CHAPTER TWO

GRAPHICS PRIMITIVES

(Design library)

Amatullah

INTRODUCTION

The computer graphics user will not, in general, have to start from scratch in preparing for a project. Instead, he will have a graphics system available. This system will include special hardware for output and input of pictorial information and software routines for performing the basic graphics operations.

The purpose of a graphics system is to make programming easier for the user. Just as high-level computer languages make programming easier by supplying powerful operations and constructs which match the requirements of the problem, a graphics system supplies operations and constructs suited to the creation of graphical images to enhance the development of a graphics program. In fact, some graphics systems are in the form of special high-level graphics languages, languages suitable for solving graphics problems. Other graphics systems are in the form of extensions to general-purpose high-level languages such as FORTRAN, PL/I, or PASCAL. Such extensions may be made through a ''package'' of subprograms or by the addition of new language constructs.

While the form may vary between different graphics systems, there are certain basic operations which can almost always be found. These are operations such as moving the pen (or electron beam), drawing a line, writing a character or a string of text, and changing the line style. In this chapter we shall begin constructing our own graphics system. We shall start by looking in more detail at some of the types of display devices. We shall present algorithms for some basic graphics operations, and we shall introduce the concepts of device independence and of a display file.

DISPLAY DEVICES

We saw in the last chapter how computer graphics images are composed of a finite number of picture elements, or pixels. A display with good resolution might have 1000 divisions in both the x and y directions. The screen would then have 1000 × 1000 or 1 million, pixels. Each pixel requires at least one bit of intensity information, light or dark, and further bits are needed if shades of gray or different colors are desired. Thus, if we actually store the information for each pixel in the computer's memory, a lot of memory may be required. This is, in fact, what is done in some *raster graphics* displays. As we said in Chapter 1, the portion of the memory used to hold the pixels is called the frame buffer. The memory is usually scanned and displayed by *direct memory access*, that is, special hardware, independent of the central processor (leaving the processor free for generation of the images). (See Figure 2-1.)

In the raster display, the frame buffer may be examined to determine what is currently being displayed. Surfaces, as well as lines, may be displayed on raster display devices. Since images may be displayed on television-style picture tubes, raster display devices can often take advantage of the technological research and mass production of the television industry. The raster terminal can also display color images. One of the problems with the raster display is the time which may be required to alter every pixel whenever the image is changed. Another disadvantage is the cost of the required memory. This has been eased in some displays by using a coarse resolution (fewer pixels). However, technological progress has steadily brought the price of memory down, and at this point in time it appears that raster graphics will play a dominant role in the future.

In the past, the cost of memory made the raster display seem much less promising. Different designs for graphics displays were developed in an effort to reduce expenses. One approach was to let the display medium remember the image, instead of using the computer memory. This is what plotters do. A pen is lowered onto paper and moved under the direction of a vector generation algorithm. (See Figure 2-2.)

Once the line is drawn, the ink on the paper "remembers" it, and the computer need not consider it further. The main disadvantage to this approach is that once drawn, a line cannot be easily removed. If we wish to change the image on the plotter by removing a line, we must get a fresh piece of paper and redraw the picture (without the removed line). This can be time-consuming (and can use a lot of paper). For this reason, plotters are not the best devices for interactive graphics.

The first "low-cost" CRT display (under $5000) was produced by Tektronix. Because of good resolution and low cost, these terminals became widely accepted and may be commonly found today. The terminals use special cathode ray tubes called *direct view storage tubes (DVST)*, which behave much the same way as a plotter. An elec-

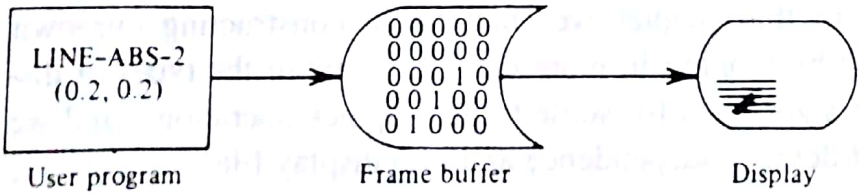


FIGURE 2-1
Raster display system.

raster → all information are define (matrix)
non-raster → command interpreted.

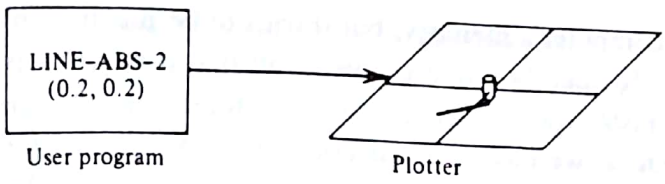


FIGURE 2-2
Plotting system.

tron beam is directed at the surface of the screen. The position of the beam is controlled by electric or magnetic fields within the tube. Once the screen phosphors of this special tube have been illuminated by the electron beam, they stay lit. (See Figure 2-3.)

As was the case with the plotter, one cannot alter a DVST image except by erasing the entire screen and drawing it again. This can be done faster than on a plotter, but the process is still time-consuming, making interaction difficult and eliminating these devices from use in real-time animation.

A display device which stores the image (as plotters and storage tubes do) but allows selective erasing is the *plasma panel*. The plasma panel contains a gas at low pressure sandwiched between horizontal and vertical grids of fine wires. A large voltage difference between a horizontal and vertical wire will cause the gas to glow as it does in a neon street sign. A lower voltage will not start a glow but will maintain a glow once started. Normally the wires have this low voltage between them. To set a pixel, the voltage is increased momentarily on the wires that intersect the desired point. To extinguish a pixel, the voltage on the corresponding wires is reduced until the glow cannot be maintained. Plasma panels are very durable and are often used for military applications. They have also been used in the PLATO educational system.

A device which is now becoming economical is the *liquid crystal display*. This is a flat panel display technology, which makes it less bulky than CRTs. Also, because of its low voltage and power requirements, it is lighter in weight, making it the display of choice where portability is required. In a liquid crystal display, light is either transmitted or blocked, depending upon the orientation of molecules in the liquid crystal. An electrical signal can be used to change the molecular orientation, turning a pixel on or off. The material is sandwiched between horizontal and vertical grids of electrodes which are used to select the pixel.

At the present time, liquid crystal televisions are available, but the display is formed on a single semiconductor wafer, which limits its size. Larger but slower displays are also available (640 × 250 pixels). This technology is still young and may compete with CRT displays in the future.

Another approach to the display of graphical information, which has had a profound effect on today's graphics methods, is the *vector refresh display*. The vector re-

FIGURE 2-3
Direct view storage tube system.

fresh display stores the image in the computer's memory, but it tries to be much more efficient about it than a raster display. To specify a line segment, all that is required is the coordinates of its endpoints. The raster frame buffer, however, stores not only the endpoints but also all pixels in between, as well as all pixels not on the line. The vector refresh display stores only the commands necessary for drawing the line segments. The input to the vector generator is saved, instead of the output. These commands are saved in what is called the *display file*. They are examined and the lines are drawn using a vector-generating algorithm. This is done on a normal cathode ray tube, so the image quickly fades. In order to present a steady image, the display must be drawn repeatedly. This means that a vector generator must be applied to all of the lines in an image fast enough to draw the entire image before flicker is noticeable (more than 30 times a second). To do this, the vector generators of refresh displays are usually implemented in hardware. (See Figure 2-4.)

Refresh displays do allow real-time alteration of the image. They also have some disadvantages. The images formed are composed of line segments, not surfaces, and a complex display may flicker because it will take a long time to analyze and draw it.

The concept of a display file has proved to be a useful one. It provides an interface between the image specification process and the image display process. It also defines a compact description of the image, which may be saved for later display. The display-file idea may be applied to devices other than refresh displays. Such files are sometimes called *pseudo display files*, or *metafiles*. Standards are currently under development for metafiles to aid in the transport of images between computers.

Devices such as vector refresh displays, pen plotters, and DVSTs may only directly support *line* drawing. They do not provide support for the solid areas which can be constructed on raster displays. Such line-drawing devices are called *calligraphic* displays.

