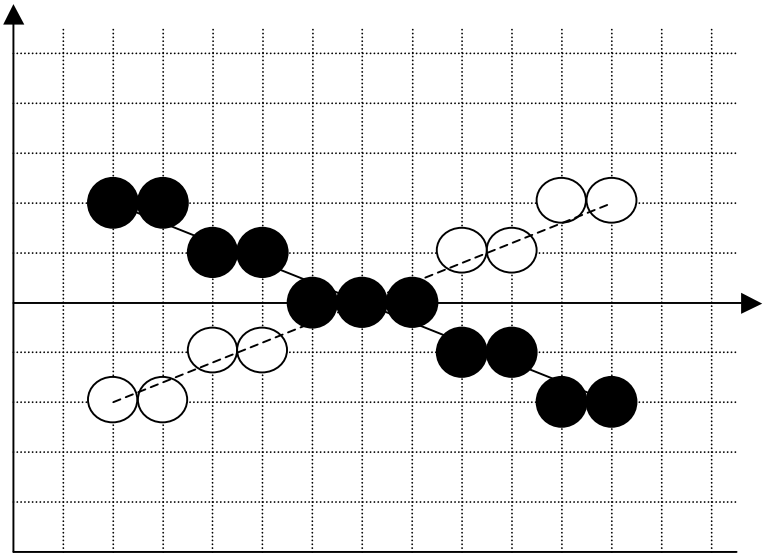


So to summarize, we wanted to draw a line from (2,3) to (5,10). Since the slope was greater than 1 we exchanged the x and y values and calculate the pixels for the line from (3,2) to (10,5). These are (3,2), (4,2), (5,3), (6,3), (7,4), (8,4), (9,5), and (10,5). Then we exchanged their x and y values to get the pixels that we displayed, (2,3), (2,4), (3,5), (3,6), (4,7), (4,8), (5,9), and (5,10). In effect we have reflected the line that we want around the diagonal line y = x, which turned a slope of greater than 1 into a slope of less than 1.

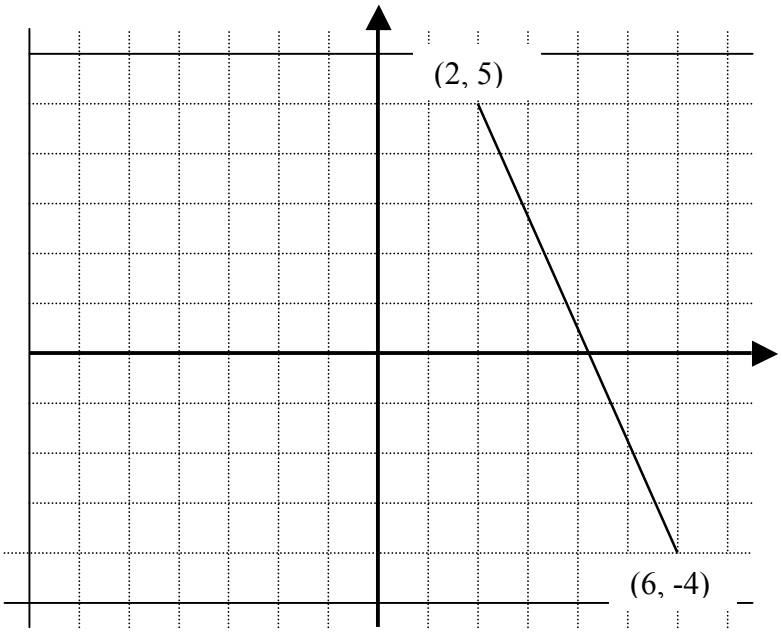
3. slope < 0: If the slope is less than zero we can reflect the line around the x axis to get a line whose slope is greater than zero, generate the points there, and then reflect back around the x axis to get the displayed pixels. What that means is that instead of using Bresenham from (x0, y0) to (x1, y1), we will use the algorithm on the line from (x0, -y0) to (x1, -y1). Using the same notation that I used before, and wanting the line from (2,2) to (12,-2), we compute the pixels for (2,-2) to (12,2) using Bresenham, as shown below:



giving the values (2,-2), (3,-2), (4,-1), (5,-1), (6,0), (7,0), (8,0), (9,1), (10,1), (11,2), and (12,2). Changing the signs of the y values gives the displayed line with pixels (2,2), (3,2), (4,1), (5,1), (6,0), (7,0), (8,0), (9,-1), (10,-1), (11,-2), and (12,-2).

