Chapter 2
2D Graphics

Abstract Objects can be drawn, edited and transformed in 2D and 3D spaces. One can create objects ranging from simple 2D primitives to complex 3D environments. Planar objects, sometimes called 2D shapes, are created on a single plane (usually the *xy*-plane) while 3D objects utilize all three dimensions of space. Focusing on planar objects, 2D primitives like lines, circles, etc. may be used to create more complex shapes. Thus, it is important to know how to create/draw such primitives as groups of pixels utilizing their equations; a process that is referred to as *scan conversion* (Working with graphics as mathematical equations is referred to as *vector graphics* while working with them as a series of pixels is called *raster graphics*.). A line can be scan-converted (i.e., expressed as a set of pixels that approximate its path) if we know its endpoints, which define its equation. Likewise, a circle is scan-converted if we know the location of its center as well as its radius; i.e., components that define its equation. The same concept applies to other primitives. Different operations may be applied to objects in 2D space. For example, an object can be clipped using a clip window or a clip polygon to preserve a part of that object. (Some systems refer to the clipping process as *trimming*.) Other operations that may be applied to such objects are called *transformations*. Such operations include translating (i.e., moving), rotating, scaling, stretching, reflecting (i.e., mirroring) and shearing the objects. In this chapter, we will look at how to create 2D primitives like lines and circles. Also, we will talk about polygons and polylines and how to clip line segments and polygons. We will dedicate Chap. 3 to talk about different transformations in 2D space. 3D object creation and operations will be discussed in subsequent chapters. Before starting the discussion, we should mention that there are two different Cartesian coordinate systems that could be used. These are left-handed and right-handed coordinate systems (see Sect. B.1.1). In the right-handed coordinate system, the origin is at the lower left corner of the screen and the *y*-axis is pointing upwards. On the contrary, in the left-handed coordinate system, the origin is at the upper left corner of the screen and the *y*-axis is pointing downwards. In this chapter, we will show a line drawn in both systems only once. Afterwards, we will stick to the right-handed coordinate system to avoid confusion. (The usual convention working with raster images is to use the

R. Elias, *Digital Media*, DOI: 10.1007/978-3-319-05137-6_2,

left-handed coordinate system where the origin is at the upper left corner and the y-axis is pointing downwards. However, switching between the left- and right-hand coordinate system is easy and can be done using Eq. (B.7).)

2.1 Lines

Drawing a line on a computer screen given its two endpoints is done by intensifying (or turning on) pixels along the path of this line. Figure 2.1a depicts an example of a line starting at pixel $[x_0, y_0]^T$ and ending at pixel $[x_1, y_1]^T$ where these pairs indicate the column and row numbers of the pixel respectively. The pixels to be intensified (i.e., those that are shaded in Fig. 2.1) could be *exactly on* or *close to* the linear path. Such a line can be expressed as

$$y - y_0 = \underbrace{\frac{y_1 - y_0}{x_1 - x_0}}_{slope}(x - x_0). \tag{2.1}$$

This can be written as

$$y = m(x - x_0) + y_0, \tag{2.2}$$

where $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$ is the slope of the line.

Note that the coordinate system used in Fig. 2.1a is a right-handed coordinate system where the origin is at the lower left corner of the screen and the y-axis is pointing upwards. (This is the coordinate system that will be used throughout this book unless otherwise specified.) Alternatively, the same line is drawn using a left-handed coordinate system in Fig. 2.1b where the origin is at the upper left corner and the y-axis is pointing downwards. Consequently, the slope of the line (i.e., m) in these cases is <1. Practically, the cases where the line is horizontal, vertical or having a slope of ± 1 are handled as special cases.

2.1.1 Digital Differential Analyzer Algorithm

A simple and direct way to draw the line is performed by looping or iterating through Eq. (2.2), incrementing the value of x from x_0 to x_1 (i.e., moving from one column to the next) and obtaining the corresponding y to plot the pixel $[x, y]^T$. There are some remarks to be mentioned here.

1. The x-value is an integer value as it is a column number; however, the resulting y-value could be a floating point number. In this case, the y-value must be rounded to the nearest integer in order to have the value $\lfloor y + 0.5 \rfloor$ where $\lfloor . \rfloor$ denotes the floor operation.

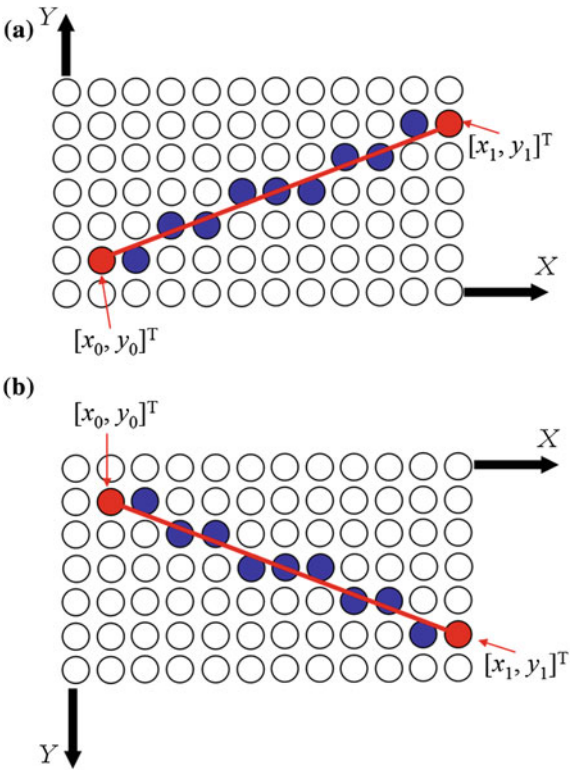

Fig. 2.1 A 2D line as a path from $[x_0, y_0]^T$ to $[x_1, y_1]^T$. The slope of this line is <1. **a** The line is drawn using the *right-handed* coordinate system where the y-axis is pointing upwards. **b** The line is drawn using the *left-handed* coordinate system where the y-axis is pointing downwards

2. At each iteration, the slope m is re-calculated although the slope is a constant number that does not change for the entire line. This could be time consuming. Thus, the value m should be pre-calculated before looping.

   Using Eq. (2.2) at iteration i, we can write

$$y_i = m(x_i - x_0) + y_0 = mx_i + \underbrace{y_0 - mx_0}_{B}, \tag{2.3}$$

where $B = y_0 - mx_0$ is the y-intercept, which is a constant value for the line. Since the x increment is 1, the subsequent value of y is obtained as

$$y_{i+1} = mx_{i+1} + B = m(x_i + 1) + B = \underbrace{mx_i + B}_{y_i} + m = y_i + m. \tag{2.4}$$

This means that subsequent y-values can be calculated by adding the value of the slope m to the previous y-value at each iteration. Thus, for $|m| \leq 1$ as in Fig. 2.1, the algorithm can be written as follows:

Algorithm 2.1 *DDA1 algorithm*

Input: x_0, y_0, x_1, y_1
1: $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$
2: $y = y_0$
3: **for** $(x = x_0$ to $x_1)$ **do**
4: Plot $[x, \lfloor y + 0.5 \rfloor]^T$
5: $y = y + m$
6: **end for**

end

This algorithm is referred to as the *digital differential analyzer (DDA) algorithm*. Note that for $|m| \leq 1$, a fraction is added to the previous y-value each time. This means that, after rounding, y may maintain the same value as in the previous iteration or be incremented by 1.

Example 2.1: [*DDA line drawing algorithm*]
Using the DDA line drawing algorithm, determine the pixels along a line segment that goes from $[3, 4]^T$ to $[8, 6]^T$.

Solution 2.1: According to Algorithm 2.1, the first step is to calculate the slope m where

$$m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0} = \frac{6 - 4}{8 - 3} = 0.4.$$

The second step is to loop along the x-direction from 3 to 8 using a step size of 1 and get the corresponding y-value by adding the value m to the previous y as listed in the following table:

3: x	4: $\lfloor y + 0.5 \rfloor$	4: Plot	5: y
3	4	$[3, 4]^T$	$4.0 + 0.4 = 4.4$
4	4	$[4, 4]^T$	$4.4 + 0.4 = 4.8$
5	5	$[5, 5]^T$	$4.8 + 0.4 = 5.2$
6	5	$[6, 5]^T$	$5.2 + 0.4 = 5.6$
7	6	$[7, 6]^T$	$5.6 + 0.4 = 6.0$
8	6	$[8, 6]^T$	$6.0 + 0.4 = 6.4$

Each exact value of y appearing in the last column is rounded and used in the next row to plot a pixel. Thus, the line segment is approximated by the pixels $[3, 4]^T$, $[4, 4]^T$, $[5, 5]^T$, $[6, 5]^T$, $[7, 6]^T$ and $[8, 6]^T$. □

Algorithm 2.1 assumes that $|m| \leq 1$. On the other hand, if $|m| > 1$, it can be written that

$$x_i = \frac{1}{m}(y_i - B)\,. \tag{2.5}$$

Consequently, x_{i+1} is calculated as by incrementing the value of y by 1; so, we can write

$$x_{i+1} = \frac{1}{m}(y_{i+1} - B) = \frac{1}{m}(y_i + 1 - B) = \underbrace{\frac{1}{m}(y_i - B)}_{x_i} + \frac{1}{m} = x_i + \frac{1}{m}. \tag{2.6}$$

Thus, the roles of x and y are reversed by assigning an increment of 1 to y and an increment of $\frac{1}{m}$ to x; consequently, the DDA algorithm can be modified as follows:

Algorithm 2.2 *DDA2 algorithm*

Input: x_0, y_0, x_1, y_1
1: $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$
2: $x = x_0$
3: **for** $(y = y_0$ to $y_1)$ **do**
4: Plot $[\lfloor x + 0.5 \rfloor, y]^T$
5: $x = x + \frac{1}{m}$
6: **end for**

end

2.1.2 Bresenham's Algorithm

In Example 2.1, notice that a difference could arise between the accurate value of y and its integer value to be used for plotting or displaying the pixel. This difference represents an error value. Also, note that the y-value is incremented by 1 only if the fraction included in the accurate value of y is ≥ 0.5; otherwise, y remains unchanged.

This suggests that each time x is incremented, the slope m can be added to an *error* value (indicating the vertical distance between the rounded y-value and the exact y-value) and if the new *error* is ≥ 0.5 (i.e., the line gets closer to the next y-value), y is incremented by 1 while the error is decremented by 1. This idea is listed in Algorithm 2.3.

Algorithm 2.3 *Modified DDA algorithm*

Input: x_0, y_0, x_1, y_1
1: $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$
2: $y = y_0$
3: $error = 0$
4: **for** $(x = x_0$ to $x_1)$ **do**
5: Plot $[x, y]^T$

```
    error = error + m
    if (error ≥ 0.5) then
        y = y + 1
        error = error − 1
    end if
end for
```

end

Example 2.2: [*DDA and Bresenham's algorithm*] Re-solve Example 2.1 using Algorithm 2.3

Solution 2.2: The values are shown in the next table where the initial values for x, y and *error* are x_0, y_0 and 0 respectively.

4: x	5: y	5: Plot	6: *error*	8: y	9: *error*
3	4	$[3, 4]^T$	$0 + 0.4 = 0.4$		
4	4	$[4, 4]^T$	$0.4 + 0.4 = 0.8$	$4 + 1 = 5$	$0.8 - 1 = -0.2$
5	5	$[5, 5]^T$	$-0.2 + 0.4 = 0.2$		
6	5	$[6, 5]^T$	$0.2 + 0.4 = 0.6$	$5 + 1 = 6$	$0.6 - 1 = -0.4$
7	6	$[7, 6]^T$	$-0.4 + 0.4 = 0$		
8	6	$[8, 6]^T$	$0 + 0.4 = 0.4$		

□

Algorithm 2.3 does not handle the general case of a line and cannot be used to draw an arbitrary line. In particular, there are three cases that need to be considered to generalize it. In a right-handed coordinate system, these cases are:

1. if the line goes up but the slope $m > 1$ or $|\Delta y| > |\Delta x|$ (i.e., a steep line) as shown in Fig. 2.2a;
2. if the line slopes upwards but heads in the opposite direction where $x_0 > x_1$ as shown in Fig. 2.2b; and
3. if the line goes down where $y_0 > y_1$ as shown in Fig. 2.2c.

The solutions to the three cases mentioned above are the following:

1. If the line is steep, reflect it about the line $y = x$ in order to obtain a line with smaller slope (i.e., with $m < 1$). This is done by swapping x_0 with y_0 and x_1 with y_1. The x and y parameters are swapped again for plotting (i.e., to plot $[y, x]^T$).
2. If the line slopes upwards but heads in the opposite direction, swap the endpoints $[x_0, y_0]^T$ and $[x_1, y_1]^T$.
3. If the line goes down, decrement y by 1 instead of incrementing it (i.e., step y by -1 instead of 1).

Algorithm 2.4 accommodates these changes.

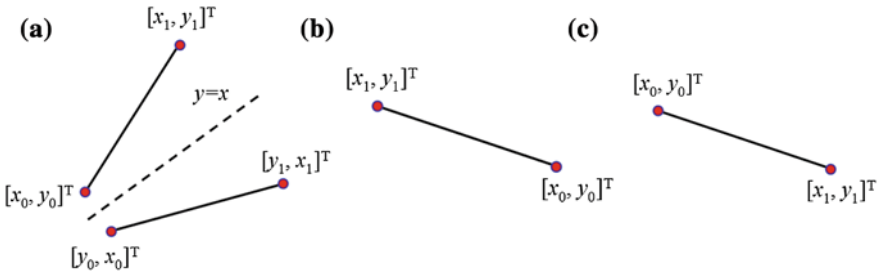

Fig. 2.2 Cases considered to generalize the line algorithm. **a** If the slope is >1. **b** If the line slopes upwards but heads in the opposite direction. **c** If the line goes down

Algorithm 2.4 *Modified Algorithm 2.3*

Input: x_0, y_0, x_1, y_1
$\Delta x = x_1 - x_0$
$\Delta y = y_1 - y_0$
$steep = \left|\frac{\Delta y}{\Delta x}\right| > 1$
if $(steep = TRUE)$ **then** ..... $\Longleftarrow$ Fig. 2.2a
    swap (x_0, y_0)
    swap (x_1, y_1)
end if

if $(x_0 > x_1)$ **then** ..... $\Longleftarrow$ Fig. 2.2b
    swap (x_0, x_1)
    swap (y_0, y_1)
end if

if $(y_0 > y_1)$ **then** ..... $\Longleftarrow$ Fig. 2.2c
    $\delta y = -1$
else
    $\delta y = 1$
end if

$m = \frac{|\Delta y|}{\Delta x} = \frac{|y_1 - y_0|}{x_1 - x_0}$
$y = y_0$
$error = 0$

for $(x = x_0$ to $x_1)$ **do**
    if $(steep = TRUE)$ **then**
        Plot $[y, x]^T$
    else
        Plot $[x, y]^T$
    end if

```
    error = error + m
    if (error ≥ 0.5) then
        y = y + δy
        error = error − 1
    end if
end for
```

end

The main source of problem with Algorithm 2.4 is that it works with floating-point numbers (e.g., *m* and *error*), which slows the process down and may result in error accumulation. Working with integer numbers will be much faster and more accurate. Switching to integers can be achieved easily by multiplying *m* and *error* by the denominator of the slope; i.e., Δx. Also, both sides of the condition *error* $\geq$0.5 are doubled to get rid of the fraction. Such an algorithm is referred to as *Bresenham's algorithm* (Bresenham 1965). It is listed in Algorithm 2.5.

Algorithm 2.5 *Bresenham's algorithm*

```
Input: x0, y0, x1, y1
steep = |y1 − y0| > |x1 − x0|
if (steep = TRUE) then
    swap (x0, y0)
    swap (x1, y1)
end if

if (x0 > x1) then
    swap (x0, x1)
    swap (y0, y1)
end if

if (y0 > y1) then
    δy = −1
else
    δy = 1
end if

Δx = x1 − x0
Δy = |y1 − y0|
y = y0
error = 0

for (x = x0 to x1) do
    if (steep = TRUE) then
        Plot [y, x]^T
    else
```

```
27:         Plot [x, y]^T
28:     end if
29:     error = error + Δy
30:     if (2 × error ≥ Δx) then
31:         y = y + δy
32:         error = error − Δx
33:     end if
34: end for
```

end

Note that Bresenham's algorithm can be tuned for integer computations of circumferential pixels in circles.

Example 2.3: [*Bresenham's line drawing algorithm*] You are asked to draw a line segment between the points $[1, 1]^T$ and $[4, 3]^T$. Use Bresenham's line drawing algorithm to specify the locations of pixels that should approximate the line.

Solution 2.3: We will follow the steps of Algorithm 2.5. The line is not steep as $|y_1 - y_0| < |x_1 - x_0|$. The y-step is 1 as $y_0 < y_1$. We have

$$\begin{aligned} \delta y &= 1, \\ \Delta x = x_1 - x_0 &= 4 - 1 = 3, \\ \Delta y = y_1 - y_0 &= 3 - 1 = 2, \\ y &= 1, \\ error &= 0. \end{aligned}$$

The following table shows the loop along the x-direction from 1 to 4 (i.e., the loop that starts at Line 23 in Algorithm 2.5):

23: x	27: Plot	29: *error*	31: y	32: *error*
1	$[1, 1]^T$	$0 + 2 = 2$	$1 + 1 = 2$	$2 - 3 = -1$
2	$[2, 2]^T$	$-1 + 2 = 1$		
3	$[3, 2]^T$	$1 + 2 = 3$	$2 + 1 = 3$	$3 - 3 = 0$
4	$[4, 3]^T$	$0 + 2 = 2$	$3 + 1 = 4$	$2 - 3 = -1$

Thus, the line is approximated by the pixels $[1, 1]^T$, $[2, 2]^T$, $[3, 2]^T$ and $[4, 3]^T$. □

2.1.3 The Midpoint Algorithm

The *midpoint algorithm* (Pitteway 1967; Aken 1984; Van Aken and Novak 1985) can be used instead of Bresenham's algorithm to approximate straight lines. Actually, it produces the same line pixels; however, a difference does exist. In Bresenham's

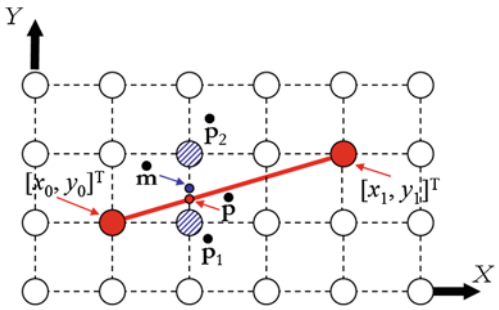

Fig. 2.3 $\dot{\mathbf{m}} = [x_0 + 1, y_0 + \frac{1}{2}]^T$ is the midpoint between $[x_0 + 1, y_0]^T$ and $[x_0 + 1, y_0 + 1]^T$

algorithm, the smallest y-difference between the actual accurate y-value and the rounded integer y-value is used to pick up the nearest pixel to approximate the line. On the other hand, in the midpoint technique, we determine which side of the linear equation the midpoint between pixels lies.

