# Scan-Conversion

## Topics

- Scan-Converting
- a Point,
- a Line,
- a Circle,
- an Ellipse,
- Arcs and Sectors
- a Rectangle
- a Character

# Scan-Converting a Point

- A mathematical point (x, y)
  - where x and y are real numbers within an image area,
  - needs to be scan-converted to a pixel at location (x’, y’).
- This may be done by making
  - x’ to be the integer part of x and
  - y’ the integer part of y.
- In other words,
  - x’ = Floor(x) and y’ = Floor(y),
  - the function Floor() returns the largest integer
  - that is less than or equal to the argument.

# Scan-Converting a Point

- All points that satisfy

  x’<= x <=x’+1 and

  y’ <= y <=y’+1 are mapped to pixel (x’, y’).

For example,

- point $p_1$ (1.7, 0.8) is represented by pixel (1, 0).
- point $p_2$ (2.3, 1.9) is represented by pixel (2, 1).
- point $p_3$ (3.7, 4.9) is represented by pixel (3, 4).
- point $p_4$ (7.6, 8.9) is represented by pixel (7, 8).

# Scan-Converting a Line

- 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.
- Two endpoints are described by $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$.
- The line equation describes the coordinates of all points the lie between the two endpoints.

# Scan-Converting a Line

- The slope-intercept equation is not suitable for vertical lines.
- Horizontal, vertical, and diagonal ($|m|=1$) lines can and often should, be handled as special cases without going through the following scan-conversion algorithms.
- A line connects two points.
- It is a basic element in graphics.
- To draw a line, you need two points between which you can draw a line.
- In the following three algorithms,
  - we refer the one point of line as $(x_0,y_0)$ and
  - the second point of line as $(x_1, y_1)$.

# Scan-Converting a Line

**Algorithms are:**

- DDA (Digital Differential Analyzer) Algorithm
- Bresenham’s Line Algorithm
- Mid-Point Algorithm

# DDA(Digital Differential Analyzer) Algorithm

Digital Differential Analyzer(*DDA)* algorithm is the simple line generation algorithm which is explained step by step here.

**Step 1:** Get the input of two end points $(x_0, y_0)$ and $(x_1, y_1)$.
**Step 2**: Calculate the difference between two end points.

$dx = x_1 - x_0$

$dy = y_1 - y_0$

**Step 3**: Based on the calculated difference in Step 2, you need to identify the number of steps to put pixel. If dx > dy, then you need more steps in x coordinate; otherwise in y coordinate.

```
if (dx > dy)
        Steps = absolute(dx);
else
        Steps = absolute(dy);
```

# DDA(Digital Differential Analyzer) Algorithm

**Step 4**: Calculate the increment in x coordinate and y coordinate.

```
Xincrem ent = dx / (float) steps;
Yincrem ent = dy / (float) steps;
```

**Step 5:** Put the pixel by successfully incrementing x and y coordinates accordingly and complete the drawing of the line.

```
for(int v=0; v < Steps; v++)
{
        x = x + Xincrem ent;
        y = y + Yincrem ent;
        putpixel(x,y);
}
```

# Bresenham's Line Algorithm

- The Bresenham algorithm is another incremental scan conversion algorithm.
- The big advantage of this algorithm is that, it uses only integer calculations.
- Moving across the x axis in unit intervals and at each step choose between two different y coordinates.

**Step 1**: Input the two end-points of line, storing the left end-point in $(x0, y0)$.
**Step 2**: Plot the point $(x0, y0)$.
**Step 3**: Calculate the constants dx, dy, 2dy, and $2dy - 2dx$ and get the first value for the decision parameter as –

$$p0 = 2dy - dx$$

# Bresenham's Line Algorithm

**Step 4**: At each $X_k$ along the line, starting at k = 0, perform the following test −

If $p_k < 0$, the next point to plot is $(x_k + 1, y_k)$ and

$$p_{k+1} = p_k + 2dy$$

Otherwise, $p_{k+1} = p_k + 2dy - 2dx$

**Step 5:** Repeat step 4 $dx - 1$ times.

For m > 1, find out whether you need to increment x while incrementing y each time.

After solving, the equation for decision parameter $P_k$ will be very similar, just the x and y in the equation gets interchanged.

# Bresenham's Line Algorithm

In short,

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 $o<m<1$ can be stated as follows:

```
int x = x1, y =y1;
int dx = x2 - x1, dy = y2 - y1, dT = 2(dy - dx), dS = 2dy;
int d = 2dy – dx;
setPixel(x,y);
while(x<x2)
{
        x++;
        if(d<0)
                d = d + dS;
        else
        {
                y++;
                d = d + dT;
        }
        setPixel(x, y);
}
```

# Bresenham's Line Algorithm: Description

- 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 **$0^0$ to $45^0$** angle range.
- For example,
  - a line from **$(x_1', y_1')$ to $(x_2', y_2')$** with $-1 <= m < 0$
  - has a horizontally mirrored counterpart
  - from $(x_1', -y_1')$ to $(x_2', -y_2')$ with $0 <= m < 1$.

# Bresenham's Line Algorithm: Description

- We can simply use the algorithm
  - to scan-convert this counterpart,
  - but negate the y coordinate at the end of each iteration
  - to set the right pixel for the line.
- For a line whose slope is in the $45^0$ to $90^0$ range,
  - we can obtain its mirrored counterpart
  - by exchanging the x and y coordinates of its endpoints.
- We can then scan-convert this counterpart
  - but we must exchange x and y in the call to setPixel.

