# Rasterization: Lines and Circles

We'll discuss various ways of drawing straight lines on a raster. First, we'll describe a simple algorithm which allows to draw a line using one or two floating point additions per pixel drawn. Then, we'll discuss the Bresenham's algorithm, allowing to draw a line with integer endpoints using only cheap integer additions. Then, we'll see how the jagged lines can be smoothed using a simple antialiasing idea based on the Bresenham's algorithm's decision variable. Finally, we'll describe a variant of the Bresenham's algorithm allowing to draw circles.

Before we start, notice that there is no unique and perfect way of drawing a line on a raster: Figure 1 shows that there are at least two reasonably looking possibilities. The algorithms presented here lead to something looking more like the first solution (upper line).

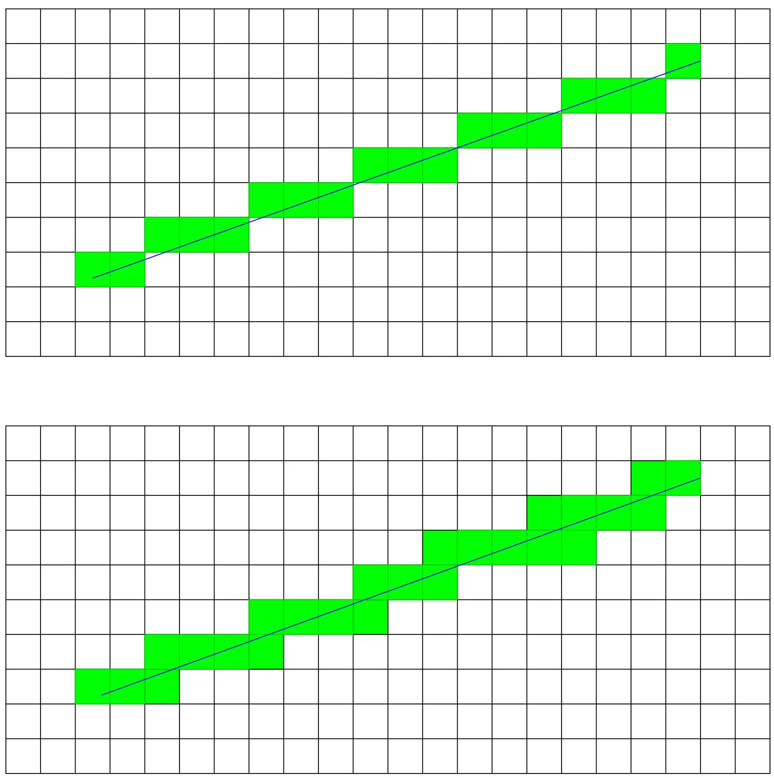
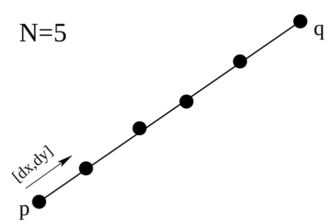


Figure 1: Two ways of scan converting a line segment: for each vertical row of pixels shade the one which is the closest to line in the vertical direction (top) and shade all pixels intersected by the line (bottom).

# 1 DDA algorithm (Digital Differential Analyzer)

Say that the line you want to draw starts at the point $p = (p_x, p_y)$ and ends at $q = (q_x, q_y)$. We'll move from $p$ to $q$ in small steps, each of which moves us by the vector $\vec{pq}/N$ where $N$ is a certain integer (We'll figure out later what it should be). For each point on the way, we find the pixel containing it (if the pixels have centers at integers, this can be done by rounding the coordinates to the nearest integer). All such pixels are shaded. This is shown in Figure 2.




```
x:=p_x;
y:=p_y;
dx := (q_x−p_x)/N;
dy := (q_y−p_y)/N;
for i:=0 to N do
    plot(Round(x),Round(y));
    x := x + dx;
    y := y + dy;
```

Figure 2: The DDA algorithm. The figure shows the points whose pixels are plotted if $N = 5$.

