
CHAPTER ONE

GEOMETRY AND LINE GENERATION

INTRODUCTION

Perhaps our age will be known as the Information Revolution or the Computer Revolution, for we are witnessing a remarkable growth and development of computer technology and applications. The computer is an information processing machine, a tool for storing, manipulating, and correlating data. We are able to generate or collect and process information on a scope never before possible. This information can help us make decisions, understand our world, and control its operation. But as the volume of information increases, a problem arises. How can this information be efficiently and effectively transferred between machine and human? The machine can easily generate tables of numbers hundreds of pages long. But such a printout may be worthless if the human reader does not have the time to understand it. Computer graphics strikes directly at this problem. It is a study of techniques to improve communication between human and machine. A graph may replace that huge table of numbers and allow the reader to note the relevant patterns and characteristics at a glance. Giving the computer the ability to express its data in pictorial form can greatly increase its ability to provide information to the human user. This is a passive form of graphics, but communication can also be a two-way process. It may be convenient and appropriate to input graphical information to the computer. Thus there are both graphical input and graphical output devices. It is often desirable to have the input from the user alter the output presented by the machine. A dialogue can be established through the graphics medium. This is termed *interactive computer graphics* because the user interacts with the machine. Computer graphics allows communication through pictures, charts, and diagrams. It offers

a vital alternative to the typewriter's string of symbols. The old adage "A picture is worth a thousand words" is certainly true. Through computer graphics we can pilot spaceships; walk through buildings which have yet to be built; and watch bridges collapse, stars being born, and the rotations of atoms. There are many applications for computer graphics. Management information may be displayed as charts and diagrams. Scientific theories and models may be described in pictorial form. (See Plates 1 through 4.) In computer-aided design we can display an aircraft wing, a highway layout, a printed circuit board, a building "blueprint," or a machine part. (See Plate 15.) Maps can be created for all kinds of geographic information. Diagrams and simulations can enrich classroom instruction. The computer has become a new tool for the artist and animator. (See Plates 5 through 14.) And in video games, computer graphics provides a new form of entertainment.

Over the years many graphics display devices have been developed. There are also many software packages and graphics languages. The problem with such diversity is that it makes it difficult to transfer a graphics program from one installation to another. In the late 1970s, the Graphics Standards Planning Committee of the Association for Computing Machinery developed a proposal for a standard graphics system called the CORE system. This system provided a standardized set of commands to control the construction and display of graphic images. The commands were independent of the device used to create or to display the image and independent of the language in which the graphics program was written. The CORE system defined basic graphics primitives from which more complex or special-purpose graphics routines could be built. The idea was that a program written for the CORE system could be run on any installation using that system. The CORE system contained mechanisms for describing and displaying both two-dimensional and three-dimensional structures. However, it was developed just before the reduction in the cost of computer memory made possible economical raster displays (which allow solid and colored areas to be drawn). It therefore lacked the primitives for describing areas and could only create line drawings. Extensions were soon proposed to provide the CORE system with raster imaging primitives.

A second standard called the graphics kernel system (GKS) was developed in Europe, and it has been steadily gaining in popularity. The GKS system was heavily influenced by CORE, and the minimal GKS implementation is essentially identical to CORE's two-dimensional subset. The GKS standard did contain primitives for imaging areas and colors, but it did not contain the constructs for three-dimensional objects. It introduced the concept of a workstation, which allowed a single graphics program to control several graphics terminals.

Another graphics standard is the programmer's hierarchical interactive graphics standard (PHIGS). It takes input and output functions and viewing model from CORE and GKS, but it is a programmer's toolbox, intended for programming graphics applications. It contains enhanced graphics program structuring features. Two other graphics standards are the computer graphics metafile (CGM) and the computer graphics interface (CGI). The CGM is a file format for picture information that allows device-independent capture, storage, and transfer. The CGI is a companion standard which provides a procedural interface for the CGM primitives.

In this book we are going to present the algorithms for constructing a graphics system which have the flavor of the CORE and GKS standards and, in some areas, go beyond them.

We begin our discussion of computer graphics with the fundamental question of how to locate and display points and line segments. There are several hardware devices (graphics terminals) which may be used to display computer-generated images. Some of these will be discussed in Chapter 2. Before we talk about the devices which display points, we shall review the basic geometry which underlies all of our techniques. We shall consider what points and lines are and how we can specify and manipulate them. We conclude this chapter with a discussion of how the mathematical description of these fundamental geometric building blocks can be implemented on an actual display device. Algorithms are presented for carrying out such an implementation for a line printer or common cathode ray tube (CRT) display. These algorithms will allow us (if needed) to use the line printer or CRT as a somewhat crude, but effective, graphics display device for demonstrating the graphics principles described in the rest of the text.

LINES

We can specify a point (a position in a plane) with an ordered pair of numbers (x, y) , where x is the horizontal distance from the origin and y is the vertical distance. Two points will specify a line. Lines are described by equations such that if a point (x, y) satisfies the equations, then the point is on the line. If the two points used to specify a line are (x_1, y_1) and (x_2, y_2) , then an equation for the line is given by

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1} \quad (1.1)$$

This says that the slope between any point on the line and (x_1, y_1) is the same as the slope between (x_2, y_2) and (x_1, y_1) .

There are many equivalent forms for this equation. Multiplying by the denominators gives the form

$$(x - x_1)(y_2 - y_1) = (y - y_1)(x_2 - x_1) \quad (1.2)$$

A little more algebra solving for y gives

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1 \quad (1.3)$$

or

$$\boxed{y = mx + b} \quad (1.4)$$

where

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

and

$$b = y_1 - mx_1$$

This is called the slope-intercept form of the line. The slope m is the change in height divided by the change in width for two points on the line (the rise over the run). The intercept b is the height at which the line crosses the y axis. This can be seen by noting that the point $(0, b)$ satisfies the equation of the line.

A different form of the line equation, called the general form, may be found by multiplying out the factors in Equation 1.2 and collecting them on one side of the equal sign.

$$(y_2 - y_1)x - (x_2 - x_1)y + x_2y_1 - x_1y_2 = 0 \quad (1.5)$$

or

$$\boxed{rx + sy + t = 0} \quad (1.6)$$

where possible values for r , s , and t are

$$r = (y_2 - y_1)$$

$$s = -(x_2 - x_1)$$

$$t = x_2y_1 - x_1y_2$$

We say *possible* values because we see that multiplying r , s , and t by any common factor will produce a new set of r' , s' , and t' values which will still satisfy Equation 1.6 and, therefore, also describe the same line. The values for r , s , and t are sometimes chosen so that

$$\boxed{r^2 + s^2 = 1} \quad (1.7)$$

Comparing Equations 1.4 and 1.6 we see that

$$\boxed{m = -\frac{r}{s}}$$

and

$$\boxed{b = -\frac{t}{s}} \quad (1.8)$$

Can we determine where two lines will cross? Yes, it is fairly easy to determine where two lines will cross. By the two lines crossing we mean that they share some point in common. That point will satisfy both of the equations for the two lines. The problem is to find this point. Suppose we give the equations for the two lines in their slope-intercept form:

$$\text{line 1: } y = m_1x + b_1$$

$$\text{line 2: } y = m_2x + b_2$$

(1.9)

Now if there is some point (x_i, y_i) shared by both lines, then

$$y_i = m_1x_i + b_1 \quad \text{and} \quad y_i = m_2x_i + b_2$$

(1.10)

will both be true. Equating over y_i gives

$$m_1x_i + b_1 = m_2x_i + b_2$$

(1.11)

Solving for x_i yields

$$x_i = \frac{b_2 - b_1}{m_1 - m_2} \quad (1.12)$$

Substituting this into the equation for either line 1 or line 2 gives

$$y_i = \frac{b_2 m_1 - b_1 m_2}{m_1 - m_2} \quad (1.13)$$

Therefore, the point

$$\left(\frac{b_2 - b_1}{m_1 - m_2}, \frac{b_2 m_1 - b_1 m_2}{m_1 - m_2} \right) \quad (1.14)$$

is the intersection point. Note that two parallel lines will have the same slope. Since such lines will not intersect, it is not at all surprising that the above expression results in a division by zero. When no point exists, we cannot solve for it.

