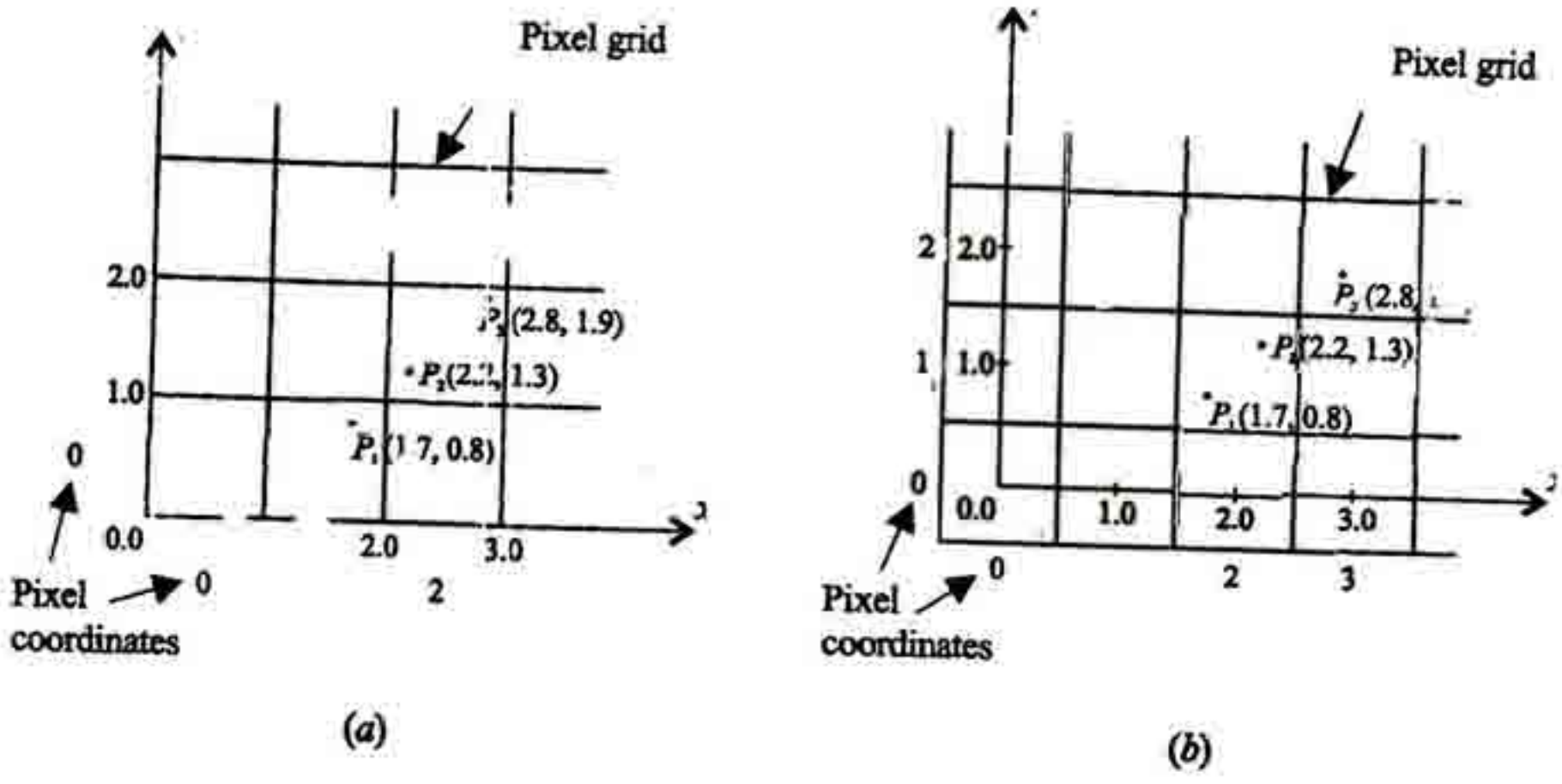

Fig. 3-1 Scan-converting points.

