Chapter 2.

Scan Conversion

The process of representing continuous graphics objects as a collection of discrete pixels is called scan conversion. Many scan conversion algorithms are implemented in computer hardware or firmware. However if a specific graphics computer does not have the hardware conversion algorithm the scan conversion algorithm can be implemented in software

2.1 Scan-Converting Lines

On the screen we only have pixels with integer coordinates (in some range) only. So in order to display points with real coordinates, we have to round or truncate them. This is called *rastering*.

A line in computer graphics typically refers to a line segment, which is a portion of a straight line that extends indefinitely in opposite directions. It is defined by its two endpoints and the line equation *y= mx+ b*, where *m* is called the slope and *b* the y-intercept of the line.

When we draw the line on the screen, we have to pick points with integer coordinates as close as possible to the line. If the slope satisfies $-1 \leq m \leq 1$, then we should pick in any column with given integer value x one point as close as possible to the corresponding point on the line. Its y-coordinate is given by $\bar{y} = ROUND(mx + b)$. If the slope satisfies $|m| > 1$ we may exchange x and y, in order to reduce the problem to the particular case.

While this approach is mathematically sound, it involves floating-point computation (multiplication and addition) in every step that uses the line equation since m and b are generally real numbers. The challenge is to find a way to achieve the same goal as quickly as possible.

2.1.1 Digital Differential Analyzer (DDA) Method

The digital differential analyzer (DDA) algorithm is an incremental scan-conversion method. Such an approach is characterized by performing calculations at each step using results from the preceding step.

The basis of the DDA method is to take unit steps along one coordinate and compute the corresponding values along the other coordinate. The unit steps are always along the coordinate of greatest change, e.g. if $dx = 10$ and $dy = 5$, then we would take unit steps along x and compute the steps along y.

Suppose at step i we have calculated (x_i, y_i) to be a point on the line. Since the next point (x_{i+1}, y_{i+1}) should satisfy

$\frac{\Delta y}{\Delta x} = m$ where $\Delta y = y_{i+1} - y_i$ and $\Delta x = x_{i+1} - x_i$,

we have

$y_{i+1} = y_i + m\Delta x$

or $x_{i+1} = x_i + \Delta y / m$

Considering a line with positive slope, if $m \leq 1$, we sample at unit x intervals (i.e., $dx = 1$) and compute successive y values as

$$y_{i+1} = y_i + m$$

Subscript i takes integer values starting from 0 (for the 1st point), and increases by 1 until endpoint is reached. y value is rounded off to nearest integer to correspond to a screen pixel.

For lines with $m > 1$, we reverse the role of x and y (i.e., we sample at $dy = 1$) and calculate consecutive x values as

$$x_{i+1} = x_i + 1/m$$

Similar calculations are carried out to determine pixel positions along a line with negative slope.

The DDA algorithm is faster than the direct use of the line equation since it calculates points on the line without any floating-point multiplication. However, a floating-point addition is still needed in determining each successive point. Furthermore, cumulative error due to limited precision in the floating-point representation may cause calculated points to drift away from their true position when the line is relatively long.