# 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 to reflecting each calculated point
  - around each $45^0$ axis.

- For example, if **point 1** were calculated
  - with a circle algorithm,
  - seven more points could be found by reflection.

# Scan-Converting a Circle

- The reflection is accomplished by reversing the x, y coordinates as in **point 2**,
  - reversing the x, y coordinates and reflecting about the y axis as in **point 3**,
  - reflecting about the y axis as in **point 4**,
  - switching the signs of x and y as in **point 5**,
  - reversing the x, y coordinates and reflecting about the x axis as in **point 6**,
  - reversing the x, y coordinates and reflecting about the y axis as in **point 7**, and
  - reflecting about the x axis as in **point 8**.

# Scan-Converting a Circle …

**To summarize,**

$p_1 = (x, y)$ $\quad$ $p_5 = (-x, -y)$

$p_2 = (y, x)$ $\quad$ $p_6 = (-y, -x)$

$p_3 = (-y, x)$ $\quad$ $p_7 = (y, -x)$

$p_4 = (-x, y)$ $\quad$ $p_8 = (x, -y)$

**To summarize,**

$p_1 = (8, 2)$ $\quad$ $p_5 = (-8, -2)$

$p_2 = (2, 8)$ $\quad$ $p_6 = (-2, -8)$

$p_3 = (-2, 8)$ $\quad$ $p_7 = (2, -8)$

$p_4 = (-8, 2)$ $\quad$ $p_8 = (8, -2)$

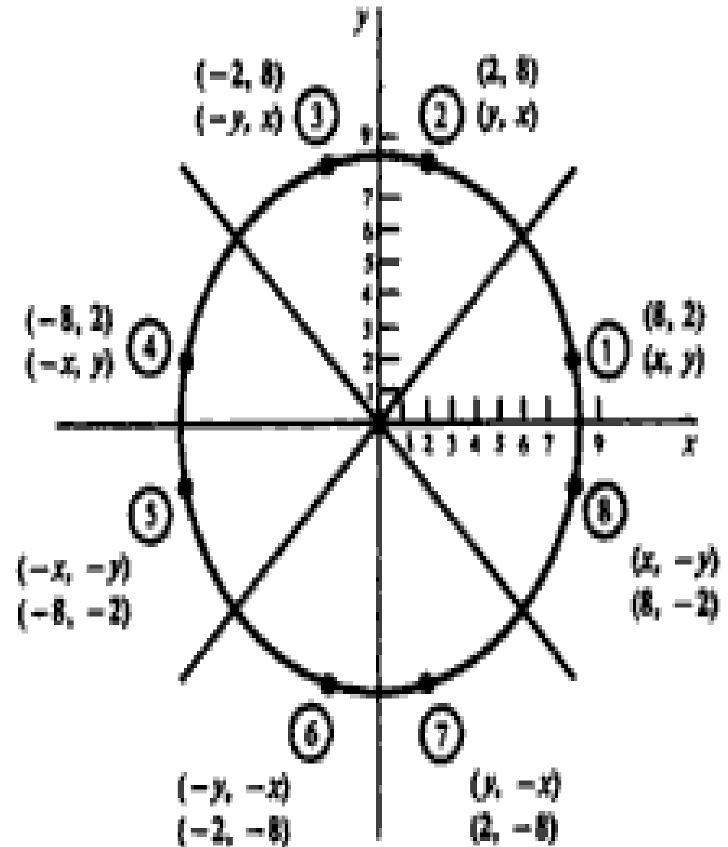



Fig. 3-4 Eight-way symmetry of a circle.

# Defining a Circle

- There are **two standard method**s of mathematically defining a circle centered at the origin.
  - **Polynomial Method**
  - **Trigonometric Method**

- 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 and
$r$ = the circle radius.

# Defining a Circle

- With this method, each x coordinate in the sector,
  - from $90^0$ to $45^0$, is found by stepping
  - x from 0 to $r/(\sqrt{2})$, and
  - each y coordinate is found by evaluating $\sqrt{(r^2-x^2)}$ 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.

# Defining a Circle

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

  $x = r \cos\theta$ and

  $y = r \sin\theta$

  where $\theta$ = current angle

  $r$ = circle radius

  $x$ = x coordinate

  $y$ = y coordinate

- By this method, $\theta$ is stepped from 0 to $\pi/4$, and
  - each value of x and y is calculated.
- However, computation of the values of $\sin\theta$ and
  - $\cos\theta$ is even more time-consuming
  - than the calculations required by the first method.

# Mathematical Problems

# Solved Problems

## Problem 01:

**3.1** The endpoints of a given line are (0, 0) and (6, 18). Compute each value of $y$ as $x$ steps from 0 to 6 and plot the results.

**SOLUTION**

An equation for the line was not given. Therefore, the equation of the line must be found. The equation of the line ($y = mx + b$) is found as follows. First the slope is found:

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{18 - 0}{6 - 0} = \frac{18}{6} = 3$$

Next, the $y$ intercept $b$ is found by plugging $y_1$ and $x_1$ into the equation $y = 3x + b$: $0 = 3(0) + b$. Therefore, $b = 0$, so the equation for the line is $y = 3x$ (see Fig. 3-32).