All the graphics display systems we have described can display the image as it is being constructed. When a command to draw a line segment is issued, the line segment can immediately appear on the screen or paper. The major difference is in whether the display may be altered. To change a vector refresh display, one needs only to change the display file. To change a raster display, one alters the frame buffer. A similar alteration can change a plasma panel. However, to change a plotter or a DVST, one must first call a new-frame routine, which shifts the paper or erases the screen, and then one must redraw the entire image.

We should mention one further class of display device. They are raster printers and plotters that create the image in a single sweep across the page. There are a number of technologies which fall into this category including film printers, laser printers,

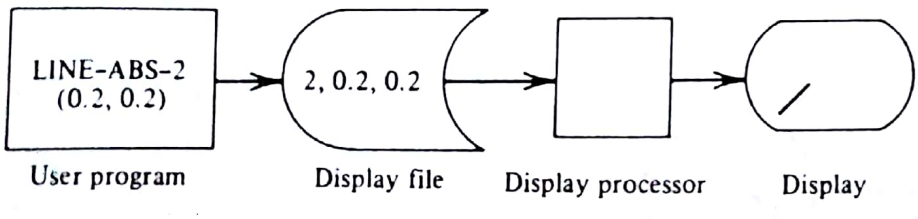


FIGURE 2-4
Vector refresh display system.

electrostatic plotters, thermal and thermal-transfer printers, ink-jet printers, and impact dot-matrix printers. The devices can range in resolution from about 100 pixels per inch for dot-matrix printers to over 1000 pixels per inch for film printers. Dot-matrix printers have an array of wires which can be individually triggered to press an inked ribbon to make a dot on the paper. By sweeping the array across the paper, images can be formed. Ink-jet printers form tiny droplets of ink which can be guided to the paper to form dots. They use nozzles which move across the paper. Laser printers are built on top of copier technology. Instead of copying the light pattern reflected from a piece of paper, a laser is used to supply the light pattern. A rotating mirror sweeps the laser in a raster pattern, and a light valve turns the beam on or off to form the image which is "copied." Film printers use a laser scanning system similar to laser printers, but they focus the laser directly on photographic film to form the picture. Thermal printers have a print head which can burn tiny dots on a heat-sensitive paper. Thermal-transfer printers have a similar print head, but it is used to melt dots of wax-based pigment onto the paper. Electrostatic plotters use an array of wires to which a voltage may be selectively applied. As the paper passes across the wires, it is given a pattern of electrostatic charges. The paper then passes through a liquid toner which is attracted to the pattern on the paper.

As we saw in Chapter 1, one way to produce an image on such raster devices is to first construct it in a frame buffer. The image is constructed by altering the contents of this array, just as may be done for a raster display. The difference from a raster display is that the user will not be able to watch the image being formed. When the user has completed the image, it may output in the order required for printing. These devices require an additional procedure for showing the image.

By creating a frame buffer, alphanumeric terminals and printers may be used for graphics output. They have the same behavior as raster printing devices. The resolution of such devices is usually not very good, but they are readily available and may be sufficient for the user's needs.

PRIMITIVE OPERATIONS

Regardless of the differences in display devices, most graphics systems offer a similar set of graphics primitive commands (although the form of the commands may differ between systems). The first primitive command we shall consider is that for drawing a line segment. While a segment may be specified by its two endpoints, it is often the case that the segments drawn will be connected end to end. (See Figure 2-5.)

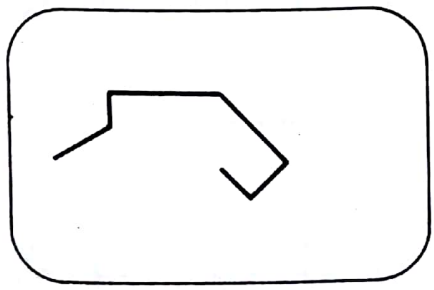


FIGURE 2-5
Connected line segments.

The final point of the last segment becomes the first point of the next segment. To avoid specifying this point twice, the system can keep track of the *current* pen or electron beam position. The command then becomes: draw a line from the current position to the point specified. This is

LINE-ABS-2(X, Y)

and is called an *absolute line command* because the actual coordinates of the final position are passed. (See Figure 2-6.)

There is also a *relative line command*. In this command we only indicate how far to move from the current position (see Figure 2-7):

LINE-REL-2(DX, DY)

The actual endpoint of the segment may be determined from the current position and the relative specification. If we let (XC, YC) denote the current position, then

LINE-REL-2(DX, DY)

is the same as

LINE-ABS-2(XC + DX, YC + DY)

The above procedures are fine for producing a connected string of line segments, but it may happen that we wish to draw two disconnected segments. (See Figure 2-8.)

This can be accomplished by the same mechanism if we picture these two segments as connected by a middle segment which happens to be invisible. We have commands for moving the pen position without leaving a line. Again there can be both absolute and relative moves.

MOVE-ABS-2(X, Y)
MOVE-REL-2(DX, DY)

We can construct a line drawing (say of a house) by a series of line and move commands. If these commands are located in a subprogram, then each time a subprogram is called, an image of the house is produced. If absolute commands are used, then the house will always be located at the same position on the screen. If, however,

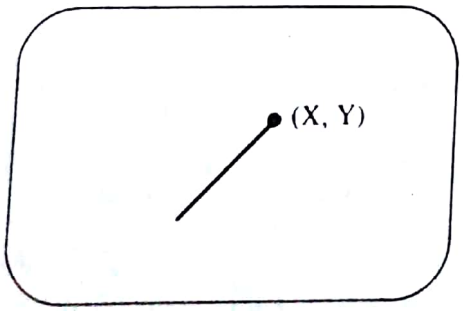


FIGURE 2-6
The absolute line command.

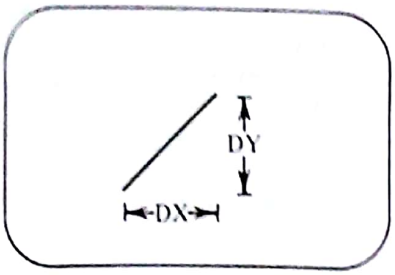


FIGURE 2-7
The relative line command.

only relative commands are used, then the position of the house will depend upon the position of the pen (or beam) at the time when the subprogram was entered. This may be used for the construction of pictures made of repeated instances of basic components. The subprogram for each type of component should be written using only relative commands. Drawing the entire picture is reduced to positioning the beam and calling subprograms. For example, suppose we write the following subprogram for a house.

```
Subprogram House
BEGIN
   LINE-REL-2(0,0.2);
   LINE-REL-2(0.1,0.1);
   LINE-REL-2(0.1,–0.1);
   LINE-REL-2(0,–0.2);
   LINE-REL-2(–0.2,0);
END;
```

This will start at the current pen position, which will become the lower-left corner of the drawing. It will draw the left wall, the roof, the right wall, and, finally, the floor. (See Figure 2-9.)

Since only relative commands were used, we can draw three houses by simply calling this subprogram at three different starting positions. (See Figure 2-10.)

```
BEGIN
   MOVE-ABS-2(0.1,0.2);
   HOUSE;
   MOVE-ABS-2(0.4,0.2);
   HOUSE;
   MOVE-ABS-2(0.7,0.2);
   HOUSE;
END;
```

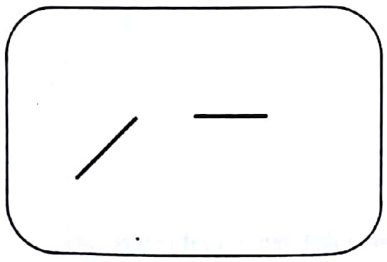


FIGURE 2-8
Disconnected line segments.