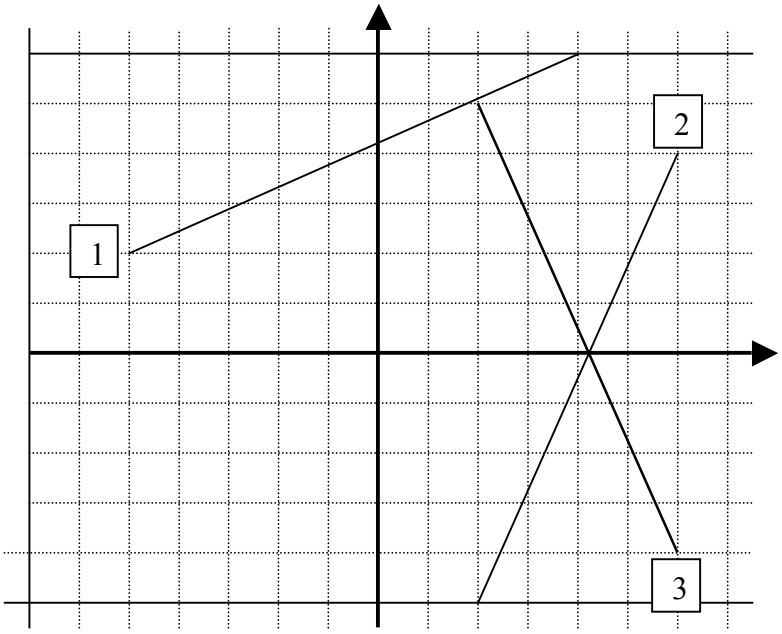


Example: Use Bresenham to generate the line from (6,-4) to (2,5).

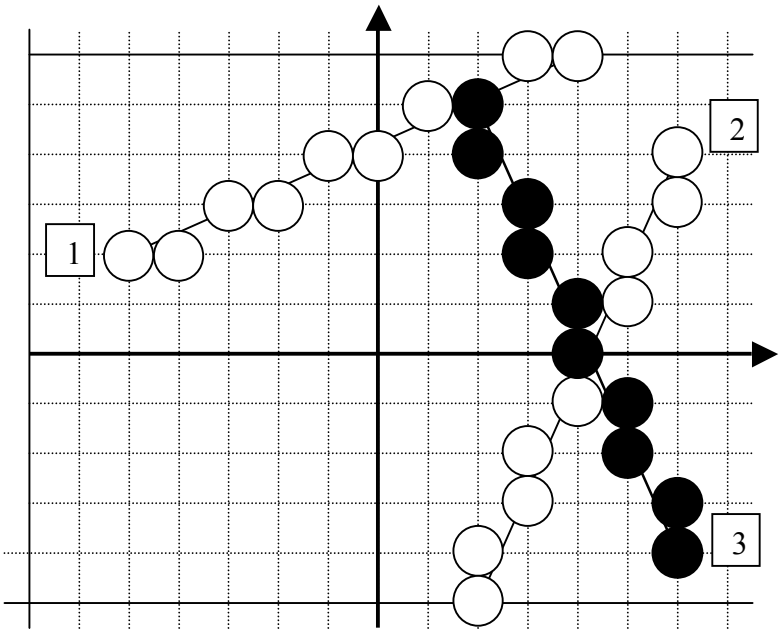


Everything is wrong here. The line goes right to left, but we can fix that by using the line from (2,5) to (6,-4). The slope is negative, and so we mirror around the x axis, and use the line from (2,-5) to (6,4). The slope of this line is greater than 1, so we mirror around y = x, and use Bresenham for the line from (-5,2) to (4,6). This satisfies everything, and so we can finally run the algorithm.

The three lines are shown in the next figure. First we will use Bresenham to give us the pixel addresses on the line labeled 1, then exchange *x* and *y* to get the pixel addresses on the line labeled 2, and then switch the signs of the *y* values to get the line labeled 3, which is the line that we wanted. In effect the first operation is a mirror back around the line $y = x$, and the second is a mirror back around the *x* axis.