If the Equation 1.6 form is used to describe the lines, then similar algebra yields an intersection point which is given by

$$\left(\frac{s_1 t_2 - s_2 t_1}{s_2 r_1 - s_1 r_2}, \frac{t_1 r_2 - t_2 r_1}{s_2 r_1 - s_1 r_2} \right) \quad (1.15)$$

LINE SEGMENTS

What are *line segments*? Our equations for lines specify all points in a given direction. The lines extend forever both forward and backward. This is not exactly what we need for graphics. We would like to display only pieces of lines. Let's consider only those points on a line which lie between two endpoints p_1 and p_2 . (See Figure 1-1.)

This is called a line segment. A line segment may be specified by its two endpoints. From these endpoints we can determine the equation of the line. From this equation and the endpoints we can decide if any point is or is not on the segment. If the endpoints are $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ and these yield equation $y = mx + b$ (or $rx + sy + t = 0$), then another point $p_3 = (x_3, y_3)$ lies on the segment if

1. $y_3 = mx_3 + b$ (or $rx_3 + sy_3 + t = 0$)
2. $\min(x_1, x_2) \leq x_3 \leq \max(x_1, x_2)$
3. $\min(y_1, y_2) \leq y_3 \leq \max(y_1, y_2)$

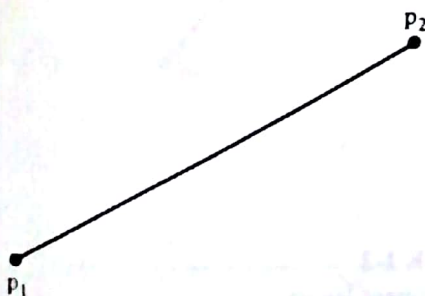


FIGURE 1-1
A line segment.

Here the notation $\min(x_1, x_2)$ means the smallest of x_1 and x_2 and $\max(x_1, x_2)$ means the largest of the two numbers.

There is one more useful form of the line equation called the parametric form because the x and y values on the line are given in terms of a parameter u . This form is convenient when considering line segments because we can construct it such that line-segment points correspond to values of the parameter between 0 and 1. Suppose we want the line segment between (x_1, y_1) and (x_2, y_2) . We wish the x coordinate to go uniformly from x_1 to x_2 . This may be expressed by the equation

$$x = x_1 + (x_2 - x_1)u \quad (1.16)$$

When u is 0, x is x_1 . As u increases to 1, x moves uniformly to x_2 . But for a line segment, we must have the y coordinate moving from y_1 to y_2 at the same time as x changes.

$$y = y_1 + (y_2 - y_1)u \quad (1.17)$$

The two equations together describe a straight line. This can be shown by equating over u . A little algebra recovers the line equation. Note that with this form, we can generate the point on the line segment by letting u sweep from 0 to 1; also, given the parameter value for a point on the line, we can easily test to see if the point lies within the segment boundaries.

How long is a line segment? If we are given the two endpoints of a line segment p_1 and p_2 we can determine its length L . Construct a right triangle p_1p_2A by attaching a vertical line to p_2 and a horizontal line to p_1 . (See Figure 1-2.)

The Pythagorean theorem states that the square of the length of the hypotenuse (p_1p_2) is equal to the sum of the squares of the lengths of the two adjacent sides (p_1A and p_2A). If we call the coordinates of p_1 (x_1, y_1) , and the coordinates of p_2 (x_2, y_2) , then A will have coordinates (x_2, y_1) , and the length of the segment L will be given by

$$L^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 \quad (1.18)$$

so

$$L = [(x_2 - x_1)^2 + (y_2 - y_1)^2]^{1/2} \quad (1.19)$$

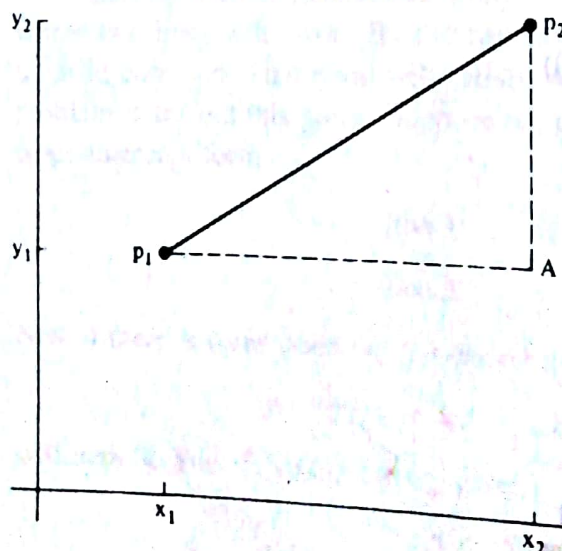


FIGURE 1-2
Line segment length.

What is the *midpoint* of a line segment? The midpoint of a line segment is often useful and easy to calculate. The point halfway between the endpoints of a segment will have an x coordinate halfway between the x coordinates of the endpoints, and a y coordinate halfway between the y coordinates of the endpoints. (See Figure 1-3.) Therefore, the midpoint is

$$(x_m, y_m) = \left[\frac{(x_1 + x_2)}{2}, \frac{(y_1 + y_2)}{2} \right] \quad (1.20)$$

PERPENDICULAR LINES

Can we tell if two lines are *perpendicular*? We can determine if two lines are perpendicular by examining their slopes. Suppose we have two lines

$$y = m_1x + b_1$$

and

$$y = m_2x + b_2$$

(1.21)

If the first line is perpendicular to the second, then a line parallel to the first (that is, a line with the same slope) will also be perpendicular to the second. For example, $y = m_1x$ should be perpendicular to $y = m_2x + b_2$. The same argument applies to the second line: $y = m_2x$ will be perpendicular to $y = m_1x$. Now these are two lines which intersect at the origin. (See Figure 1-4.)

Consider a point (x_1, y_1) on the line $y = m_1x$ so that $y_1 = m_1x_1$ and a point (x_2, y_2) on $y = m_2x$ so that $y_2 = m_2x_2$. The three points (x_1, y_1) , (x_2, y_2) , and $(0, 0)$ form a triangle. If the two lines are perpendicular, they will form a right triangle and the Pythagorean theorem will apply. The distance between $(0, 0)$ and (x_1, y_1) squared plus the distance between $(0, 0)$ and (x_2, y_2) squared will equal the square of the hypotenuse between (x_1, y_1) and (x_2, y_2) . The distance formula gives

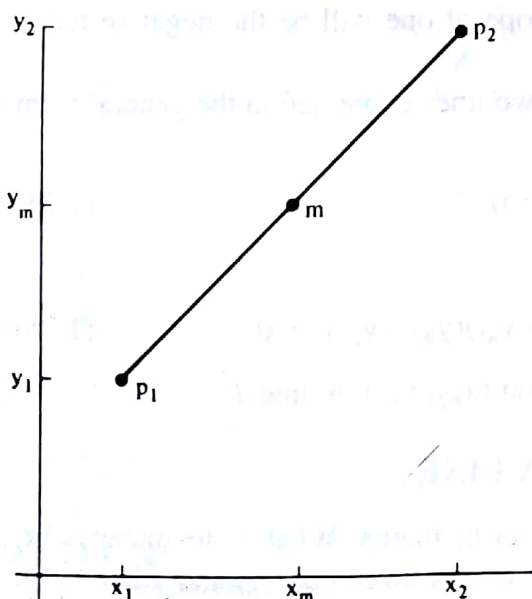


FIGURE 1-3

The midpoint of a line segment.