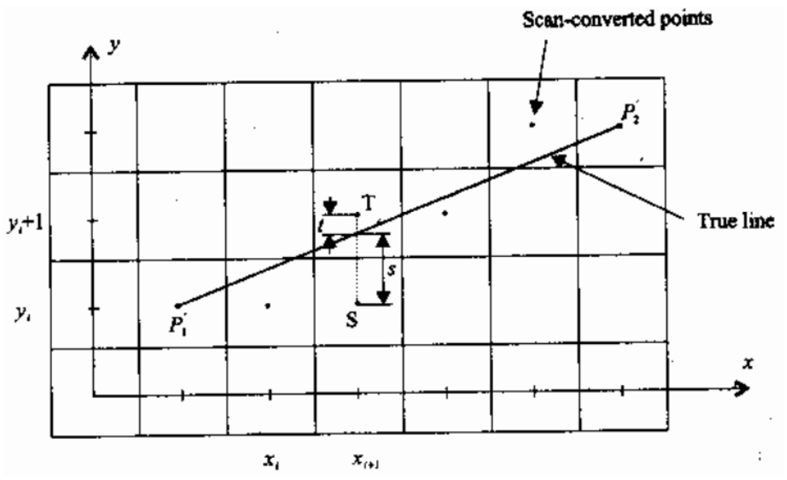
The complete DDA algorithm is given below.

```
void line_dda(int xa,int ya,int xb,int yb){
        int dx = xb-xa,  dy = yb-ya, steps, k;
        float xincrement, yincrement, x = xa, y = ya;

        if(abs(dx) > abs(dy))
                        steps = abs(dx);
                else    steps = abs(dy);

        xincrement = dx/steps;
        yincrement = dy/steps;

        putpixel(round(x), round(y))

        for(k=0; k<steps; k++){
                x += xincrement;
                y += yincrement;
                putpixel(round(x),round(y));
        }
}
```

2.1.2 Bresenham's Line Algorithm

Bresenham's line algorithm is a highly efficient incremental method for scan-converting lines. It produces mathematically accurate results using only integer addition, subtraction, and multiplication by 2, which can be accomplished by a simple arithmetic shift operation.

The method works as follows. Assume that we want to scan-convert the line shown in the figure below, where $0 < m < 1$. We start with pixel $P'_1(x'_1, y'_1)$, then select subsequent pixels as we work our way to the right, one pixel position at a time in the horizontal direction towards $P'_2(x'_2, y'_2)$.



Once a pixel is chosen at any step, the next pixel is either the one to its right (which constitutes a lower bound for the line) or the one to its right and up (which constitutes an upper bound for the line) due to the limit on m. The line is best approximated by those pixels that fall the least distance from its true path between $P'_1$and P'_2.

Using the notation of the above figure, the coordinates of the last chosen pixel upon entering step i are (x_i, y_i). Our task is to choose the next one between the bottom pixel S and the top pixel T. If S is chosen, we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$. If T is chosen, we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i + 1$.

The actual y coordinate of the line at $x = x_i + 1$ is $y = mx_{i+1} + b = m(x_i + 1) + b$. The distance from S to the actual line in the y direction is $s = y - y_i$. The distance from T to the actual line in the y direction is $t = (y_i + 1) - y$.

Now consider the difference between these two distance values: $s - t$. When $(s - t) < 0$, we have $s < t$ and the closest pixel is S. Conversely, when $(s - t) > 0$, we have $s > t$ and the closest pixel is T. We also choose T when $s - t = 0$. This difference is

$$s - t = (y - y_i) - [(y_i + 1) - y]$$

$$= 2y - 2y_i - 1 = 2m(x_i + 1) + 2b - 2y_i - 1$$

Substituting m by $\Delta y/\Delta x$ and introducing a decision variable $d_i = \Delta x(s - t)$, which has the same sign as $(s - t)$ since Δx is positive in our case, we have

$$d_i = 2\Delta y * x_i - 2\Delta x * y_i + C \text{ where } C = 2\Delta y + \Delta x(2b - 1)$$

Similarly, we can write the decision variable d_{i+1} for the next step as

$$d_{i+1} = 2\Delta y * x_{i+1} - 2\Delta x * y_{i+1} + C$$

Then, $\quad d_{i+1} - d_i = 2\Delta y\,(x_{i+1} - x_i) - 2\Delta x\,(y_{i+1} - y_i)$

Since $x_{i+1} = x_i + 1$, we have $\quad d_{i+1} = d_i + 2\Delta y - 2\Delta x\,(y_{i+1} - y_i)$