Consider Fig. 2.3 where a line is to be drawn from $[x_0, y_0]^T$ to $[x_1, y_1]^T$. When x is incremented at the second iteration as done before, the exact intersection happens at point $\dot{\mathbf{p}}$. Thus, there will be two choices for the pixels to be picked up at $x = x_0 + 1$; these choices are $\dot{\mathbf{p}}_1$ and $\dot{\mathbf{p}}_2$ (hatched in Fig. 2.3). The main idea of the previous formulation is to calculate the distances from $\dot{\mathbf{p}}$ to $\dot{\mathbf{p}}_1$ and from $\dot{\mathbf{p}}$ to $\dot{\mathbf{p}}_2$. According to the smaller distance, a pixel will be selected. Hence, $\dot{\mathbf{p}}_1$ is selected in our case.

In the midpoint algorithm, the equation of the line spanning from $[x_0, y_0]^T$ to $[x_1, y_1]^T$ is expressed [as in Eq. (2.3)] as

$$y = mx + B = \frac{\Delta y}{\Delta x}x + B, \tag{2.7}$$

where m is the slope; $\Delta x = x_1 - x_0$; $\Delta y = y_1 - y_0$; and B is the y-intercept. Equation (2.7) can be re-written in implicit form (i.e., $ax + by + c = 0$) as

$$\underbrace{\Delta y}_{a}\ x \underbrace{-\Delta x}_{b} y + \underbrace{\Delta x B}_{c} = 0, \tag{2.8}$$

where $a = \Delta y$; $b = -\Delta x$; and $c = \Delta x B$. Now, the midpoint $\dot{\mathbf{m}} = [x_0 + 1, y_0 + \frac{1}{2}]^T$ between $[x_0 + 1, y_0]^T$ and $[x_0 + 1, y_0 + 1]^T$ is applied to Eq. (2.8) to get

$$\begin{aligned} \mathscr{F}\left(x_0 + 1, y_0 + \frac{1}{2}\right) = d_{\dot{\mathbf{m}}} &= \Delta y(x_0 + 1) - \Delta x\left(y_0 + \frac{1}{2}\right) + \Delta x B \\ &= a(x_0 + 1) + b\left(y_0 + \frac{1}{2}\right) + c. \end{aligned} \tag{2.9}$$

There are three possibilities for $d_{\dot{\mathbf{m}}}$:

$$d_{\dot{\mathbf{m}}} = \begin{cases} +ve, \text{ the line is passing above } \dot{\mathbf{m}}\text{; thus, } \dot{\mathbf{p}}_2 \text{ is selected;} \\ -ve, \text{ the line is passing below } \dot{\mathbf{m}}\text{; thus, } \dot{\mathbf{p}}_1 \text{ is selected;} \\ 0, \text{ the midpoint } \dot{\mathbf{m}} \text{ is exactly on the line; thus, select either } \dot{\mathbf{p}}_1 \text{ or } \dot{\mathbf{p}}_2. \end{cases}$$

Choosing the pixel in the next column depends on whether $\dot{\mathbf{p}}_1$ or $\dot{\mathbf{p}}_2$ has been chosen. Assume that $\dot{\mathbf{p}}_1$ is selected. The midpoint $[x_0 + 2, y_0 + \frac{1}{2}]^T$ between $[x_0 + 2, y_0]^T$ and $[x_0 + 2, y_0 + 1]^T$ is applied to Eq. (2.8) to get

$$\begin{aligned} \mathscr{F}\left(x_0 + 2, y_0 + \frac{1}{2}\right) &= a(x_0 + 2) + b\left(y_0 + \frac{1}{2}\right) + c \\ &= \underbrace{a(x_0 + 1) + b\left(y_0 + \frac{1}{2}\right) + c}_{d_{\dot{\mathbf{m}}}} + a \\ &= d_{\dot{\mathbf{m}}} + a \\ &= d_{\dot{\mathbf{m}}} + \Delta y. \end{aligned} \tag{2.10}$$

On the other hand, if $\dot{\mathbf{p}}_2$ is selected, the midpoint $[x_0 + 2, y_0 + \frac{3}{2}]^T$ between $[x_0 + 2, y_0 + 1]^T$ and $[x_0 + 2, y_0 + 2]^T$ is applied to Eq. (2.8) to get

$$\begin{aligned} \mathscr{F}\left(x_0 + 2, y_0 + \frac{3}{2}\right) &= a(x_0 + 2) + b\left(y_0 + \frac{3}{2}\right) + c \\ &= \underbrace{a(x_0 + 1) + b\left(y_0 + \frac{1}{2}\right) + c}_{d_{\dot{\mathbf{m}}}} + a + b \\ &= d_{\dot{\mathbf{m}}} + a + b \\ &= d_{\dot{\mathbf{m}}} + \Delta y - \Delta x. \end{aligned} \tag{2.11}$$

This means that the subsequent sign of $d_{\dot{\mathbf{m}}}$ can be obtained by incrementing it by either Δy if $\dot{\mathbf{p}}_1$ is selected or $\Delta y - \Delta x$ if $\dot{\mathbf{p}}_2$ is selected. An initial value for $d_{\dot{\mathbf{m}}}$ can be obtained using the start point $[x_0, y_0]^T$ and Eq. (2.9). So,

$$\begin{aligned} \mathscr{F}\left(x_0 + 1, y_0 + \frac{1}{2}\right) &= a(x_0 + 1) + b\left(y_0 + \frac{1}{2}\right) + c \\ &= \underbrace{ax_0 + by_0 + c}_{\mathscr{F}(x_0, y_0)} + a + \frac{b}{2}, \end{aligned} \tag{2.12}$$

where $a + \frac{b}{2} = \Delta y - \frac{\Delta x}{2}$ is the initial estimate of $d_{\dot{\mathbf{m}}}$. The midpoint algorithm is listed in Algorithm 2.6.

Algorithm 2.6 *Midpoint algorithm—floating-point version*

Input: x_0, y_0, x_1, y_1
1: $\Delta x = x_1 - x_0$
2: $\Delta y = y_1 - y_0$
3: $d_{\dot{\mathbf{m}}} = \Delta y - \frac{\Delta x}{2}$
4: $d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_1) = \Delta y$
5: $d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_2) = \Delta y - \Delta x$
6: $y = y_0$
7:
8: **for** $(x = x_0$ to $x_1)$ **do**
9: Plot $[x, y]^T$
10: **if** $(d_{\dot{\mathbf{m}}} \leq 0)$ **then**
11: $d_{\dot{\mathbf{m}}} = d_{\dot{\mathbf{m}}} + d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_1)$
12: **else**
13: $d_{\dot{\mathbf{m}}} = d_{\dot{\mathbf{m}}} + d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_2)$
14: $y = y + 1$
15: **end if**
16: **end for**

end

Algorithm 2.6 uses floating-point numbers which slows the process down. This can be easily overcome as listed in Algorithm 2.7.

Algorithm 2.7 *Midpoint algorithm – integer version*

Input: x_0, y_0, x_1, y_1
1: $\Delta x = x_1 - x_0$
2: $\Delta y = y_1 - y_0$
3: $d_{\dot{\mathbf{m}}} = 2\Delta y - \Delta x$
4: $d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_1) = 2\Delta y$
5: $d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_2) = 2\Delta y - 2\Delta x$
6: $y = y_0$
7:
8: **for** $(x = x_0$ to $x_1)$ **do**
9: Plot $[x, y]^T$
10: **if** $(d_{\dot{\mathbf{m}}} \leq 0)$ **then**
11: $d_{\dot{\mathbf{m}}} = d_{\dot{\mathbf{m}}} + d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_1)$
12: **else**
13: $d_{\dot{\mathbf{m}}} = d_{\dot{\mathbf{m}}} + d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_2)$
14: $y = y + 1$
15: **end if**
16: **end for**

end

Example 2.4: [*Midpoint line drawing algorithm*]
You are asked to draw a line segment between the points $[1, 1]^T$ and $[4, 3]^T$. Use the midpoint line drawing algorithm to specify the locations of pixels that should approximate the line.

Solution 2.4: We will follow the steps of Algorithm 2.7. So,

$$\begin{aligned} \Delta x &= x_1 - x_0 = 4 - 1 = 3, \\ \Delta y &= y_1 - y_0 = 3 - 1 = 2, \\ d_{\dot{\mathbf{m}}} &= 2\Delta y - \Delta x = 2 \times 2 - 3 = 1, \\ d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_1) &= 2\Delta y = 2 \times 2 = 4, \\ d_{\dot{\mathbf{m}}}(\dot{\mathbf{p}}_2) &= 2\Delta y - 2\Delta x = 2 \times 2 - 2 \times 3 = -2, \\ y &= 1. \end{aligned}$$

The following table shows the loop along the x-direction from 1 to 4 (i.e., the loop that starts at Line 8 in Algorithm 2.7):

8: x	9: Plot	11: $d_{\dot{\mathbf{m}}}$	13: $d_{\dot{\mathbf{m}}}$	14: y
1	$[1, 1]^T$		$1 - 2 = -1$	$1 + 1 = 2$
2	$[2, 2]^T$	$-1 + 4 = 3$		
3	$[3, 2]^T$		$3 - 2 = 1$	$2 + 1 = 3$
4	$[4, 3]^T$		$1 - 2 = -1$	$3 + 1 = 4$

Thus, the line is approximated by the pixels $[1, 1]^T$, $[2, 2]^T$, $[3, 2]^T$ and $[4, 3]^T$. This is the same result obtained by Bresenham's algorithm in Example 2.3. □

2.2 Circles

As shown in Fig. 2.4, a pixel $[x, y]^T$ on the circumference of a circle can be estimated if the center $[x_0, y_0]^T$, the radius r and the angle θ are known. The location of the pixel $[x, y]^T$ is obtained using trigonometric functions as

$$\begin{aligned} x &= x_0 + r\cos(\theta), \\ y &= y_0 + r\sin(\theta), \end{aligned} \tag{2.13}$$

which represent the parametric form of a circle where the parameter is the angle θ.

In order to draw the whole circle, Eq. (2.13) can be used iteratively with different θ-values going from 0° to 360° (i.e., $\theta \in [0°, 360°)$ or $0 \leq \theta < 2\pi$).[1] The problem with this approach is that working with trigonometric functions is time-consuming. The calculations could be very slow especially if the angle increment is small. On the other hand, if the angle increment is large, the algorithm will be fast; however,

[1] $[0°, 360°)$ represents a half-open interval where 0° is included while 360° is excluded.

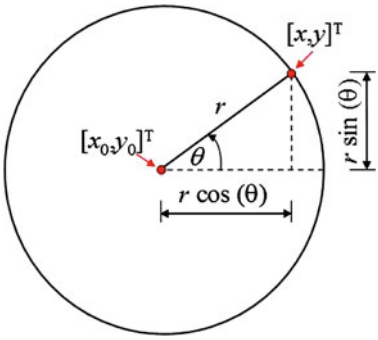

Fig. 2.4 A pixel $[x, y]^T$ on the circumference of a circle can be estimated if the center $[x_0, y_0]^T$, the radius r and the angle θ are known

some circumferential pixels may be missed. Also, if the radius value is too large, some pixels may be skipped as well. There must be a more efficient way to draw a circle.

2.2.1 Two-Way Symmetry Algorithm

A circle equation may be expressed as

$$(x - x_0)^2 + (y - y_0)^2 = r^2, \tag{2.14}$$

where $[x_0, y_0]^T$ is the center point of the circle, r is its radius in pixels and $[x, y]^T$ is a point on the circumference. Equation (2.14) can be re-written in an explicit form to solve for y in terms of x as

$$y = y_0 \pm \sqrt{r^2 - (x - x_0)^2}. \tag{2.15}$$

This explicit circle equation can be used to draw the circle by iterating along x-direction. Assuming that the center of the circle $[x_0, y_0]^T$ is at the origin $[0, 0]^T$, the x-values lie in the interval $[-r, r]$. Each iteration results in *two* values for y which makes it faster than the previous method. Hence, comes the term *two-way symmetry* approach. However, discontinuities may appear with this approach when the slope is <1 (i.e., $\left|\frac{y-y_0}{x-x_0}\right| < 1$). An example showing a quarter of a circle is shown in Fig. 2.5 where discontinuities appear. Note that there is only one pixel marked for each x in this upper quarter of the circle.

Algorithm 2.8 calculates the points comprising a circle centered at the origin and then moves the points estimated by the amount $[x_0, y_0]^T$ (i.e., the center of the circle) to get the correct positions.

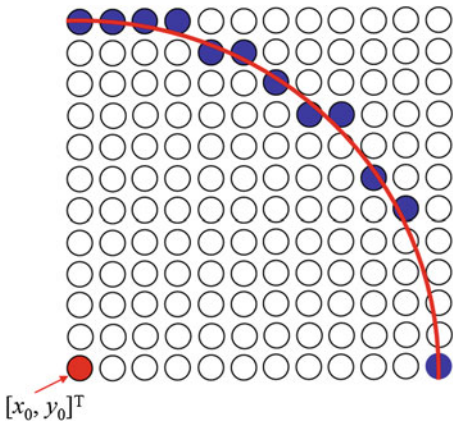

Fig. 2.5 Two-way symmetry approach results in discontinuities where $\left|\frac{y-y_0}{x-x_0}\right| < 1$. Pixels selected to represent this quarter are shaded

Algorithm 2.8 *Two-way symmetry algorithm*

Input: x_0, y_0, r
1: Plot $[x_0 + r, y_0]^T$
2: Plot $[x_0 - r, y_0]^T$
3: **for** $(x = -r + 1$ to $r - 1)$ **do**
4: $\quad y = \lfloor\sqrt{r^2 - x^2} + 0.5\rfloor$
5: $\quad$ Plot $[x_0 + x, y_0 + y]^T$
6: $\quad$ Plot $[x_0 + x, y_0 - y]^T$
7: **end for**

end

Note that the start and endpoints of the horizontal diameter are treated as special cases before entering the loop. This is to avoid reflecting these points about themselves. Also, be careful that Algorithm 2.8 does not check if the points estimated lie inside the boundaries of the screen or viewport. In order to accommodate this constraint for a point $[x, y]^T$, check if $x \geq 0$, $y \geq 0$, $x < x_{max}$ and $y < y_{max}$ where x_{max} and y_{max} are the width and height of the screen or viewport in pixels.

Example 2.5: [*2-way symmetry algorithm*]
A circle having a radius of 5 pixels and centered at $[3, 4]^T$ is to be drawn on a computer screen. Use the 2-way symmetry algorithm to determine what pixels should constitute the circle.

Solution 2.5: The start and endpoints of the horizontal diameter are $[8, 4]^T$ and $[-2, 4]^T$. The rest of the points are listed in the following table:

3: x	4: y	5: Plot	6: Plot
-4	$\lfloor\sqrt{5^2-(-4)^2}+0.5\rfloor=3$	$[-1,7]^T$	$[-1,1]^T$
-3	$\lfloor\sqrt{5^2-(-3)^2}+0.5\rfloor=4$	$[0,8]^T$	$[0,0]^T$
-2	$\lfloor\sqrt{5^2-(-2)^2}+0.5\rfloor=5$	$[1,9]^T$	$[1,-1]^T$
-1	$\lfloor\sqrt{5^2-(-1)^2}+0.5\rfloor=5$	$[2,9]^T$	$[2,-1]^T$
0	$\lfloor\sqrt{5^2-0^2}+0.5\rfloor=5$	$[3,9]^T$	$[3,-1]^T$
1	$\lfloor\sqrt{5^2-1^2}+0.5\rfloor=5$	$[4,9]^T$	$[4,-1]^T$
2	$\lfloor\sqrt{5^2-2^2}+0.5\rfloor=5$	$[5,9]^T$	$[5,-1]^T$
3	$\lfloor\sqrt{5^2-3^2}+0.5\rfloor=4$	$[6,8]^T$	$[6,0]^T$
4	$\lfloor\sqrt{5^2-4^2}+0.5\rfloor=3$	$[7,7]^T$	$[7,1]^T$

Of course, all points with negative coordinates are disregarded. Hence, the points considered are $[8,4]^T$, $[0,8]^T$, $[0,0]^T$, $[1,9]^T$, $[2,9]^T$, $[3,9]^T$, $[4,9]^T$, $[5,9]^T$, $[6,8]^T$, $[6,0]^T$, $[7,7]^T$ and $[7,1]^T$. □

2.2.2 Four-Way Symmetry Algorithm

The symmetry of the circle can be utilized such that the circle may be split into four quadrants. In this four-way symmetry approach, the pixels of only one quadrant are estimated as done with the two-way symmetry approach while the pixels in the remaining three quadrants are mirrored/reflected. In this case, the number of iterations is reduced along the x-direction if compared with the two-way symmetry approach to almost half the number of iterations. This makes this approach faster than the previous one; however, it keeps the same problem of discontinuities as shown in Fig. 2.6 where the upper two quadrants of a circle are drawn using the four-way symmetry approach.

Assuming that the center of the circle $[x_0, y_0]^T$ is at the origin $[0,0]^T$, Eq. (2.15) can be used to get a point $[x, y]^T$ on the circumference. This is followed by mirroring this point to obtain three other points $[x, -y]^T$, $[-x, y]^T$ and $[-x, -y]^T$ in the other three quadrants. In this case, the x-values lie within the interval $[0, r]$. In the general case where the circle is centered at an arbitrary point $[x_0, y_0]^T$, we calculate the points of the circle as if it is centered at the origin and then move the points estimated by the vector $[x_0, y_0]^T$ that represents the center point to get the correct positions. Algorithm 2.9 lists the four-way symmetry algorithm.

Algorithm 2.9 *Four-way symmetry algorithm*

Input: x_0, y_0, r
1: Plot $[x_0, y_0 + r]^T$
2: Plot $[x_0, y_0 - r]^T$
3: Plot $[x_0 + r, y_0]^T$
4: Plot $[x_0 - r, y_0]^T$
5: **for** ($x = 1$ to $r - 1$) **do**

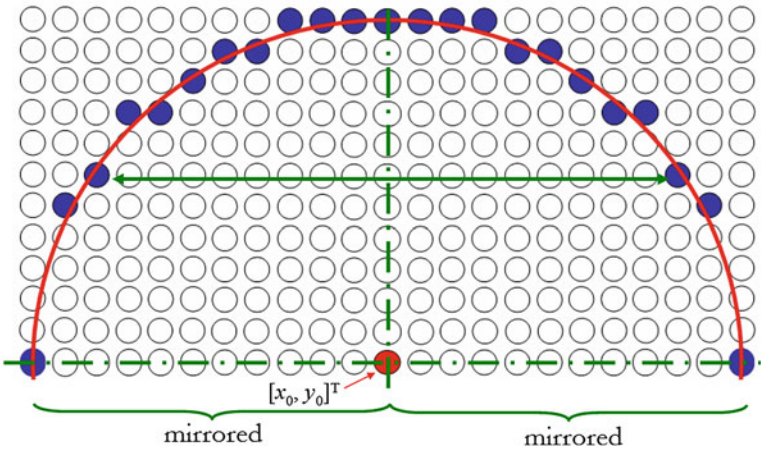

Fig. 2.6 Four-way symmetry approach. Only two quadrants are shown

6: $y = \lfloor \sqrt{r^2 - x^2} + 0.5 \rfloor$
7: Plot $[x_0 + x, y_0 + y]^T$
8: Plot $[x_0 + x, y_0 - y]^T$
9: Plot $[x_0 - x, y_0 + y]^T$
10: Plot $[x_0 - x, y_0 - y]^T$
11: **end for**

end

Note that the start and endpoints of each quadrant are treated as special cases before entering the loop as done before with the two-way symmetry approach. This is to avoid reflecting these points about themselves. Also, you may tune Algorithm 2.9 to check if the points estimated lie inside the boundaries of the screen or viewport as mentioned before with the two-way symmetry approach.