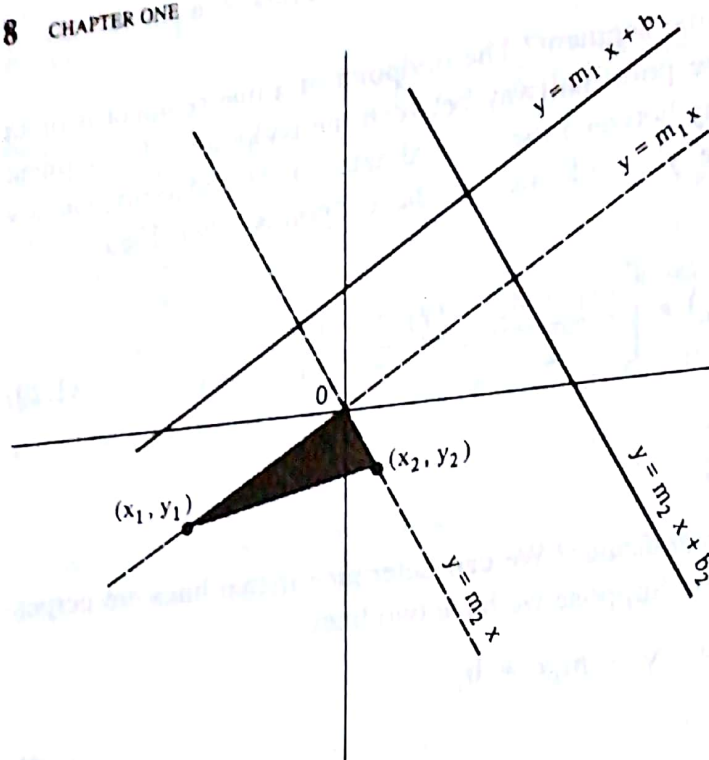


FIGURE 1-4
Construction for test for perpendicular lines.

$$x_1^2 + y_1^2 + x_2^2 + y_2^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2 \quad (1.22)$$

Simplifying gives

$$0 = -2y_1y_2 - 2x_1x_2 \quad (1.23)$$

or

$$\frac{y_1}{x_1} = -\frac{x_2}{y_2}$$

but since $y_1 = m_1x_1$ and $y_2 = m_2x_2$, we have

$$m_1 = -\frac{1}{m_2} \quad (1.24)$$

Therefore, if two lines are perpendicular, the slope of one will be the negative reciprocal of the slope of the other.

From Equations 1.8 and 1.22 we see that two lines expressed in the general form for a line are perpendicular if

$$r_1r_2 + s_1s_2 = 0 \quad (1.25)$$

It also follows that

$$(x_{b1} - x_{a1})(x_{b2} - x_{a2}) + (y_{b1} - y_{a1})(y_{b2} - y_{a2}) = 0 \quad (1.26)$$

for (x_{a1}, y_{a1}) and (x_{b1}, y_{b1}) on line 1, (x_{a2}, y_{a2}) and (x_{b2}, y_{b2}) on line 2.

DISTANCE BETWEEN A POINT AND A LINE

We shall derive one more formula in our review of geometry. What is the distance between a point and a line in a plane? Suppose we have a point (x_0, y_0) and a line $rx +$

$sy + t = 0$, where r , s , and t were chosen to satisfy Equation 1.7. We can find a line which is perpendicular to the given line and contains the given point. It is

$$-sx + ry + (sx_0 \pm ry_0) = 0 \quad (1.27)$$

We can determine the intersection point of the original line and this perpendicular. (See Figure 1-5.) Using Expression 1.15 we find it to be

$$(s(sx_0 - ry_0) - rt, -st - r(sx_0 - ry_0)) \quad (1.28)$$

Now we can use the distance formula (Equation 1.19) to determine the distance between the point (x_0, y_0) and this intersection point. This is what we mean by the distance between the point and the line.

$$L = (\{x_0 - [s(sx_0 - ry_0) - rt]\}^2 + \{y_0 - [-st - r(sx_0 - ry_0)]\}^2)^{1/2} \quad (1.29)$$

This will reduce down to

$$L = |rx_0 + sy_0 + t| \quad (1.30)$$

Notice that this is just the magnitude of the value obtained by substituting the coordinates of the point into the expression for the line. When the expression is zero, the point is on the line, while other values give the distance of the point from the line. Remember that this only works because we have chosen values for r , s , and t which

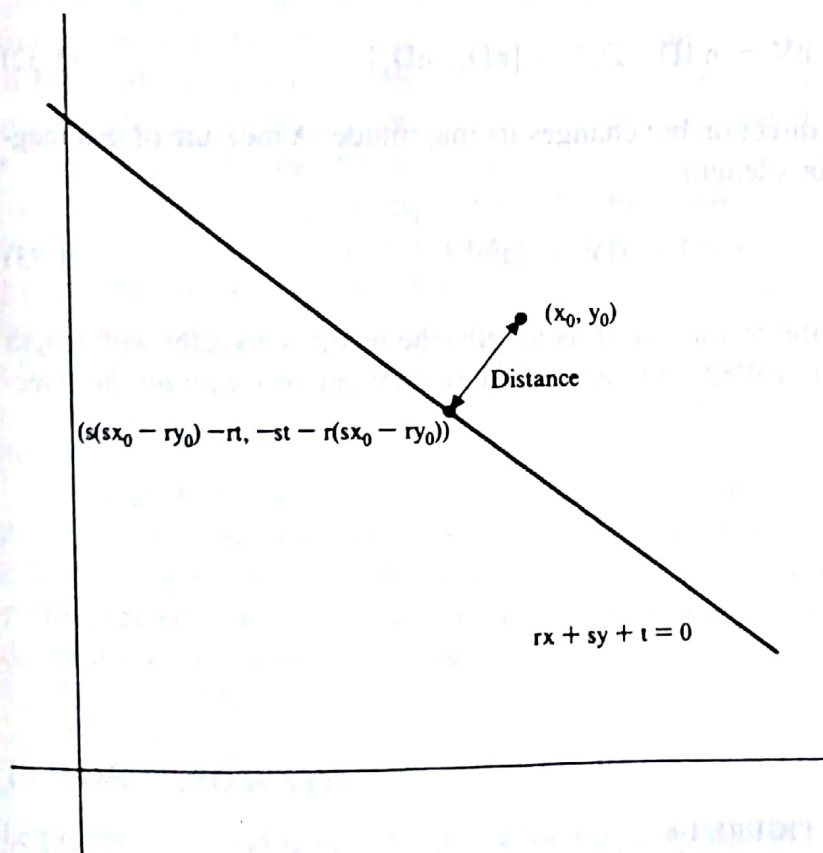


FIGURE 1-5

Distance between a point and a line.

satisfy Equation 1.7, and in fact this simple distance relationship is the motivation for this choice.

VECTORS

What is a vector? A vector has a single direction and a length. A vector may be denoted $[D_x, D_y]$, where D_x indicates how far to move along the x-axis direction and D_y indicates how far to move along the y-axis direction. (See Figure 1-6.)

Unlike line segments, vectors have no fixed position in space. They tell us how far and what direction to move, but they do not tell us where to start. The idea of a vector is useful because it closely parallels the manner in which a pen draws lines on paper or an electron beam draws lines on a cathode ray tube. The command to the pen may be to move so far from its current position in a given direction.

Two vectors may be added by adding their respective components.

$$V_1 + V_2 = [D_{x1}, D_{y1}] + [D_{x2}, D_{y2}] = [D_{x1} + D_{x2}, D_{y1} + D_{y2}] \quad (1.31)$$

We can picture this in terms of pen movements. Suppose we start at some point A. The first vector moves the pen from point A to point B; the second, from point B to point C. The right-hand side of the above equation produces a single vector which will move the pen directly from point A to point C.

We can also multiply a vector by a number by multiplying each of its components.

$$nV = n[D_x, D_y] = [nD_x, nD_y] \quad (1.32)$$

This preserves the vector's direction but changes its magnitude. A measure of that magnitude is given by the vector's length.

$$|V| = (D_x^2 + D_y^2)^{1/2} \quad (1.33)$$

If we multiply a vector by the reciprocal of its length, the result is a vector with length equal to 1. Such vectors are called unit vectors. They conveniently capture the direction information.

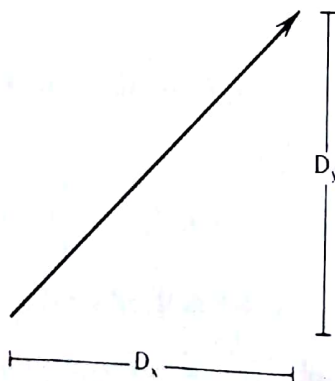


FIGURE 1-6
A vector.

We sometimes use vectors as a shorthand way of expressing operations on all coordinates. For example, the parametric equations for a line (Equations 1.16 and 1.17) can be combined into the vector form

$$\boxed{V = V_1 + u(V_2 - V_1)} \quad (1.34)$$

where $V = [x, y]$, $V_1 = [x_1, y_1]$, and $V_2 = [x_2, y_2]$.

We shall consider additional vector operations in later chapters as the need arises.