If the chosen pixel is the top pixel T (meaning that $d_i > 0$) then $y_{i+1} = y_i + 1$ and so

$$d_{i+1} = d_i + 2(\Delta y - \Delta x)$$

On the other hand, if the chosen pixel is the bottom pixel S (meaning that $d_i < 0$) then $y_{i+1} = y_i$ and so

$$d_{i+1} = d_i + 2\Delta y$$

Hence we have

$$d_{i+1} = \begin{cases} d_i + 2(\Delta y - \Delta x) & \text{if } d_i \geq 0 \\ d_i + 2\Delta y & \text{if } d_i < 0 \end{cases}$$

Finally, we calculate d_1, the base case value for this recursive formula, from the original definition of the decision variable d_i:

$$d_1 = \Delta x[2m(x_1 + 1) + 2b - 2y_1 - 1] = \Delta x[2(mx_1 + b - y_1) + 2m - 1]$$

Since $mx_1 + b - y_1 = 0$, we have

$$d_1 = 2\Delta y - \Delta x$$

In summary, Bresenham's algorithm for scan-converting a line from $P'_1(x'_1, y'_1)$, to $P'_2(x'_2, y'_2)$, with $x'_1 < x'_2$ and $0 < m < 1$ can be stated as follows:

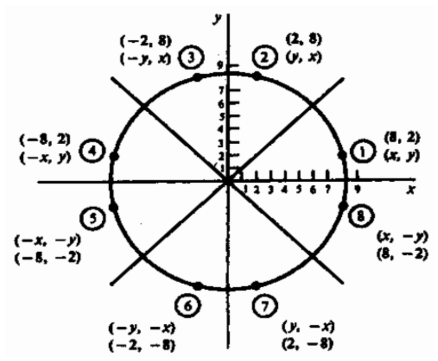
```
int $x = x'_1$, $y = y'_1$;
int $dx = x'_2 - x'_1$, $dy = y'_2 - y'_1$, $dT = 2(dy - dx)$, $dS = 2dy$;
int $d = 2dy - dx$;
setPixel($x$, $y$);
while ($x < x'_2$) {
    $x$++;
    if ($d < 0$)
        $d = d + dS$;
    else {
        $y$++;
        $d = d + dT$;
    }
    setPixel($x$, $y$);
}
```

Here we first initialize decision variable d and set pixel P'$_1$. During each iteration of the while loop, we increment x to the next horizontal position, then use the current value of d to select the bottom or top (increment y) pixel and update d, and at the end set the chosen pixel.

As for lines that have other m values we can make use of the fact that they can be mirrored either horizontally, vertically, or diagonally into this 0° to 45° angle range.

2.2 Scan-Converting a Circle

A circle is a symmetrical figure. Any circle-generating algorithm can take advantage of the circle's symmetry to plot eight points for each value that the algorithm calculates. Eight-way symmetry is used by reflecting each calculated point around each 45^o axis.



$P_1 = (x, y)$	$P_5 = (-x, -y)$
$P_2 = (y, x)$	$P_6 = (-y, -x)$
$P_3 = (-y, x)$	$P_7 = (y, -x)$
$P_4 = (-x, y)$	$P_8 = (x, -y)$

There are two standard methods of mathematically defining a circle cantered at the origin. The first method defines a circle with the second-order polynomial equation

$$y^2 = r^2 - x^2$$

where x = the x coordinate
y = the y coordinate
r = the circle radius

With this method, each y coordinate is found, from 90^o to 45^o, by evaluating the function for each step of x. This is a very inefficient method, however, because for each point both x and r must be squared and subtracted from each other; then the square root of the result must be found.