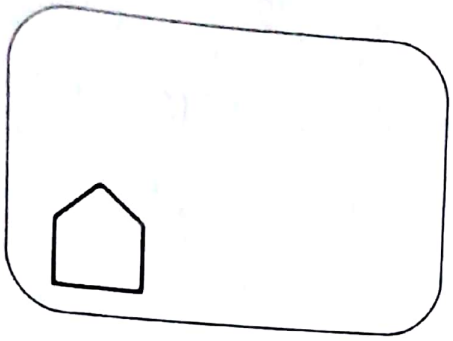


FIGURE 2-9
Subprogram HOUSE.

THE DISPLAY-FILE INTERPRETER

While it might be possible to have the LINE command directly alter the frame buffer, we shall not organize our system in this manner. Instead, we shall insert an intermediate step. We shall have LINE and MOVE commands store their information in what we shall call our display file. We shall then use the information in the display file to create the image. There are several reasons for using this two-step process. First, not every display device has a frame buffer. Different display devices require different programs to drive them. By isolating the driving program in the second of the two steps, we achieve some measure of device independence. Second, it will make it easy for us to change the position, size, and orientation of the image. These image transformations will be carried out during the second step. The techniques involved will be covered in Chapter 4. For now, we shall concentrate on the form of the display file. In Chapter 5 we shall learn how to structure the image and to carry out transformations on portions of it. The display file will support this structuring.

The display file will contain the information necessary to construct the picture. The information will be in the form of instructions such as "draw a line," or "move the pen." Saving instructions such as these usually takes much less storage than saving the picture itself. These instructions can be thought of as a program for creating the image. Each instruction indicates a MOVE, or a LINE, action for the display device. We shall write a *display-file interpreter* to convert these instructions into actual images. (See Figure 2-11.)

Our interpreter may be thought of as a machine which executes these instructions. The result of execution is a visual image. In some graphics systems there is, in fact, a separate computer, called the *display processor*, which is located in the graphics terminal and which is used just for this purpose. In other systems, the behavior of a display processor is simulated. Where there is a separate display processor, some care must be taken when the display file is modified since the display processor may currently be executing the instructions being changed.

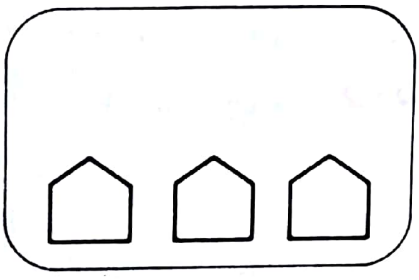


FIGURE 2-10
Three calls of HOUSE, each with a different initial pen position.

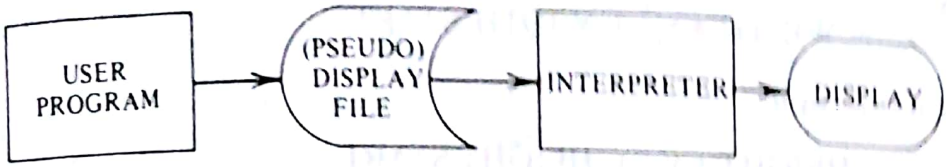


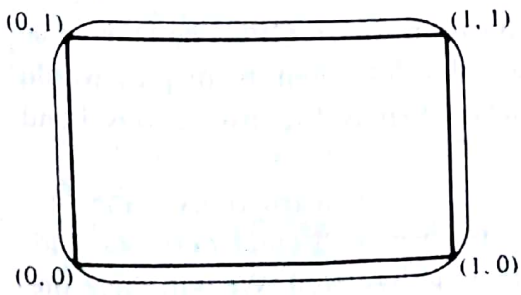
FIGURE 2-11
Display file and interpreter.

Our display-file interpreter serves as an interface between our graphics program and the display device. If we write a graphics program for a particular display device, chances are that it will not run on a different display. The portability of the program is limited. If, on the other hand, we write a program which generates display-file code, all we need is an interpreter for each device which converts our "standard" display instructions to the actions of the particular device. We think of the display device and its interpreter as a machine upon which any standard program may run. The display-file instructions may actually be saved in a file, either for display later or for transfer to another machine. Such files of imaging instructions are sometimes called *metafiles*.

NORMALIZED DEVICE COORDINATES (0 to 1)

Different display devices may have different screen sizes as measured in pixels. If we wish our programs to be device-independent, we should specify the coordinates in some units other than pixels and then use the interpreter to convert these coordinates to the appropriate pixel values for the particular display we are using. The device-independent units are called the normalized device coordinates. In these units, the screen measures 1 unit wide and 1 unit high. The lower-left corner of the screen is the origin, and the upper-right corner is the point (1, 1). The point (0.5, 0.5) is in the center of the screen no matter what the physical dimensions or resolution of the actual display device may be. (See Figure 2-12.)

The interpreter uses a simple linear formula to convert from the normalized device coordinates to the actual device coordinates. Suppose that for the actual display the index of the leftmost pixel is WIDTH-START and that there are WIDTH pixels in the horizontal direction. Suppose also that the bottommost pixel is HEIGHT-START and the number of pixels in the vertical direction is HEIGHT. In the normalized coordinates the screen is 1 unit wide, but in the actual coordinates it is WIDTH units wide so the normalized x position should be multiplied by WIDTH/1 to convert to actual screen units. At position $x_n = 0$ in normalized coordinates we should get x_s = WIDTH-START in actual screen coordinates, so the conversion formula should be



x max value 1
y max value 1

FIGURE 2-12
Normalized device coordinates.

$$x_s = \text{WIDTH} * x_n + \text{WIDTH-START}$$

Similarly for the vertical direction

$$y_s = \text{HEIGHT} * y_n + \text{HEIGHT-START}$$

One problem in setting up the formula to convert from normalized to device coordinates is that the display surfaces are often not square. The ratio of the height to the width is called the display's aspect ratio. If we have a display which is not square, we can either use the display's full height and width in the conversion formula or use numbers which correspond to a square area. If we use the full dimensions, we take full advantage of the display area, but the image will be stretched or squashed. If we use a square area of the display, the image is correctly proportioned, but some of the display area is wasted. If we use a square area larger than the actual display, we may use all of the screen and have a properly proportioned image, but the image may not entirely fit on the display.

DISPLAY-FILE STRUCTURE

Now let us consider the structure of the display file. Each display-file command contains two parts, an *operation code (opcode)*, which indicates what kind of command it is (e.g., LINE or MOVE), and *operands*, which are the coordinates of a point (x, y). The display file is made up of a series of these instructions. One possible method for storing these instructions is to use three separate arrays: one for the operation code (DF-OP), one for the x coordinate (DF-X), and one for the y coordinate (DF-Y). To piece together the seventh display-file instruction, we would get the seventh element from each of the three arrays DF-OP[7], DF-X[7], and DF-Y[7]. The display file must be large enough to hold all the commands needed to create our image.

We must assign meaning to the possible operation codes before we can proceed to interpret them. At this point there are only two possible instructions to consider, MOVE and LINE. We need to consider only absolute MOVE and LINE commands, since relative commands can be converted to absolute commands before they are entered into the display file. Let us define an opcode of 1 to mean a MOVE command and an opcode of 2 to mean a LINE command. A command to move to position x = 0.3 and y = 0.7 would look like 1, 0.3, 0.7. The statements

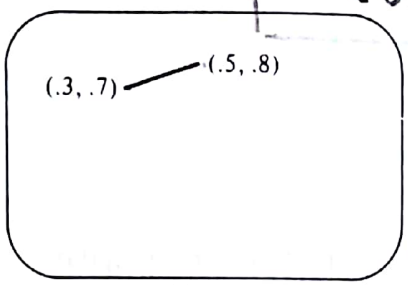
```
DF-OP[3] ← 1;
DF-X[3] ← 0.3;
DF-Y[3] ← 0.7;
```

would store this instruction in the third display-file position. If DF-OP[4] had the value 2, DF-X[4] had the value 0.5, and DF-Y[4] had the value 0.8, then the display would show a line segment from (0.3, 0.7) to (0.5, 0.8) when display-file instructions 3 and 4 were interpreted. (See Figure 2-13.)