PIXELS AND FRAME BUFFERS

How does all this apply to an actual graphics display? To begin with, the mathematical notion of an infinite number of infinitesimal points does not carry over to the actual display. We cannot represent an infinite number of points on a computer, just as we cannot represent an infinite quantity of numbers. The machine is finite, and we are limited to a finite number of points making up each line (usually no more than a few hundred to a few thousand). The maximum number of distinguishable points which a line may have is a measure of the resolution of the display device. The greater the number of points, the higher the resolution. This limitation in the number of points may not bother us too much, because the human eye does not notice much detail finer than 1000 points per line segment. Since we must build our lines from a finite number of points, each point must have some size and so is not really a point at all. It is called a *pixel* (short for picture element). The pixel is the smallest addressable screen element. It is the smallest piece of the display screen which we can control. Each pixel has a name or address. The names which identify pixels correspond to the coordinates which identify points. Computer graphics images are made by setting the intensity and color of the pixels which compose the screen. We draw line segments by setting the intensities, that is, the brightness, of a string of pixels between a starting pixel and an ending pixel. We can think of the display screen as a grid, or array, of pixels. We shall give integer coordinate values to each pixel. Starting at the left with 1, we shall number each column. Starting at the bottom with 1, we shall number each row. The coordinate (i, j) will then give the column and row of a pixel. Each pixel will be centered at its coordinates. (See Figure 1-7.)

We may wish to place the intensity values for all pixels into an array in our computer's memory. Our graphics display device can then access this array to determine the intensity at which each pixel should be displayed. This array, which contains an internal representation of the image, is called the *frame buffer*. It collects and stores pixel values for use by the display device.

VECTOR GENERATION

The process of "turning on" the pixels for a line segment is called *vector generation*. If we know the endpoints which specify the segment, how do we decide which pixels

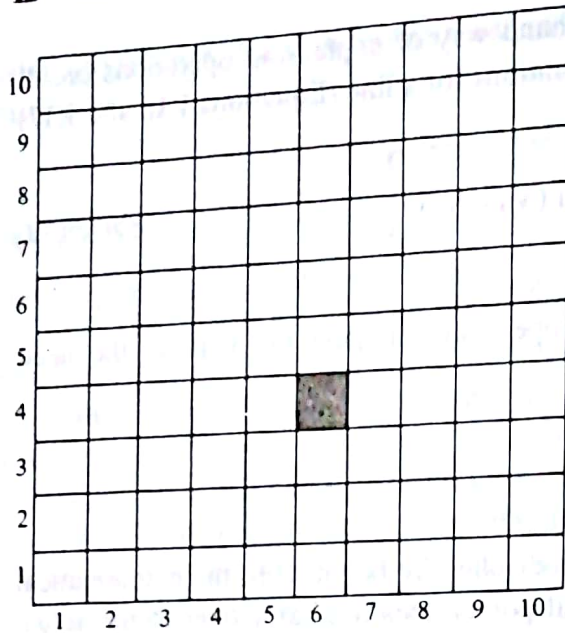


FIGURE 1-7

A pixel.

should have their intensity changed? There are several approaches to this problem. We shall present two examples here. The first is a general algorithm, while the second is a more efficient version for the case where the line segment endpoints are integers.

The problem is to select pixels which lie near to the line segment. We might try to turn on every pixel through which the line segment passes, but there is a problem with this approach. It would not be easy to find all such pixels, and since vector generation may be performed often, especially for animated displays or complex images, we want it to be efficient. Another problem is that the apparent thickness of the line would change with slope and position. An alternative would be to step along the columns of pixels, and for each column ask which row is closest to the line. We could then turn on the pixel in that row and column. We know how to find the row because we can place the x value corresponding to the column into the line equation, solve for y , and note which row it is in. This will work for lines with slopes between -1 and 1 (lines which are closer to being horizontal than vertical). But for the steeply rising or falling lines, the method will leave gaps. This failing can be overcome if we divide the lines into two classes. For the gentle slopes ($-1 < m < 1$) there are more columns than rows. These are the line segments where the length of the x component $D_x = (x_b - x_a)$ is longer than the length of the y component $D_y = (y_b - y_a)$, that is, $|D_x| > |D_y|$. For these cases, we step across the columns and solve for the rows. For the sharp slopes where $|D_x| \leq |D_y|$, we step up the rows and solve for the columns.

It would still be very inefficient if we actually had to solve the line equation to determine every pixel, but fortunately we can avoid this by taking uniform steps and using what we learn about the position of the line at each column (or row) to determine its position at the next column (or row). Consider the gentle slope case where we step across the columns. Each time we move from one column to the next, the value of x changes by 1 . But if x always changes by exactly 1 , then y will always change by exactly m (the slope). The change in y is

$$\begin{aligned} y_{i+1} - y_i &= (mx_{i+1} + b) - (mx_i + b) \\ &= m(x_{i+1} - x_i) = m(1) = m \end{aligned} \quad (1.35)$$

This means that as we step along the columns, we can find the new value of y for the line by just adding m to the y position at the previous column. So our approach so far is first to find the column at the left end of the line segment and the y value at that column, and then to step through the columns which the line segment crosses, adding m to the y value each time to get a new y value. The y values will be used to select a row at each column, and we turn on a pixel at the selected row and column. (See Figure 1-8.)

The vector generation algorithms (and curve generation algorithms) which step along the line (or curve) to determine the pixels which should be turned on are sometimes called *digital differential analyzers* (DDAs). The name comes from the fact that we use the same technique as a numerical method for solving differential equations. For a line segment, we solve the differential equation for a straight line and plot the result instead of printing it.

We shall make one further refinement in our approach. Instead of determining the full y value of the line segment for each column, we shall only keep track of the current height above the closest row boundary. At each step, the height increases by m along with y . We can check the height at each step to see if we have moved into a new row. If we have entered a new row, then we change the row value used to select pixels and also subtract 1 from our height so that we will now be measuring height from the new row boundary. For lines with negative slopes, we could let the height decrease and check for crossing of a lower row boundary, but instead we use the absolute value of

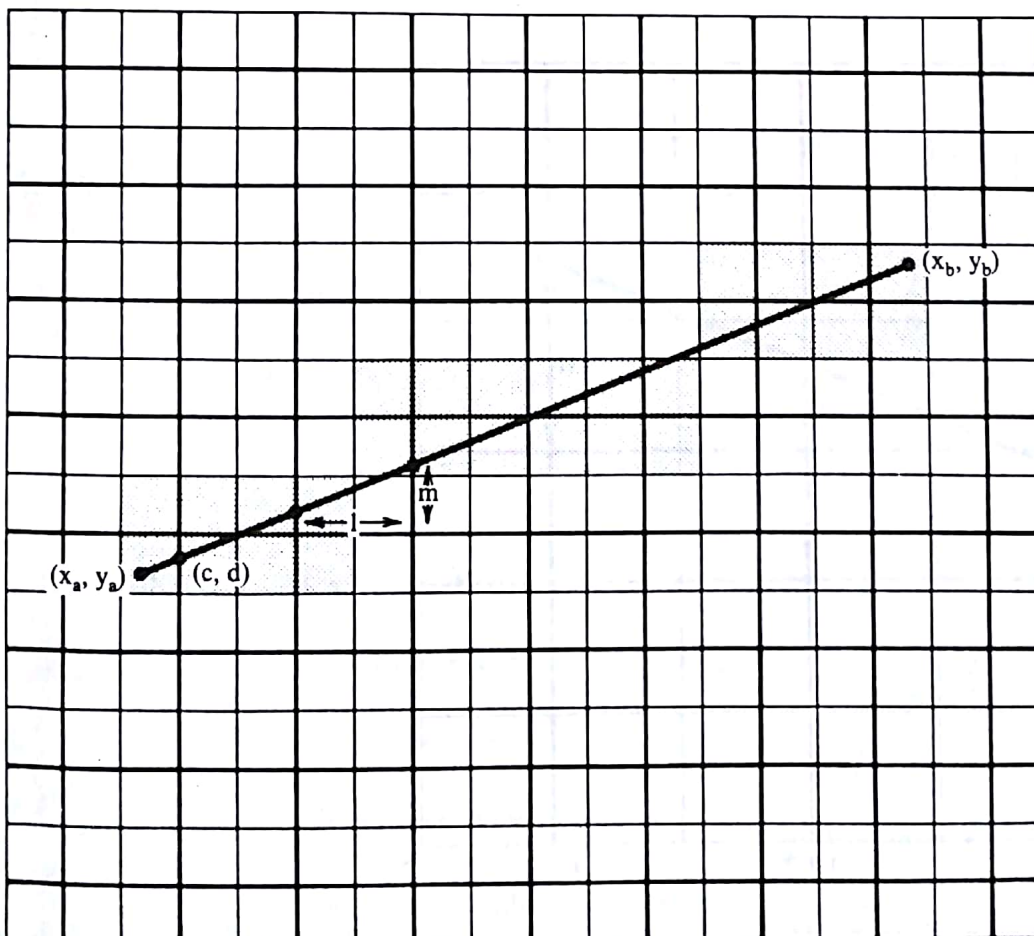


FIGURE 1-8

Turn on pixels with centers closest to the line segment.