# Solved Problems

## Problem 02:

**3.2** What steps are required to plot a line whose slope is between 0° and 45° using the slope–intercept equation?

**SOLUTION**

1. Compute $dx$: $dx = x_2 - x_1$.
2. Compute $dy$: $dy = y_2 - y_1$.
3. Compute $m$: $m = dy/dx$.
4. Compute $b$: $b = y_1 - m \times x_1$.
5. Set $(x, y)$ equal to the lower left-hand endpoint and set $x_{end}$ equal to the largest value of $x$. If $dx < 0$, then $x = x_2$, $y = y_2$, and $x_{end} = x_1$. If $dx > 0$, then $x = x_1$, $y = y_1$, and $x_{end} = x_2$.
6. Test to determine whether the entire line has been drawn. If $x > x_{end}$, stop.
7. Plot a point at the current $(x, y)$ coordinates.
8. Increment $x$: $x = x + 1$.
9. Compute the next value of $y$ from the equation $y = mx + b$.
10. Go to step 6.

# Solved Problems

## Problem 03:

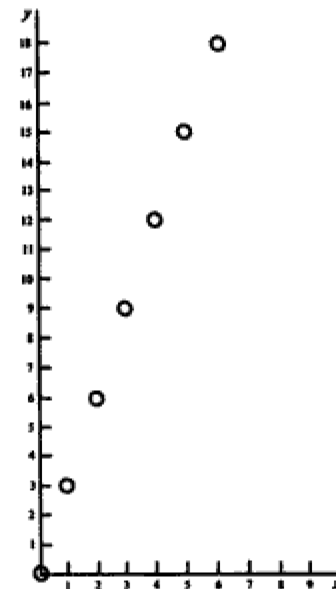
**3.3** Use pseudo-code to describe the steps that are required to plot a line whose slope is between 45° and −45° (i.e., $|m| > 1$) using the slope–intercept equation.

### SOLUTION

Presume $y_1 < y_2$ for the two endpoints $(x_1, y_1)$ and $(x_2, y_2)$:

```
int x = x1, y = y1;
float xf, m = (y2 - y1)/(x2 - x1), b = y1 - m x1;
setPixel(x, y);
while (y < y2) {
    y++;
    xf = (y - b)/m;
    x = Floor(xf + 0.5);
    setPixel(x, y);
}
```

| $y = 3x + 0$ | $x$ |
|---|---|
| 0 | 0 |
| 3 | 1 |
| 6 | 2 |
| 9 | 3 |
| 12 | 4 |
| 15 | 5 |
| 18 | 6 |



# Solved Problems

## Problem 04:

**3.6** What steps are required to plot a line whose slope is between 0° and 45° using Bresenham's method?

**SOLUTION**

1. Compute the initial values:

$$dx = x_2 - x_1 \qquad Inc_2 = 2(dy - dx)$$
$$dy = y_2 - y_1 \qquad d = Inc_1 - dx$$
$$Inc_1 = 2dy$$

2. Set $(x, y)$ equal to the lower left-hand endpoint and $x_{\text{end}}$ equal to the largest value of $x$. If $dx < 0$, then $x = x_2$, $y = y_2$, $x_{\text{end}} = x_1$. If $dx > 0$, then $x = x_1$, $y = y_1$, $x_{\text{end}} = x_2$.
3. Plot a point at the current $(x, y)$ coordinates.
4. Test to see whether the entire line has been drawn. If $x = x_{\text{end}}$, stop.
5. Compute the location of the next pixel. If $d < 0$, then $d = d + Inc_1$. If $d \geq 0$, then $d = d + Inc_2$, and then $y = y + 1$.
6. Increment $x$: $x = x + 1$.
7. Plot a point at the current $(x, y)$ coordinates.
8. Go to step 4.

# Solved Problems

## Problem 05:

**3.7** Indicate which raster locations would be chosen by Bresenham's algorithm when scan-converting a line from pixel coordinate (1, 1) to pixel coordinate (8, 5).

**SOLUTION**

First, the starting values must be found. In this case

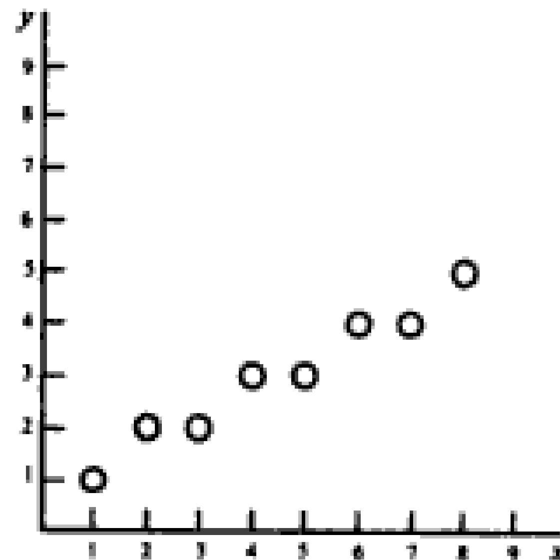
$$dx = x_2 - x_1 = 8 - 1 = 7 \qquad dy = y_2 - y_1 = 5 - 1 = 4$$

Therefore:

$$Inc_1 = 2dy = 2 \times 4 = 8$$
$$Inc_2 = 2(dy - dx) = 2 \times (4 - 7) = -6$$
$$d = Inc_1 - dx = 8 - 7 = 1$$

The following table indicates the values computed by the algorithm (see also Fig. 3-33).