The second method of defining a circle makes use of trigonometric functions

$$x = r\cos\theta \qquad y = r\sin\theta$$

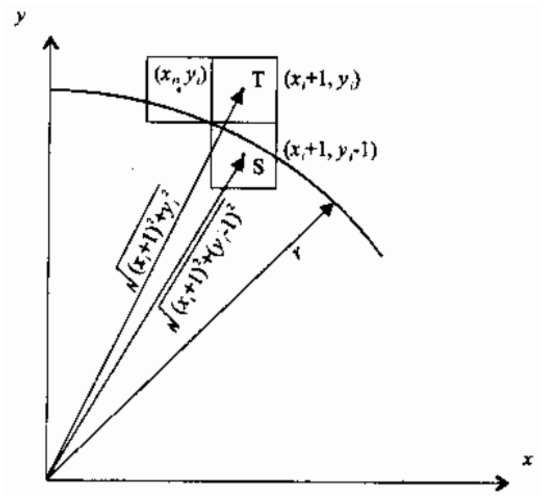
where θ = current angle
r = circle radius
x = x coordinate
y = y coordinate

By this method, θ is stepped from 0 to π/4, and each value of x and y is calculated. However, computation of the values of sinθ and cosθ is even more time consuming than the calculations required by the first method.

2.2.1 Bresenham's Circle Algorithm

Scan-converting a circle using Bresenham's algorithm works as follows. If the eight-way symmetry of a circle is used to generate a circle, points will only have to be generated through a 45^0 angle. And, if points are generated from 90^0 to 45^0, moves will be made only in the (+x) and (-y) directions.

The best approximation of the true circle will be described by those pixels in the raster that fall the least distance from the true circle. Notice that, if points are generated from 90º and 45^0, each new point closest to the true circle can be found by taking either of two actions: (1) move in the x direction one unit or (2) move in the x direction one unit and move in the negative y direction one unit. Therefore, a method of selecting between these two choices is all that is necessary to find the points closest to the true circle.



Assume that (x_i, y_i) are the coordinates of the last scan-converted pixel upon entering step *i*. Let the distance from the origin to pixel T squared minus the distance to the true circle squared = D(T). Then let the distance from the origin to pixel S squared minus the distance to the true circle squared = D(S). As the coordinates of T are $(x_i + 1, y_i)$ and those of S are $(x_i + 1, y_i - 1)$, the following expressions can be developed:

$$D(T) = (x_i + 1)^2 + y_i^2 - r^2 \qquad D(S) = (x_i + 1)^2 + (y_i - 1)^2 - r^2$$

This function D provides a relative measurement of the distance from the centre of a pixel to the true circle. Since D(T) will always be positive (T is outside the true circle) and D(S) will always be negative (S is inside the true circle), a decision variable di may be defined as follows:

$$d_i = D(T) + D(S)$$

Therefore

$$d_i = 2(x_i + 1)^2 + y_i^2 + (y_i - 1)^2 - 2r^2$$

When $d_i < 0$, we have |D(T)| < |D(S)| and pixel T is chosen. When $d_i \geq 0$, we have |D(T)| ≥ | D(S)| and pixel S is selected. We can also write the decision variable d_{i+1} for the next step:

$$d_{i+1} = 2(x_{i+1} + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2r^2$$

Hence

$$d_{i+1} - d_i = 2(x_{i+1} + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2(x_i + 1)^2 - y_i^2 - (y_i - 1)^2$$

Since $x_{i+1} = x_i + 1$, we have

$$d_{i+1} = d_i + 4x_i + 2(y_{i+1}^2 - y_i^2) - 2(y_{i+1} - y_i) + 6$$

If T is the chosen pixel (meaning that $d_i < 0$) then $y_{i+1} = y_i$ and so

$$d_{i+1} = d_i + 4x_i + 6$$

On the other hand, if S is the chosen pixel (meaning that $d_i \geq 0$) then $y_{i+1} = y_i - 1$ and so

$$d_{i+1} = d_i + 4(x_i - y_i) + 10$$

Hence we have

$$d_{i+1} = \begin{cases} d_i + 4x_i + 6 & \text{if } d_i < 0 \\ d_i + 4(x_i - y_i) + 10 & \text{if } d_i \geq 0 \end{cases}$$