Example 2.6: [*4-way symmetry algorithm*]
A circle having a radius of 5 pixels and centered at $[3, 4]^T$ is to be drawn on a computer screen. Use the 4-way symmetry algorithm to determine what pixels should constitute the circle.

Solution 2.6: The start and endpoints of each quadrant are $[3, 9]^T$, $[3, -1]^T$, $[8, 4]^T$ and $[-2, 4]^T$. The rest of the points are listed in the following table:

5: x	6: y	7: Plot	8: Plot	9: Plot	10: Plot
1	$\lfloor \sqrt{5^2 - 1^2} + 0.5 \rfloor = 5$	$[4, 9]^T$	$[4, -1]^T$	$[2, 9]^T$	$[2, -1]^T$
2	$\lfloor \sqrt{5^2 - 2^2} + 0.5 \rfloor = 5$	$[5, 9]^T$	$[5, -1]^T$	$[1, 9]^T$	$[1, -1]^T$
3	$\lfloor \sqrt{5^2 - 3^2} + 0.5 \rfloor = 4$	$[6, 8]^T$	$[6, 0]^T$	$[0, 8]^T$	$[0, 0]^T$
4	$\lfloor \sqrt{5^2 - 4^2} + 0.5 \rfloor = 3$	$[7, 7]^T$	$[7, 1]^T$	$[-1, 7]^T$	$[-1, 1]^T$

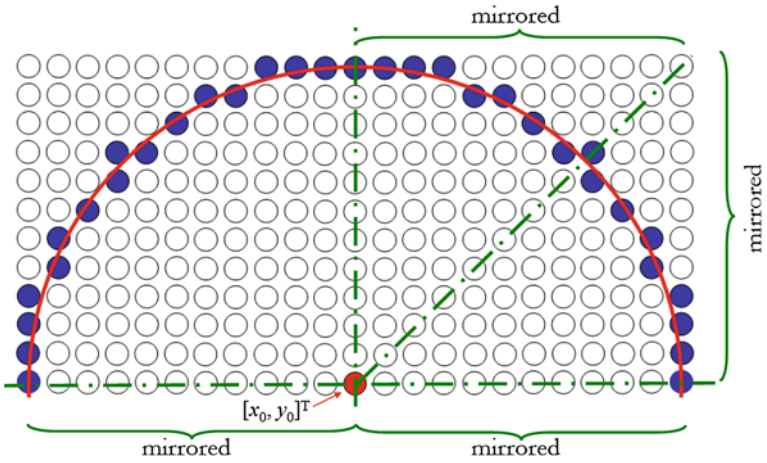

Fig. 2.7 Eight-way symmetry approach. Only four octants are shown

We discard all points with negative coordinates. Hence, the points considered are $[3, 9]^T$, $[8, 4]^T$, $[4, 9]^T$, $[2, 9]^T$, $[5, 9]^T$, $[1, 9]^T$, $[6, 8]^T$, $[6, 0]^T$, $[0, 8]^T$, $[0, 0]^T$, $[7, 7]^T$ and $[7, 1]$. Note that those are the same points obtained in Example 2.5 using the 2-way symmetry algorithm but with less number of iterations. □

2.2.3 Eight-Way Symmetry Algorithm

The four-way symmetry approach can be further enhanced by splitting the circle into eight octants instead of only four quadrants. This is referred to as the *eight-way symmetry* approach in which the pixels of only one octant are estimated as done before while the pixels in the remaining seven octants are mirrored/reflected. In this case, the number of loop iterations is reduced more than before. This makes this approach much faster than the previous ones. In addition, it avoids the problem of discontinuities as shown in Fig. 2.7 where the upper four octants of a circle are drawn using the eight-way symmetry approach.

Assuming that the center of the circle $[x_0, y_0]^T$ is at the origin $[0, 0]^T$, Eq. (2.15) can be used to get a point $[x, y]^T$ on the circumference. This is followed by mirroring this point to obtain seven other points $[x, -y]^T$, $[-x, y]^T$, $[-x, -y]^T$, $[y, x]^T$, $[y, -x]^T$, $[-y, x]^T$ and $[-y, -x]^T$ in the other seven octants. Note that the x-values within the loop lie in the interval $\left[1, \left\lfloor \frac{r}{\sqrt{2}} \right\rfloor\right]$.

If the circle is centered at an arbitrary point $[x_0, y_0]^T$, we calculate the points comprising the circle as if it is centered at the origin and then move the points estimated by the amount $[x_0, y_0]^T$ to get the correct positions. Algorithm 2.10 lists

the eight-way symmetry algorithm, which results in better and faster-to-generate circles.

Algorithm 2.10 *Eight-way symmetry algorithm*

Input: x_0, y_0, r
1: Plot $[x_0, y_0 + r]^T$
2: Plot $[x_0, y_0 - r]^T$
3: Plot $[x_0 + r, y_0]^T$
4: Plot $[x_0 - r, y_0]^T$
5: $x = 1$
6: $y = \lfloor \sqrt{r^2 - x^2} + 0.5 \rfloor$
7:
8: **while** $(x < y)$ **do**
9: Plot $[x_0 + x, y_0 + y]^T$
10: Plot $[x_0 + x, y_0 - y]^T$
11: Plot $[x_0 - x, y_0 + y]^T$
12: Plot $[x_0 - x, y_0 - y]^T$
13: Plot $[x_0 + y, y_0 + x]^T$
14: Plot $[x_0 + y, y_0 - x]^T$
15: Plot $[x_0 - y, y_0 + x]^T$
16: Plot $[x_0 - y, y_0 - x]^T$
17: $x = x + 1$
18: $y = \lfloor \sqrt{r^2 - x^2} + 0.5 \rfloor$
19: **end while**
20:
21: **if** $x = y$ **then**
22: Plot $[x_0 + x, y_0 + y]^T$
23: Plot $[x_0 + x, y_0 - y]^T$
24: Plot $[x_0 - x, y_0 + y]^T$
25: Plot $[x_0 - x, y_0 - y]^T$
26: **end if**

end

Notice that the start and endpoints of each octant are treated as special cases as done with the two- and four-way symmetry approaches to avoid reflecting these points about themselves. (Check Lines 1–4 and 22–25.) You may also check if the points estimated lie inside the boundaries of the screen or viewport by testing if $x \geq 0$, $y \geq 0$, $x < x_{max}$ and $y < y_{max}$ where x_{max} and y_{max} are the width and height of the screen in pixels.

Example 2.7: [*8-way symmetry algorithm*]
A circle having a radius of 5 pixels and centered at $[3, 4]^T$ is to be drawn on a computer screen. Use the 8-way symmetry algorithm to determine what pixels should constitute the circle.

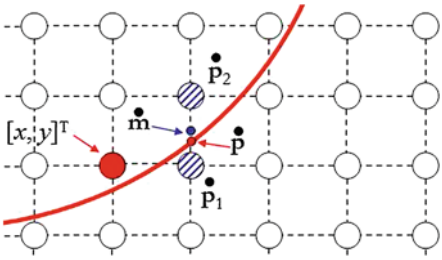

Fig. 2.8 $\dot{\mathbf{m}} = [x+1, y+\frac{1}{2}]^T$ is the midpoint between $[x+1, y]^T$ and $[x+1, y+1]^T$

Solution 2.7: The start and endpoints of each quadrant are $[3, 9]^T$, $[3, -1]^T$, $[8, 4]^T$ and $[-2, 4]^T$. The rest of the points are listed in the following table:

5/17: x	6/18: y	9: Plot	10: Plot	11: Plot	12: Plot	13: Plot	14: Plot	15: Plot	16: Plot
1	5	$[4, 9]^T$	$[4, -1]^T$	$[2, 9]^T$	$[2, -1]^T$	$[8, 5]^T$	$[8, 3]^T$	$[-2, 5]^T$	$[-2, 3]^T$
2	5	$[5, 9]^T$	$[5, -1]^T$	$[1, 9]^T$	$[1, -1]^T$	$[8, 6]^T$	$[8, 2]^T$	$[-2, 6]^T$	$[-2, 2]^T$
3	4	$[6, 8]^T$	$[6, 0]^T$	$[0, 8]^T$	$[0, 0]^T$	$[7, 7]^T$	$[7, 1]^T$	$[-1, 7]^T$	$[-1, 1]^T$
4	3				no iteration as $x > y$				

After neglecting all points with negative coordinates, the points considered are $[3, 9]^T$, $[8, 4]^T$, $[4, 9]^T$, $[2, 9]^T$, $[8, 5]^T$, $[8, 3]^T$, $[5, 9]^T$, $[1, 9]^T$, $[8, 6]^T$, $[8, 2]^T$, $[6, 8]^T$, $[6, 0]^T$, $[0, 8]^T$, $[0, 0]^T$, $[7, 7]^T$ and $[7, 1]^T$. Notice that the discontinuity problem has been overcome in this case. (Check out points $[8, 2]^T$, $[8, 3]^T$, $[8, 5]^T$ and $[8, 6]^T$.) In addition, calculations have been done with a faster algorithm.

Also, notice that the number of iterations along the x-direction may be calculated as

$$\left\lfloor \frac{r}{\sqrt{2}} \right\rfloor = \left\lfloor \frac{5}{\sqrt{2}} \right\rfloor = 3. \qquad \square$$

2.2.4 The Midpoint Algorithm

The *midpoint algorithm* used to draw lines in Sect. 2.1.3 can be modified to draw circles (Foley et al. 1995). Similar to the 8-way symmetry algorithm (Sect. 2.2.3), only one octant is considered. The rest of the circle is obtained by symmetry as done before. Similar to the midpoint technique used to draw lines, we determine on which side of the circle equation the midpoint between pixels lies.

Consider Fig. 2.8 where a part of a circle is shown and where a pixel $[x, y]^T$ is determined to belong to the circle. When x is incremented at the next iteration as done before, the exact intersection happens at point $\dot{\mathbf{p}}$. Thus, there will be two choices for the pixels to be picked up at $x + 1$; these choices are $\dot{\mathbf{p}}_1$ and $\dot{\mathbf{p}}_2$.

The implicit form of a circle equation is given by

$$\mathscr{F}(x, y) = x^2 + y^2 - r^2 = 0, \tag{2.16}$$

where r is the radius of the circle and $[x, y]^T$ is a point on the circle. Now, the midpoint $\dot{\mathbf{m}} = [x + 1, y + \frac{1}{2}]^T$ between $[x + 1, y]^T$ and $[x + 1, y + 1]^T$ is applied to Eq. (2.16) to get

$$\mathscr{F}\left(x + 1, y + \frac{1}{2}\right) = d_{\dot{\mathbf{m}}} = (x + 1)^2 + \left(y + \frac{1}{2}\right)^2 - r^2. \tag{2.17}$$

There are three possibilities for $d_{\dot{\mathbf{m}}}$:

$$d_{\dot{\mathbf{m}}} = \begin{cases} +ve, \ \dot{\mathbf{m}} \text{ is outside the circle; thus, } \dot{\mathbf{p}}_2 \text{ is selected;} \\ -ve, \ \dot{\mathbf{m}} \text{ is inside the circle; thus, } \dot{\mathbf{p}}_1 \text{ is selected;} \\ 0, \ \text{the midpoint } \dot{\mathbf{m}} \text{ is exactly on the circle; thus, select either } \dot{\mathbf{p}}_1 \text{ or } \dot{\mathbf{p}}_2. \end{cases}$$

Choosing the pixel in the next column depends on whether $\dot{\mathbf{p}}_1$ or $\dot{\mathbf{p}}_2$ has been selected. Assume that $\dot{\mathbf{p}}_1$ is selected. The midpoint $[x + 2, y + \frac{1}{2}]^T$ between $[x + 2, y]^T$ and $[x + 2, y + 1]^T$ is applied to Eq. (2.16) to get

$$\begin{aligned} \mathscr{F}\left(x + 2, y + \frac{1}{2}\right) &= (x + 2)^2 + \left(y + \frac{1}{2}\right)^2 - r^2 \\ &= \underbrace{(x + 1)^2 + \left(y + \frac{1}{2}\right)^2 - r^2}_{d_{\dot{\mathbf{m}}}} + 2x + 3 \\ &= d_{\dot{\mathbf{m}}} \ + \ 2x + 3. \end{aligned} \tag{2.18}$$

On the other hand, if $\dot{\mathbf{p}}_2$ is selected, the midpoint $[x+2, y+\frac{3}{2}]^T$ between $[x+2, y+1]^T$ and $[x + 2, y + 2]^T$ is applied to Eq. (2.16) to get

$$\begin{aligned} \mathscr{F}\left(x + 2, y + \frac{3}{2}\right) &= (x + 2)^2 + \left(y + \frac{3}{2}\right)^2 - r^2 \\ &= \underbrace{(x + 1)^2 + \left(y + \frac{1}{2}\right)^2 - r^2}_{d_{\dot{\mathbf{m}}}} + 2x + 2y + 5 \\ &= d_{\dot{\mathbf{m}}} \ + 2x \ + 2y \ + 5. \end{aligned} \tag{2.19}$$

This means that the subsequent sign of $d_{\dot{\mathbf{m}}}$ can be obtained by incrementing it by either $2x + 3$ if $\dot{\mathbf{p}}_1$ is selected or $2x + 2y + 5$ if $\dot{\mathbf{p}}_2$ is selected where $[x, y]^T$ is the previous point on the circle (i.e., the increments are functions rather than constants as in line midpoint algorithm). Assuming that the center point is at the origin, an initial value for $d_{\dot{\mathbf{m}}}$ can be obtained using the lowest point $[0, -r]^T$ and Eq. (2.17)

where the next midpoint lies at $[1, -r + \frac{1}{2}]^T$. So,

$$\begin{aligned} \mathscr{F}\left(1, -r + \frac{1}{2}\right) &= 1^2 + \left(-r + \frac{1}{2}\right)^2 - r^2 \\ &= \underbrace{\frac{5}{4} - r}_{d_{\text{m}}}, \end{aligned} \tag{2.20}$$

where $\frac{5}{4} - r$ is the initial estimate of d_{m}. The midpoint algorithm for circles is listed in Algorithm 2.11. This algorithm assumes that the center of the circle is at $[x_0, y_0]^T$.

Algorithm 2.11 *Midpoint algorithm for circles – floating-point version*

Input: x_0, y_0, r
1: $d_{\text{m}} = \frac{5}{4} - r$
2: $x = 0$
3: $y = -r$
4:
5: Plot $[x_0, y_0 + r]^T$
6: Plot $[x_0, y_0 - r]^T$
7: Plot $[x_0 + r, y_0]^T$
8: Plot $[x_0 - r, y_0]^T$
9:
10: **while** $(x < -(y + 1))$ **do**
11: **if** $(d_{\text{m}} < 0)$ **then**
12: $d_{\text{m}} = d_{\text{m}} + 2x + 3$
13: **else**
14: $d_{\text{m}} = d_{\text{m}} + 2x + 2y + 5$
15: $y = y + 1$
16: **end if**
17: $x = x + 1$
18: Plot $[x_0 + x, y_0 + y]^T$
19: Plot $[x_0 + x, y_0 - y]^T$
20: Plot $[x_0 - x, y_0 + y]^T$
21: Plot $[x_0 - x, y_0 - y]^T$
22: Plot $[x_0 + y, y_0 + x]^T$
23: Plot $[x_0 + y, y_0 - x]^T$
24: Plot $[x_0 - y, y_0 + x]^T$
25: Plot $[x_0 - y, y_0 - x]^T$
26: **end while**

end

Notice that checking for the number of iterations $\left\lfloor \frac{r}{\sqrt{2}} \right\rfloor$ may be used to replace the condition of the "while" loop on Line 10. (See Problem 2.8.)

Algorithm 2.11 uses floating-point numbers which slows the process down. The initial value of $d_{\dot{m}}$ contains a fraction; thus, a new variable $h_{\dot{m}} = d_{\dot{m}} - \frac{1}{4}$ is used (Foley et al. 1995) to replace the value of $d_{\dot{m}}$ by $h_{\dot{m}} + \frac{1}{4}$. Hence, Line 1 in Algorithm 2.11 can be $h_{\dot{m}} = 1 - r$. In addition, the condition $d_{\dot{m}} < 0$ of Line 11 is changed to $h_{\dot{m}} < -\frac{1}{4}$. However, notice that $h_{\dot{m}}$ is initialized to an integer (i.e., $1 - r$) and is incremented by integers (either by $2x + 3$ or by $2x + 2y + 5$). So, the condition $h_{\dot{m}} < -\frac{1}{4}$ can be modified to $h_{\dot{m}} < 0$. Algorithm 2.12 applies all these changes.

Algorithm 2.12 *Midpoint algorithm for circles – integer version*

Input: x_0, y_0, r
1: $h_{\dot{m}} = 1 - r$
2: $x = 0$
3: $y = -r$
4:
5: Plot $[x_0, y_0 + r]^T$
6: Plot $[x_0, y_0 - r]^T$
7: Plot $[x_0 + r, y_0]^T$
8: Plot $[x_0 - r, y_0]^T$
9:
10: **while** $(x < -(y + 1))$ **do**
11: **if** $(h_{\dot{m}} < 0)$ **then**
12: $h_{\dot{m}} = h_{\dot{m}} + 2x + 3$
13: **else**
14: $h_{\dot{m}} = h_{\dot{m}} + 2x + 2y + 5$
15: $y = y + 1$
16: **end if**
17: $x = x + 1$
18: Plot $[x_0 + x, y_0 + y]^T$
19: Plot $[x_0 + x, y_0 - y]^T$
20: Plot $[x_0 - x, y_0 + y]^T$
21: Plot $[x_0 - x, y_0 - y]^T$
22: Plot $[x_0 + y, y_0 + x]^T$
23: Plot $[x_0 + y, y_0 - x]^T$
24: Plot $[x_0 - y, y_0 + x]^T$
25: Plot $[x_0 - y, y_0 - x]^T$
26: **end while**

end

Example 2.8: [*Midpoint circle drawing algorithm*]
A circle having a radius of 5 pixels and centered at $[3, 4]^T$ is to be drawn on a computer screen. Use the midpoint algorithm to determine what pixels should approximate the circle.