| $d$ | $x$ | $y$ |
|---|---|---|
| $1$ | 1 | 1 |
| $1 + Inc_2 = -5$ | 2 | 2 |
| $-5 + Inc_1 = 3$ | 3 | 2 |
| $3 + Inc_2 = -3$ | 4 | 3 |
| $-3 + Inc_1 = 5$ | 5 | 3 |
| $5 + Inc_2 = -1$ | 6 | 4 |
| $-1 + Inc_1 = 7$ | 7 | 4 |
| $7 + Inc_2 = 1$ | 8 | 5 |



# Solved Problems

## Problem 06:

**3.10** What steps are required to generate a circle using the polynomial method?

**SOLUTION**

1. Set the initial variables: $r$ = circle radius; $(h, k)$ = coordinates of the circle center; $x = 0$; $i$ = step size; $x_{end} = r/\sqrt{2}$.
2. Test to determine whether the entire circle has been scan-converted. If $x > x_{end}$, stop.
3. Compute the value of the $y$ coordinate, where $y = \sqrt{r^2 - x^2}$.
4. Plot the eight points, found by symmetry with respect to the center $(h, k)$, at the current $(x, y)$ coordinates:

$$\begin{array}{ll} \text{Plot}(x + h, y + k) & \text{Plot}(-x + h, -y + k) \\ \text{Plot}(y + h, x + k) & \text{Plot}(-y + h, -x + k) \\ \text{Plot}(-y + h, x + k) & \text{Plot}(y + h, -x + k) \\ \text{Plot}(-x + h, y + k) & \text{Plot}(x + h, -y + k) \end{array}$$

5. Increment $x$: $x = x + i$.
6. Go to step 2.

# Solved Problems

## Problem 07:

**3.11** What steps are required to scan-convert a circle using the trigonometric method?

**SOLUTION**

1. Set the initial variables: $r$ = circle radius; $(h, k)$ = coordinates of the circle center; $i$ = step size; $\theta_{end} = \pi/4$ radians = 45°; $\theta = 0$.
2. Test to determine whether the entire circle has been scan-converted. If $\theta > \theta_{end}$, stop.
3. Compute the value of the $x$ and $y$ coordinates:
4. Plot the eight points, found by symmetry with respect to the center $(h, k)$, at the current $(x, y)$ coordinates:

$$\begin{array}{ll} \text{Plot}(x+h, y+k) & \text{Plot}(-x+h, -y+k) \\ \text{Plot}(y+h, x+k) & \text{Plot}(-y+h, -x+k) \\ \text{Plot}(-y+h, x+k) & \text{Plot}(y+h, -x+k) \\ \text{Plot}(-x+h, y+k) & \text{Plot}(x+h, -y+k) \end{array}$$

5. Increment $\theta$: $\theta = \theta + i$.
6. Go to step 2.

# Solved Problems

## Problem 08:

**3.12** What steps are required to scan-convert a circle using Bresenham's algorithm?

**SOLUTION**

1. Set the initial values of the variables: $(h, k)$ = coordinates of circle center; $x = 0$; $y$ = circle radius $r$; and $d = 3 - 2r$.
2. Test to determine whether the entire circle has been scan-converted. If $x > y$, stop.
3. Plot the eight points, found by symmetry with respect to the center $(h, k)$, at the current $(x, y)$ coordinates:

$$\begin{array}{ll} \text{Plot}(x + h, y + k) & \text{Plot}(-x + h, -y + k) \\ \text{Plot}(y + h, x + k) & \text{Plot}(-y + h, -x + k) \\ \text{Plot}(-y + h, x + k) & \text{Plot}(y + h, -x + k) \\ \text{Plot}(-x + h, y + k) & \text{Plot}(x + h, -y + k) \end{array}$$

4. Compute the location of the next pixel. If $d < 0$, then $d = d + 4x + 6$ and $x = x + 1$. If $d \geq 0$, then $d = d + 4(x - y) + 10$, $x = x + 1$, and $y = y - 1$.
5. Go to step 2.

# Solved Problems

## Problem 09:

**3.20** What steps are required to generate an ellipse using the polynomial method?

**SOLUTION**

1. Set the initial variables: $a$ = length of major axis; $b$ = length of minor axis; $(h, k)$ = coordinates of ellipse center; $x = 0$; $i$ = step size; $x_{end} = a$.
2. Test to determine whether the entire ellipse has been scan-converted. If $x > x_{end}$, stop.
3. Compute the value of the $y$ coordinate:

$$y = b\sqrt{1 - \frac{x^2}{a^2}}$$

4. Plot the four points, found by symmetry, at the current $(x, y)$ coordinates:

$$\text{Plot}(x + h, y + k) \qquad \text{Plot}(-x + h, -y + k)$$
$$\text{Plot}(-x + h, y + k) \qquad \text{Plot}(x + h, -y + k)$$

5. Increment $x$: $x = x + i$.
6. Go to step 2.

# Solved Problems

## Problem 10:

**3.21** What steps are required to scan-convert an ellipse using the trigonometric method?

**SOLUTION**

1. Set the initial variables: $a$ = length of major axis; $b$ = length of minor axis; $(h, k)$ = coordinates of ellipse center; $i$ = counter step size; $\theta_{end} = \pi/2$; $\theta = 0$.
2. Test to determine whether the entire ellipse has been scan-converted. If $\theta > \theta_{end}$, stop.
3. Compute the values of the $x$ and $y$ coordinates:

$$x = a\cos(\theta) \qquad y = b\sin(\theta)$$

4. Plot the four points, found by symmetry, at the current $(x, y)$ coordinates:

$$\text{Plot}(x+h, y+k) \qquad \text{Plot}(-x+h, -y+k)$$
$$\text{Plot}(-x+h, y+k) \qquad \text{Plot}(x+h, -y+k)$$