We will assume, in the following sections, that this second approach to coordinate system alignment is used. Thus all pixels are centered at the integer values of a continuous coordinate system where abstract graphical primitives are defined.

3.2 SCAN-CONVERTING A LINE

A line in computer graphics typically refers to a line segment, which is a portion of a straight line that extends indefinitely in opposite directions. It is defined by its two endpoints and the line equation $y = mx + b$, where m is called the slope and b the y intercept of the line. In Fig. 3-2 the two endpoints are described by $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. The line equation describes the coordinates of all the points that lie between the two endpoints.

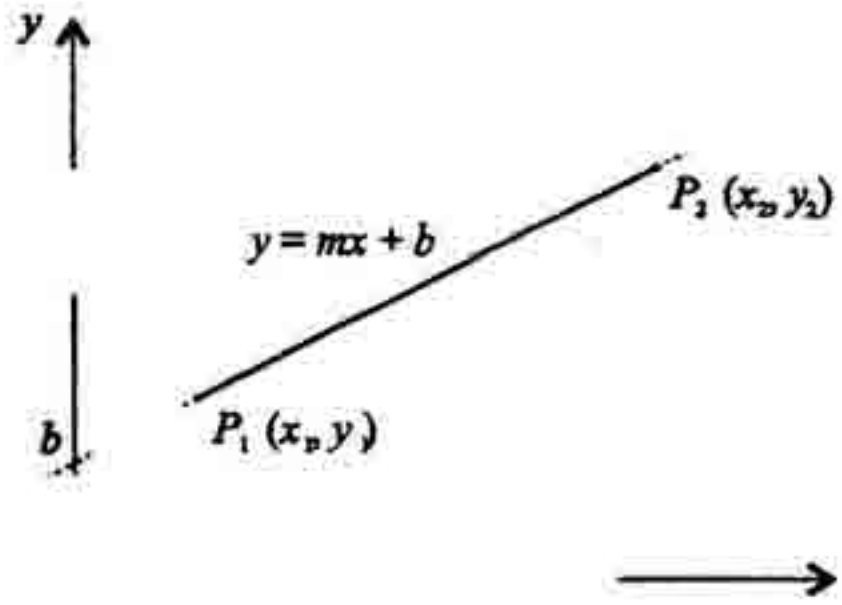

Fig. 3-2 Defining a line.

A note of caution: this slope–intercept equation is not suitable for vertical lines. Horizontal, vertical, and diagonal ($|m| = 1$) lines can, and often should, be handled as special cases without going through the following scan-conversion algorithms. These commonly used lines can be mapped to the image space in a straightforward fashion for high execution efficiency.

Direct Use of the Line Equation

A simple approach to scan-converting a line is to first scan-convert P_1 and P_2 to pixel coordinates (x_1', y_1') and (x_2', y_2'), respectively; then set $m = (y_2' - y_1')/(x_2' - x_1')$ and $b = y_1' - mx_1'$. If $|m| \leq 1$, then for

every integer value of x between and excluding x'_1 and x'_2, calculate the corresponding value of y using the equation and scan-convert (x, y). If $|m| > 1$, then for every integer value of y between and excluding y'_1 and y'_2, calculate the corresponding value of x using the equation and scan-convert (x, y).