Solution 2.8: The initial value of $h_{\dot{m}}$ is given by

$$h_{\text{m}} = 1 - r = 1 - 5 = -4.$$

The initial values for x and y are 0 and -5 respectively. The start and endpoints of each quadrant are $[3, 9]^T$, $[3, -1]^T$, $[8, 4]^T$ and $[-2, 4]^T$. The rest of the points are listed in the following table:

10: x	12/14: h_{m}	15: y	17: x	18: Plot	19: Plot	20: Plot	21: Plot	22: Plot	23: Plot	24: Plot	25: Plot
0	-1		1	$[4, -1]^T$	$[4, 9]^T$	$[2, -1]^T$	$[2, 9]^T$	$[-2, 5]^T$	$[-2, 3]^T$	$[8, 5]^T$	$[8, 3]$
1	4		2	$[5, -1]^T$	$[5, 9]^T$	$[1, -1]^T$	$[1, 9]^T$	$[-2, 6]^T$	$[-2, 2]^T$	$[8, 6]^T$	$[8, 2]$
2	3	-4	3	$[6, 0]^T$	$[6, 8]^T$	$[0, 0]^T$	$[0, 8]^T$	$[-1, 7]^T$	$[-1, 1]^T$	$[7, 7]^T$	$[7, 1]$
3					no iteration as $x = -(y + 1)$						

Of course, all points with negative coordinates are disregarded. Hence, the points considered are $[3, 9]^T$, $[8, 4]^T$, $[4, 9]^T$, $[2, 9]^T$, $[8, 5]^T$, $[8, 3]^T$, $[5, 9]^T$, $[1, 9]^T$, $[8, 6]^T$, $[8, 2]^T$, $[6, 0]^T$, $[6, 8]^T$, $[0, 0]^T$, $[0, 8]^T$, $[7, 7]^T$ and $[7, 1]^T$. Note that those are the same points obtained in Example 2.7 using the 8-way symmetry algorithm but produced by a faster integer algorithm. □

2.3 Polygons

A 2D *polygon* is a closed planar path composed of a number of sequential straight line segments. Each line segment is called a *side* or an *edge*. The intersections between line segments are called *vertices*. The end vertex of the last edge is the start vertex of the first edge. A polygon encloses an area.

A *polyline* is similar to a polygon except that the end vertex of the last edge does not have to be the start vertex of the first edge. Hence, unlike polygons, no area is expected to emerge for a polyline.

2.3.1 Convexity Versus Concavity

A polygon may be *convex* or *concave*. Examples of convex and concave polygons are shown in Fig. 2.9. A polygon is convex if the line connecting any two interior points is included completely in the interior of the polygon. Each interior angle in a convex polygon must be $\leq 180°$; otherwise, the polygon is considered concave.

A 2D polygon is stored as a set of vertex coordinates (i.e., $[x_i, y_i]^T$ where i is the vertex number). These coordinates can be used to determine whether the polygon is convex or concave. Cross product or linear equations of the edges can be used to answer the question of convexity/concavity of a polygon.

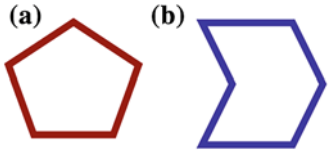

Fig. 2.9 Examples of polygons. **a** Convex polygon where any interior angle is $\leq 180°$. **b** Concave polygon where at least one of the interior angles is $>180°$

Cross product: This method proceeds as follows:

1. For each edge:

   a. Define a vector for the edge $\mathbf{e}_i$ connecting the vertices $\dot{\mathbf{v}}_i$ and $\dot{\mathbf{v}}_{i+1}$. This is expressed as

   $$\mathbf{e}_i = \dot{\mathbf{v}}_{i+1} - \dot{\mathbf{v}}_i. \tag{2.21}$$

   b. Compute the 2D cross product (Sect. A.4.3.4) along consecutive edges $\mathbf{e}_{i-1}$ and $\mathbf{e}_i$ as

   $$\begin{aligned} \mathbf{e}_{i-1} \times \mathbf{e}_i &= [\dot{\mathbf{v}}_i - \dot{\mathbf{v}}_{i-1}] \times [\dot{\mathbf{v}}_{i+1} - \dot{\mathbf{v}}_i] \\ &= \begin{bmatrix} x_i - x_{i-1} \\ y_i - y_{i-1} \end{bmatrix} \times \begin{bmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{bmatrix} \\ &= (x_i - x_{i-1})(y_{i+1} - y_i) - (y_i - y_{i-1})(x_{i+1} - x_i). \end{aligned} \tag{2.22}$$

2. The polygon is convex if and only if all signs of cross products along all edges are the same; otherwise, the polygon is concave.

Linear equations: This method proceeds as follows:

1. For each edge:

   a. Estimate the linear equation of the edge connecting the vertices $\dot{\mathbf{v}}_i = [x_i, y_i]^T$ and $\dot{\mathbf{v}}_{i+1} = [x_{i+1}, y_{i+1}]^T$. This is expressed in explicit form as

   $$y = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i \tag{2.23}$$

   or in $ax + by + c = 0$ implicit form as

   $$\underbrace{(y_{i+1} - y_i)}_{a} x + \underbrace{(x_i - x_{i+1})}_{b} y + \underbrace{y_i x_{i+1} - x_i y_{i+1}}_{c} = 0. \tag{2.24}$$

   Another way to get the same equation is to calculate the cross product (Sect. A.4.3.4) of the homogeneous points (Sect. B.7) $\mathbf{v}_i = [x_i, y_i, 1]^T$ and $\mathbf{v}_{i+1} = [x_{i+1}, y_{i+1}, 1]^T$. Thus,

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \mathbf{v}_i \times \mathbf{v}_{i+1} = \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \times \begin{bmatrix} x_{i+1} \\ y_{i+1} \\ 1 \end{bmatrix}. \tag{2.25}$$

b. Apply each of the remaining vertices to the linear equation estimated above.

2. The polygon is convex if and only if all signs obtained are the same for each single edge; otherwise, the polygon is concave.

Example 2.9: [*Polygon concavity/convexity*]
At least two different methods may be used to decide whether or not a 2D polygon is concave or convex. Apply each of them to the 2D polygon specified by the vertices $[0, 0]^T$, $[5, 0]^T$, $[-1, 5]^T$ and $[-1, -5]^T$. Based on your calculations, determine whether this polygon is concave or convex.

Solution 2.9: This problem can be solved using 2D cross product or using linear equations.

1. Using 2D cross product:
   a. Substitute $\dot{\mathbf{v}}_{i-1} = [-1, -5]^T$, $\dot{\mathbf{v}}_i = [0, 0]^T$ and $\dot{\mathbf{v}}_{i+1} = [5, 0]^T$ in Eq. (2.22) and calculate $\mathbf{e}_{i-1} \times \mathbf{e}_i$ as

$$\begin{aligned} \mathbf{e}_{i-1} \times \mathbf{e}_i &= [\dot{\mathbf{v}}_i - \dot{\mathbf{v}}_{i-1}] \times [\dot{\mathbf{v}}_{i+1} - \dot{\mathbf{v}}_i] \\ &= (x_i - x_{i-1})(y_{i+1} - y_i) - (y_i - y_{i-1})(x_{i+1} - x_i) \\ &= (0 - (-1))(0 - 0) - (0 - (-5))(5 - 0) = -25 \Rightarrow -ve. \end{aligned}$$

   b. Substitute $\dot{\mathbf{v}}_{i-1} = [0, 0]^T$, $\dot{\mathbf{v}}_i = [5, 0]^T$ and $\dot{\mathbf{v}}_{i+1} = [-1, 5]^T$ in Eq. (2.22) and calculate $\mathbf{e}_{i-1} \times \mathbf{e}_i$ as

$$\begin{aligned} \mathbf{e}_{i-1} \times \mathbf{e}_i &= [\dot{\mathbf{v}}_i - \dot{\mathbf{v}}_{i-1}] \times [\dot{\mathbf{v}}_{i+1} - \dot{\mathbf{v}}_i] \\ &= (x_i - x_{i-1})(y_{i+1} - y_i) - (y_i - y_{i-1})(x_{i+1} - x_i) \\ &= (5 - 0)(5 - 0) - (0 - 0)(-1 - 5) = +25 \Rightarrow +ve. \end{aligned}$$

   The substitutions result in different signs. This implies that it is a concave polygon.
2. Using linear equations:
   a. Consider the linear equation given the points $[x_i, y_i]^T = [-1, -5]^T$ and $[x_{i+1}, y_{i+1}]^T = [0, 0]^T$. Using Eq. (2.23), we have

$$\begin{aligned} y &= \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i \\ y &= 5x + 5 - 5 \\ y - 5x &= 0. \end{aligned}$$

b. Apply the other two points ($[5, 0]^T$ and $[-1, 5]^T$) to the previous equation to get

$$\begin{aligned} y - 5x &= 0 \\ 0 - 5(5) &= -25 \Rightarrow -ve, \\ 5 - 5(-1) &= +10 \Rightarrow +ve. \end{aligned}$$

The substitutions result in different signs. This implies that it is a concave polygon. □

2.4 Line Clipping

Given a 2D line or a group of 2D lines, a clip rectangle or window can be used to clip those lines so that only lines or portions of lines inside the clip window are preserved while the other lines or portions of lines are removed. Such an approach is referred to as a *2D line clipping* algorithm. It should be noted that even though there are many algorithms for rectangle and polygon clipping, a line clipping algorithm can be used repeatedly to clip polygons of any shape approximated by line segments.

An example of line clipping is shown in Fig. 2.10 where lines preserved after clipping appear thicker. There are three distinctive cases that may be observed:

1. Both endpoints of the line are *inside* the clip rectangle as line $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$ shown in Fig. 2.10.
2. One endpoint is *inside* the clip rectangle while the other endpoint is *outside* the rectangle as line $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$ shown in Fig. 2.10.
3. Both endpoints of the line are *outside* the clip rectangle as lines $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$ and $\overline{\dot{\mathbf{g}}\dot{\mathbf{h}}}$ shown in Fig. 2.10.

Dealing with each of the previous cases is different.

Both endpoints are inside the clip rectangle: A line is completely inside a clip rectangle or window if both endpoints are inside the window. Assume that the clip rectangle spans from $[x_{min}, y_{min}]^T$ to $[x_{max}, y_{max}]^T$ and a line goes from $[x_0, y_0]^T$ to $[x_1, y_1]^T$, the line is preserved and *trivially accepted* if all the following eight inequalities hold:

$$\begin{aligned} x_{min} \leq x_0 \leq x_{max}, \\ y_{min} \leq y_0 \leq y_{max}, \\ x_{min} \leq x_1 \leq x_{max}, \\ y_{min} \leq y_1 \leq y_{max}. \end{aligned} \tag{2.26}$$

Only one endpoint is inside the clip rectangle: A portion of the line is inside the clip rectangle in case only one endpoint is inside the clip rectangle. Assume that the clip rectangle spans from $[x_{min}, y_{min}]^T$ to $[x_{max}, y_{max}]^T$ and a line goes from $[x_0, y_0]^T$ to $[x_1, y_1]^T$. Perform the following steps:

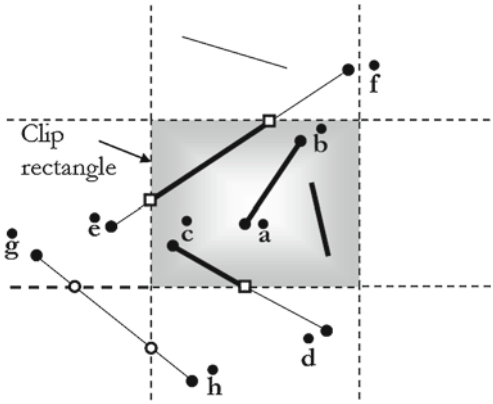

Fig. 2.10 Example of 2D line clipping. Lines remained after clipping appear thicker

1. Check if only one endpoint falls inside the clip rectangle. The endpoint $[x_0, y_0]^T$ is inside the clip rectangle if the following tests are true:

$$\begin{aligned} x_{min} \leq x_0 \leq x_{max}, \\ y_{min} \leq y_0 \leq y_{max}. \end{aligned} \tag{2.27}$$

   The endpoint $[x_1, y_1]^T$ is inside the clip rectangle if the following tests are true:

$$\begin{aligned} x_{min} \leq x_1 \leq x_{max}, \\ y_{min} \leq y_1 \leq y_{max}. \end{aligned} \tag{2.28}$$

   Note that either Inequality (2.27) *or* Inequality (2.28) must be true but *not* both; otherwise, we will go back to the first case where both endpoints are inside the clip rectangle. (Note that either Inequality (2.27) or Inequality (2.28) can be used to keep or neglect isolated points; a process that is referred to as *point clipping*.)
2. Detect the intersection point between the line and the clip rectangle. Many methods exist to detect line intersections. The most suitable method in this situation is using the linear parametric form. There are four intersection points between the line and each of the clip rectangle edges to be detected. The intersection point $[x, y]^T$ along the line that goes from $[x_0, y_0]^T$ to $[x_1, y_1]^T$ can be expressed in parametric form as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + t_{line} \begin{bmatrix} x_1 - x_0 \\ y_1 - y_0 \end{bmatrix}, \tag{2.29}$$

   where t_{line} is a parameter that determines the location of the point $[x, y]^T$ along the line such that $t_{line} \in [0, 1]$. The intersection point $[x, y]^T$ along the lower and upper horizontal clip rectangle edges can be expressed in parametric forms respectively as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{min} \end{bmatrix} + t_{edge1} \begin{bmatrix} x_{max} - x_{min} \\ y_{min} - y_{min} \end{bmatrix} \tag{2.30}$$

and

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{max} \end{bmatrix} + t_{edge2} \begin{bmatrix} x_{max} - x_{min} \\ y_{max} - y_{max} \end{bmatrix}, \tag{2.31}$$

where t_{edge1} and t_{edge2} are parameters that determine the location of the intersection point $[x, y]^T$ along the clip rectangle lower and upper edges respectively such that $t_{edge1} \in [0, 1]$ and $t_{edge2} \in [0, 1]$. Note that a value of either 0 or 1 means that the line is going through a corner of the clip rectangle. Similarly, the intersection point $[x, y]^T$ along the left and right vertical clip rectangle edges can be expressed in parametric forms respectively as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{min} \end{bmatrix} + t_{edge3} \begin{bmatrix} x_{min} - x_{min} \\ y_{max} - y_{min} \end{bmatrix} \tag{2.32}$$

and

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{max} \\ y_{min} \end{bmatrix} + t_{edge4} \begin{bmatrix} x_{max} - x_{max} \\ y_{max} - y_{min} \end{bmatrix}, \tag{2.33}$$

where t_{edge3} and t_{edge4} are parameters that determine the location of the point $[x, y]^T$ along the edges such that $t_{edge3} \in [0, 1]$ and $t_{edge4} \in [0, 1]$. To obtain the point $[x, y]^T$, solve Eq. (2.29) with each of Eqs. (2.30)–(2.33).

3. Check which intersection point $[x, y]^T$ falls inside the boundaries of the clip rectangle. Intersection points may occur at line and/or edge extensions. If this is the case, the following will be true: $t_{line} \notin [0, 1]$ and/or $t_{edgei} \notin [0, 1]$ where i is the edge number. Thus, checking the value of t_{line} and t_{edgei} against the interval [0,1] determines whether the intersection point falls within the clip rectangle. Another way to determine if an intersection point $[x, y]^T$ falls within the clip rectangle is by checking

$$\begin{aligned} x_{min} \leq x \leq x_{max}, \\ y_{min} \leq y \leq y_{max}, \end{aligned} \tag{2.34}$$

which must be true so that the point $[x, y]^T$ is determined as falling within the clip rectangle.

Finally, the portion of the line from the intersection point to the inside endpoint is kept while the rest of the line is removed.

Both endpoints are outside the clip rectangle: If both endpoints are outside the clip rectangle, the line may be completely outside the clip rectangle as line $\overline{\dot{\mathbf{g}}\dot{\mathbf{h}}}$ in Fig. 2.10 or part of the line is inside the clip rectangle as line $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$ in the same figure. In this case, the intersection points between the line and the clip rectangle must be detected as in the previous case. However, in the current case we must differentiate between two type of intersection points.

1. The first type appears in white squares in Fig. 2.10. These intersection points fall within the boundaries of the clip rectangle (i.e., $t_{edgei} \in [0, 1]$ where i is the edge

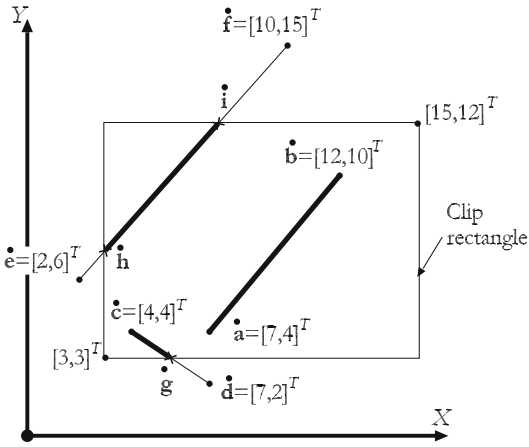

Fig. 2.11 Three lines clipped by a rectangle

number). The portion of the line connecting these intersection points is inside the clip rectangle and must be kept.

2. The second type appears in white circles in Fig. 2.10. These intersection points are located outside the boundaries of the clip rectangle (i.e., $t_{edgei} \notin [0, 1]$ where i is the edge number). The whole line must be removed.

The previous brute-force idea is inefficient and expensive as many calculations must be performed. The next example clarifies this claim.

Example 2.10: [*Line clipping*]

Figure 2.11 shows three line segments $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$, $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$ and $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$ where $\dot{\mathbf{a}} = [7, 4]^T$, $\dot{\mathbf{b}} = [12, 10]^T$, $\dot{\mathbf{c}} = [4, 4]^T$, $\dot{\mathbf{d}} = [7, 2]^T$, $\dot{\mathbf{e}} = [2, 6]^T$ and $\dot{\mathbf{f}} = [10, 15]^T$. If a clip rectangle spanning from $[3, 3]^T$ to $[15, 12]^T$ is used to clip these lines, what lines or portions of lines are preserved and kept?

Solution 2.10: **Working with line $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$**: Check if endpoints fall within the boundaries of the clip rectangle:

$$
\begin{aligned}
x_{min} \leq x_{\dot{\mathbf{a}}} \leq x_{max} &\Longrightarrow 3 \leq 7 \leq 15 \Longrightarrow true,\\
y_{min} \leq y_{\dot{\mathbf{a}}} \leq y_{max} &\Longrightarrow 3 \leq 4 \leq 12 \Longrightarrow true,\\
x_{min} \leq x_{\dot{\mathbf{b}}} \leq x_{max} &\Longrightarrow 3 \leq 12 \leq 15 \Longrightarrow true,\\
y_{min} \leq y_{\dot{\mathbf{b}}} \leq y_{max} &\Longrightarrow 3 \leq 10 \leq 12 \Longrightarrow true.
\end{aligned}
$$

As all inequalities result in true condition, we conclude that the line $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$ falls completely within the boundaries of the clip rectangle and must be preserved.