In order for the lines to be continuous, we need to ensure that we do not jump over entire row or column of pixels as we move from $p$ to $q$. On the other hand, we should not be overcautious, since too many steps will mean more computation than necessary. This means that the best choice for $N$ is

$$N = \lceil max(|p_x - q_x|, |p_y - q_y|) \rceil.$$

It ensures that both coordinates of the vector $[dx, dy]$ have magnitude less than 1. This is why no jumps over rows or columns of pixels occur.

Notice that if both endpoints have integer coordinates then, with the $N$ defined above, either $dx$ or $dy$ is equal to 1 (since $N$ is either $|p_x - q_x|$ or $|p_y - q_y|$). Therefore, we can actually compute one of the coordinates using integer arithmetic. There is also a possibility of computing both using integers but this requires a division operation for each plotted pixel (why?).

# 2 Bresenham's Algorithm

The Bresenham's algorithm uses only a few integer additions per shaded pixel. It assumes that the endpoints of the line have integer coordinates. We will assume that the line starts at $p = (0, 0)$ and ends at a point $q = (q_x, q_y)$ having integer coordinates and contained in the first octant of the coordinate system. That is, we'll assume $q_x >= q_y >= 0$ (Figure 3). All other cases can be reduced to the first-quadrant case by applying symmetries across the axes and translations.

Let's look at the centers of the pixels (round dots in Figure 4). They form a square grid of points. Let's call a vertical line passing through centers of

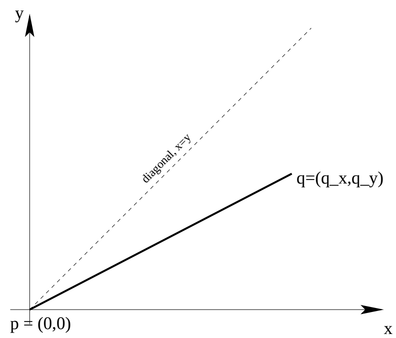



Figure 3: A line having the endpoint in the first quadrant

the pixels a scanline. We'll plot one pixel on each scanline intersecting the line segment to be drawn. The center of the plotted pixel will be the closest to the intersection of the scanline and the line segment (black dots in Figure 4 are the plotted pixels). Imagine that we follow the plotted points from left to right. As we do that, we either have to move horizontally to the next pixel on the right or along the diagonal to the one immediately above the one on the right. Thus, at each step, we need to increment the $x$ coordinate and possibly increment the $y$ coordinate (unless a horizontal move happens).

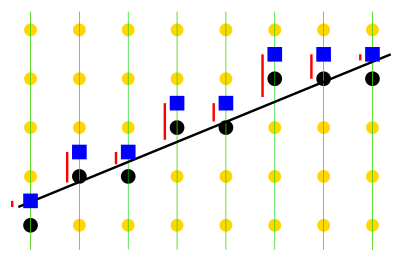


Figure 4: A line having the endpoint in the first quadrant

In the Bresenham's algorithm, we shall walk through the scanlines intersecting the line from left to right. At each step, we will decide whether we have to move horizontally or along the diagonal. The decision will be based on the *decision variable*. The decision variable will be the difference of the $y$ coordinates of the midpoint of the interval connecting the center of the pixel plotted and the pixel immediately above it (blue squares) and the intersection of the line and the scanline. In Figure 4, the value of the decision variables are the heights of the red bars. Because we wish to plot pixels closest to the line along scanlines, our algorithm has to ensure that all the decision variables are positive (geometrically: we need to plot the points in such a way that all the blue squares are above the line). It is not hard to see that if we move horizontally, then the value of the decision variable decreases by the slope of the line ($p_y/p_x$):

the blue square does not change height and the line-scanline intersection moves up by $p_y/p_x$. If the new value of the decision variable computed this way is negative, this would mean we rather need to move along the diagonal (to keep it positive). Consequently, the decision variable needs to be increased by 1. The algorithm is given in Figure 5.

```
d := 1/2; { q_x }              // initial value of the decision variable;
y := 0;                        // y−coordinate of the plotted points
for x:=0 to q_x do
  plot(x,y);
  d := d − q_y/q_x; { d−2*q_y} // update the decision variable; pretend that we move right for nov
  if d<0 then                  // if d<0 then we need to move along the diagonal
    y := y+1;
    d := d+1;     {d+2*q_x}
```