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

DDA Algorithm

The digital differential analyzer (DDA) algorithm is an incremental scan-conversion method. Such an approach is characterized by performing calculations at each step using results from the preceding step. Suppose at step i we have calculated (x_i, y_i) to be a point on the line. Since the next point (x_{i+1}, y_{i+1}) should satisfy $\Delta y/\Delta x = m$ where $\Delta y = y_{i+1} - y_i$ and $\Delta x = x_{i+1} - x_i$, we have

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

or

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

These formulas are used in the DDA algorithm as follows. When $|m| \leq 1$, we start with $x = x'_1$ (assuming that $x'_1 < x'_2$) and $y = y'_1$, and set $\Delta x = 1$ (i.e., unit increment in the x direction). The y coordinate of each successive point on the line is calculated using $y_{i+1} = y_i + m$. When $|m| > 1$, we start with $x = x'_1$ and $y = y'_1$ (assuming that $y'_1 < y'_2$), and set $\Delta y = 1$ (i.e., unit increment in the y direction). The x coordinate of each successive point on the line is calculated using $x_{i+1} = x_i + 1/m$. This process continues until x reaches x'_2 (for the $|m| \leq 1$ case) or y reaches y'_2 (for the $|m| > 1$ case) and all points found are scan-converted to pixel coordinates.

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

Bresenham's Line Algorithm

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

The method works as follows. Assume that we want to scan-convert the line shown in Fig. 3-3 where $0 < m < 1$. We start with pixel $P'_1(x'_1, y'_1)$, then select subsequent pixels as we work our way to the right, one pixel position at a time in the horizontal direction towards $P'_2(x'_2, y'_2)$. Once a pixel is chosen at any step, the next pixel is either the one to its right (which constitutes a lower bound for the line) or the one to its right and up (which constitutes an upper bound for the line) due to the limit on m. The line is best approximated by those pixels that fall the least distance from its true path between P'_1 and P'_2.

Using the notation of Fig. 3-3, the coordinates of the last chosen pixel upon entering step i are (x_i, y_i). Our task is to choose the next one between the bottom pixel S and the top pixel T. If S is chosen, we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$. If T is chosen, we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i + 1$. The actual y coordinate of the line at $x = x_{i+1}$ is $y = mx_{i+1} + b = m(x_i + 1) + b$. The distance from S to the actual line in the y direction is $s = y - y_i$. The distance from T to the actual line in the y direction is $t = (y_i + 1) - y$.