Finally, we set (0, *r*) to be the starting pixel coordinates and compute the base case value d_1 for this recursive formula from the original definition of d_i:

$$d_1 = 2(0 + 1)^2 + r^2 + (r - 1)^2 - 2r^2 = 3 - 2r$$

We can now summarize the algorithm for generating all the pixel coordinates in the 90° to 45° octant that are needed when scan-converting a circle of radius *r*:

```
int x = 0, y = r, d = 3 − 2r;
while (x <= y) {
    setPixel(x, y);
    if (d < 0)
        d = d + 4x + 6;
    else {
      d = d + 4(x − y) +10;
      y−−;
      }
    x++;
}
```

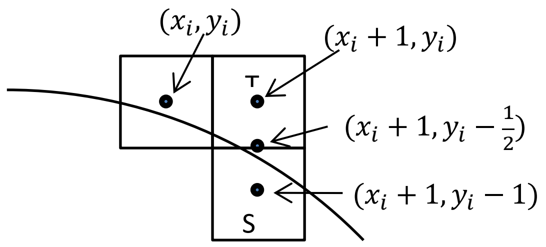
Note that during each iteration of the while loop we first set a pixel whose position has already been determined, starting with (0, *r*). We then test the current value of decision variable *d* in order to update *d* and determine the proper *y* coordinate of the next pixel. Finally we increment *x*.

2.2.2 Midpoint Circle Algorithm

Midpoint circle algorithm is another incremental circle algorithm that is very similar to Bresenham's approach. It is based on the following function for testing the spatial relationship between an arbitrary point U, y) and a circle of radius r centered at the origin:

$$f(x,y) = \begin{cases} < 0 & \text{for } (x,y) \text{ inside the circle} \\ = 0 & \text{for } (x,y) \text{ on the circle} \\ > 0 & \text{for } (x,y) \text{ outside the circle} \end{cases}$$

Now consider the coordinates of the point halfway between pixel T and pixel S: $(x_i + 1, y - \frac{1}{2})$



This is called the midpoint and we use it to define a decision parameter:

$$P_i = f\left(x_i + 1, y - \frac{1}{2}\right) = (x_i + 1)^2 + (y_i - \frac{1}{2})^2 - r^2$$

If P_i is negative, the midpoint is inside the circle, and we choose pixel T. On the other hand, if P_i is positive (or equal to zero), the midpoint is outside the circle (or on the circle), and we choose pixel S. Similarly, the decision parameter for the next step is

$$P_{i+1} = (x_{i+1} + 1)^2 + (y_{i+1} - \frac{1}{2})^2 - r^2$$

Since $x_{i+1} = x_i + 1$, we have

$$P_{i+1} - P_i = [(x_i + 1) + 1]^2 + (x_i + 1)^2 + (y_{i+1} - \frac{1}{2})^2 - (y_i - \frac{1}{2})^2$$

Hence

$$P_{i+1} = P_i + 2(x_i + 1) + 1 + (y_{i+1}^2 - y_i^2) - (y_{i+1} - y_i)$$

If pixel T is chosen (meaning $P_i < 0$), we have $y_{i+1} = y_i$. On the other hand, if pixel S is chosen (meaning $P_i \geq 0$), we have $y_{i+1} = y_i - 1$. Thus

$$P_{i+1} = \begin{cases} P_i + 2(x_i + 1) + 1 & if\ P_i < 0 \\ P_i + 2(x_i + 1) + 1 - 2(y_i - 1) & if\ P_i \geq 0 \end{cases}$$

We can continue to simplify this and get,

$$P_{i+1} = \begin{cases} P_i + 2x_i + 3 & if\ P_i < 0 \\ P_i + 2(x_i - y_i) + 5 & if\ P_i \geq 0 \end{cases}$$

Finally, we compute the initial value for the decision parameter using the original definition of P_i and $(0, r)$:

$$P_1 = (0+1)^2 + (r - \tfrac{1}{2})^2 - r^2 = \tfrac{5}{4} - r$$

One can see that this is not really integer computation. However, when r is a integer we can simply set $P_1 = 1 - r$.