CEILING \rightarrow 3.5 hai tu return 3 karega
 Floor \rightarrow 3.5 hai tu 4 return karega

height and slope and check as it steps up. This is in order to be consistent with the algorithm presented in the next section.

For lines with sharp slopes, a similar procedure is used, only the roles of x and y are exchanged and a new "height" is found for a new row by adding $1/m$ to the height value for the old row.

Now a few more words on how our particular vector generator finds the starting point. For the gentle slope cases, we shall turn on pixels in the columns that have a center line which crosses the line segment. We shall center the columns and rows on integer coordinate values, so if x_a is the left end of the line segment and x_b is the right end, then columns between $c = \text{CEILING}(x_a)$ and $f = \text{FLOOR}(x_b)$ inclusive are affected. (The function CEILING returns the smallest integer which is greater than or equal to its argument, and FLOOR returns the largest integer which is less than or equal to its argument.) Our starting y position will correspond to the point at $x = c$, and might not be the endpoint y_a . The starting y value may be determined from the line equation as follows:

$$\begin{aligned} d &= mc + b = mc + (y_a - mx_a) \\ &= y_a + m(c - x_a) \end{aligned} \quad (1.36)$$

To find the index of the closest row, we round the value of y . Rounding may be done by adding 0.5 and then using the FLOOR function.

$$r = \text{FLOOR}(y + 0.5) \quad (1.37)$$

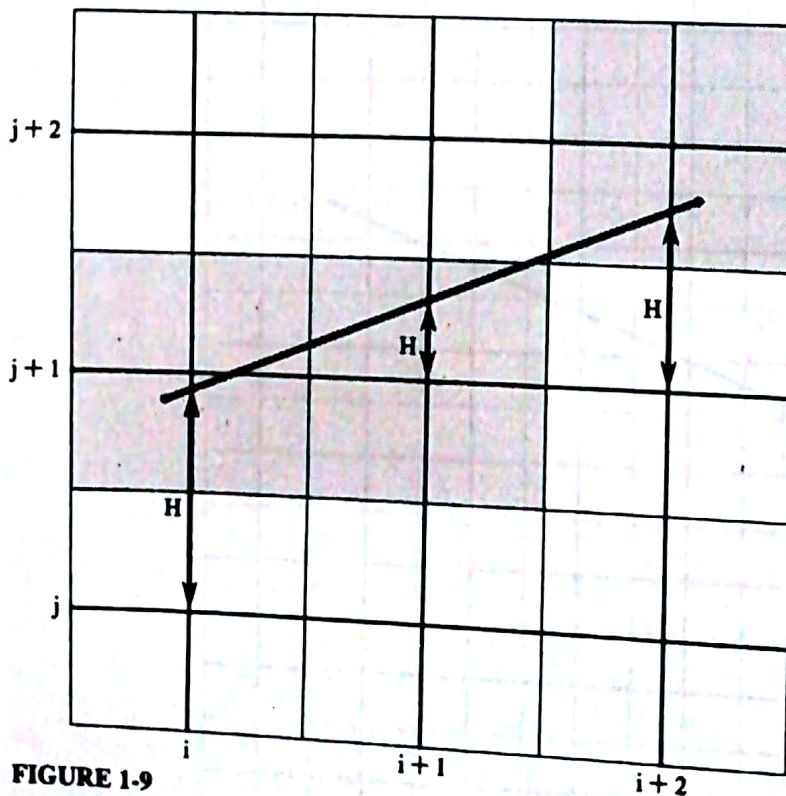


FIGURE 1-9
Height of the line above pixel-row boundaries.

$r = \text{ceiling}(y - 0.5)$

To find the height of y above the row boundary, we take the difference of y (where the line is) and r (the row index). If we then add m to this, we have the height value which should be checked for the next column and so on. (See Figure 1-9.)

We shall now present the full algorithm.

Imp 1.1 Algorithm VECGEN(XA, YA, XB, YB, INTENSITY) For changing pixel values of the frame buffer along a line segment **DDA**

Arguments XA and YA are the coordinates of one endpoint
XB and YB are the coordinates of the other endpoint (x_a, y_a) (x_b, y_b)
INTENSITY is the intensity setting to be used for the vector

Global FRAME the two-dimensional frame buffer array **(display info)**
Local DX, DY the vector to be drawn
R and C the row and column indices for the pixel to be changed
F the stopping index
D the line segment coordinate at the starting point
H the difference between the line segment and the row index
M the slope of the line segment
M1 the change in H when a boundary is crossed

BEGIN

determine the components of the vector

$DX \leftarrow XB - XA;$

$DY \leftarrow YB - YA;$

decide on whether to step across columns or up rows

IF $|DX| > |DY|$ **THEN**

BEGIN

the gentle slope case

$M \leftarrow DY / DX;$

set up of starting point depends on which point is leftmost

IF $DX > 0$ **THEN**

BEGIN

$C \leftarrow \text{CEILING}(XA);$

$D \leftarrow YA + M * (C - XA);$

$F \leftarrow \text{FLOOR}(XB);$

END

ELSE

BEGIN

$C \leftarrow \text{CEILING}(XB);$

$D \leftarrow YB + M * (C - XB);$

$F \leftarrow \text{FLOOR}(XA);$

END;

$R \leftarrow \text{FLOOR}(D + 0.5);$

$H \leftarrow R - D + M;$

IF $M > 0$ **THEN**

BEGIN

the positive slope case

$M1 \leftarrow M - 1;$



now step through the columns

WHILE $C \leq F$ **DO**

```

BEGIN
    set the nearest pixel in the frame buffer
    FRAME[C, R] ← INTENSITY;
    next column
    C ← C + 1;
    should row change
    IF H ≥ 0.5 THEN
        BEGIN
            R ← R + 1;
            H ← H + M1;
        END
    ELSE H ← H + M;
    END;
END;
ELSE (M < 0)
    BEGIN
        then negative slope case
        M ← -M;
        H ← -H;
        M1 ← M - 1;
        WHILE C ≤ F DO
            BEGIN
                set the nearest pixel in the frame buffer
                FRAME[C, R] ← INTENSITY;
                next column
                C ← C + 1;
                should row change
                IF H > 0.5 THEN
                    BEGIN
                        R ← R - 1;
                        H ← H + M1;
                    END
                ELSE H ← H + M;
                END;
            END;
        END;
    END
ELSE
    BEGIN
        the sharp slope case
        IF DY = 0.0 THEN RETURN;
        here the above steps are repeated
        with the roles of x and y interchanged
    END;
    RETURN;
END;

```

The actual algorithm contains a couple of things which were not brought up in the above discussion. The determination of the starting values is complicated by not

knowing a priori whether XA is left or right of XB, and YA above or below YB. The algorithm also contains a test on the sign of the slope. It is used to select a loop which moves the line in the proper direction.