5. Increment $\theta$: $\theta = \theta + i$.
6. Go to step 2.

# Solved Problems

## Problem 11:

**3.24** What steps are required to scan-convert an arc using the trigonometric method?

**SOLUTION**

1. Set the initial variables: $a$ = major axis; $b$ = minor axis; $(h, k)$ = coordinates of arc center; $i$ = step size; $\theta$ = starting angle; $\theta_1$ = ending angle.
2. Test to determine whether the entire arc has been scan-converted. If $\theta > \theta_1$, stop.
3. Compute the values of the $x$ and $y$ coordinates:

$$x = a\cos(\theta) + h \qquad y = a\sin(\theta) + k$$

4. Plot the points at the current $(x, y)$ coordinates: Plot$(x, y)$.
5. Increment $\theta$: $\theta = \theta + i$.
6. Go to step 2.

(Note: for the arc of a circle $a = b$ = circle radius $r$.)

# Solved Problems

## Problem 12:

**3.25** What steps are required to generate an arc of a circle using the polynomial method?

**SOLUTION**

1. Set the initial variables: $r$ = radius; $(h, k)$ = coordinates of arc center; $x$ = $x$ coordinate of start of arc; $x_1$ = $x$ coordinate of end of arc; $i$ = counter step size.
2. Test to determine whether the entire arc has been scan-converted. If $x > x_1$, stop.
3. Compute the value of the $y$ coordinate:

$$y = \sqrt{r^2 - x^2}$$

4. Plot at the current $(x, y)$ coordinates:

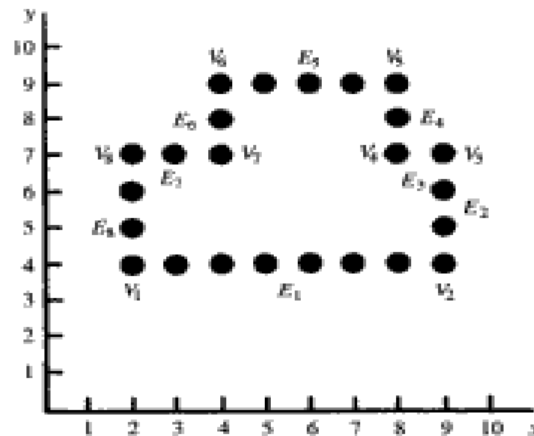
$$\text{Plot}(x + h, y + k)$$

5. Increment $x$: $x = x + i$.
6. Go to step 2.

# Solved Problems

## Problem 13:

**3.30** The coordinates of the vertices of a polygon are shown in Fig. 3-39. (*a*) Write the initial edge list for the polygon. (*b*) State which edges will be active on scan lines $y = 6, 7, 8, 9,$ and $10$.



**SOLUTION**

(*a*) Column $x$ contains the $x$ coordinate of the corresponding edge's lower endpoint. Horizontal edges are not included.

| Edge | $y_{min}$ | $y_{max}$ | $x$ | $1/m$ |
|---|---|---|---|---|
| $E_2$ | 4 | 7 | 9 | 0 |
| $E_8$ | 4 | 7 | 2 | 0 |
| $E_4$ | 7 | 9 | 8 | 0 |
| $E_6$ | 7 | 9 | 4 | 0 |

# Solved Problems

(*b*) An edge becomes active when the scan line value $y$ equals the edge's $y_{min}$ value. The edge remains active until the scan line value $y$ goes beyond the edge's $y_{max}$ value. Therefore, the active edges for $y = 6, 7, 8, 9$, and 10 appears as follows.

At $y = 6$, $E_2$ and $E_3$.

At $y = 7$, $y = y_{max}$ for both edges $E_2$ and $E_3$ so they remain active. Also at $y = 7$, edges $E_4$ and $E_6$ become active.

At $y = 8$, $E_2$ and $E_3$ are removed from the edge list. $E_4$ and $E_6$ remain active.

At $y = 9$, the active edges remain the same. At $y = 10$, edges $E_2$ and $E_4$ are removed from the edge list and the edge list becomes empty.

# Solved Problems

## Problem 14:

**3.33** Write a pseudo-code procedure for generating the Koch curve $K_n$ (after the one in the text for generating $C_n$).

**SOLUTION**

```
Koch-curve (float x, y, len, alpha; int n)
{
   if (n > 0) {
      len = len/3;
      Koch-curve(x, y, len, alpha, n − 1);
      x = x + len*cos(alpha);
      y = y + len*sin(alpha);
      Koch-curve(x, y, len, alpha − 60, n − 1);
      x = x + len*cos(alpha − 60);
      y = y + len*sin(alpha − 60);
      Koch-curve(x, y, len, alpha + 60, n − 1);
      x = x + len*cos(alpha + 60);
      y = y + len*sin(alpha + 60);
      Koch-curve(x, y, len, alpha, n − 1);
   } else
      line(x, y, x + len*cos(alpha), y + len*sin(alpha));
}
```

# Solved Problems

## Problem 15:

**3.34** Presume that the following statement produces a filled triangle with vertices at $(x_1, y_1)$, $(x_2, y_2)$, and $(x_3, y_3)$:

triangle($x_1, y_1, x_2, y_2, x_3, y_3$)

Write a pseudo-code procedure for generating the Sierpinski gasket $S_n$ (after the procedure in the text for generating $C_n$).