Working with line $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$: Check if endpoints fall within the boundaries of the clip rectangle:

$$\begin{aligned}
x_{min} \leq x_{\dot{\mathbf{c}}} \leq x_{max} &\Longrightarrow 3 \leq 4 \leq 15 \Longrightarrow true,\\
y_{min} \leq y_{\dot{\mathbf{c}}} \leq y_{max} &\Longrightarrow 3 \leq 4 \leq 12 \Longrightarrow true,\\
x_{min} \leq x_{\dot{\mathbf{d}}} \leq x_{max} &\Longrightarrow 3 \leq 7 \leq 15 \Longrightarrow true,\\
y_{min} \leq y_{\dot{\mathbf{d}}} \leq y_{max} &\Longrightarrow 3 \leq 2 \leq 12 \Longrightarrow false.
\end{aligned}$$

Hence, the endpoint $\dot{\mathbf{c}}$ falls within the clip rectangle while the endpoint $\dot{\mathbf{d}}$ is outside it. Now, check the intersection point between the line and the lower horizontal edge:

$$\begin{bmatrix} x_{\dot{\mathbf{c}}} \\ y_{\dot{\mathbf{c}}} \end{bmatrix} + t_{line} \begin{bmatrix} x_{\dot{\mathbf{d}}} - x_{\dot{\mathbf{c}}} \\ y_{\dot{\mathbf{d}}} - y_{\dot{\mathbf{c}}} \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{min} \end{bmatrix} + t_{edge1} \begin{bmatrix} x_{max} - x_{min} \\ y_{min} - y_{min} \end{bmatrix}$$
$$\begin{bmatrix} 4 \\ 4 \end{bmatrix} + t_{line} \begin{bmatrix} 7-4 \\ 2-4 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix} + t_{edge1} \begin{bmatrix} 15-3 \\ 3-3 \end{bmatrix}$$

or

$$\begin{aligned}
4 + 3\, t_{line} &= 3 + 12\, t_{edge1}\\
4 - 2\, t_{line} &= 3.
\end{aligned}$$

Solving these two equations results in values of $\frac{1}{2}$ and $\frac{2.5}{12}$ for t_{line} and t_{edge1} respectively. Since $t_{line} \in [0, 1]$ and $t_{edge1} \in [0, 1]$, the intersection point falls within the boundaries of the clip rectangle. Thus, using the line equation and $t_{line} = \frac{1}{2}$, the intersection point $[x, y]^T$ will be

$$\begin{aligned}
\begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} x_{\dot{\mathbf{c}}} \\ y_{\dot{\mathbf{c}}} \end{bmatrix} + t_{line} \begin{bmatrix} x_{\dot{\mathbf{d}}} - x_{\dot{\mathbf{c}}} \\ y_{\dot{\mathbf{d}}} - y_{\dot{\mathbf{c}}} \end{bmatrix}\\
&= \begin{bmatrix} 4 \\ 4 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 7-4 \\ 2-4 \end{bmatrix} = \begin{bmatrix} 5.5 \\ 3 \end{bmatrix}.
\end{aligned}$$

The same point is obtained when using $t_{edge1} = \frac{2.5}{12}$ with the parametric form of the edge. That is

$$\begin{aligned}
\begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} x_{min} \\ y_{min} \end{bmatrix} + t_{edge1} \begin{bmatrix} x_{max} - x_{min} \\ y_{min} - y_{min} \end{bmatrix}\\
&= \begin{bmatrix} 3 \\ 3 \end{bmatrix} + \frac{2.5}{12} \begin{bmatrix} 15-3 \\ 3-3 \end{bmatrix} = \begin{bmatrix} 5.5 \\ 3 \end{bmatrix}.
\end{aligned}$$

Because there is one endpoint inside the clip rectangle and one endpoint outside it, there is at most one intersection point that falls inside the borders of the clip rectangle. In other words, there is no need to check for intersection points with the rest of the edges. As $[4, 4]^T$ is the inside point, then the portion from $[4, 4]^T$ to $[5.5, 3]^T$ is kept while the rest of the line (i.e., from $[5.5, 3]^T$ to $[7, 2]^T$) is removed.

Working with line $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$: Check if endpoints fall within the boundaries of the clip rectangle:

$$\begin{aligned}
x_{min} \le x_{\dot{\mathbf{e}}} \le x_{max} &\Longrightarrow 3 \le 2 \le 15 \;\; \Longrightarrow \textit{false},\\
y_{min} \le y_{\dot{\mathbf{e}}} \le y_{max} &\Longrightarrow 3 \le 6 \le 12 \;\; \Longrightarrow \textit{true},\\
x_{min} \le x_{\dot{\mathbf{f}}} \le x_{max} &\Longrightarrow 3 \le 10 \le 15 \Longrightarrow \textit{true},\\
y_{min} \le y_{\dot{\mathbf{f}}} \le y_{max} &\Longrightarrow 3 \le 15 \le 12 \Longrightarrow \textit{false}.
\end{aligned}$$

Thus, both endpoints are outside the clip rectangle. Hence, the line $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$ may be completely outside the clip rectangle or intersects its boundaries at two intersection points. We check the intersection point between the line and the lower horizontal edge:

$$\begin{bmatrix} x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{e}}} \end{bmatrix} + t_{line} \begin{bmatrix} x_{\dot{\mathbf{f}}} - x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{f}}} - y_{\dot{\mathbf{e}}} \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{min} \end{bmatrix} + t_{edge1} \begin{bmatrix} x_{max} - x_{min} \\ y_{min} - y_{min} \end{bmatrix}$$
$$\begin{bmatrix} 2 \\ 6 \end{bmatrix} + t_{line} \begin{bmatrix} 10 - 2 \\ 15 - 6 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix} + t_{edge1} \begin{bmatrix} 15 - 3 \\ 3 - 3 \end{bmatrix}$$

or

$$\begin{aligned}
2 + 8\, t_{line} &= 3 + 12\, t_{edge1}\\
6 + 9\, t_{line} &= 3.
\end{aligned}$$

The value of t_{line} is $-\frac{1}{3}$, which does not belong to the interval [0,1]; hence, this intersection point is outside the clip rectangle. Check the next possible intersection between the line and the upper horizontal edge:

$$\begin{bmatrix} x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{e}}} \end{bmatrix} + t_{line} \begin{bmatrix} x_{\dot{\mathbf{f}}} - x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{f}}} - y_{\dot{\mathbf{e}}} \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{max} \end{bmatrix} + t_{edge2} \begin{bmatrix} x_{max} - x_{min} \\ y_{max} - y_{max} \end{bmatrix}$$
$$\begin{bmatrix} 2 \\ 6 \end{bmatrix} + t_{line} \begin{bmatrix} 10 - 2 \\ 15 - 6 \end{bmatrix} = \begin{bmatrix} 3 \\ 12 \end{bmatrix} + t_{edge2} \begin{bmatrix} 15 - 3 \\ 12 - 12 \end{bmatrix}$$

or

$$\begin{aligned}
2 + 8\, t_{line} &= 3 + 12\, t_{edge2}\\
6 + 9\, t_{line} &= 12.
\end{aligned}$$

Solving these two equations results in values of $\frac{2}{3}$ and $\frac{13}{36}$ for t_{line} and t_{edge2} respectively. Since $t_{line} \in [0, 1]$ and $t_{edge2} \in [0, 1]$, the intersection point falls within the boundaries of the clip rectangle. Thus, using the line equation with $t_{line} = \frac{2}{3}$, the intersection point $[x, y]^T$ will be

$$\begin{aligned}
\begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{e}}} \end{bmatrix} + t_{line} \begin{bmatrix} x_{\dot{\mathbf{f}}} - x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{f}}} - y_{\dot{\mathbf{e}}} \end{bmatrix}\\
&= \begin{bmatrix} 2 \\ 6 \end{bmatrix} + \frac{2}{3} \begin{bmatrix} 10 - 2 \\ 15 - 6 \end{bmatrix} = \begin{bmatrix} 7.3333 \\ 12 \end{bmatrix}.
\end{aligned}$$

The same point is obtained when using $t_{edge2} = \frac{13}{36}$ with the parametric form of the edge. That is

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{max} \end{bmatrix} + t_{edge2} \begin{bmatrix} x_{max} - x_{min} \\ y_{max} - y_{max} \end{bmatrix}$$
$$= \begin{bmatrix} 3 \\ 12 \end{bmatrix} + \frac{13}{36} \begin{bmatrix} 15 - 3 \\ 12 - 12 \end{bmatrix} = \begin{bmatrix} 7.3333 \\ 12 \end{bmatrix}.$$

Because both endpoints are outside the clip rectangle and t_{edge2} is neither 0 nor 1, we expect that there is only one more intersection point that falls inside the borders of the clip rectangle. So, we will go for the third edge (i.e., the left vertical edge):

$$\begin{bmatrix} x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{e}}} \end{bmatrix} + t_{line} \begin{bmatrix} x_{\dot{\mathbf{f}}} - x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{f}}} - y_{\dot{\mathbf{e}}} \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{min} \end{bmatrix} + t_{edge3} \begin{bmatrix} x_{min} - x_{min} \\ y_{max} - y_{min} \end{bmatrix}$$
$$\begin{bmatrix} 2 \\ 6 \end{bmatrix} + t_{line} \begin{bmatrix} 10 - 2 \\ 15 - 6 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \end{bmatrix} + t_{edge3} \begin{bmatrix} 3 - 3 \\ 12 - 3 \end{bmatrix}$$

or

$$2 + 8\, t_{line} = 3$$
$$6 + 9\, t_{line} = 3 + 9\, t_{edge3}.$$

Solving these two equations results in values of $\frac{1}{8}$ and $\frac{11}{24}$ for t_{line} and t_{edge3} respectively. Since $t_{line} \in [0, 1]$ and $t_{edge2} \in [0, 1]$, the intersection point falls within the boundaries of the clip rectangle. So, using the line equation with $t_{line} = \frac{1}{8}$, the intersection point $[x, y]^T$ will be estimated as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{e}}} \end{bmatrix} + t_{line} \begin{bmatrix} x_{\dot{\mathbf{f}}} - x_{\dot{\mathbf{e}}} \\ y_{\dot{\mathbf{f}}} - y_{\dot{\mathbf{e}}} \end{bmatrix}$$
$$= \begin{bmatrix} 2 \\ 6 \end{bmatrix} + \frac{1}{8} \begin{bmatrix} 10 - 2 \\ 15 - 6 \end{bmatrix} = \begin{bmatrix} 3 \\ 7.125 \end{bmatrix}.$$

The same point is obtained when using $t_{edge3} = \frac{11}{24}$ with the parametric form of the edge. That is

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{min} \end{bmatrix} + t_{edge3} \begin{bmatrix} x_{min} - x_{min} \\ y_{max} - y_{min} \end{bmatrix}$$
$$= \begin{bmatrix} 3 \\ 3 \end{bmatrix} + \frac{11}{24} \begin{bmatrix} 3 - 3 \\ 12 - 3 \end{bmatrix} = \begin{bmatrix} 3 \\ 7.125 \end{bmatrix}.$$

So, the portion from $[3, 7.125]^T$ to $[7.3333, 12]^T$ is kept while the portions from $[2, 6]^T$ to $[3, 7.125]^T$ and from $[7.3333, 12]^T$ to $[10, 15]^T$ are removed. □

In the literature, there are many 2D *line clipping* algorithms such as Liang-Barsky algorithm (Liang and Barsky 1984), Nicholl-Lee-Nicholl algorithm (Nicholl et al.

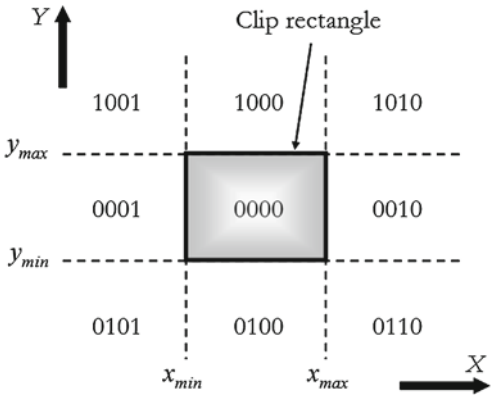

Fig. 2.12 The space is divided into nine regions where the middle region represents the clip rectangle, window, polygon or the viewport. Each region is assigned a 4-bit outcode

Table 2.1 Assigning outcodes for the nine regions

Bit	Value	Meaning
3	= 1, if region is above the top edge	if $y > y_{max}$
	= 0, otherwise	if $y \leq y_{max}$
2	= 1, if region is below the bottom edge	if $y < y_{min}$
	= 0, otherwise	if $y \geq y_{min}$
1	= 1, if region is right to the right edge	if $x > x_{max}$
	= 0, otherwise	if $x \leq x_{max}$
0	= 1, if region is left to the left edge	if $x < x_{min}$
	= 0, otherwise	if $x \geq x_{min}$

Bit 3 represents the most-significant bit while bit 0 represents the least-significant bit

1987), Cyrus-Beck algorithm (Cyrus and Beck 1978) and Cohen-Sutherland algorithm. We will discuss the Cohen-Sutherland algorithm in more details below.

2.4.1 Cohen-Sutherland Algorithm

In Cohen-Sutherland clipping algorithm, the 2D space is divided into nine regions where the *middle region* represents the clip rectangle, window, polygon or the viewport as shown in Fig. 2.12.

Each region is assigned a 4-bit outcode. Each binary digit indicates where the region is with respect to the clip rectangle that is assigned the outcode 0000. The bits are arranged from left to right as top, bottom, right and left. Assuming that the clip rectangle spans from $[x_{min}, y_{min}]^T$ to $[x_{max}, y_{max}]^T$ as shown in Fig. 2.12, a point $[x, y]^T$ is assigned the bit values listed in Table 2.1 starting from the most-significant bit to the least-significant bit. For example, 1010 represents any point in the top right region while 0110 represents any point in the bottom right region. The outcodes are shown in Fig. 2.12. Note that in some implementations of this algorithm, the bits are arranged from left to right as left, right, bottom and top instead of top, bottom, right and left (which we are using). Also, some implementations deals with the

Table 2.2 OR truth table and AND truth table

a	b	a OR b	a AND b
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

y-axis as pointing downwards. However, these discrepancies should not affect the final outcome of clipped lines.
The Cohen-Sutherland clipping algorithm clips a line as follows:

1. Determine the outcode for each endpoint. An outcode for a point $[x, y]^T$ is obtained by retrieving the *sign bit* of the following values:

$$\begin{aligned} &\text{Bit 3} \Longrightarrow y_{max} - y, \\ &\text{Bit 2} \Longrightarrow y - y_{min}, \\ &\text{Bit 1} \Longrightarrow x_{max} - x, \\ &\text{Bit 0} \Longrightarrow x - x_{min}, \end{aligned} \tag{2.35}$$

   where bit 3 represents the most-significant bit while bit 0 represents the least-significant bit. A sign bit is 1 for negative values and 0 otherwise. An alternative way to calculate the outcode is by ORing values. If *outcode* is initialized to 0000, it will take the following values:

$$\begin{aligned} &outcode = \begin{cases} outcode \text{ OR } 1000, & \text{if } y > y_{max}; \\ outcode \text{ OR } 0100, & \text{if } y < y_{min}, \end{cases} \\ &\text{then} \\ &outcode = \begin{cases} outcode \text{ OR } 0010, & \text{if } x > x_{max}; \\ outcode \text{ OR } 0001, & \text{if } x < x_{min}. \end{cases} \end{aligned} \tag{2.36}$$

   An algorithm to obtain the outcode for a point $[x, y]^T$ given the lower left and upper right corners (i.e., $[x_{min}, y_{min}]^T$ and $[x_{max}, y_{max}]^T$) of the clip rectangle is listed in Algorithm 2.13.
2. Consider the two outcodes determined above.
   a. If both endpoints are in the clip rectangle (i.e., having the same outcode of 0000), bitwise-OR the bits. This results in a value of 0000. In this case, *trivially accept* the line. The OR truth table is listed in Table 2.2.
   b. Otherwise, if both endpoints are outside the clip rectangle (i.e., having outcodes other than 0000), bitwise-AND the bits. If this results in a value other than 0000, *trivially reject* the line. The AND truth table is listed in Table 2.2.
   c. Otherwise, segment the line using the edges of the clip rectangle:
      i. Find an endpoint $[x_0, y_0]^T$ that is outside the clip rectangle (i.e., an outpoint) where its outcode $\neq 0000$ (at least one endpoint is outside the clip rectangle).
      ii. Find the intersection point $[x, y]^T$ between the line and the clip rectangle. If the outpoint is $[x_0, y_0]^T$ and the other endpoint is $[x_1, y_1]^T$, the intersection point is given by:

$$\dot{\mathbf{p}}_{top} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + \frac{y_{max} - y_0}{m} \\ y_{max} \end{bmatrix},$$

$$\dot{\mathbf{p}}_{bottom} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + \frac{y_{min} - y_0}{m} \\ y_{min} \end{bmatrix}, \tag{2.37}$$

$$\dot{\mathbf{p}}_{right} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{max} \\ y_0 + m(x_{max} - x_0) \end{bmatrix},$$

$$\dot{\mathbf{p}}_{left} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_0 + m(x_{min} - x_0) \end{bmatrix},$$

where $\dot{\mathbf{p}}_{top}$, $\dot{\mathbf{p}}_{bottom}$, $\dot{\mathbf{p}}_{right}$ and $\dot{\mathbf{p}}_{left}$ are the intersection points with the top, bottom, right and left edges of the clip rectangle respectively; $[x_{min}, y_{min}]^T$ and $[x_{max}, y_{max}]^T$ are the lower left and upper right points of the clip rectangle respectively; and m is the line slope.

iii. The portion from the outpoint to the intersection point should be removed or rejected. The outpoint is replaced by the intersection point. Re-estimate the outcode for the outpoint.

iv. Go to Step 2.

3. If trivially accepted, draw the line.

Algorithm 2.14 lists the steps of the Cohen-Sutherland clipping algorithm (Foley et al. 1995).