Figure 5: A line having the endpoint in the first quadrant

To make it useinteger operations only we can scale the decision variable by $2q_x$, which leads to replacing the right-hand sides of the assignments updating the $d$ variable with what is given in red in Figure 5.

## 3 Bresenham's Algorithm with Antialiasing

Antialiasing attempts to reduce the jaggy appearence of the rasterized lines. While there are many ways of doing this, one which particularly well incorporates into the Bresenham's algorithm works by distributing the intensity between two pixels on each scanline rather than assigning all of it to one. Two closest pixels closest to the line are shaded for each scanline. One is the pixel plotted by the algorithm in Figure 5. The other one is either the one immediately above it or below it, depending on whether the decision variable is less or greater than .5 (Figure 7). The intensities assigned to the pixels depend on their distances to the intersection point of the scanline and the line being drawn.

## 4 Bresenham Algorithm for Circles

Let's say we want to scan-convert a circle centered at $(0, 0)$ with an integer radius $R$ (Figure 7). We'll see that the ideas we previously used for line scan-conversion can be made work for this task. First of all, notice that the interior of the circle is characterized by the inequality $D(x, y) := x^2 + y^2 - R^2 < 0$. We'll use $D(x, y)$ to derive our decision variable.

First, let's think how to plot pixels close to the 1/8 of the circle marked red in the figure. The range of the $x$ coordinates for such pixels is from 0 to $R/\sqrt{2}$. We'll go over vertical scanlines through the centers of the pixels and, for each such scanline, compute the pixel on that line which is the closest to the scanline-circle intersection point (black dots in Figure 8). All such pixels will be plotted by our procedure.

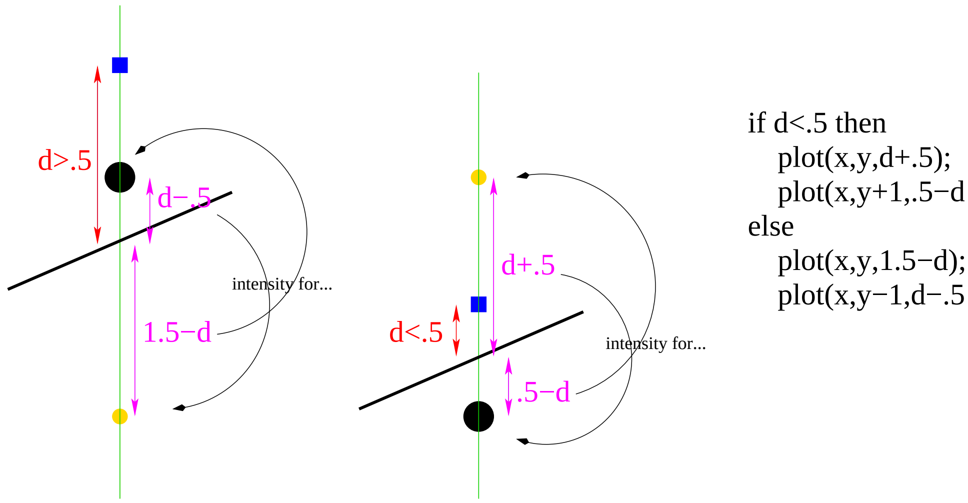



Figure 6: Two cases which need to be addressed in the antialiased variant of the Bresenham's Algorithm: Left: the plotted point is above the line; in this case we need to distribute intensity between this one and the one immediately below it. Center: the plotted point is below the line; in this case we need to distribute intensity between this one and the one immediately below it. Right: here is what needs to replace the plot(x,y) call in Fig. 5 if antialiased lines are to be produced (the third argument of plot is the intensity to be assigned to a pixel).