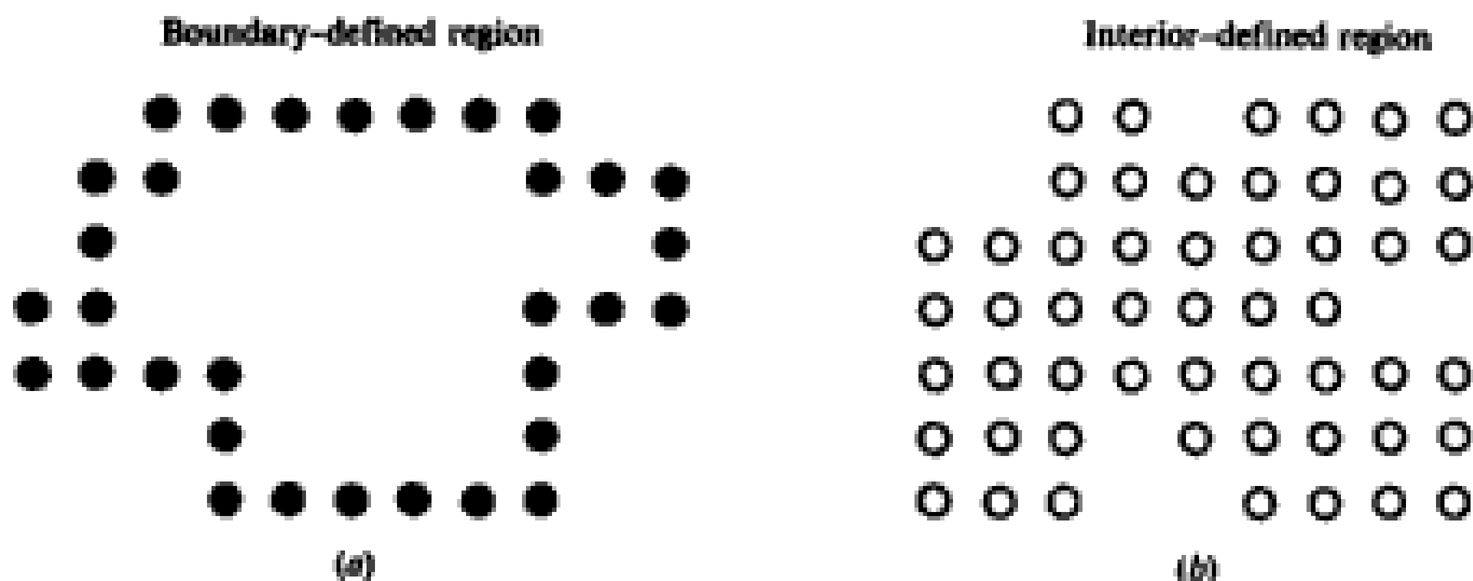
**SOLUTION**

```
S-Gasket (float x1, y1, x2, y2, x3, y3; int n)
{
  float x12, y12, x13, y13, x23, y23;
  if (n > 0) {
    x12 = (x1 + x2)/2;
    y12 = (y1 + y2)/2;
    x13 = (x1 + x3)/2;
    y13 = (y1 + y3)/2;
    x23 = (x2 + x3)/2;
    y23 = (y2 + y3)/2;
    S-Gasket(x1, y1, x12, y12, x13, y13, n - 1);
    S-Gasket(x12, y12, x2, y2, x23, y23, n - 1);
    S-Gasket(x13, y13, x23, y23, x3, y3, n - 1);
  } else
    triangle(x1, y1, x2, y2, x3, y3);
}
```

# Thanks

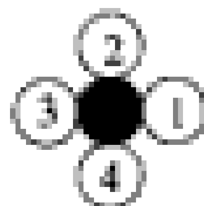
## 3.7 REGION FILLING

Region filling is the process of "coloring in" a definite image area or region. Regions may be defined at the pixel or geometric level. At the pixel level, we describe a region either in terms of the bounding pixels that outline it or as the totality of pixels that comprise it (see Fig, 3-17). In the first case the region is called boundary-defined and the collection of algorithms used for filling such a region are collectively called boundary-fill algorithms. The other type of region is called an interior-defined region and the accompanying algorithms are called flood-fill algorithms. At the geometric level a region is defined or enclosed by such abstract contouring elements as connected lines and curves. For example, a polygonal region, or a filled polygon, is defined by a closed polyline, which is a polyline (i.e., a series of sequentially connected lines) that has the end of the last line connected to the beginning of the first line.