Algorithm 2.13 *Outcode calculation algorithm*

Input: $x, y, x_{min}, x_{max}, y_{min}, y_{max}$
Output: *outcode*
1: $outcode = 0000$
2: **if** $(y > y_{max})$ **then**
3: $outcode = outcode$ OR 1000
4: **else if** $(y < y_{min})$ **then**
5: $outcode = outcode$ OR 0100
6: **end if**
7: **if** $(x > x_{max})$ **then**
8: $outcode = outcode$ OR 0010
9: **else if** $(x < x_{min})$ **then**
10: $outcode = outcode$ OR 0001
11: **end if**
12: **return** *outcode*

end

Algorithm 2.14 *Cohen-Sutherland clipping algorithm*

Input: $x_0, y_0, x_1, y_1, x_{min}, y_{min}, x_{max}, y_{max}$

```
outcode0 = Call Algorithm 2.13 for [x0, y0]^T
outcode1 = Call Algorithm 2.13 for [x1, y1]^T
done = FALSE
in = FALSE

while (done = FALSE) do
    if (outcode0 OR outcode1 = 0000) then
        done = TRUE
        in = TRUE
    else if (outcode0 AND outcode1 ≠ 0000) then
        done = TRUE
    else
        m = (y1 − y0)/(x1 − x0)
        if (outcode0 ≠ 0000) then
            outcode = outcode0
        else
            outcode = outcode1
        end if

        if (outcode AND 1000 ≠ 0000) then
            x = x0 + (ymax − y0)/m
            y = ymax
        else if (outcode AND 0100 ≠ 0000) then
            x = x0 + (ymin − y0)/m
            y = ymin
        else if (outcode AND 0010 ≠ 0000) then
            x = xmax
            y = y0 + m(xmax − x0)
        else
            x = xmin
            y = y0 + m(xmin − x0)
        end if

        if (outcode = outcode0) then
            x0 = x
            y0 = y
            outcode0 = Call Algorithm 2.13 for [x0, y0]^T
        else
            x1 = x
            y1 = y
            outcode1 = Call Algorithm 2.13 for [x1, y1]^T
        end if
```

43: **end if**
44: **end while**
45:
46: **if** ($in = TRUE$) **then**
47: Call Algorithm 2.7 with parameters x_0, y_0, x_1, y_1
48: **end if**

end

One point to be mentioned here is that Algorithm 2.14 computes the slope m on Line 13. This operation may be repeated for the same line if it intersects the clip rectangle more than once. In this case, it is better to compute the slope before the loop. However, if the slope is computed before the loop and the line is trivially accepted or trivially rejected, the slope will be computed but will never be used.

Example 2.11: [*Cohen-Sutherland clipping algorithm—outcodes*]

Figure 2.13 shows four line segments $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$, $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$, $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$ and $\overline{\dot{\mathbf{g}}\dot{\mathbf{h}}}$ where $\dot{\mathbf{a}} = [2, 6]^T$, $\dot{\mathbf{b}} = [4, 10]^T$, $\dot{\mathbf{c}} = [7, 18]^T$, $\dot{\mathbf{d}} = [10, 10]^T$, $\dot{\mathbf{e}} = [10, 1]^T$, $\dot{\mathbf{f}} = [18, 10]^T$, $\dot{\mathbf{g}} = [12, 12]^T$ and $\dot{\mathbf{h}} = [14, 10]^T$. If a clip rectangle spanning from $[5, 3]^T$ to $[15, 15]^T$ is used to clip these lines utilizing the Cohen-Sutherland clipping algorithm, get the outcodes for each of the endpoints.

Solution 2.11: Applying Algorithm 2.13 and since $[x_{min}, y_{min}]^T = [5, 3]^T$ and $[x_{max}, y_{max}]^T = [15, 15]^T$, the outcodes are listed in the following table:

Point	$[x, y]^T$	Condition	Outcode
$\dot{\mathbf{a}}$	$[2, 6]^T$	$x < x_{min}$	0001
$\dot{\mathbf{b}}$	$[4, 10]^T$	$x < x_{min}$	0001
$\dot{\mathbf{c}}$	$[7, 18]^T$	$y > y_{max}$	1000
$\dot{\mathbf{d}}$	$[10, 10]^T$	within	0000
$\dot{\mathbf{e}}$	$[10, 1]^T$	$y < y_{min}$	0100
$\dot{\mathbf{f}}$	$[18, 10]^T$	$x > x_{max}$	0010
$\dot{\mathbf{g}}$	$[12, 12]^T$	within	0000
$\dot{\mathbf{h}}$	$[14, 10]^T$	within	0000

□

Example 2.12: [*Cohen-Sutherland clipping algorithm—line category*] In Example 2.11, determine which lines are trivially accepted/rejected and which lines need intersection determination.

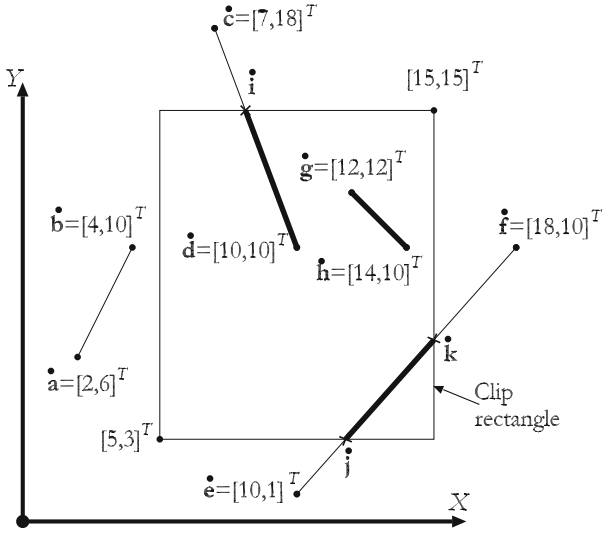

Fig. 2.13 Four lines clipped by a rectangle

Solution 2.12: In order to determine line categories, we apply the first part of Algorithm 2.14 (i.e., performing ORing and ANDing operations).

Line	ORing	ANDing	Decision
$\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$	0001	0001	Trivially reject
$\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$	1000	0000	Intersections
$\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$	0110	0000	Intersections
$\overline{\dot{\mathbf{g}}\dot{\mathbf{h}}}$	0000		Trivially accept

Hence, only $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$ and $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$ need further intersection determination. □

Example 2.13: [*Cohen-Sutherland clipping algorithm—intersection points*] In Example 2.11, if the lines to be clipped intersect the clip rectangle, determine the intersection points between those lines and clip rectangle.

Solution 2.13: As we found in Example 2.12, intersections will be performed only for $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$ and $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$.

Working with line $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$: The slope is obtained as

$$m = \frac{y_{\dot{\mathbf{d}}} - y_{\dot{\mathbf{c}}}}{x_{\dot{\mathbf{d}}} - x_{\dot{\mathbf{c}}}} = \frac{10 - 18}{10 - 7} = -\frac{8}{3} = -2.6667.$$

We get the outpoint which has the outcode that is not equal to 0000 as

$$outcode = outcode_{\dot{\mathbf{c}}} = 1000.$$

Since

$$outcode \text{ AND } 1000 = 1000,$$

then the line extending from the outpoint (i.e., $\dot{\mathbf{c}}$) to $\dot{\mathbf{d}}$ intersects the top edge of the clip rectangle. The intersection point $\dot{\mathbf{i}}$ between the line and the clip rectangle is obtained as

$$x_{\dot{\mathbf{i}}} = x_{\dot{\mathbf{c}}} + \frac{y_{max} - y_{\dot{\mathbf{c}}}}{m} = 7 + \frac{15 - 18}{-8/3} = 8\frac{1}{8} = 8.125,$$
$$y_{\dot{\mathbf{i}}} = y_{max} = 15.$$

Thus, the intersection point $\dot{\mathbf{i}}$ is $[8.125, 15]^T$. The intersection points between $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$ and the clip rectangle are obtained similarly. They are listed in the table below.

Line	Outpoint	m	Edge	Intersection
$\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$	$\dot{\mathbf{c}}$	$-\frac{8}{3}$	Top	$\dot{\mathbf{i}} = [8.125, 15]^T$
$\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$	$\dot{\mathbf{e}}$	$\frac{9}{8}$	Bottom	$\dot{\mathbf{j}} = [11.7778, 3]^T$
$\overline{\dot{\mathbf{j}}\dot{\mathbf{f}}}$	$\dot{\mathbf{f}}$	$\frac{9}{8}$	Right	$\dot{\mathbf{k}} = [15, 6.625]^T$

□

Example 2.14: [*Cohen-Sutherland clipping algorithm*]

Figure 2.11 shows three line segments $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$, $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$ and $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$ where $\dot{\mathbf{a}} = [7, 4]^T$, $\dot{\mathbf{b}} = [12, 10]^T$, $\dot{\mathbf{c}} = [4, 4]^T$, $\dot{\mathbf{d}} = [7, 2]^T$, $\dot{\mathbf{e}} = [2, 6]^T$ and $\dot{\mathbf{f}} = [10, 15]^T$. If a clip rectangle spanning from $[3, 3]^T$ to $[15, 12]^T$ is used to clip these lines utilizing the Cohen-Sutherland clipping algorithm, what lines or portions of lines are preserved and kept?

Solution 2.14: Working with line $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$:

1. Get the outcodes for the endpoints $\dot{\mathbf{a}}$ and $\dot{\mathbf{b}}$:

$$outcode_{\dot{\mathbf{a}}} = 0000,$$
$$outcode_{\dot{\mathbf{b}}} = 0000.$$

2. Perform a bitwise-ORing operation between $outcode_{\dot{\mathbf{a}}}$ and $outcode_{\dot{\mathbf{b}}}$:

$$outcode_{\dot{\mathbf{a}}} \text{ OR } outcode_{\dot{\mathbf{b}}} = 0000 \text{ OR } 0000 = 0000.$$

Since the result of the bitwise-ORing operation is 0000, both endpoints are in the clip rectangle.

3. Therefore, the line $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$ is trivially accepted.

Working with line $\overline{\dot{\mathbf{c}}\dot{\mathbf{d}}}$:

1. Get the outcodes for the endpoints $\dot{\mathbf{c}}$ and $\dot{\mathbf{d}}$:

$$\begin{aligned} outcode_{\dot{\mathbf{c}}} &= 0000, \\ outcode_{\dot{\mathbf{d}}} &= 0100. \end{aligned}$$

2.

   a. Perform a bitwise-ORing operation between $outcode_{\dot{\mathbf{c}}}$ and $outcode_{\dot{\mathbf{d}}}$:

$$outcode_{\dot{\mathbf{c}}} \text{ OR } outcode_{\dot{\mathbf{d}}} = 0000 \text{ OR } 0100 = 0100.$$

   b. Since the bitwise-ORing operation results in a value that is not 0000, perform a bitwise-ANDing operation between $outcode_{\dot{\mathbf{c}}}$ and $outcode_{\dot{\mathbf{d}}}$:

$$outcode_{\dot{\mathbf{c}}} \text{ AND } outcode_{\dot{\mathbf{d}}} = 0000 \text{ AND } 0100 = 0000.$$

   c.

      i. Since the bitwise-ANDing operation results in 0000, get the slope m of the line, which is computed as

$$m = \frac{y_{\dot{\mathbf{d}}} - y_{\dot{\mathbf{c}}}}{x_{\dot{\mathbf{d}}} - x_{\dot{\mathbf{c}}}} = \frac{2-4}{7-4} = -\frac{2}{3} = -0.6667.$$

      ii. Get the outpoint which has the outcode that is not equal to 0000 as

$$outcode_{\dot{\mathbf{d}}} = 0100.$$

      iii. Since

$$outcode_{\dot{\mathbf{d}}} \text{ AND } 0100 = 0100,$$

      then the line extending from the outpoint (i.e., $\dot{\mathbf{d}}$) to $\dot{\mathbf{c}}$ intersects the bottom edge of the clip rectangle. Split the line at the intersection point between the line and the clip rectangle. Utilizing the value of m, the intersection point $[x, y]^T$ is obtained as

$$\begin{aligned} x &= x_{\dot{\mathbf{d}}} + \frac{y_{min} - y_{\dot{\mathbf{d}}}}{m} = 7 + \frac{3-2}{-2/3} = 5.5, \\ y &= y_{min} = 3. \end{aligned}$$

      In other words, the intersection point is $[5.5, 3]^T$. Let us call this point $\dot{\mathbf{g}}$.

iv. Remove the part between $\dot{\mathbf{d}} = [7, 2]^T$ and $\dot{\mathbf{g}} = [5.5, 3]^T$. The outcode for the intersection point now is

$$outcode_{\dot{\mathbf{g}}} = 0000.$$

v. The line to be tested now is extending from $\dot{\mathbf{c}} = [4, 4]^T$ to $\dot{\mathbf{g}} = [5.5, 3]^T$. The outcode for each of its endpoints is 0000.

3. The procedure is applied again to line $\overline{\dot{\mathbf{c}}\dot{\mathbf{g}}}$. Hence, line $\overline{\dot{\mathbf{c}}\dot{\mathbf{g}}}$ is trivially accepted.

Working with line $\overline{\dot{\mathbf{e}}\dot{\mathbf{f}}}$:

1. Get the outcodes for the endpoints $\dot{\mathbf{e}}$ and $\dot{\mathbf{f}}$:

$$outcode_{\dot{\mathbf{e}}} = 0001,$$
$$outcode_{\dot{\mathbf{f}}} = 1000.$$

2.

a. Perform a bitwise-ORing operation between $outcode_{\dot{\mathbf{e}}}$ and $outcode_{\dot{\mathbf{f}}}$:

$$outcode_{\dot{\mathbf{e}}} \text{ OR } outcode_{\dot{\mathbf{f}}} = 0001 \text{ OR } 1000 = 1001.$$

b. Since the bitwise-ORing operation results in a value that is not 0000, perform a bitwise-ANDing operation between $outcode_{\dot{\mathbf{e}}}$ and $outcode_{\dot{\mathbf{f}}}$:

$$outcode_{\dot{\mathbf{e}}} \text{ AND } outcode_{\dot{\mathbf{f}}} = 0001 \text{ AND } 1000 = 0000.$$

c.

i. Since the bitwise-ANDing operation results in 0000, get the slope m of the line that is obtained as

$$m = \frac{y_{\dot{\mathbf{f}}} - y_{\dot{\mathbf{e}}}}{x_{\dot{\mathbf{f}}} - x_{\dot{\mathbf{e}}}} = \frac{15 - 6}{10 - 2} = \frac{9}{8} = 1.125.$$

ii. check the first point (i.e., $\dot{\mathbf{e}} = [2, 6]^T$) if it has an outcode that is not equal to 0000 as

$$outcode_{\dot{\mathbf{e}}} = 0001.$$

iii. Since

$$outcode_{\dot{\mathbf{e}}} \text{ AND } 0001 = 0001,$$

then the line intersects the left edge of the clip rectangle. Split the line at the intersection point between the line and the clip rectangle. Utilizing the slope m, the intersection point $[x, y]^T$ is estimated as

$$x = x_{min} = 3,$$
$$y = y_{\dot{\mathbf{e}}} + m(x_{min} - x_{\dot{\mathbf{e}}}) = 6 + \tfrac{9}{8}(3-2) = 7\tfrac{1}{8} = 7.125.$$

In other words, the intersection point is $[3, 7.125]^T$. Let us call this point $\dot{\mathbf{h}}$.

iv. Remove the part between $\dot{\mathbf{e}} = [2, 6]^T$ and $\dot{\mathbf{h}} = [3, 7.125]^T$. The outcode for the intersection point now is

$$outcode_{\dot{\mathbf{h}}} = 0000.$$

v. The line to be tested now is extending from $\dot{\mathbf{h}} = [3, 7.125]^T$ to $\dot{\mathbf{f}} = [10, 15]^T$.

Working with line $\overline{\dot{\mathbf{h}}\dot{\mathbf{f}}}$:

a. Perform a bitwise-ORing operation between $outcode_{\dot{\mathbf{h}}}$ and $outcode_{\dot{\mathbf{f}}}$:

$$outcode_{\dot{\mathbf{h}}} \text{ OR } outcode_{\dot{\mathbf{f}}} = 0000 \text{ OR } 1000 = 1000.$$

b. Since the bitwise-ORing operation results in a value that is not 0000, perform a bitwise-ANDing operation between $outcode_{\dot{\mathbf{h}}}$ and $outcode_{\dot{\mathbf{f}}}$:

$$outcode_{\dot{\mathbf{h}}} \text{ AND } outcode_{\dot{\mathbf{f}}} = 0000 \text{ AND } 1000 = 0000.$$

c.

i. Since the bitwise-ANDing operation results in 0000, get the slope m of the line that is obtained as

$$m = \frac{9}{8} = 1.125.$$

ii. Get the outpoint (i.e., $\dot{\mathbf{f}}$) as it has an outcode that is not equal to 0000 as

$$outcode_{\dot{\mathbf{f}}} = 1000.$$

iii. Since

$$outcode_{\dot{\mathbf{f}}} \text{ AND } 1000 = 1000,$$

then the line intersects the top edge of the clip rectangle. Split the line at the intersection point between the line and the clip rectangle. Using the slope m, the intersection point is estimated as

$$x = x_{\dot{\mathbf{f}}} + \frac{y_{max} - y_{\dot{\mathbf{f}}}}{m} = 10 + \frac{12 - 15}{9/8} = 7\frac{1}{3} = 7.3333,$$
$$y = y_{max} = 12.$$

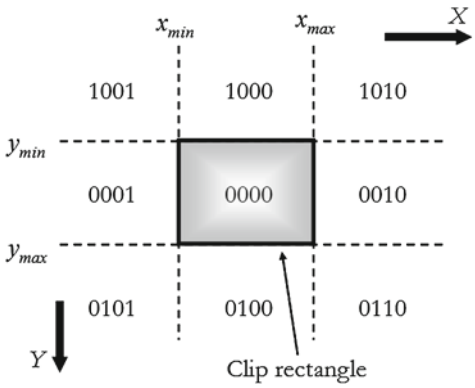

Fig. 2.14 In a left-handed coordinate system, the y-axis is pointing downwards. The space is divided into nine regions where the middle region represents the clip rectangle, window, polygon or the viewport. Each region is assigned a 4-bit outcode

In other words, the intersection point is $[7.3333, 12]^T$. Let us call this point $\dot{\mathbf{i}}$.

iv. Remove the part between $\dot{\mathbf{f}} = [10, 15]^T$ and $\dot{\mathbf{i}} = [7.3333, 12]^T$. The outcode for the intersection point now is

$$outcode_{\dot{\mathbf{i}}} = 0000.$$

v. The line to be tested now is extending from $\dot{\mathbf{h}} = [3, 7.125]^T$ to $\dot{\mathbf{i}} = [7.3333, 12]^T$. The outcode for each of its endpoints is 0000.

3. The procedure is applied again to line $\overline{\dot{\mathbf{h}}\dot{\mathbf{i}}}$. Hence, line $\overline{\dot{\mathbf{h}}\dot{\mathbf{i}}}$ is trivially accepted.