Let us develop the algorithms for inserting display-file instructions. Line segments require two endpoints for their specification, but we shall enter only one endpoint and assume that the other endpoint is the current pen position. We will therefore

DF-OP	DF-X	DF-Y
1	0.3	0.7
2	0.5	0.8

Display file instructions



Result

FIGURE 2-13
Display-file instructions.

need variables DF-PEN-X and DF-PEN-Y to keep track of the current pen position. We will need to know this position for the conversion of relative commands to absolute commands. We shall also need a variable FREE to indicate where the next free (unused) cell of the display file is located. These variables, together with the display file itself, are used by several different routines and must be maintained between accesses. They are therefore presented as global variables.

The first algorithm we will consider actually puts an instruction into the display file.

2.1 Algorithm PUT-POINT(OP, X, Y) Place an instruction into the display file

Arguments OP, X, Y the instruction to be entered
Global DF-OP, DF-X, DF-Y the three display-file arrays
FREE the position of the next free cell
Constant DFSIZE the length of the display-file arrays

```
BEGIN
   IF FREE > DFSIZE THEN RETURN ERROR 'DISPLAY FILE FULL';
   DF-OP[FREE] ← OP;
   DF-X[FREE] ← X;
   DF-Y[FREE] ← Y;
   FREE ← FREE + 1;
   RETURN;
END;
```

This algorithm stores the operation code and the coordinates of the specified position in the display file. The pointer FREE to the next free cell is incremented so that it will be in the correct position for the next entry.

We also wish to access elements in the display file. We isolate the accessing mechanism in a separate routine so that any changes in the data structure used for the display file will not affect the rest of our graphics package.

2.2 Algorithm GET-POINT (NTH, OP, X, Y) Retrieve the NTH instruction from the display file

Arguments NTH the number of the desired instruction.
OP, X, Y the instruction to be returned
Global arrays DF-OP, DF-X, DF-Y the display file

```
BEGIN
    OP ← DF-OP[NTH];
    X ← DF-X[NTH];
    Y ← DF-Y[NTH];
    RETURN;
END;
```

Our MOVE and LINE instructions must update the current pen position and enter a command into the display file. If the update of the pen position is done first, then the new pen position will serve as the operand for the display-file instruction. It will prove convenient in later chapters to have a separate routine which takes the operation code and the pen position and enters them into the display file as an instruction.

2.3 Algorithm DISPLAY-FILE-ENTER(OP) Combine operation and position to form an instruction and save it in the display file

```
Argument    OP the operation to be entered
Global      DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
    PUT-POINT(OP, DF-PEN-X, DF-PEN-Y);
    RETURN;
END;
```

Using DISPLAY-FILE-ENTER to place instructions in the display file, the absolute MOVE routine becomes the following:

2.4 Algorithm MOVE-ABS-2(X, Y) User routine to save an instruction to move the pen

```
Arguments   X, Y the point to which to move the pen
Global      DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
    DF-PEN-X ← X;
    DF-PEN-Y ← Y;
    DISPLAY-FILE-ENTER(1);
    RETURN;
END;
```

The point (DF-PEN-X, DF-PEN-Y) is keeping track of where we wish the pen to go. By setting (DF-PEN-X, DF-PEN-Y) to (X, Y), we are saying the pen is to be at position (X, Y).

The algorithm for entering a LINE command is similar.

2.5 Algorithm LINE-ABS-2(X, Y) User routine to save a command to draw a line

```
Arguments   X, Y the point to which to draw the line
Global      DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
    DF-PEN-X ← X;
    DF-PEN-Y ← Y;
    DISPLAY-FILE-ENTER(2)
    RETURN;
END;
```

Again by changing DF-PEN-X and DF-PEN-Y we indicate that the pen will be placed at (X, Y), but by entering an operation code of 2 instead of 1, we instruct the interpreter to draw a line as the pen is moved.

We can also write algorithms for the relative commands.

2.6 Algorithm MOVE-REL-2(DX, DY) User routine to save a command to move the pen

```
Arguments   DX, DY the change in the pen position
Global      DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
   DF-PEN-X ← DF-PEN-X + DX;
   DF-PEN-Y ← DF-PEN-Y + DY;
   DISPLAY-FILE-ENTER(1);
   RETURN;
END;
```

2.7 Algorithm LINE-REL-2(DX, DY) User routine to save a command to draw a line

```
Arguments   DX, DY the change over which to draw a line
Global      DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
   DF-PEN-X ← DF-PEN-X + DX;
   DF-PEN-Y ← DF-PEN-Y + DY;
   DISPLAY-FILE-ENTER(2);
   RETURN;
END;
```

The relative LINE and MOVE routines act like the absolute routines in that they tell where the pen is to be placed and how it is to get there. They differ in that the new pen position is calculated as an offset to the old pen position.

DISPLAY-FILE ALGORITHMS

The above algorithms indicate how to enter instructions into the display file. The other half of the problem is to analyze the display file and perform the commands.

Now let us consider algorithms for our display-file interpreter. The interpreter will read instructions from a portion of the display file and carry out the appropriate LINE and MOVE commands by calls on a vector-generating subroutine such as those described in Chapter 1.

The routine which actually causes the LINE or MOVE to be carried out on the display can depend upon the nature of the display device and upon its software. Below are versions of these algorithms which may be used with a vector generator. The DOMOVE routine has only to update the current pen position. The arithmetic involved is the conversion from normalized device coordinates to the actual screen coordinates.

2.8 Algorithm DOMOVE(X, Y) Perform a move of the pen

```
Arguments   X, Y point to which to move the pen (normalized coordinates)
Global      FRAME-PEN-X, FRAME-PEN-Y the pen position (actual screen coordi-
            nates)
```

Normalized coordinate to change in actual co

```
                WIDTH, HEIGHT the screen dimensions
                WIDTH-START, HEIGHT-START coordinates of the lower-left corner
                WIDTH-END, HEIGHT-END coordinates of the upper-right corner
BEGIN
   FRAME-PEN-X ← MAX(WIDTH-START, MIN(WIDTH-END, X * WIDTH +
                        WIDTH-START));
   FRAME-PEN-Y ← MAX(HEIGHT-START, MIN(HEIGHT-END, Y * HEIGHT +
                        HEIGHT-START));
   RETURN;
END;
```

In this algorithm we see the formula for converting the normalized coordinate values of the arguments into actual screen coordinates. The MAX and MIN functions have been added to the formula as a safeguard. They prevent it from ever generating a value outside the bounds of the actual display. If (X, Y) were to correspond to a point outside the screen area, the MAX and MIN functions would "clamp" the corresponding screen coordinate position to the display boundary. In algorithm 2.8, it was assumed that WIDTH-START was less than WIDTH-END, and HEIGHT-START less than HEIGHT-END; however, this is not the case for some devices. In general, we use the smallest boundary in the MAX test and the largest boundary in the MIN test.

The DOLINE algorithm updates the pen position and calls the Bresenham algorithm (or some other vector generator) to place the line segment in the frame buffer. It, too, performs a conversion from normalized device coordinates to the actual screen coordinates. Since the Bresenham algorithm is used in this example, the coordinates must be rounded to integer pixel positions.

2.9 Algorithm DOLINE(X, Y) This routine draws a line

```
Arguments   X, Y point to which to draw the line (normalized coordinates)
Global      FRAME-PEN-X, FRAME-PEN-Y the pen position (screen coordinates)
            WIDTH, HEIGHT the screen dimensions
            WIDTH-START, HEIGHT-START coordinates of the lower-left corner
            WIDTH-END, HEIGHT-END coordinates of the upper-right corner
            LINECHR the style of the line
Local       X1, Y1 the old endpoint of the line segment
BEGIN
   X1 ← FRAME-PEN-X;
   Y1 ← FRAME-PEN-Y;
   FRAME-PEN-X ← MAX(WIDTH-START, MIN(WIDTH-END, X * WIDTH +
                        WIDTH-START));
   FRAME-PEN-Y ← MAX(HEIGHT-START, MIN(HEIGHT-END, Y * HEIGHT +
                        HEIGHT-START));
   BRESENHAM(INT(X1 + 0.5), INT(Y1 + 0.5),
                        INT(FRAME-PEN-X + 0.5), INT(FRAME-PEN-Y + 0.5),
                        LINECHR);
                        RETURN;
END;
```

Now we can write the interpreter, the routine which examines the display file and calls DOMOVE and DOLINE according to the instructions that it discovers. It will

prove useful in later chapters to be able to interpret just a portion of the display file. We shall therefore pass as arguments to the interpreter the starting position START and how many instructions to interpret COUNT.