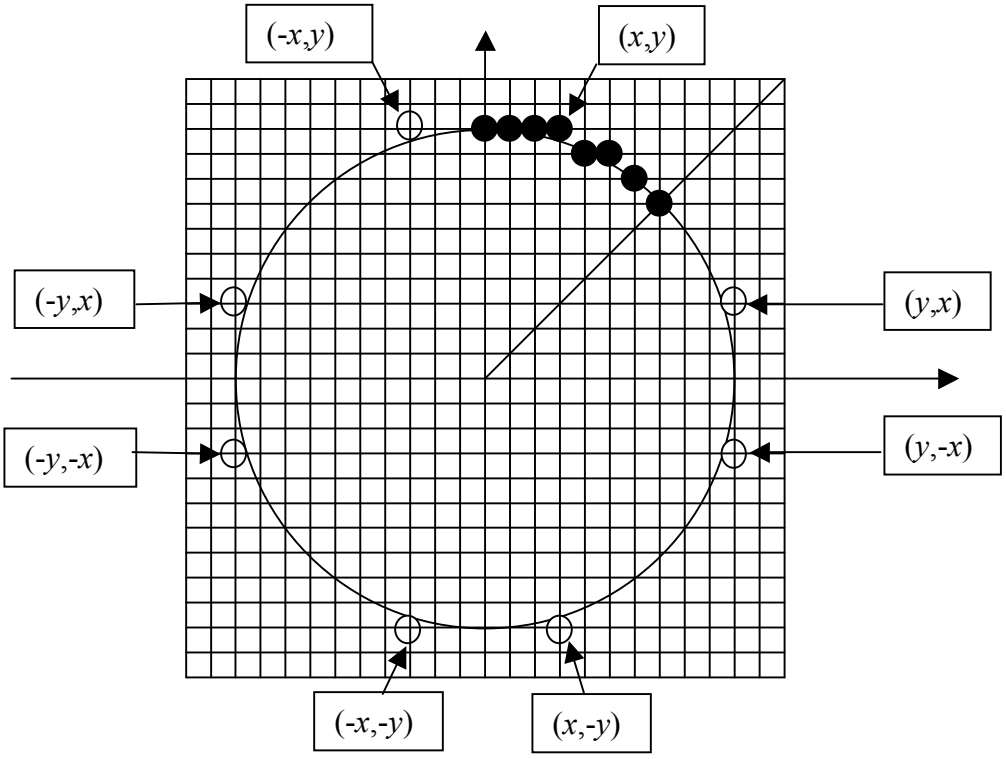
In our first Bresenham algorithm example we showed that for (-5, 2) to (4, 6) Bresenham generates the ten pixels (-5, 2), (-4, 2), (-3, 3), (-2, 3), (-1, 4), (0, 4), (1, 5), (2, 5), (3, 6), and (4, 6). Mirroring back around the line x = y, by exchanging x and y values, gives the pixels on the intermediate line as (2, -5), (2, -4), (3, -3), (3, -2), (4, -1), (4, 0), (5, 1), (5, 2), (6, 3), and (6, 4). Switching the signs of the y values completes the whole process, and gives us the pixels (2, 5), (2, 4), (3, 3), (3, 2), (4, 1), (4, 0), (5, -1), (5, -2), (6, -3), and (6, -4). These three lines are shown below.



5. Efficiency considerations for Circle Generators

In both the DDA section and the Bresenham section we will assume that the equation of the circle is $x^2 + y^2 = R^2$. I.e., we have a circle with its center at the origin and radius R. In a later section we'll generalize this to circles with equation $(x - c_x)^2 + (y - c_y)^2 = R^2$, which is a circle with radius R and center at (cx, cy), by generating the circle centered at the origin and then translating it to the correct center.

An important efficiency improvement for both DDA and Bresenham is to only calculate pixel addresses in the primary octant of the circle (shown below) and then use symmetry to calculate the points in the other seven octants. Since the circle is centered at the origin, as we discussed above, if we generate a point, (x, y), then the other seven points which are combinations of positive and negative x and y values can be drawn without further computations.



So it is sufficient to just generate about $\frac{1}{8}$ of the points on the circle, and then output seven additional points for each one.

6. DDA Algorithm for Circles Centered at the Origin

The advantages of using Bresenham are even greater for drawing circles (and other conics) than they are for lines. The DDA algorithm for circles, which we'll describe below, has to use square roots to compute y values for different x's, whereas Bresenham once again only uses integer addition. First, however, we'll look at the DDA algorithm to ensure that it generates the same points as Bresenham does when we see that.