Thus, the preserved lines are

1. $\overline{\dot{\mathbf{a}}\dot{\mathbf{b}}}$: extending from $\dot{\mathbf{a}} = [7, 4]^T$ to $\dot{\mathbf{b}} = [12, 10]^T$;
2. $\overline{\dot{\mathbf{c}}\dot{\mathbf{g}}}$: extending from $\dot{\mathbf{c}} = [4, 4]^T$ to $\dot{\mathbf{g}} = [5.5, 3]^T$; and
3. $\overline{\dot{\mathbf{h}}\dot{\mathbf{i}}}$: extending from $\dot{\mathbf{h}} = [3, 7.125]^T$ to $\dot{\mathbf{i}} = [7.3333, 12]^T$. □

Example 2.15: [*Cohen-Sutherland clipping algorithm—outcodes in a left-handed coordinate system*]
Assume that a left-handed coordinate system is used to assign outcode values to different regions of the Cohen-Sutherland clipping algorithm. Modify Table 2.1 so that each region keeps its outcode (e.g., the upper left region is assigned 1001, the lower right region is assigned 0110, etc.).

Solution 2.15: Since the y-axis is pointing downwards in a left-handed coordinate system as shown in Fig. 2.14, Table 2.1 is modified as follows:

□

2.5 Polygon Clipping

A 2D polygon represented by a set of three or more vertices $(\dot{\mathbf{v}}_i | i \in \{0, 1, 2, \ldots, n-1\}$ where n is the number of vertices) can be clipped by another polygon. The output

Bit	Value	Meaning
3	= 1, if region is above the top edge	if $y < y_{min}$
	= 0, otherwise	if $y \geq y_{min}$
2	= 1, if region is below the bottom edge	if $y > y_{max}$
	= 0, otherwise	if $y \leq y_{max}$
1	= 1, if region is right to the right edge	if $x > x_{max}$
	= 0, otherwise	if $x \leq x_{max}$
0	= 1, if region is left to the left edge	if $x < x_{min}$
	= 0, otherwise	if $x \geq x_{min}$

of this clipping process is one or more polygons. The polygon after clipping may include vertices that are not part of the original vertices (i.e., new vertices may be created).

There are many 2D *polygon clippers* or *polygon clipping algorithms* such as Sutherland-Hodgman algorithm (Sutherland and Hodgman 1974), Patrick-Gilles Maillot algorithm (Maillot 1992), and Weiler-Atherton algorithm. We will discuss the Weiler-Atherton algorithm in more details.

2.5.1 Weiler-Atherton Algorithm

In Weiler-Atherton algorithm (Weiler and Atherton 1977), there are two types of polygons; a subject polygon that is to be clipped and a clip polygon or window as shown in Fig. 2.15a. The goal is to obtain the subject polygon after clipping as shown in Fig. 2.15b. In this algorithm, polygons are clockwise-oriented while holes are counter-clockwise-oriented. (Some researchers work with counter-clockwise-oriented polygons; however, this should not make a difference in the final outcome.) Also, in this algorithm, a polygon is represented as a circular list of vertices. The algorithm can be summarized by *walking* along the polygon boundaries as follows:

1. Compute the intersection points between the subject and clip polygons as depicted in Fig. 2.15c. Many methods to estimate the intersection points may be used. For example, given a subject edge $\overline{\dot{\mathbf{v}}_1\dot{\mathbf{v}}_2}$ bounded by $[x_{\dot{\mathbf{v}}_1}, y_{\dot{\mathbf{v}}_1}]^T$ and $[x_{\dot{\mathbf{v}}_2}, y_{\dot{\mathbf{v}}_2}]^T$ and a clipping horizontal (or vertical) edge $\overline{\dot{\mathbf{c}}_1\dot{\mathbf{c}}_2}$ bounded by $[x_{\dot{\mathbf{c}}_1}, y_{\dot{\mathbf{c}}_1}]^T$ and $[x_{\dot{\mathbf{c}}_2}, y_{\dot{\mathbf{c}}_2}]^T$, the intersection point $\dot{\mathbf{p}} = [x, y]^T$ may be estimated as done in Eq. (2.37). Alternatively, cross product (Sect. A.4.3.3) of homogeneous points (Sect. B.7) can also be used with lines with general slope values as

$$\mathbf{p} = [\mathbf{v}_1 \times \mathbf{v}_2] \times [\mathbf{c}_1 \times \mathbf{c}_2], \tag{2.38}$$

where $\times$ denotes the cross product; and $\mathbf{p}$, $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{c}_1$ and $\mathbf{c}_2$ are the homogeneous representation of the intersection point, subject polygon vertices and clip polygon vertices (i.e., $\mathbf{p} = [x, y, 1]^T$, $\mathbf{v}_1 = [x_{\mathbf{v}_1}, y_{\mathbf{v}_1}, 1]^T$, $\mathbf{v}_2 = [x_{\mathbf{v}_2}, y_{\mathbf{v}_2}, 1]^T$, $\mathbf{c}_1 = [x_{\mathbf{c}_1}, y_{\mathbf{c}_1}, 1]^T$ and $\mathbf{c}_2 = [x_{\mathbf{c}_2}, y_{\mathbf{c}_2}, 1]^T$).

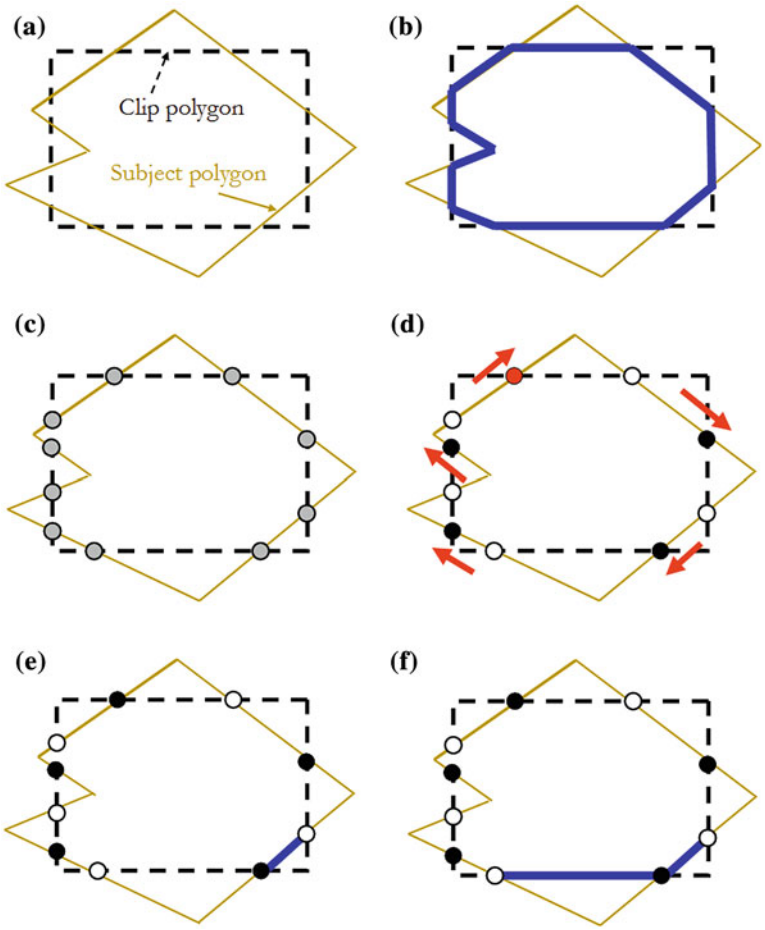

Fig. 2.15 Weiler-Atherton algorithm. **a** The original subject and clip polygons. **b** The subject polygon after clipping. **c** The intersection points between the subject and clip polygons are computed. **d** Points where subject polygon enters clipping window are represented by *white* circles. **e** Out-to-in: Record clipped point and follow subject polygon boundary in a *clockwise* direction. **f** In-to-out: Record clipped point and follow clip polygon boundary in a *clockwise* direction

When a new intersection is detected, a new false vertex is added to the circular lists of the vertices representing the polygons. Links are established to permit travelling between polygons.

2. *Walking* along the boundaries of the subject polygon in a clockwise direction, mark points where the subject polygon enters the clip polygon. These points are represented in Fig. 2.15d as white circles where the previous subject vertex is outside the clip polygon and/or the following subject vertex is inside the clip polygon. The points where the subject polygon leaves the clip polygon are represented as black circles; i.e., when the previous subject vertex is inside the clip

polygon and/or the following subject vertex is outside the clip polygon. (One may prefer to *walking* along the boundaries of the clip polygon and detect intersections where clip polygon enters and leaves the subject polygon.) Note that intersections alternate from entering to leaving as the number of intersections is always even.

3. There are two types of point pairs; out-to-in and in-to-out:
   a. Out-to-in pair (i.e., from white to black circles): At a white circle (i.e., entering point), follow the subject polygon vertices in its circular list until the next leaving intersection. Figure 2.15e shows this case as following the subject polygon boundary in a clockwise direction.
   b. In-to-out pair (i.e., from black to white circles): At a black circle (i.e., leaving point), follow the clip polygon vertices in its circular list until the next entering intersection. Figure 2.15f shows this case as following the clip polygon boundary in a clockwise direction.
4. Repeat Step 3 until there are no more pairs to process.

The result of the above operation is the subject polygon after clipping as shown in Fig. 2.15b. The process is summarized in Algorithm 2.15.

Algorithm 2.15 *Weiler-Atherton algorithm*

Input: Circular lists of vertices representing the subject and clip polygons
Output: Lists of vertices representing the clipped polygons

```
Get intersections between subject and clip polygons and add them to both lists.
Along the subject polygon, determine entering and leaving intersections.
while more vertices to process do
    If an entering or subject vertex is encountered, follow the subject circular list.
    If a leaving or clip vertex is encountered, follow the clip circular list.
    A loop of vertices is complete when arriving at the start vertex.
end while
```

end

Example 2.16: [*Weiler-Atherton clipping—circular lists and insertion of false vertices*]

Consider an irregular subject polygon to be clipped by a rectangular clip polygon as shown in Fig. 2.16. What are the steps that should be followed to populate circular lists for both the subject and clip polygons?

Solution 2.16: The steps are shown in Fig. 2.17.

1. To simplify the discussion, the vertices of both subject and clip polygons are numbered as shown in Fig. 2.17a and two circular lists are created; one for each polygon. The sequence of vertices of the subject polygon in a clockwise order is “0,” “1,” “2,” “3” and “4.” The sequence of vertices of the clip polygon in a clockwise order is “a,” “b,” “c” and “d.” The circular lists are shown in Fig. 2.17b.

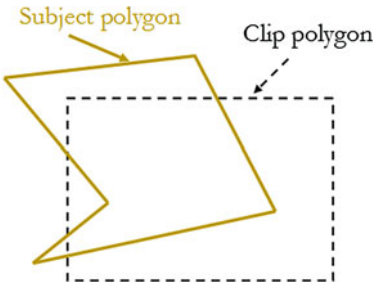

Fig. 2.16 An irregular subject polygon is to be clipped by a rectangular clip polygon

2. The first intersection marked "e" is detected (Fig. 2.17c) and a new false vertex is linked to both lists as shown in Fig. 2.17d. The subject sequence now becomes "0," "e," "1," "2," "3" and "4" and the clip sequence becomes "a," "e," "b," "c" and "d."
3. Along the clockwise direction, the next intersection marked "f" is detected (Fig. 2.17e) and a new false vertex is linked to both lists as shown in Fig. 2.17f. Notice that "f" is between "1" and "2" in the subject polygon and between "d" and "a" in the clip polygon. Thus, the subject sequence now becomes "0," "e," "1," "f," "2," "3" and "4" and the clip sequence becomes "a," "e," "b," "c," "d" and "f."
4. The following intersection marked "g" is detected (Fig. 2.17g) and a new false vertex is linked to both lists as shown in Fig. 2.17h. The subject sequence now becomes "0," "e," "1," "f," "2," "g," "3" and "4" and the clip sequence becomes "a," "e," "b," "c," "d," "f" and "g."
5. The next intersection marked "h" is detected (Fig. 2.17i) and a new false vertex is linked to both lists as shown in Fig. 2.17j. The subject sequence now becomes "0," "e," "1," "f," "2," "g," "3," "h" and "4" and the clip sequence becomes "a," "e," "b," "c," "d," "f", "g" and "h."

In Fig. 2.17, the false vertices "e," "f," "g" and "h" are shown in gray. Also, in order to make the distinction clear, the edges and links of the subject polygon are illustrated as solid lines while the edges and links of the clip polygon are illustrated as dashed lines. Notice that this whole process is represented by Line 1 in Algorithm 2.15. □

Example 2.17: [*Weiler-Atherton clipping—clipped loops*]
Building on Example 2.16, determine the clipped polygon parts (i.e., loops of vertices).

Solution 2.17: The steps are shown in Fig. 2.18.

1. The entering and leaving vertices are determined. In Fig. 2.18a, b, the entering vertices "e" and "g" are marked in white and the leaving vertices "f" and "h" are marked in black.
2. Starting at the *entering* vertex "e," the *subject* polygon border is followed as shown in Fig. 2.18c. This process is performed on the *subject* polygon circular list by following the link out of that vertex. This is shown in Fig. 2.18d.

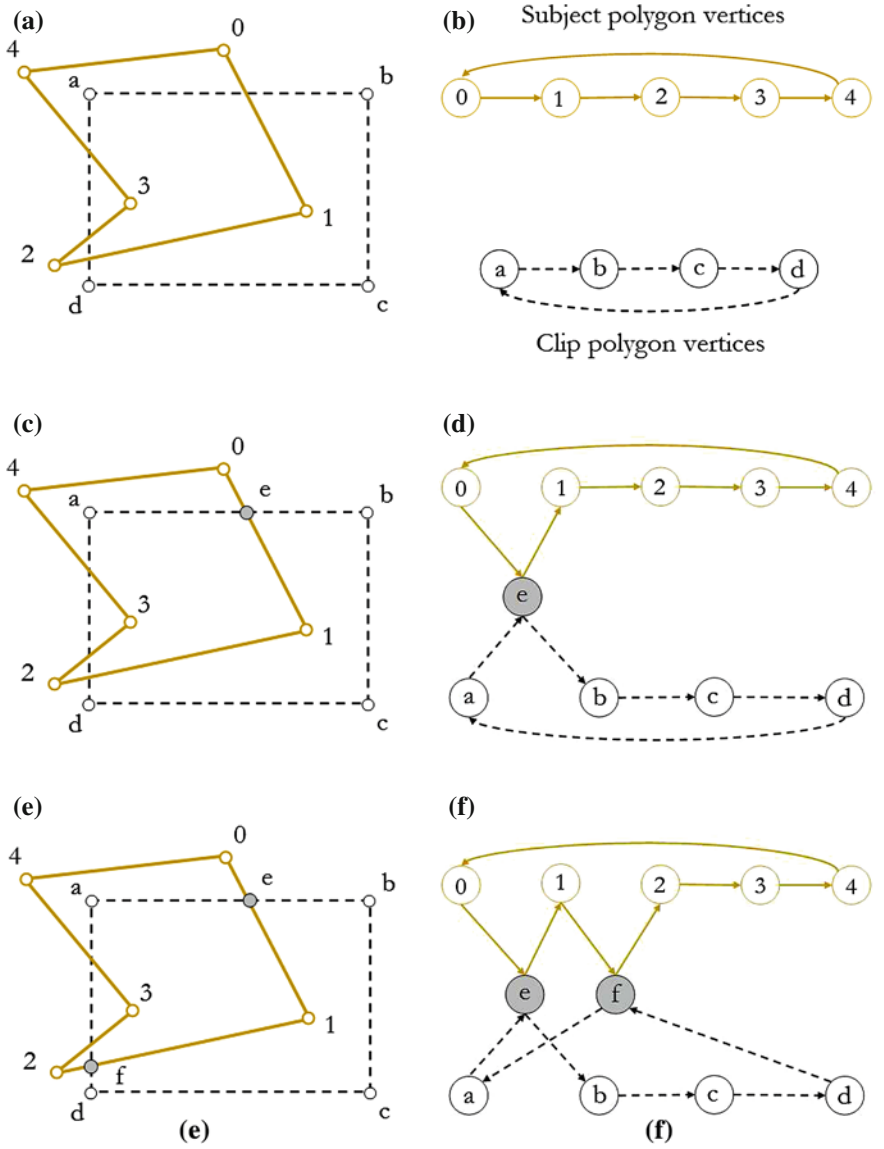

Fig. 2.17 Getting intersection points and establishing *circular lists* for subject and clip polygons. This process is represented by Line 1 in Algorithm 2.15

3. Continuing from the subject vertex "1," the *subject* polygon is followed as in the previous step. This is shown in Fig. 2.18e, f.
4. At the *leaving* vertex "f," the *clip* polygon border is followed as shown in Fig. 2.18g. This process is performed on the *clip* polygon circular list by following the link out of that vertex. This is shown in Fig. 2.18h.

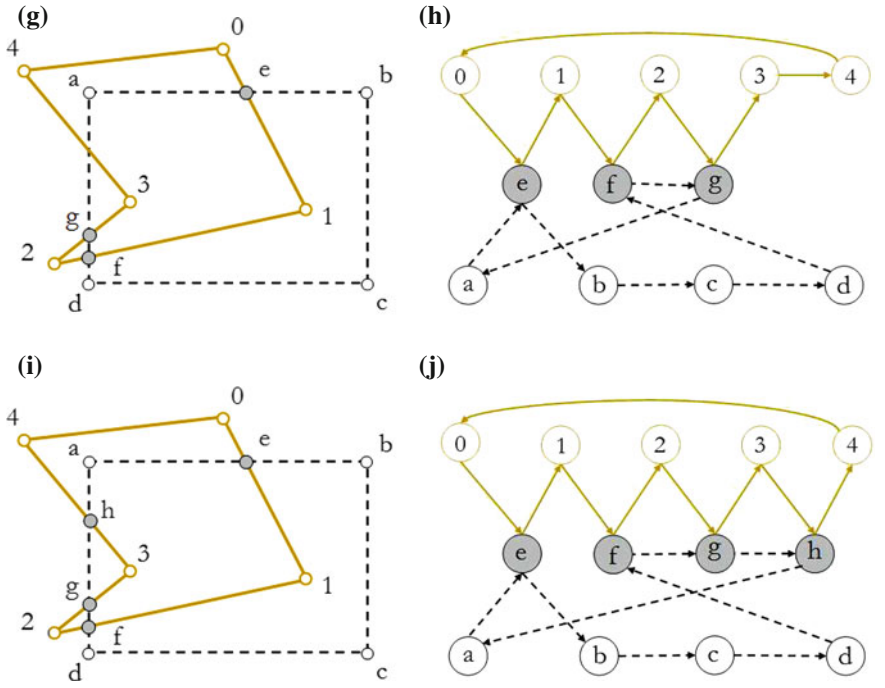

Fig. 2.17 (continued)

5. As in Step 2 above, from the *entering* vertex "g," the *subject* polygon border is followed as shown in Fig. 2.18i. This process is performed on the *subject* polygon circular list by following the link out of that vertex. This is shown in Fig. 2.18j.
6. As in Step 3, continuing from the subject vertex "3," the *subject* polygon is followed as shown in Fig. 2.18k, l.
7. As in Step 4, at the *leaving* vertex "h," the *clip* polygon is followed as shown in Fig. 2.18m, n.
8. The loop is complete by following the *clip* path from the *clip* vertex "a" to the *entering* vertex "e," which is the start vertex in Step 1.
9. The final loop (i.e., the clipped part of the polygon) now is "e," "1," "f," "g," "3," "h," "a" and "e."