2.10 Algorithm INTERPRET(START, COUNT) Scan the display file performing the instructions

```
Arguments   START the starting index of the display-file scan
            COUNT the number of instructions to be interpreted
Local       NTH the display-file index
            OP, X, Y, the display-file instruction
BEGIN
   a loop to do all desired instructions
   FOR NTH = START TO START + COUNT - 1 DO
      BEGIN
         GET-POINT(NTH, OP, X, Y);
         IF OP = 1 THEN DOMOVE(X, Y)
         ELSE IF OP = 2 THEN DOLINE(X, Y)
               ELSE RETURN ERROR 'OP-CODE ERROR';
      END;
   RETURN;
END;
```

A loop steps through all the desired display-file instructions, retrieving them by the GET-POINT routine. The algorithm identifies each instruction as being either a MOVE or a LINE instruction by examining its opcode and calls the routine DOMOVE or DOLINE to actually perform the appropriate action upon the display.

DISPLAY CONTROL

In order to show the picture described in the display file, one might have to do three things. First, the current display may have to be cleared; second, the display file must be interpreted; and third, on some devices (such as line printers and standard CRT terminals) an explicit action is required to show the contents of the frame buffer. For the convenience of the user we shall combine these operations under a single routine. Before presenting the algorithm for this routine, however, we should say a little more about clearing the display (or frame buffer). We may not want to clear the display every time we interpret the display file. If the only changes to the picture have been additions, then there is no need to clear and redraw the rest of the image. In fact, quite complex drawings may be generated a piece at a time. Of course, some changes made to the image do require clearing and redrawing. Deleting a portion of the drawing—or shifting its position, size, or orientation—may require a new image starting with a clear screen. We shall see how to perform these changes in Chapters 4 and 5; the important thing to note here is that the request for a clear screen may be incorporated as part of these changes. In these cases, screen clearing can be done automatically. Still, the user should also be able to explicitly request an erasure of the screen. In Chapter 1 we introduced a display-clearing routine which the user might call; however, sometimes the point at which the user's program discovers that an erasure is needed may

occur before the desired time at which the actual clearing should occur. We shall handle clearing of the frame by means of a flag (a variable with the value true or false). We use a true value to indicate that the screen should be cleared before interpreting the display file and a false value to mean that the display-file instructions may be drawn "on top of" the old image. If the machine discovers that an erasure will be needed, it sets the flag to true. If the user decides that there should be a new frame, he sets the flag to true. Nothing happens to the display until the display file is ready to be interpreted. At this point the erase flag is checked, and if true, the frame is cleared and the flag is changed back to false.

2.11 Algorithm NEW-FRAME User routine to indicate that the frame buffer should be cleared before showing the display file

```
Global      ERASE-FLAG a flag to indicate whether the frame should be cleared
BEGIN
   ERASE-FLAG ← TRUE;
   RETURN;
END;
```

Now we wish to combine the erasing of the frame buffer, if needed, the interpretation of the display file, and the displaying of the new frame buffer into a single routine called MAKE-PICTURE-CURRENT.

2.12 Algorithm MAKE-PICTURE-CURRENT User routine to show the current display file

```
Global      FREE the index of the next free display-file cell
            ERASE-FLAG indicates if frames should be cleared
BEGIN
   IF ERASE-FLAG THEN
      BEGIN
         ERASE;
         ERASE-FLAG ← FALSE;
      END;
   IF FREE > 1 THEN INTERPRET(1, FREE – 1);
   DISPLAY;
   FREE ← 1;
   RETURN;
END;
```

The algorithm first checks for an erasure, as we have discussed. It next checks to be sure the display file is not empty; if it is not, the commands within it are interpreted. Next, any actions necessary to show the results of the interpretation are taken by means of the DISPLAY routine. If a graphics display device which immediately shows the result of each command is used, then the call of DISPLAY is unnecessary or DISPLAY becomes a "dummy" do-nothing routine. Finally, the display-file index is reset to 1, indicating an empty display file ready to accept the next image-drawing commands.

There is one more routine needed to implement this stage of our graphics system. We have some global variables which should be given initial values. This is done by the following algorithm.

2.13 Algorithm INITIALIZE-2A Initialization of variables for line drawings

```
Global      FREE the index of the next free display-file cell
            DF-PEN-X, DF-PEN-Y the display-file pen position
BEGIN
   INITIALIZE-1;
   FREE ← 1;
   DF-PEN-X ← 0;
   DF-PEN-Y ← 0;
   NEW-FRAME;
   RETURN;
END;
```

TEXT

Another primitive operation is that of text output. Most graphics displays involve textual, as well as graphical, data. Labels, instructions, commands, values, messages, and so on, may be presented with the image. The primitive command involved here is the output of a character or a string of characters. While one usually has a command available for output of an entire string, there may sometimes be only a command available for output of a single character. A procedure to apply such a primitive to an entire string is not difficult to construct. The characters themselves may be drawn by either the dot-matrix or the stroke method. Their patterns are often copied from memory into the frame buffer or created by special character generation hardware, although software subroutines using line segments are also to be found. The advantage of hardware is speed and a saving of display-file memory. With sophisticated displays there may be options which the user must specify. Among such options are the spacing of the characters, the size of the characters, the ratio of their height to their width, the slope of the character string, the orientation of the characters, and, possibly, which font is to be used.

We will extend our interpreter to include the output of text. We will do this by extending the number of operation codes to include one code for each character. The operand for an instruction will determine where the character will be placed on the screen. We shall use as the opcode for a character the negative of its ASCII character code. (See Figure 2-14.) Our opcodes will range between –32 and –126 inclusive. Using the ASCII character code should facilitate conversion between the instruction and the character value for output in most systems. We shall exclude the ASCII codes for control characters so that a valid code must be less than –31 (the codes 0 through –31 will be used for line and polygon style changes). Character codings other than ASCII can be used, provided that the codes can be mapped to values less than –31.

A character command, then, has an opcode that specifies which character is to be displayed and x, y operands which tell where on the screen that character should be placed. A word or string of text is stored as a sequence of individual character instructions. Since we save the position of each character, we have control over the character spacing within the string. Since we save y position as well as x position, we can cause our strings to be written vertically or diagonally.