The error of being $^1/_4$ less than the precise value does not prevent P_1 from getting the appropriate sign. It does not affect the rest of the scan-conversion process either, because the decision variable is only updated with integer increments in subsequent steps.

The following is a description of this midpoint circle algorithm that generates the pixel coordinates in the 90° to 45° octant:

```
int x = 0, y = r, p = 1 − r;
while (x < = y) {
   setPixel(x, y);
   if (p < 0)
      p = p + 2x + 3;
   else {
      p = p + 2(x − y) + 5;
      y − − ;
   }
   x ++ ;
}
```

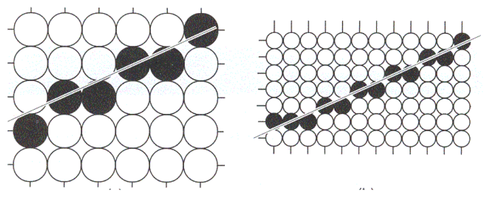
2.3 Aliasing and Antialiasing

Since the resolution of a screen is limited, we can see jaggies (or staircases) on a line drawn using the algorithms described before. This undesirable effect is due to the all-or-nothing approach to scan conversion. This phenomenon is called aliasing.

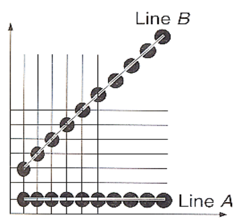
To reducing aliasing, there are some antialiasing methods.

2.3.1 Increasing Resolution

By increasing the resolution, the appearance of the lines will be improved but the jaggies effect would not be removed. Also, this will require more memory and increase scan conversion time.



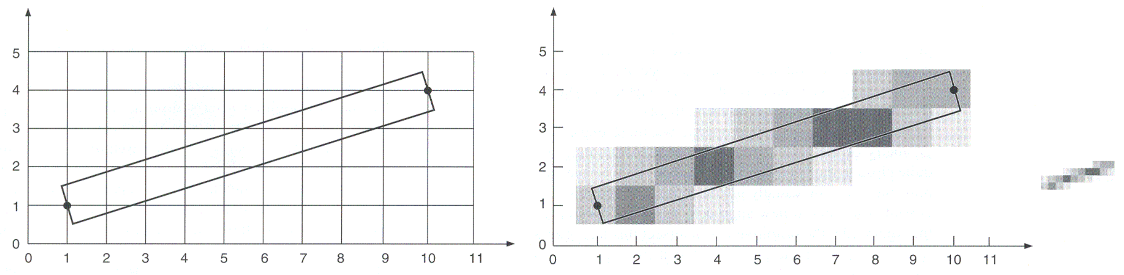
2.3.2 Varying Intensity of Lines as a Function of Slope



Since the densities of the pixels along lines with different slopes are different, slope line intensities are adjusted according to horizontal and vertical lines. Horizontal and vertical lines with minimum intensity and 45° lines are given max intensity. The number of pixels turned on for a 45° line is the same as that for a horizontal but the length of this 45° line is $\sqrt{2}$ times longer, so its intensity has to be increased by $\sqrt{2}$ times.

2.3.3 Overlapping Axes

A pixel is a spot covering a small area of the screen. Lines have a width equal to that of a pixel. We may think that any line as a rectangle covering a portion of a grid, which may represents a screen, as shown in the figure below.



Then only the squares (or pixels) covered by line will be turned on. To perform antialiasing, the intensity of each square depends on how much the square is covered by the line. If a square is totally inside the line, it will have 100% of intensity; otherwise, it will only have an intensity proportional to the area covered.

The problem of this method is that it will complicate the scan conversion algorithm a lot.