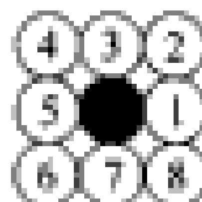


#### 4-Connected vs. 8-Connected

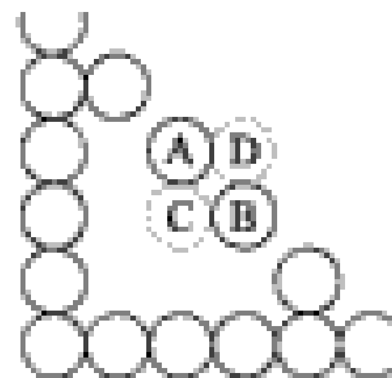
An interesting point here is that, while a geometrically defined contour clearly separates the interior of a region from the exterior, ambiguity may arise when an outline consists of discrete pixels in the image space. There are two ways in which pixels are considered connected to each other to form a "continuous" boundary. One method is called 4-connected, where a pixel may have up to four neighbors [see Fig. 3-18(*a*)]; the other is called 8-connected, where a pixel may have up to eight neighbors [see




(*a*)




(*b*)




(*c*)

**Fig. 3-18** 4-connected vs. 8-connected pixels.

### A Boundary-fill Algorithm

This is a recursive algorithm that begins with a starting pixel, called a seed, inside the region. The algorithm checks to see if this pixel is a boundary pixel or has already been filled. If the answer is no, it fills the pixel and makes a recursive call to itself using each and every neighboring pixel as a new seed. If the answer is yes, the algorithm simply returns to its caller.

This algorithm works elegantly on an arbitrarily shaped region by chasing and filling all non-boundary pixels that are connected to the seed, either directly or indirectly through a chain of neighboring relations. However, a straightforward implementation can take time and memory to execute due to the potentially high number of recursive calls, especially when the size of the region is relatively large.

### A Flood-fill Algorithm

This algorithm also begins with a seed (starting pixel) inside the region. It checks to see if the pixel has the region's original color. If the answer is yes, it fills the pixel with a new color and uses each of the pixel's neighbors as a new seed in a recursive call. If the answer is no, it returns to the caller.

This method shares great similarities in its operating principle with the boundary-fill algorithm. It is particularly useful when the region to be filled has no uniformly colored boundary. On the other hand, a

### A Scan-line Algorithm

In contrast to the boundary-fill and flood-fill algorithms that fill regions defined at the pixel level in the image space, this algorithm handles polygonal regions that are geometrically defined by the coordinates of their vertices (along with the edges that connect the vertices). Although such regions can be filled by first scan-converting the edges to get the boundary pixels and then applying a boundary-fill algorithm to finish the job, the following is a much more efficient approach that makes use of the information regarding edges that are available during scan conversion to facilitate the filling of interior pixels.

We represent a polygonal region in terms of a sequence of vertices $V_1, V_2, V_3, \ldots$, that are connected by edges $E_1, E_2, E_3, \ldots$, (see Fig. 3-19). We assume that each vertex $V_i$ has already been scan-converted to integer coordinates $(x_i, y_i)$.

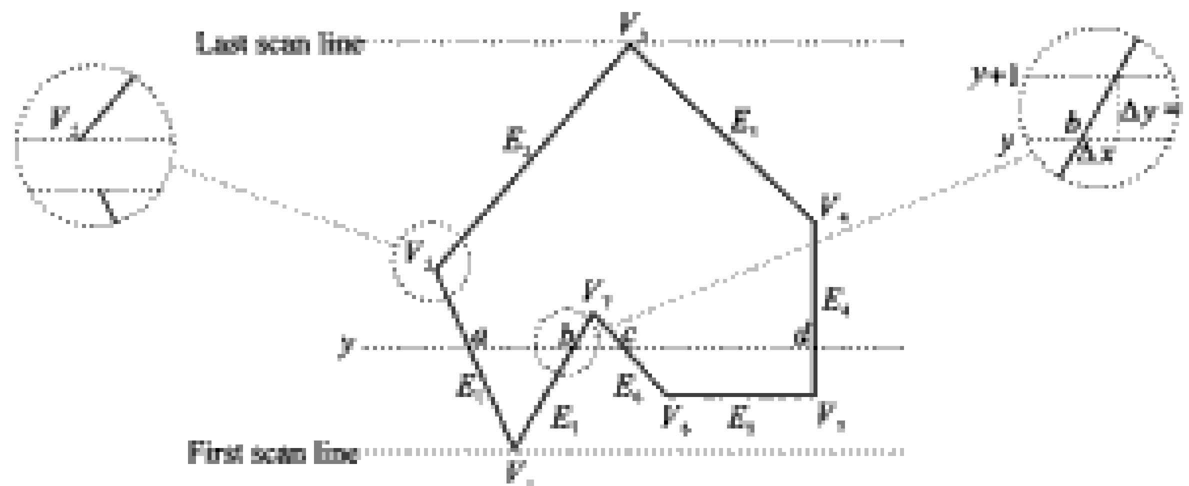



**Fig. 3-19** Scan-converting a polygonal region.

Table 3-1 An edge list.