We have not introduced any mechanism for specifying the character orientation. Some display devices allow characters to be drawn at any angle. Others allow only 90-

-32 to -126 are reserved for character.

space	32	8	56	P	80	h	104
!	33	9	57	Q	81	i	105
"	34	:	58	R	82	j	106
#	35	;	59	S	83	k	107
$	36	<	60	T	84	l	108
%	37	=	61	U	85	m	109
&	38	>	62	V	86	n	110
'	39	?	63	W	87	o	111
(	40	@	64	X	88	p	112
)	41	A	65	Y	89	q	113
*	42	B	66	Z	90	r	114
+	43	C	67	[	91	s	115
,	44	D	68	\	92	t	116
-	45	E	69	]	93	u	117
.	46	F	70	^	94	v	118
/	47	G	71	_	95	w	119
0	48	H	72	`	96	x	120
1	49	I	73	a	97	y	121
2	50	J	74	b	98	z	122
3	51	K	75	c	99	{	123
4	52	L	76	d	100	|	124
5	53	M	77	e	101	}	125
6	54	N	78	f	102	~	126
7	55	O	79	g	103		

FIGURE 2-14
ASCII character codes.

degree increments, while many (such as normal output devices) display only upright characters. To maintain device independence and to try to simplify things a little, we shall require all characters within a string to be oriented in the same direction and, in fact, will design our algorithms for a device which allows only upright characters. Writing a string of text can be rather cumbersome if each character requires its own procedure call. We shall therefore write a routine for placing entire strings of text into the display file through a series of character commands. It will be up to this routine to automatically shift the position of each character to achieve the desired spacing. The spacing between characters will be given by global variables XCHRSP and YCHRSP. Of course, we need a routine to set these spacing parameters to whatever values we desire. The CORE and GKS graphics systems have rich selections of text formatting operations; they allow changing the size, orientation, spacing, and font of the character and also the direction of the line of text. We will not be that ambitious here. We shall only control the character spacing and line direction. We shall determine the direction of the line of text from the orientation of the characters. There is a command SET-CHARUP(DX, DY) in which [DX, DY] is a vector specifying the "UP" direction for the characters. The text line then prints to the "right" of this direction. For our system, we will not try to change character orientation with respect to line direction, so a SET-CHARUP command looks a bit awkward. Nevertheless, we shall use it because of its correspondence to graphics standards.

Spacing in the direction of the text line can be changed by the SET-CHARSPACE(CHARSPACE) command. The system will space over one character width automatically. The CHARSPACE parameter indicates any additional spacing. This spacing is specified in units of character size, so a CHARSPACE of 0.5 would mean spacing an additional one-half character (a total of 1.5 character widths between character centers). The CHARSPACE parameter is measured in terms of character

width instead of normalized device coordinates, so a widely spaced line on one device will look just as widely spaced on every other device, even though the width of the characters (in normalized device coordinates) may differ. Our problem is to convert the CHARUP and CHARSPACE specifications, and our knowledge about the size of a character, into distances to step in the x and y direction between each character in the string (XCHRSP, YCHRSP). (See Figure 2-15 and Figure 2-16.)

The calculation of step size is complicated by the fact that we are not rotating the character symbols and that the width and height of a character are often unequal. If the text string is to be printed horizontally, then we shall step by the character width. If the string is to be printed vertically, then character height should be the default step size. At other orientations, we wish to use some default step size between the width and height. To use as much of the width as we are stepping horizontally and as much of the height as we are stepping vertically, we calculate a default step size as

$$\text{DEFAULT-STEP} = \frac{|(\text{CHAR-WIDTH})(\text{DY})| + |(\text{CHAR-HEIGHT})(\text{DX})|}{(\text{DX}^2 + \text{DY}^2)^{1/2}} \qquad (2.1)$$

Here DX and DY are from the CHARUP vector, and are perpendicular to the line direction; that is why they appear to be reversed.

This default step size can be increased by the user's CHAR-SEPARATION factor.

$$\text{TRUE-STEP} = \text{DEFAULT-STEP}\,(1 + \text{CHAR-SEPARATION}) \qquad (2.2)$$

Finally, we decompose the step size into horizontal and vertical components. Once again, since [DX, DY] is a vector perpendicular to the direction of the string, DY will act as the x component and – DX will behave as the y component.

$$\begin{aligned} \text{XCHRSP} &= \text{TRUE-STEP}\,\frac{\text{DY}}{(\text{DX}^2 + \text{DY}^2)^{1/2}} \\ \text{YCHRSP} &= \text{TRUE-STEP}\,\frac{-\text{DX}}{(\text{DX}^2 + \text{DY}^2)^{1/2}} \end{aligned} \qquad (2.3)$$

This calculation is carried out a bit more efficiently in the following algorithm. (You may notice that upon combining Equations 2.1, 2.2, and 2.3, the square root operation drops out.)

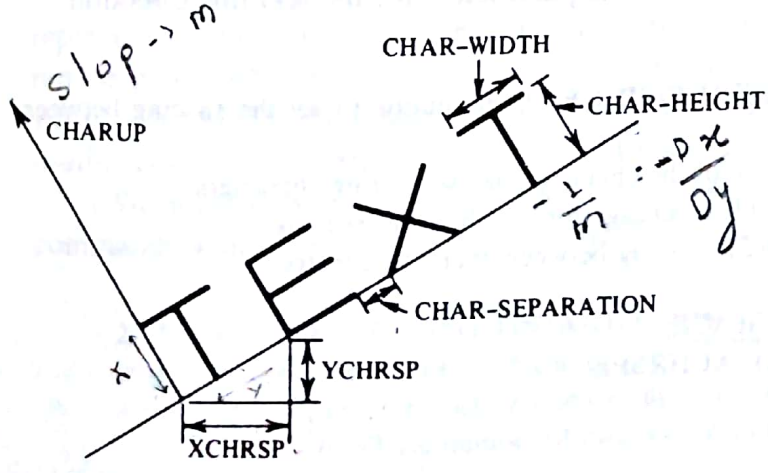


FIGURE 2-15
Text parameters for orientable characters.

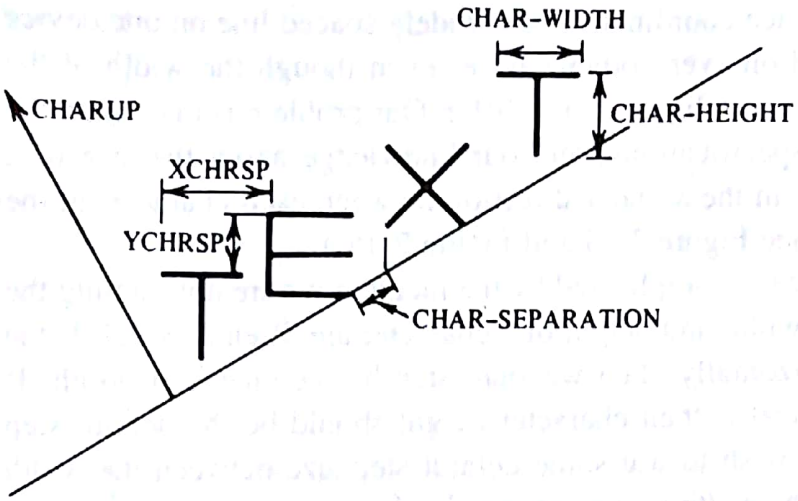


FIGURE 2-16
Text parameters for nonorientable characters.

2.14 Algorithm SET-CHARUP(DX, DY) User routine to indicate in which direction a string should be printed

```
Arguments   DX, DY a vector for the up direction of the string
Global      XCHRSP, YCHRSP spacing between character centers
            CHAR-WIDTH, CHAR-HEIGHT the character size
            CHAR-SEPARATION the proportion of character size separating characters
Local       S, S1, S2 temporary variables used to hold partially completed calculations
Constant    ROUNDOFF some small number greater than any round-off error
BEGIN
   S ← DX ↑ 2 + DY ↑ 2;
   IF S < ROUNDOFF THEN RETURN ERROR 'NO CHARUP DIRECTION';
   S1 ← (|CHAR-WIDTH * DY| + |CHAR-HEIGHT * DX|);
   S2 ← S1 * (1 + CHAR-SEPARATION) / S;
   XCHRSP ← DY * S2;
   YCHRSP ← – DX * S2;
   RETURN;
END;
```

The SET-CHARSPACE routine should not only change the global CHAR-SEPARATION value but also update the step size currently in use. In the arguments of the call to SET-CHARUP, we again see the switch of x and y which comes from specifying the "up" direction in terms of the parameters for the text line direction.

2.15 Algorithm SET-CHARSPACE(SPACE) User routine to set the spacing between characters

```
Argument    SPACE the fraction of the character size separating characters
Global      CHAR-SEPARATION storage for the character spacing
            XCHRSP, YCHRSP spacing between character centers
BEGIN
   CHAR-SEPARATION ← SPACE;
   SET-CHARUP( – YCHRSP, XCHRSP);
   RETURN;
END;
```

Now let's write the algorithm to enter a string of text into the display file, beginning at the current pen position.