The points in the primary octant, shown above, start at (0, R) above the origin, and then increase x by 1 as long as $x \leq y$ (the diagonal at the end of the primary octant. The algorithm for this is:

```
procedure DDAcircle(in int R; in colorval color) {
        declare:  int x;  float y;
        x ← 0;  y ← R;
        print8pixels(x, y, color);
        while (x < y) {
                x++;
                y = √(R * R – x * x) ;
                print8pixels(x, round(y), color);
        } end while
} end procedure DDAcircle
```

I.e., each time we are solving the circle equation $y^2 = R^2 - x^2$, taking the square root, and printing the x and the closest y, plus the seven symmetric points. The table below gives the (x, y) values which match the diagram in the last section.

x	$y = \sqrt{100 - x * x}$	round(y)
0	$\sqrt{100} = 10.00$	10
1	$\sqrt{99} = 9.95$	10
2	$\sqrt{96} = 9.80$	10
3	$\sqrt{91} = 9.54$	10
4	$\sqrt{84} = 9.17$	9
5	$\sqrt{75} = 8.66$	9
6	$\sqrt{64} = 8.00$	8
7	$\sqrt{51} = 7.14$	7

7. Bresenham Algorithm for Circles Centered at the Origin

While it is surprising that it is possible to draw lines with only integer additions using a decision variable, it is even more surprising that it is possible to do the same for circles. However, Bresenham's algorithm accomplishes this, through the use of another decision variable, *d*. Obviously if it is possible to draw a circle without the square root and round operators in the main loop then we can significantly improve the speed of the algorithm.

```
procedure BresenhamCircle(in int R; colorval color) {
        declare:  int x, y, d;

        d = 1 – R;
        x ← 0;  y ← R;
        print8pixels(x, y, color);

        while (x < y) {
                x++;
                if (d < 0) {
                        d += 2 * x + 1;  /* go East */
                } else {
                        y--;
                        d += 2 * (x – y) + 1;  /* go SouthEast */
                } end if
                Print8pixels(x, y, color) if x ≤ y;
        } end while
} end procedure BresenhamCircle
```

E.g., for $R = 10$ the table below shows that it generates the same points as DDA.

x	*y*	*d*
0	10	-9
1	10	-6
2	10	-1
3	10	6
4	9	-3
5	9	8
6	8	5

7	7	6

Notice that, as promised, Bresenham's circle algorithm uses no floats, and only integer addition and comparisons. I won't prove that it works in this class, but will leave that for CS 525.

8. Bresenham Algorithm for General Circles

For both DDA and Bresenham we assumed that the circle was centered at the origin. In general a circle will have equation

$$(x - c_x)^2 + (y - c_y)^2 = R^2$$

Which is a circle with radius R centered at the point (c_x, c_y).

To generate all of the pixels for this circle we first generate all of the points (including symmetries) for the circle, radius R, centered at the origin, and then add (c_x, c_y) to each point to get displayed pixels. E.g., for the circle

$$(x - 3)^2 + (y + 1)^2 = 100$$

We first generate the primary octant points that we generated in the last section, and their symmetries, to get the 64 points shown in the first table below, then add (3, -1) to each point to get the pixels shown in the second table.

(x, y)	(y, x)	$(y, -x)$	$(x, -y)$	$(-x, -y)$	$(-y, -x)$	$(-y, x)$	$(-x, y)$
(0, 10)	(10, 0)	(10, 0)	(0, -10)	(0, -10)	(-10, 0)	(-10, 0)	(0, 10)
(1, 10)	(10, 1)	(10, -1)	(1, -10)	(-1, -10)	(-10, -1)	(-10, 1)	(-1, 10)
(2, 10)	(10, 2)	(10, -2)	(2, -10)	(-2, -10)	(-10, -2)	(-10, 2)	(-2, 10)
(3, 10)	(10, 3)	(10, -3)	(3, -10)	(-3, -10)	(-10, -3)	(-10, 3)	(-3, 10)
(4, 9)	(9, 4)	(9, -4)	(4, -9)	(-4, -9)	(-9, -4)	(-9, 4)	(-4, 9)
(5, 9)	(9, 5)	(9, -5)	(5, -9)	(-5, -9)	(-9, -5)	(-9, 5)	(-5, 9)
(6, 8)	(8, 6),	(8, -6),	(6, -8)	(-6, -8)	(-8, -6),	(-8, 6),	(-6, 8)
(7, 7)	(7, 7)	(7, -7)	(7, -7)	(-7, -7)	(-7, -7)	(-7, 7)	(-7, 7)