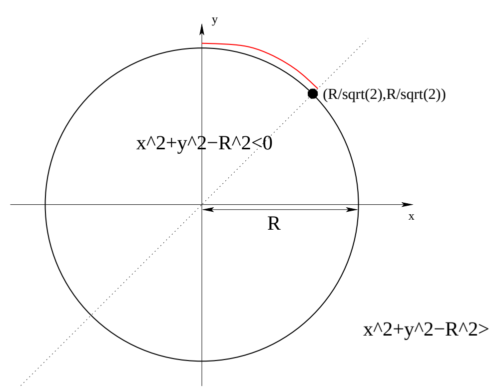



Figure 7: A circle and the description of its interior and exterior as two quadratic inequalities.

Notice that each time we move to the next scanline, the $y$-coordinate of the plotted point either stays the same or decreases by 1 (the slope of the circle there is between $-1$ and 0). To decide what needs to be done, we'll use the decision variable, which will be the value of $D(x, y)$ evaluated at the blue square, i.e. the

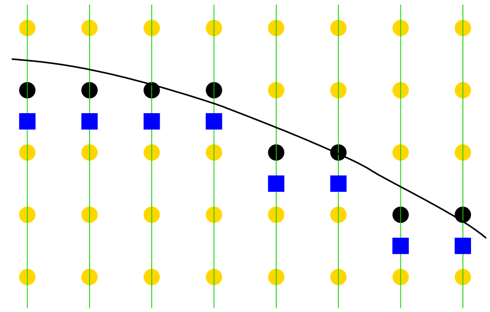


Figure 8: A circle and the description of its interior and exterior as two quadratic inequalities.

midpoint between the plotted pixel and the pixel immediately below.

The first pixel plotted is $(0, R)$ and therefore the initial value of the decision variable should be

$$D(0, R - .5) = (R - .5)^2 - R^2 = .25 - R.$$

The `y` variable, holding the second coordinates of the plotted pixels, will be initialized to $R$. Let's now think what happens after a point $(x, y)$ is plotted. First, we'll pretend that we need to move the plotted point to the right (no change in $y$) and check if this keeps the decision variable negative (we don't want any blue squares outside the circle!). If $(x, y)$ is the last plotted point, the decision variable is $D(x, y - .5)$. After we move to the right, it becomes $D(x+1, y-.5)$. Simple arithmetic shows that it increases by $D(x+1, y-.5) - D(x, y-.5) = 2x+1$. If this increase makes it positive, we'd better move down by 1 pixel. This puts the blue square at $(x+1, y-1.5)$ and means that we need to increase the decision by the previous $2x+1$ plus $D(x+1, y-1.5) - D(x+1, y-.5) = 2 - 2y$.

This leads to the algorithm in Figure 9. Clearly, to make the decision variable integer, we need to scale it by a factor of 4. All the multiplications by powers of two (when updates of the decision variables are made) can be done efficiently using shifts (<< operator in C). Eight-way symmetry is used to go from 1/8-th of the circle to the full circle.

```
y := R;
d := 1/4 − R;

for  x:=0  to  ceil(R/sqrt(2))  do
   plot_points(x,y);
   d += 2x+1;
   if (d>0)
      d += 2−2y;
      y−−;
```

```
plot_points(x,y)
   plot(x,y);
   plot(x,−y);
   plot(−x,y);
   plot(−x,−y);
   plot(y,x);
   plot(−y,x);
   plot(y,−x);
   plot(−y,−x);
```

Figure 9: Bresenham's algorithm for circles.