2.16 Algorithm TEXT(STRING) User routine to place instructions for printing a string into the display file

```
Argument   STRING the characters to be entered
Global     DF-PEN-X, DF-PEN-Y the pen position
           XCHRSP, YCHRSP the character spacing
Local      LEN the length of the string.
           X, Y the character position
           I an index to count off the characters
           CHR the character being saved
           OP the character's operation-code
BEGIN
   determine the length of the string
   LEN ← LENGTH(STRING)
   save the pen position for restoration after string is entered
   X ← DF-PEN-X;
   Y ← DF-PEN-Y;
   enter the string
   FOR I = 1 TO LEN DO
      BEGIN
         consider the ith character of the string
         CHR ← GETCHR(STRING, I);
         form its character code
         OP ← - DECODE(CHR);
         enter it into the display file
         DISPLAY-FILE-ENTER(OP);
         move the pen to the next character position
         DF-PEN-X ← DF-PEN-X + XCHRSP;
         DF-PEN-Y ← DF-PEN-Y + YCHRSP;
      END;
   restore the pen to its original position
   MOVE-ABS-2(X, Y);
   RETURN;
END;
```

The routines LENGTH, GETCHR, and DECODE depend upon how strings are represented on the particular system that we are using, and the algorithms for them will not be presented here. The LENGTH routine returns the number of characters in the string. The GETCHR routine returns the ith character in the string. The DECODE routine converts the character to the ASCII code, if necessary.

We must now modify our interpreter to be able to handle these new character commands which we are entering into the display file.

2.17 Algorithm INTERPRET(START, COUNT) (Algorithm 2.10 revisited) Scan the display file performing the instructions

```
Arguments  START the starting index of the display-file scan
           COUNT the number of instructions to be interpreted
```

Start - 10 count - 5 (Terminate pts,)
Pos. (starting loc is 10, 11, 12, 13, 14 that
4 count - 1)


```
Local       NTH the display-file index
            OP, X, Y the display-file instruction
BEGIN
  a loop to do all desired instructions
  FOR NTH = START TO START + COUNT – 1 DO
    BEGIN
      GET-POINT(NTH, OP, X, Y);
      IF OP < –31 THEN DOCHAR(OP, X, Y)
      ELSE IF OP = 1 THEN DOMOVE(X, Y)
      ELSE IF OP = 2 THEN DOLINE(X, Y)
           ELSE RETURN ERROR 'OP-CODE ERROR';
    END;
  RETURN;
END;
```

The instructions for actually putting the character into the frame buffer have been placed in the DOCHAR routine.

2.18 Algorithm DOCHAR(OP, X, Y) Place a character on the screen

```
Arguments   OP indicates which character should be used
            X, Y indicate the position on the screen
Global      WIDTH, HEIGHT the screen dimensions
            WIDTH-START, HEIGHT-START screen coordinates of lower-left corner
            WIDTH-END, HEIGHT-END screen coordinates of upper-right corner
Local       CHR the character to be displayed
            X1, Y1 screen coordinates of the character
BEGIN
  CHR ← CODE(–OP);
  X1 ← MAX(WIDTH-START, MIN(WIDTH-END, X * WIDTH +
       WIDTH-START));
  Y1 ← MAX(HEIGHT-START, MIN(HEIGHT-END, Y * HEIGHT +
       HEIGHT-START));
  GENERATE-CHAR(X1, Y1, CHR);
  RETURN;
END;
```

Once again we have system-dependent routines, which are named CODE and GENERATE-CHAR. CODE converts from ASCII to whatever form is convenient for GENERATE-CHAR. GENERATE-CHAR generates a character on the screen or in the frame buffer at position (X1, Y1).

Again we have introduced some global variables which should be initialized. The following routine takes care of this.

2.19 Algorithm INITIALIZE-2B Initialization of character parameters

```
Global      CHAR-WIDTH, CHAR-HEIGHT the character size
            CHAR-SEPARATION the proportion of character size to use for additional
            character spacing
```

```
BEGIN
    INITIALIZE-2A;
    CHAR-WIDTH ← the width of a character in normalized coordinates;
    CHAR-HEIGHT ← the height of a character in normalized coordinates;
    CHAR-SEPARATION ← 0;
    SET-CHARUP(0, 1);
    RETURN;
END;
```

THE LINE-STYLE PRIMITIVE

Many display devices offer a selection of line styles. Lines may be continuous, or they may be dashed or dotted. One may be able to select the color of the line or its intensity or thickness. It is desirable to be able to change the line style in the middle of the display process. We therefore need a display-file command for changing line style. When the interpreter encounters such a command, the line style is changed and all subsequent lines are drawn in this new style. Our display-file commands are composed of three parts, the opcode and the two operands for the x and y coordinates. We can use a special opcode to indicate change of line style (or color or intensity), but such a command would not require any operands. Some possible display-file organizations allow different-sized instructions, but we shall take the simpler alternative of providing dummy operands which are ignored. We will use codes between 0 and –15, inclusive, for change of line-style commands. This allows up to 16 possible line styles. The actual identification of opcodes with line styles will depend upon the possible line styles available. We shall, however, assume that a code of 0 corresponds to the normal straight line. This should be the default style when the system is initialized. Other line styles should correspond to the codes –1, –2, and so on. For a line printer display such as discussed in Chapter 1, the line style is the character that is placed in the frame buffer. Changing the line style is a matter of changing this character. (See Figure 2-17.)

We will augment our interpreter to process commands for changing line style. To do this, we will need two new routines, one to place the line-style commands into the display file and another to actually make the change whenever such a command is discovered by the interpreter.

While line-style opcodes in the display file are negative to distinguish them from the MOVE and LINE opcodes, there is no reason to force the user to use negative numbers in his specification. To do so would give the user one more pointless thing to remember. We therefore have the user specify the positive codes 1,2,... for line styles

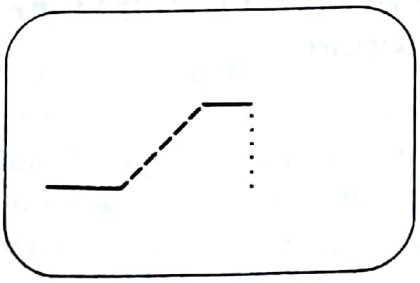


FIGURE 2-17
Changing line style.

and let the SET-LINESTYLE algorithm enter corresponding negatives into the display file. The algorithm for setting the line style would look as follows:

```
2.20 Algorithm SET-LINESTYLE(LSTYLE) User routine for changing line style
Argument    LSTYLE the user's line-style specification
BEGIN
   DISPLAY-FILE-ENTER(1 – LSTYLE);
   RETURN;
END;
```

When the interpreter discovers a line-style command, it must perform the change. We isolate this system-dependent process in a separate routine to allow easy interfacing with different display devices.

```
2.21 Algorithm DOSTYLE(OP) Routine to change the line style
Argument    OP indicates the desired line style
BEGIN
   decode OP;
   set the line style;
   RETURN;
END;
```

If the vector generation algorithms of Chapter 1 are used for a line printer or CRT with different characters representing different line styles, then algorithm 2.21 would be

```
2.21A Algorithm DOSTYLE(OP) Routine to change the line style
Global      LINECHR the line character used by the vector generator
BEGIN
   LINECHR ← LINE-CODE(1 – OP);
   RETURN;
END;
```

where LINE-CODE converts from the integers 1, 2, ... to appropriate character codes, for example '*', '+', ...