(3, 9)	(13, -1)	(13, -1)	(3, -11)	(3, -11)	(-7, -1)	(-7, -1)	(3, 9)
(4, 9)	(13, 0)	(13, -2)	(4, -11)	(2, -11)	(-7, -2)	(-7, 0)	(2, 9)
(5, 9)	(13, 1)	(13, -3)	(5, -11)	(1, -11)	(-7, -3)	(-7, 1)	(1, 9)
(6, 9)	(13, 2)	(13, -4)	(6, -11)	(0, -11)	(-7, -4)	(-7, 2)	(0, 9)
(7, 8)	(12, 3)	(12, -5)	(7, -10)	(-1, -10)	(-6, -5)	(-6, 3)	(-1, 8)
(8, 8)	(12, 4)	(12, -6)	(8, -10)	(-2, -10)	(-6, -6)	(-6, 4)	(-2, 8)
(9, 7)	(11, 5)	(11, -7)	(9, -9)	(-3, -9)	(-5, -7)	(-5, 5)	(-3, 7)
(10, 6)	(10, 6)	(10, -8)	(10, -8)	(-4, -8)	(-4, -8)	(-4, 6)	(-4, 6)

Note that the order matters. We must first generate all of the symmetries from the primary octant and then shift the points to the correct center. If we first shift the primary octant points to the correct center and then generate symmetries we will not get the correct result.

Also note that some points (on the axes and the diagonals) are generated twice. It is easy to define the function print8pixels so that this is avoided (which would be important if we were in **xor** write mode, where the second pixel displayed would delete the first).

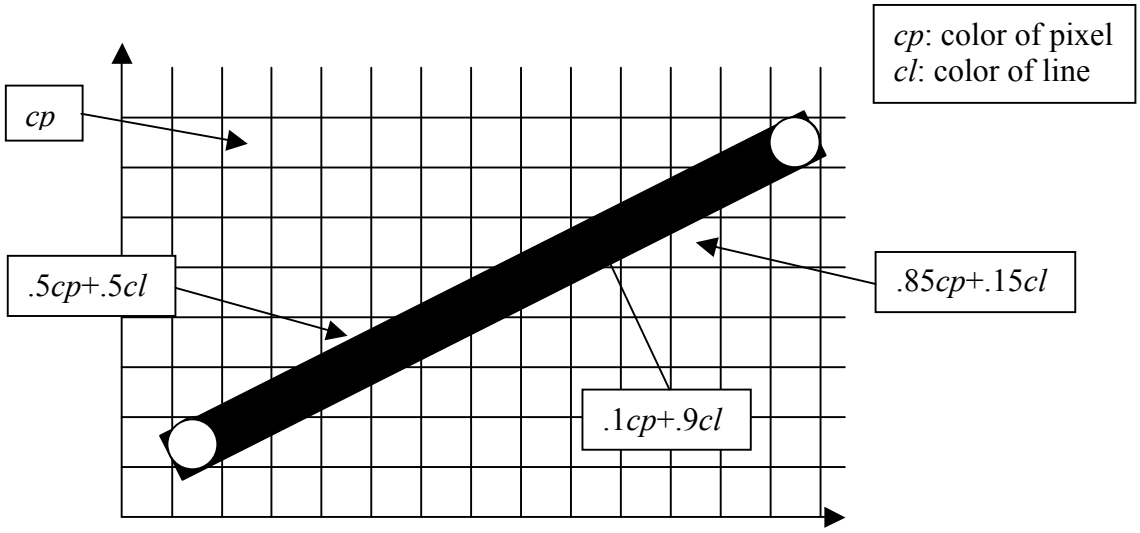
9. Other Conics

Bresenham can be modified to work for other conic curves, but this will also be left for CS 525. E.g., we can easily modify it for ellipses. One big difference, however, is that we can no longer use the eight-way symmetry of the circle, and so, for example, with ellipses we must generate a quadrant in two parts, working from the axes towards each other, and then use four-way symmetry to get the full ellipse.

10. Anti-Aliasing Lines

If it weren't for its computational cost, which is prohibitive in most applications, anti-aliasing would solve all line drawing problems. I'll just discuss drawing lines here, but the techniques apply to circles and any other edge-based figures.

There are a number of anti-aliasing techniques. Conceptually the easiest is to consider the effect of a rectangle in the line color going from the first pixel to the last, as shown below.



If a pixel is, say, 60% under this line and 40% not under the line, it will be assigned the color that is 60% line color and 40% the normal color for that pixel. On the diagram I've given some sample pixel values, where *cp* is the normal color of that pixel, and *cl* is the color of the line at that position. E.g., say that the pixel shown as being given the color .85*cp*+.15*cl* would have, without the line being drawn, the normalized RGB color (.4, .2, .6), and that the line color is a somewhat dark gray, (.4, .4, .4), then the color selected for the pixel will be (.4, .23, .57) because

$.4 \times .85 + .4 \times .15 = .4$
$.2 \times .85 + .4 \times .15 = .23$
$.6 \times .85 + .4 \times .15 = .57$