All the above steps appear as thick edges/arrows in Fig. 2.18. □

2.6 Problems

Problem 2.1: [*Bresenham's line drawing algorithm*]

Bresenham's algorithm is used to draw a line segment from $[28, 8]^T$ to $[26, 14]^T$. Determine whether the following pixels are part of the displayed line: $[28, 9]^T$, $[27, 9]^T$, $[27, 10]^T$, $[28, 10]^T$, $[26, 12]^T$, $[27, 13]^T$, $[26, 13]^T$, $[27, 14]^T$, $[27, 11]^T$ and $[27, 12]^T$.

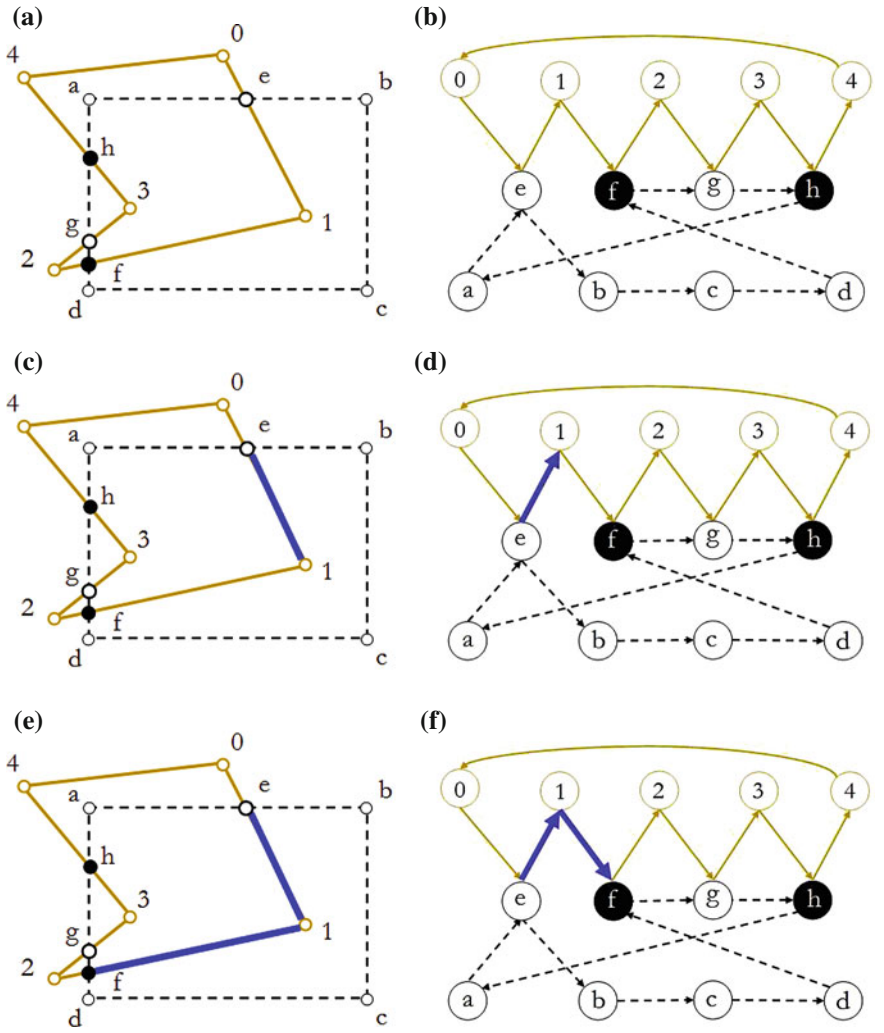

Fig. 2.18 Determining the clipped polygon parts. This process is represented by Line 2 through Line 7 in Algorithm 2.15

Problem 2.2: [*8-way symmetry algorithm*]

Figure 2.19a shows the upper left corner of a computer screen. The horizontal and vertical axes are shown with values representing pixel locations. Suppose that a curve spanning from $[5, 15]^T$ to $[15, 5]^T$ is drawn as two circle quadrants. The centers of the circles are shown as black dots. Use the 8-way symmetry algorithm to determine what pixels should constitute the curve.

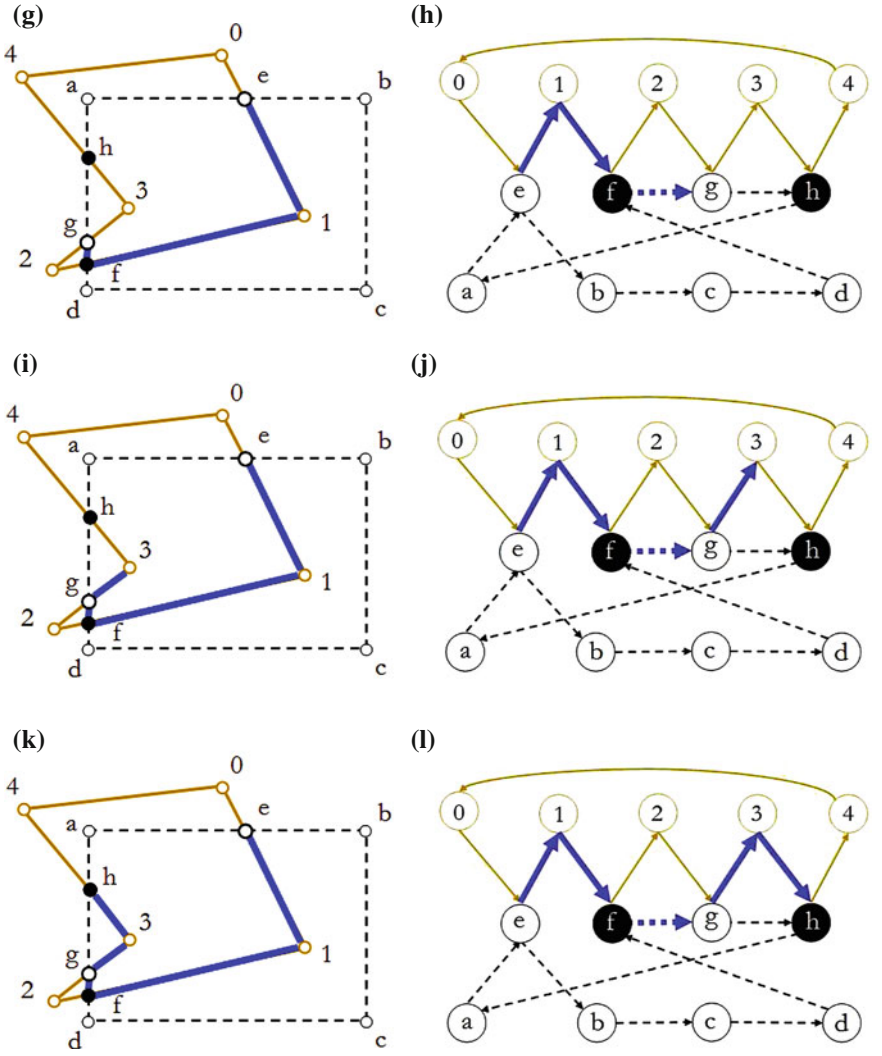

Fig. 2.18 (continued)

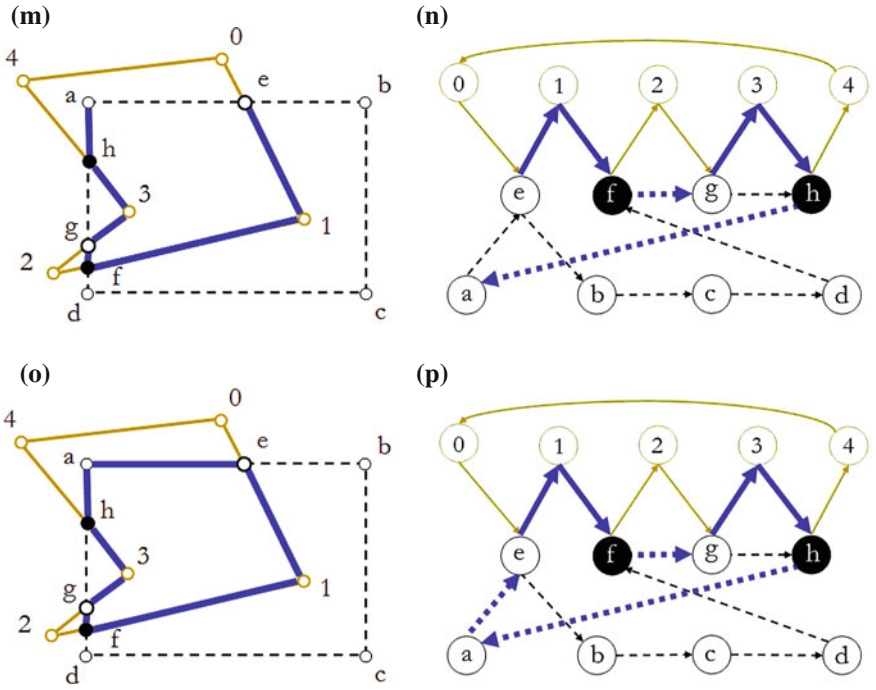

Fig. 2.18 (continued)

Problem 2.3: [*4-way symmetry algorithm*]
Re-solve Problem 2.2 using the 4-way symmetry algorithm.

Problem 2.4: [*Line and circle drawing algorithms*]
Figure 2.19b shows the upper left corner of a computer screen. The horizontal and vertical axes are shown with values representing pixel locations. Suppose that the curve shown consists of three segments two of them are line segments and the third is one-eighth of a circle whose center is shown as a black dot. Use a line drawing algorithm (of your choice) and a circle drawing algorithm (of your choice) to determine what pixels should contribute to the curve.

Problem 2.5: [*8-way symmetry algorithm*]
A circle having a radius of 5 pixels and centered at $[4, 6]^T$ is to be drawn on a computer screen. Use the 8-way symmetry algorithm to determine whether the following pixels are part of the displayed circle: $[9, 6]^T$, $[9, 5]^T$, $[9, 4]^T$, $[9, 3]^T$, $[4, 11]^T$, $[4, 1]^T$, $[7, 10]^T$, $[1, 2]^T$, $[3, 1]^T$ and $[1, 3]^T$.

Problem 2.6: [*4-way symmetry algorithm*]
Re-solve Problem 2.5 using the 4-way symmetry algorithm.

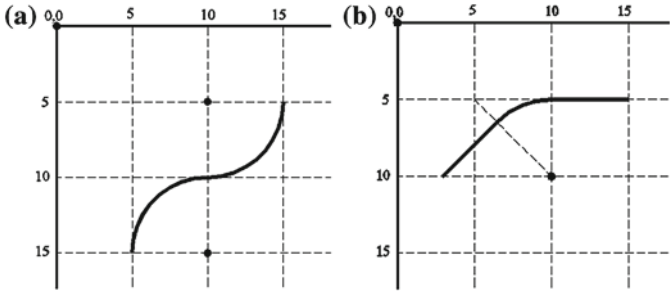

Fig. 2.19 The *upper left corner* of a computer screen

Problem 2.7: [*8-way symmetry algorithm*]

Modify Algorithm 2.10 to use a "for" loop that goes from $x = 1$ to $x = \left\lfloor \frac{r}{\sqrt{2}} \right\rfloor$ where $\lfloor . \rfloor$ is the floor operation.

Problem 2.8: [*Midpoint algorithm for circles*]

Modify Algorithm 2.11 so that the condition of the "while" loop is

$$\left(x < \left\lfloor \frac{r}{\sqrt{2}} \right\rfloor \right),$$

where x is the horizontal counter; and r is the radius of the circle.

Problem 2.9: [*Midpoint and 8-way symmetry algorithms for circles*]

In order to draw a circle, Algorithm 2.12 starts from the point at the bottom of the circle while Algorithm 2.10 starts from the point at the top. Modify Algorithm 2.12 to start from the top.

Problem 2.10: [*Polygon concavity/convexity*]

Consider the polygon defined by the vertices $[3, 3]^T$, $[6, 3]^T$, $[8, 2]^T$ and $[6, 6]^T$. Determine if this polygon is convex or concave.

Problem 2.11: [*Polygon concavity/convexity*]

Given a 2D polygon specified by the vertices $[-3, 0]^T$, $[3, -1]^T$, $[1, 0]^T$ and $[4, 2]^T$, test whether it is convex or concave.

Problem 2.12: [*Cohen-Sutherland clipping algorithm—left-handed coordinate system*]

Suppose that a clip window is indicated by its upper left and lower right corners $[100, 50]^T$ and $[300, 200]^T$. Test whether each of the following line segments can be trivially accepted in the window, trivially rejected or needs further processing:

1. A line extending from $[171, 88]^T$ to $[233, 171]^T$.
2. A line extending from $[150, 101]^T$ to $[233, 39]^T$.
3. A line extending from $[52, 15]^T$ to $[98, 45]^T$.

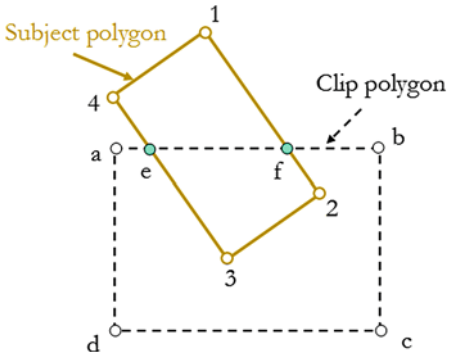

Fig. 2.20 A subject polygon to be clipped by a clip polygon

Problem 2.13: [*Cohen-Sutherland clipping algorithm—left-handed coordinate system*]

Use the Cohen-Sutherland algorithm to determine what lines or portions of lines are preserved and kept in Problem 2.12.

Problem 2.14: [*Line clipping*]

Given a clip rectangle spanning from $[3, 3]^T$ to $[15, 12]^T$, use the brute-force algorithm discussed in this chapter to determine what lines or parts of lines are preserved and kept among the following:

1. A line extending from $[15, 12]^T$ to $[17, 10]^T$.
2. A line extending from $[16, 9]^T$ to $[10, 10]^T$.
3. A line extending from $[5, 5]^T$ to $[9, 8]^T$.
4. A line extending from $[12, 14]^T$ to $[7, 2]^T$.

Problem 2.15: [*Line intersection—parametric equation*]

Using the parametric equation of a line, determine the intersection point between the two line segments $\overline{\dot{\mathbf{p}}_1\dot{\mathbf{p}}_2}$ and $\overline{\dot{\mathbf{p}}_3\dot{\mathbf{p}}_4}$ where $\dot{\mathbf{p}}_1 = [1, 1]^T$, $\dot{\mathbf{p}}_2 = [9, 9]^T$, $\dot{\mathbf{p}}_3 = [9, 1]^T$ and $\dot{\mathbf{p}}_4 = [1, 9]^T$.

Problem 2.16: [*Line intersection—homogeneous coordinates*]

Re-solve Problem 2.15 using homogeneous coordinates (see Sect. B.7).

Problem 2.17: [*Line intersection*]

Use three different methods to determine the intersection point between the two line segments $\overline{\dot{\mathbf{p}}_1\dot{\mathbf{p}}_2}$ and $\overline{\dot{\mathbf{p}}_3\dot{\mathbf{p}}_4}$ where $\dot{\mathbf{p}}_1 = [1, 1]^T$, $\dot{\mathbf{p}}_2 = [3, 3]^T$, $\dot{\mathbf{p}}_3 = [1, 3]^T$ and $\dot{\mathbf{p}}_4 = [3, 1]^T$.

Problem 2.18: [*Weiler-Atherton clipping—circular lists and insertion of false vertices*]

Consider the subject and clip polygons shown in Fig. 2.20. What are the steps that should be followed to populate circular lists for both the subject and clip polygons?

Problem 2.19: [*Weiler-Atherton clipping—clipped loops*]

Determine the clipped polygon parts (i.e., loops of vertices) in Problem 2.18.

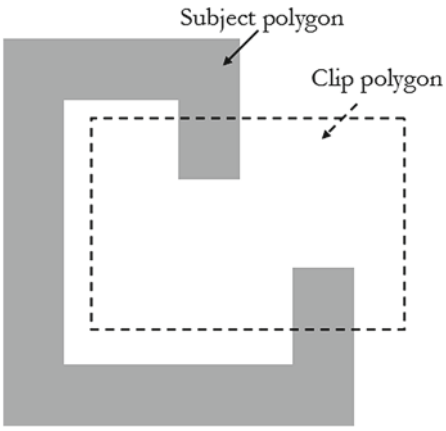

Fig. 2.21 A subject polygon to be clipped by a clip polygon

Problem 2.20: [*Weiler-Atherton clipping—circular lists and insertion of false vertices*]

Consider the subject and clip polygons shown in Fig. 2.21. What are the steps that should be followed to populate circular lists for both the subject and clip polygons?

Problem 2.21: [*Weiler-Atherton clipping—clipped loops*]

Determine the clipped polygon parts (i.e., loops of vertices) in Problem 2.20.

References

Aken, J.V. 1984. An efficient ellipse-drawing algorithm. *IEEE Computer Graphics and Applications* 4(9): 24–35.

Bresenham, J.E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal* 4(1): 25–30.

Cyrus, M., and J. Beck. 1978. Generalized two- and three-dimensional clipping. *Computers and Graphics* 3(1): 23–28.

Foley, J.D., A. van Dam, S.K. Feiner, and J. Hughes. 1995. *Computer Graphics: Principles and Practice in C*, 2nd ed. The systems programming series. Addison-Wesley, Reading, MA.

Liang, Y.-D., and B.A. Barsky. 1984. A new concept and method for line clipping. *ACM Transactions on Graphics (TOG)* 3(1): 1–22.

Maillot, P.-G. 1992. A new, fast method for 2d polygon clipping: analysis and software implementation. *ACM Transactions on Graphics (TOG)* 11(3): 276–290.

Nicholl, T.M., D.T. Lee, and R.A. Nicholl. 1987. An efficient new algorithm for 2-d line clipping: its development and analysis. *ACM SIGGRAPH Computer Graphics* 21(4): 253–262.

Pitteway, M. 1967. Algorithm for drawing ellipses or hyperbolae with a digital plotter. *Computer Journal* 10(3): 282–289.

Sutherland, I.E., and G.W. Hodgman. 1974. Reentrant polygon clipping. *Communications of the ACM* 17(1): 32–42.

Van Aken, J., and M. Novak. 1985. Curve-drawing algorithms for raster displays. *ACM Transactions on Graphics* 4(2): 147–169.

Weiler, K., and P. Atherton. 1977. Hidden surface removal using polygon area sorting. *ACM SIGGRAPH Computer Graphics* 11(2): 214–222.

http://www.springer.com/978-3-319-05136-9

Digital Media
A Problem-solving Approach for Computer Graphics
Elias, R.
2014, XXXVI, 686 p. 314 illus., Softcover
ISBN: 978-3-319-05136-9