BRESENHAM'S ALGORITHM

Imp The above algorithm was chosen because it can be revised into a very efficient and popular form known as *Bresenham's algorithm* for the special case where the coordinates of the line segment endpoints are integers. The attractiveness of Bresenham's algorithm is that it can be implemented entirely with integer arithmetic. Integer arithmetic is usually much faster than floating-point arithmetic. Furthermore, Bresenham's algorithm does not require any multiplication or division. To derive the algorithm, first consider the effect of integer coordinates on determining the starting point. The starting point will just be the endpoint of the line segment. No calculation is needed to move along the line to a pixel center, because it is already there. This eliminates one place where the line slope was needed, but we also used the slope to update H. How can we revise our test for new rows so that it requires only integer arithmetic? To simplify the discussion, consider just the gentle slope case. The test was

$$H > 0.5 \quad (1.38)$$

where

$$H \leftarrow H + M$$

or

$$H \leftarrow H + M - 1$$

at each column. Note first that we can rewrite the test as

$$H - 0.5 > 0$$

Now we can multiply by 2 to get

$$2H - 1 > 0 \quad (1.39)$$

Notice how the 0.5 fraction is removed. But H still has a fractional part, which arises from the denominator in M that is added in at each step. To remove it, we multiply by DX. Assuming we have arranged the endpoints so that DX is positive, the test is then

$$2DX H - DX > 0$$

Suppose we define G as

$$G = 2DX H - DX \quad (1.40)$$

The test is simply

$$G > 0 \quad (1.41)$$

Then how does G change from one column to the next? Solving for H gives

$$H = \frac{G + DX}{2DX} \quad (1.42)$$

If

$$H_{\text{new}} \leftarrow H_{\text{old}} + M$$

then

$$\frac{G_{\text{new}} + DX}{2DX} \leftarrow \frac{G_{\text{old}} + DX}{2DX} + \frac{DY}{DX}$$

or

$$G_{\text{new}} \leftarrow G_{\text{old}} + 2DY \quad (1.43)$$

For the case

$$H_{\text{new}} \leftarrow H_{\text{old}} + M - 1$$

we get

$$G_{\text{new}} \leftarrow G_{\text{old}} + 2DY - 2DX \quad (1.44)$$

Calculating G requires only additions and subtractions and can be done entirely with integers. The trick then is to use the test $G > 0$ to determine when a row boundary is crossed by the line instead of the test $H > 0.5$. For integer endpoints, the initial value of H is M , so the initial value of G is

$$G = 2DX \left(\frac{DY}{DX} \right) - DX = 2DY - DX \quad (1.45)$$

For each column, we check G . If it is positive, we move to the next row and add $2DY - 2DX$ to G . Otherwise, we keep the same row and add $2DY$ to G . The full algorithm follows.

1.2 Algorithm BRESENHAM(XA, YA, XB, YB, INTENSITY) For changing pixel values of the frame buffer along a line segment with integer endpoints

Arguments XA and YA are the coordinates of one endpoint

XB and YB are the coordinates of the other endpoint

INTENSITY is the intensity setting to be used for the vector

Global FRAME the two-dimensional frame buffer array

Local DX, DY the vector to be drawn

R, C the row and column indices for the pixel

F the final row or column

G for testing for a new row or column

INC1 increment for G when row or column is unchanged

INC2 increment for G when row or column changes

POS-SLOPE a flag to indicate if the slope is positive

BEGIN

determine the components of the vector

$DX \leftarrow XB - XA;$

$DY \leftarrow YB - YA;$


```

determine the sign of the slope
POS-SLOPE  $\leftarrow (DX > 0)$ ;
IF  $DY < 0$  THEN POS-SLOPE  $\leftarrow$  NOT POS-SLOPE;
decide on whether to step across columns or up rows
IF  $|DX| > |DY|$  THEN
  BEGIN
    this is the gentle slope case
    IF  $DX > 0$  THEN
      BEGIN
         $C \leftarrow XA$ ;
         $R \leftarrow YA$ ;
         $F \leftarrow XB$ ;
      END
    ELSE
      BEGIN
         $C \leftarrow XB$ ;
         $R \leftarrow YB$ ;
         $F \leftarrow XA$ ;
      END;
     $INC1 \leftarrow 2 * |DY|$ ;
     $G \leftarrow 2 * |DY| - |DX|$ ;
     $INC2 \leftarrow 2 * (|DY| - |DX|)$ ;
    IF POS-SLOPE THEN
      BEGIN
        now step across line segment
        WHILE  $C \leq F$  DO
          BEGIN
            set nearest pixel in the frame buffer
             $FRAME[C, R] \leftarrow INTENSITY$ ;
            next column
             $C \leftarrow C + 1$ ;
            should row change
            IF  $G \geq 0$  THEN
              BEGIN
                 $R \leftarrow R + 1$ ;
                 $G \leftarrow G + INC2$ ;
              END
            ELSE  $G \leftarrow G + INC1$ ;
          END;
        END;
      END
    ELSE
      BEGIN
        WHILE  $C \leq F$  DO
          BEGIN
            set nearest pixel in the frame buffer
             $FRAME[C, R] \leftarrow INTENSITY$ ;
            next column
             $C \leftarrow C + 1$ ;
            should row change

```

```

        IF G > 0 THEN
            BEGIN
                R ← R - 1;
                G ← INC2;
            END
        ELSE G ← G + INC1;
        END;
    END;
END
ELSE
    BEGIN
        this is the sharp slope case
        here the above steps are repeated
        with the roles of x and y interchanged
    END;
RETURN;
END;

```

We have seen how the BRESENHAM algorithm is more efficient than the VEC-GEN algorithm in that it requires only integer addition and subtraction. We might ask, why not always use it? Why not always round line segment endpoints to integers before vector generation? In fact, many systems will do just that, but note that in rounding the line segment endpoint positions, errors are introduced. These errors can be seen in cases where two overlapping line segments are drawn on the display (as when two objects are shown side by side). Another example occurs when a new line segment with a different INTENSITY overlaps an old line segment; we might like to change the old segment's pixels to the appropriate values for the new segment. If we deal with exactly the same line equation for both line segments, then the same pixels should be selected; but if errors in the endpoint positions are introduced, the line equations might not match, and the new line segment can leave pixels from the old line segment peeking out from behind it. Note, however, that even our VECGEN algorithm can have errors introduced in the endpoint position as the result of round-off in the floating point arithmetic of the machine.

Algorithms for vector generation (such as the ones we've seen) may be implemented in hardware where speed is important. This is usually done for displays which avoid the cost of a large frame buffer memory by letting the vector generator directly control the drawing instrument (usually an electron beam in a cathode ray tube). Instead of setting the (x, y) element in a frame buffer, the pen (or electron beam) is moved to position (x, y), where it illuminates the pixel on the screen.

ANTIALIASING OF LINES

Many displays allow only two pixel states, on or off. For these displays, lines may have a jagged or stair-step appearance when they step from one row or column to the next. The lower the resolution, the more apparent the effect. This is one aspect of a

phenomenon called *aliasing*. Aliasing produces the defects which occur when the scene being displayed changes faster or more smoothly than every two pixels. Displays which allow setting pixels to gray levels between black and white provide a means to reduce this effect. The technique is called *antialiasing*, and it uses the gray levels to gradually turn off the pixels in one row as it gradually turns on the pixels in the next. (See Figure 1-10.)

The vector generation algorithms can be modified to perform antialiasing. Remember that for gentle sloped lines, we in effect examined the line position for each column index and decided which row was closest. The line segment would lie between two pixels, and we picked one. Suppose that instead of picking the closest, we turned them both on. We should choose the intensity values according to a function of the distance between the pixel index and the line segment so that the pixel closest to the line receives most of its intensity. The sum of the intensity values for the two pixels should match the total intensity value for the line. The function used can be a simple or a complex expression based on intensity patterns, pixel shapes, and how lines cover them. In general, we want the pixel's intensity to match the amount of the line which covers its area. Antialiasing with complicated functions can still be done efficiently by storing the function values in a table. The table is then used to look up the intensity for a distance between the pixel index and the line. (See Figure 1-11.)