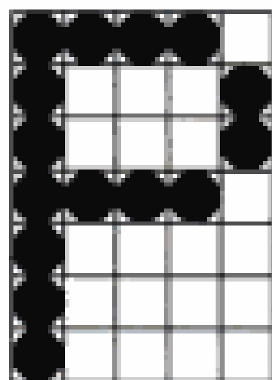
| Edge | $y_{min}$ | $y_{max}$ | $x$ coordinate of vertex with $y = y_{min}$ | $1/m$ |
|---|---|---|---|---|
| $E_1$ | $y_1$ | $y_2 - 1$ | $x_1$ | $1/m_1$ |
| $E_7$ | $y_1$ | $y_7$ | $x_1$ | $1/m_7$ |
| $E_4$ | $y_5$ | $y_4 - 1$ | $x_5$ | $1/m_4$ |
| $E_6$ | $y_6$ | $y_7$ | $x_6$ | $1/m_6$ |
| $E_2$ | $y_2$ | $y_3$ | $x_2$ | $1/m_2$ |
| $E_3$ | $y_4$ | $y_3$ | $x_4$ | $1/m_3$ |

## 3.8 SCAN-CONVERTING A CHARACTER

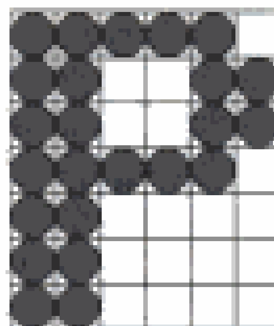
Characters such as letters and digits are the building blocks of an image's textual contents. They can be presented in a variety of styles and sizes. The overall design style of a set of characters is referred to as its typeface or font. Commonly used fonts include Arial, Century Schoolbook, Courier, and Times New Roman. In addition, fonts can vary in appearance: **bold**, *italic*, and ***bold and italic***. Character size is typically measured in height in inches, points (approximately $\frac{1}{72}$ inch), and picas (12 points).

### Bitmap Font

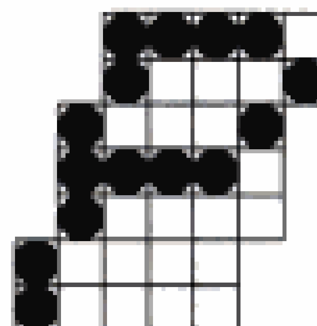
There are two basic approaches to character representation. The first is called a raster or bitmap font, where each character is represented by the on pixels in a bilevel pixel grid pattern called a bitmap (see Fig. 3-20). This approach is simple and effective since characters are defined in already-scan-converted form. Putting a character into an image basically entails a direct mapping or copying of its bitmap to a specific



'ig. 3-20 Bitmap font.



(*a*) Bold



(*b*) Italic

Fig. 3-21 Generating variations in appearance.

### Outline Font

The second character representation method is called a vector or outline font, where graphical primitives such as lines and arcs are used to define the outline of each character (see Fig. 3-22). Although an outline definition tends to be less compact than a bitmap definition and requires relatively time-consuming scan-conversion operations, it can be used to produce characters of varying size, appearance, and even orientation. For example, the outline definition in Fig. 3-22 can be resized through a scaling transformation, made into italic through a shearing transformation, and turned around with respect to a reference point through a rotation transformation (see Chap. 4).

These transformed primitives can be scan-converted directly into characters in the form of filled regions in the target image area. Or they can be used to create the equivalent bitmaps that are then used to

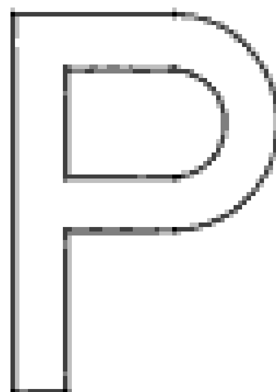


**Fig. 3-22** Outline font

## 3.9 ANTI-ALIASING

Scan conversion is essentially a systematic approach to mapping objects that are defined in continuous space to their discrete approximation. The various forms of distortion that result from this operation are collectively referred to as the aliasing effects of scan conversion.

### Staircase

A common example of aliasing effects is the staircase or jagged appearance we see when scan-converting a primitive such as a line or a circle. We also see the stair steps or "jaggies" along the border of a filled region.

## Unequal Brightness

Another artifact that is less noticeable is the unequal brightness of lines of different orientation. A slanted line appears dimmer than a horizontal or vertical line, although all are presented at the same intensity level. The reason for this problem can be explained using Fig. 3-23, where the pixels on the horizontal line are placed one unit apart, whereas those on the diagonal line are approximately 1.414 units apart. This difference in density produces the perceived difference in brightness.

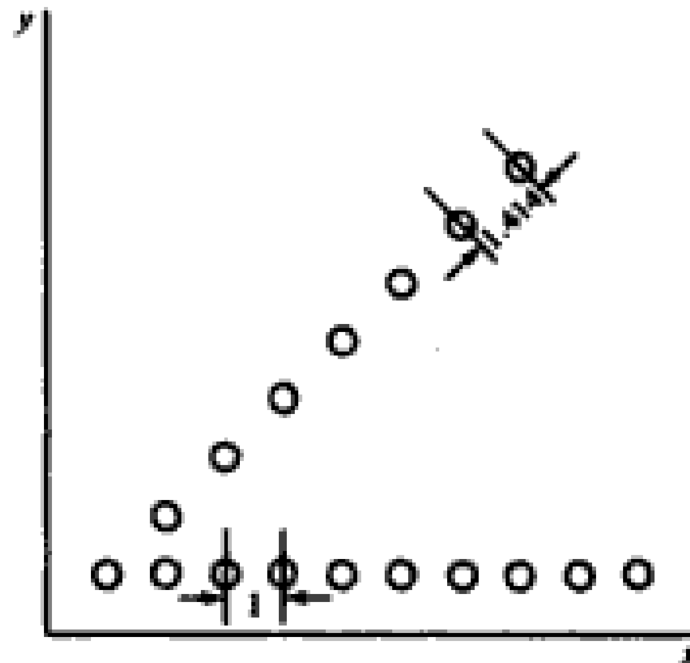



Fig. 3-23

# Thanks