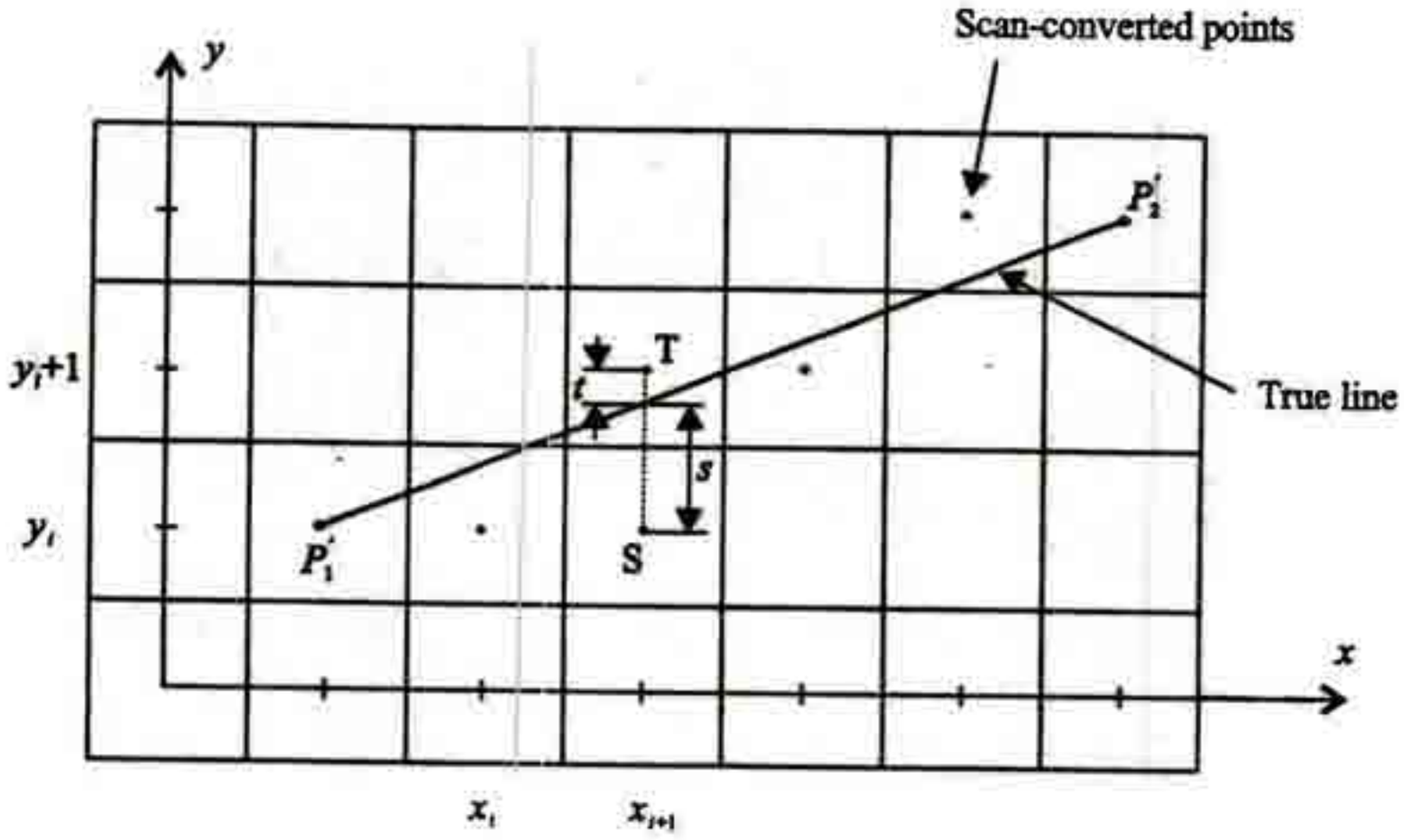

Fig. 3-3 Scan-converting a line.

Now consider the difference between these two distance values: $s - t$. When $s - t$ is less than zero, we have $s < t$ and the closest pixel is S. Conversely, when $s - t$ is greater than zero, we have $s > t$ and the closest pixel is T. We also choose T when $s - t$ is equal to zero. This difference is

$$\begin{aligned} s - t &= (y - y_i) - [(y_i + 1) - y] \\ &= 2y - 2y_i - 1 = 2m(x_i + 1) + 2b - 2y_i - 1 \end{aligned}$$

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

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

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

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

Then

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

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

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

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

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

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

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

Hence we have

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

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

$$\begin{aligned} d_1 &= \Delta x[2m(x_1 + 1) + 2b - 2y_1 - 1] \\ &= \Delta x[2(mx_1 + b - y_1) + 2m - 1] \end{aligned}$$

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

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

In summary, Bresenham's algorithm for scan-converting a line from $P_1'(x_1', y_1')$ to $P_2'(x_2', y_2')$ with $x_1' < x_2'$ and $0 < m < 1$ can be stated as follows:

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

Here we first initialize decision variable d and set pixel P_1'. During each iteration of the while loop, we increment x to the next horizontal position, then use the current value of d to select the bottom or top (increment y) pixel and update d, and at the end set the chosen pixel.

As for lines that have other m values we can make use of the fact that they can be mirrored either horizontally, vertically, or diagonally into this 0° to 45° angle range. For example, a line from (x_1', y_1') to (x_2', y_2') with $-1 < m < 0$ has a horizontally mirrored counterpart from $(x_1', -y_1')$ to $(x_2', -y_2')$ with $0 < m < 1$. We can simply use the algorithm to scan-convert this counterpart, but negate the y coordinate at the end of each iteration to set the right pixel for the line. For a line whose slope is in the 45° to 90° range, we can obtain its mirrored counterpart by exchanging the x and y coordinates of its endpoints. We can then scan-convert this counterpart but we must exchange x and y in the call to setPixel.