THICK LINE SEGMENTS

Raster displays allow the display of lines with thickness greater than one pixel. To produce a thick line segment, we can run two vector generation algorithms in parallel to find the pixels along the line edges. As we step along the line finding successive edge pixels, we must also turn on all pixels which lie between the boundaries. For a gentle sloping line between (x_a, y_a) and (x_b, y_b) with thickness w , we would have a top boundary between the points $(x_a, y_a + w_y)$ and $(x_b, y_b + w_y)$ and a lower boundary between $(x_a, y_a - w_y)$ and $(x_b, y_b - w_y)$ where w_y is given by

$$w_y = \frac{(w - 1)}{2} \frac{[(x_b - x_a)^2 + (y_b - y_a)^2]^{1/2}}{|x_b - x_a|} \quad (1.46)$$

This is the amount by which the boundary lines are moved from the line center. The $(w - 1)$ factor is the desired width minus the one-pixel thickness we automatically receive from drawing the boundary. We divide this by 2 because half the thickness will be used to offset the top boundary, and the other half to move the bottom boundary. The factor containing the x and y values is needed to find the amount to shift up and

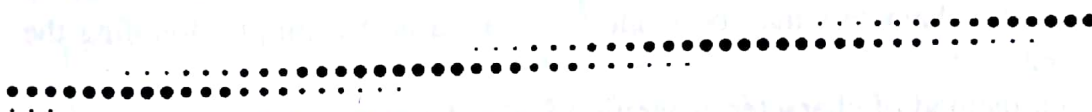
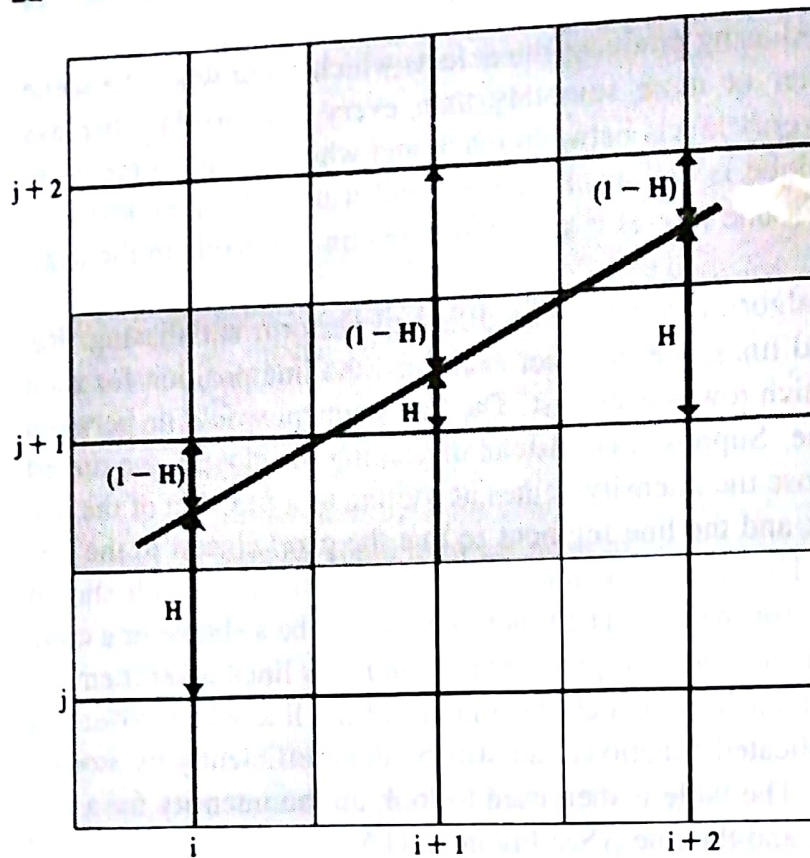


FIGURE 1-10
Antialiasing of a line.

**FIGURE 1-11**

Using vertical distance from the pixel to determine intensity.

down in order to achieve the proper width w as measured perpendicular to the line direction, not up and down. (See Figure 1-12.) Sharply sloping lines can be handled similarly with the x and y roles reversed.

CHARACTER GENERATION

Along with lines and points, strings of characters are often displayed to label and annotate drawings and to give instructions and information to the user. Characters are almost always built into the graphics display device, usually as hardware but sometimes through software. There are two primary methods for character generation. One is called the *stroke method*. This method creates characters out of a series of line segments, like strokes of a pen. We could build our own stroke-method character generator by calls to the VECGEN algorithm. We would decide what line segments are needed for each character and set up the calls to the VECGEN for each character we wished to draw. In actual graphics displays, the commands for drawing the character line segments may be in either hardware or software. The stroke method lends itself to changes of scale; the characters may be made twice as large by simply doubling the length of each segment.

The second method of character generation is the *dot-matrix* or *bitmap method*. In this scheme, characters are represented by an array of dots. An array of 5 dots wide and 7 dots high is often used, but 7×9 and 9×13 arrays are also found. High-resolution devices, such as ink-jet or laser printers, may use character arrays that are over 100

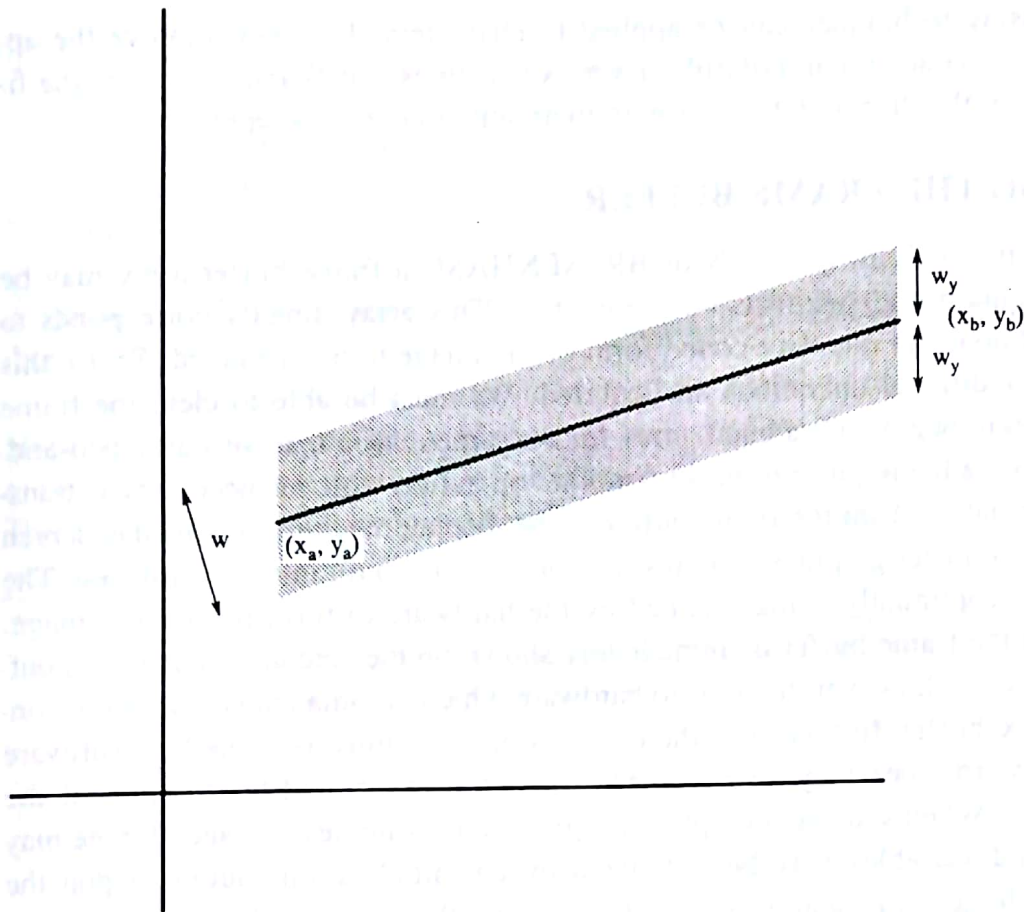


FIGURE 1-12
Thick line construction.

pixels on a side. This array is like a small frame buffer, just big enough to hold a character. The dots are the pixels for this small array. Placing the character on the screen then becomes a matter of copying pixel values from the small character array into some portion of the screen's frame buffer (usually, for common alphanumeric terminals the dot matrix is allowed to directly control the intensity of small parts of the screen, eliminating the need for a large frame buffer). The memory containing the character dot-matrix array is often a hardware device called a *character-generator chip*, but random access memory may also be used when many fonts are desired. The size of a dot is fixed, so the dot-matrix method does not lend itself to variable-sized characters. (See Figure 1-13.)

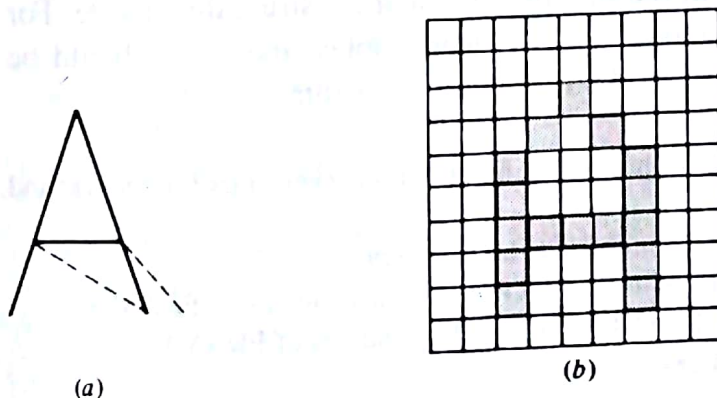


FIGURE 1-13
Character generation. (a) Stroke method;
(b) dot-matrix method.

Antialiasing techniques can be applied to characters. This can improve the appearance of the character, particularly for very small fonts and characters where the finite resolution of the display interferes with their smooth curved shapes.

DISPLAYING THE FRAME BUFFER

Imp Using algorithms such as VECGEN or BRESENHAM, a frame buffer array may be modified to contain line segments and characters. This array directly corresponds to the screen and holds an intensity-coded form of the image to be displayed. To use this device, some additional operations are required. We must be able to clear the frame buffer. We need to begin with a blank array for the same reason that we start a pen-and-ink drawing with a blank piece of paper. Another operation which is necessary is transfer of the information from the frame buffer to the display medium (the display screen or the paper). For raster graphics displays, this operation is built into the hardware. The frame buffer is continually being scanned by the hardware to form the screen image. Any change in the frame buffer is immediately shown on the screen. For graphics output on a line printer, however, there is no hardware which automatically shows the content of the frame buffer. In this case, the display operation must be done by a software routine. Finally, routines may be necessary for initialization and termination of the graphics system. When starting a graphics program, hardware devices and storage may be allocated and variables may be initialized by an initialization routine. Upon the completion of the job, the deallocation of the storage, the release of hardware devices, and other housekeeping chores may be done by a termination routine.

To complete this chapter, we would like to give the algorithms needed to obtain graphics output from a line printer or common CRT terminal. The first algorithms, the vector generators, have already been presented. We shall need a frame buffer array to hold the image. The size of this array depends upon the resolution of the display device. If a CRT displaying 24 lines of 80 characters each is used, the frame buffer might be dimensioned FRAME[80,24]. This is not the only choice. In many displays we find that setting the lower right-hand pixel will cause automatic scrolling of the image, shifting the top line off the screen. This can be prevented by not using the bottom line, and [80,23] may be more appropriate. Or we may find that the display screen is not square, and for the sake of treating the x and y directions equally, we may select a square subarea like [60,23]. A line printer may use an array FRAME[90,50] or larger, depending on the storage available and the number of characters per line. The frame buffer will be an array of characters. The INTENSITY of algorithm 1.1 will be a character such as the period or asterisk, out of which we will construct the picture. For a clear or empty frame buffer corresponding to a clear display, the array should be filled entirely with blanks. This is done by the following algorithm.

1.3 Algorithm ERASE Clears the frame buffer by assigning every pixel a background value

Global	FRAME the two-dimensional frame buffer array
	WIDTH-START and HEIGHT-START the starting indices of FRAME
	WIDTH-END and HEIGHT-END the ending indices of FRAME
Local	X, Y frame buffer indices


```

BEGIN
  FOR Y = HEIGHT-START TO HEIGHT-END DO
    FOR X = WIDTH-START TO WIDTH-END DO
      FRAME[X, Y] ← ' ';
    RETURN;
  END;

```

A call of the ERASE routine clears the display. We can use this whenever we wish to draw a new picture.

The DISPLAY routine is used to show the contents of the frame buffer on a line printer. Note that while our y coordinate begins at the bottom and increases as we move up the display, printers typically begin at the top of the page and work down. This means that the y coordinate must be displayed beginning with its high values and working down.

1.4 Algorithm DISPLAY This displays the contents of the frame buffer

```

Global  FRAME the frame buffer array
        WIDTH-START and HEIGHT-START the starting indices of FRAME
        WIDTH-END and HEIGHT-END the ending indices of FRAME
Local   X, Y the pixel being displayed
BEGIN
  FOR Y = HEIGHT-END TO HEIGHT-START DO
    PRINT FOR X = WIDTH-START TO WIDTH-END, FRAME[X, Y];
  RETURN;
END;

```

To complete this package, we must include a routine to initialize the parameters for the display size and to perform any system-dependent housekeeping. Included in the initialization should be the establishment of frame buffer size parameters and the clearing of the display.

1.5 Algorithm INITIALIZE-1

```

Global  WIDTH-START and HEIGHT-START the starting indices of FRAME
        WIDTH-END and HEIGHT-END the ending indices of FRAME
        WIDTH, HEIGHT the dimensions of FRAME
BEGIN
  perform any needed storage allocation, hardware assignment,
  or other system-dependent housekeeping;
  HEIGHT-START ← starting column index of FRAME;
  WIDTH-START ← starting row index of FRAME;
  HEIGHT-END ← ending column index of FRAME;
  WIDTH-END ← ending row index of FRAME;
  HEIGHT ← HEIGHT-END - HEIGHT-START;
  WIDTH ← WIDTH-END - WIDTH-START;
  ERASE;
  RETURN;
END;

```

1.6 Algorithm TERMINATE BEGIN

release any assigned hardware devices;
perform any necessary final housekeeping;
STOP;
END;

FURTHER READING

There are a large number of texts on analytic geometry. One book which is oriented toward computer graphics is [ROG76]. Use of a frame buffer was first described in [NOL71]. Bresenham presented his algorithm in [BRE65]. DDAs and Bresenham's algorithm are also discussed in [EAR77], [FIE85], [LOC80], and [SPR82]. An alternative approach to vector generation based on the general form of the line equation rather than on the parametric form is presented in [DAN70]. Another approach based on the structural properties of the line is described in [BRO74] and [CED79]. A symmetric algorithm which allows a sequence of lines to be erased by drawing them in reverse order with "white ink" is given in [TRA82]. An example of a 7×9 dot-matrix font may be found in [VAR71]. An example of specialized character-generation hardware may be found in [THO72]. A discussion of antialiasing lines may be found in [CRO78], [GUP81], and [PIT80]. A simple, fast algorithm for antialiased lines is given in [KET85]. Antialiasing of characters is described in [WAR80]. A more general discussion of aliasing and antialiasing is presented in [LEL80]. An early description of CORE was published in [GSPC79]. An overview of the GKS standard is given in [BON82]. The GKS standard is published in [GKS84].

- [BON82] Bono, P. R., Encarnacao, J. L., Hopgood, F. R. A., ten Hagen, P. J. W., "GKS—The First Graphics Standard," *IEEE Computer Graphics and Applications*, vol. 2, no. 7, pp. 9–23 (1982).
- [BRE65] Bresenham, J. E., "Algorithm for Computer Control of a Digital Plotter," *IBM System Journal*, vol. 4, no. 1, pp. 106–111 (1965).
- [BRO74] Brons, R., "Linguistic Methods for the Description of a Straight Line on a Grid," *Computer Graphics and Image Processing*, vol. 3, no. 1, pp. 48–62 (1974).
- [CED79] Cederberg, R. L. T., "A New Method for Vector Generation," *Computer Graphics and Image Processing*, vol. 9, no. 2, pp. 183–195 (1979).
- [CRO78] Crow, F. C., "The Use of Grayscale for Improved Raster Display of Vectors and Characters," *Computer Graphics*, vol. 12, no. 3, pp. 1–5 (1978).
- [DAN70] Danielsson, P. E., "Incremental Curve Generation," *IEEE Transactions on Computers*, vol. C-19, no. 9, pp. 783–793 (1970).
- [EAR77] Earnshaw, R. A., "Line Generation for Incremental and Raster Devices," *Computer Graphics*, vol. 11, no. 2, pp. 199–205 (1977).
- [FIE85] Field, D., "Incremental Linear Interpolation," *ACM Transactions on Graphics*, vol. 4, no. 1, pp. 1–11, (1985).
- [GKS84] "Graphics Kernel System, ANSI X3H3/83-25r3," *Computer Graphics*, vol. 18, special issue (Feb. 1984).
- [GSPC79] "Status Report of the Graphics Standards Planning Committee of ACM/SIGGRAPH," *Computer*