We must once again extend our interpreter so that it will now recognize line-style operation codes.

```
2.22 Algorithm INTERPRET(START, COUNT) (Algorithm 2.17 revisited) Scan the
display file performing the instructions
Arguments   START the starting index of the display-file scan
            COUNT the number of instructions to be interpreted
Local       NTH the display-file index
            OP, X, Y the display-file instruction
BEGIN
   a loop to do all desired instructions
   FOR NTH = START TO START + COUNT – 1 DO
```

```
        BEGIN
            GET-POINT(NTH, OP, X, Y);
            IF OP < -31 THEN DOCHAR(OP, X, Y)
            ELSE IF OP < 1 THEN DOSTYLE(OP)
                 ELSE IF OP = 1 THEN DOMOVE(X, Y)
                 ELSE IF OP = 2 THEN DOLINE(X, Y)
                      ELSE RETURN ERROR 'OP-CODE ERROR';
        END;
    RETURN;
END;
```

We shall also make a small change in the DISPLAY-FILE-ENTER algorithm so that it will distinguish between instructions which use the coordinate information (LINE and MOVE) and those which do not (STYLE). This distinction will prove useful in later chapters, where we will not enter all of the line-drawing instructions but will still wish to enter all changes of style.

2.23 Algorithm DISPLAY-FILE-ENTER(OP) (Algorithm 2.3 revisited) Combine operation and position to form an instruction and save it in the display file

```
Argument    OP the operation to be entered
Global      DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
    IF OP < 1 AND OP > -32 THEN PUT-POINT(OP, 0, 0)
    ELSE PUT-POINT(OP, DF-PEN-X, DF-PEN-Y);
    RETURN;
END;
```

There is one more routine which is needed to complete this stage of our system. We must supply initial or default values to our global variables. In addition to the previous initializations, we should now also set the initial line style.

2.24 Algorithm INITIALIZE-2 Initialization of variables for lines, characters, and style

```
BEGIN
    INITIALIZE-2B
    DOSTYLE(0)
    RETURN;
END;
```

AN APPLICATION

Let us suppose that we wanted a graphics program to draw a graph of some data. Let us outline how this might be done using some of the algorithms which we have written. To begin, we can plot the horizontal and vertical axes of the graph. These are just two lines, which may be created by a MOVE-ABS-2 to one endpoint and a LINE-ABS-2 to the other endpoint. (See Figure 2-18.)

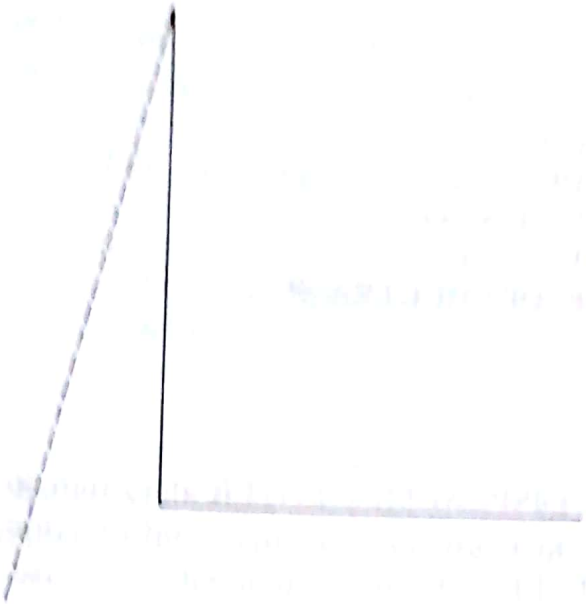


FIGURE 2-18
Drawing two axes.

```
MOVE-ABS-2(0.2, 0.8);
LINE-ABS-2(0.2, 0.2);
LINE-ABS-2(0.8, 0.2);
```

The next step might be to label the axes. This can be done by a series of TEXT commands. For example, suppose we wish to label the horizontal axis with the numbers 1 through 5. For each number, we decide where it should be placed on the graph. We issue a MOVE-ABS-2 to place the pen at this position, followed by a TEXT command to write the numeral. (See Figure 2-19.)

```
MOVE-ABS-2(0.3, 0.15);
TEXT('1');
MOVE-ABS-2(0.4, 0.15);
TEXT('2');
```

We might also want to label each axis with a string indicating what is being plotted. A MOVE-ABS-2 to the starting position followed by a TEXT command will do this for us. (See Figure 2-20.)

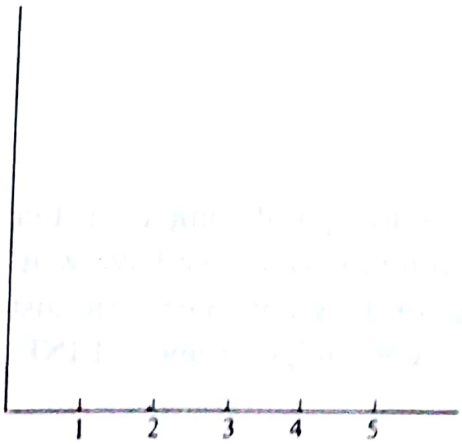


FIGURE 2-19
Labeling an axis.

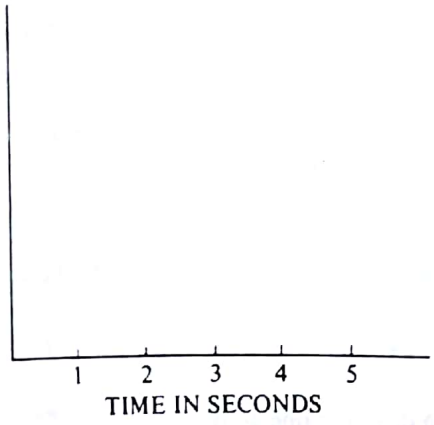


FIGURE 2-20
More labeling.

```
MOVE-ABS-2(0.35, 0.1);
TEXT('TIME IN SECONDS');
```

For the vertical axis, it may be useful to change the CHAR-UP direction so that the print line will be vertical. (See Figure 2-21.)

```
SET-CHARUP(–1, 0);
MOVE-ABS-2(0.1, 0.3);
TEXT('POSITION IN METERS');
```

Now all that is left for us to do is to plot the data. At this point we might use SET-LINESTYLE to change the style of the line so that our data curve looks different from the axes.

We shall assume that the information to be displayed came into our plotting package as an array of data values or measurements. For each of these measurements we may have to perform some arithmetic to calculate the corresponding position on our display. If we use MOVE-ABS-2 to plot the first data point, and LINE-ABS-2 on the remaining points, we shall get a sequence of line segments connecting the data values. (See Figure 2-22.)

```
SET-LINE-STYLE(2);
MOVE-ABS-2(0.2, A[1] + 0.2);
```

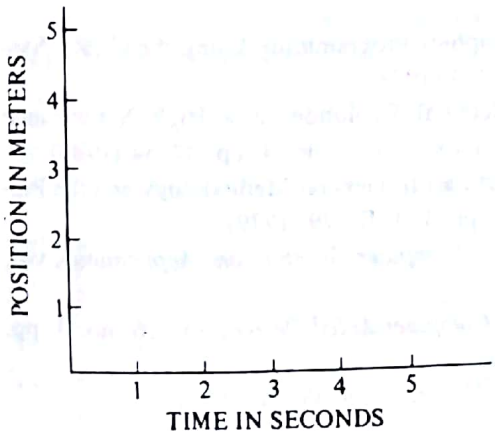


FIGURE 2-21
Labeling with a different CHAR-UP.

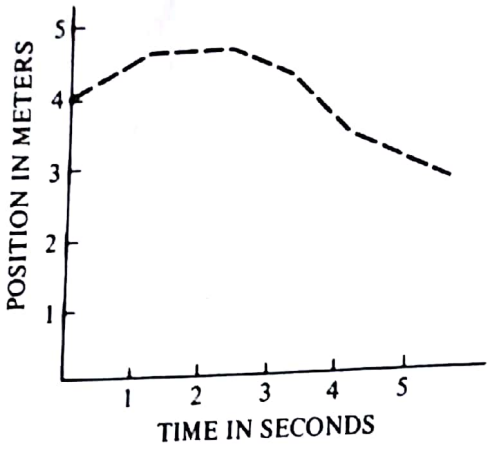


FIGURE 2-22
Plotting the curve with a different line style.

```
FOR I = 2 TO 6 DO
   LINE-ABS-2(0.1 * I + 0.1, A[I] + 0.2);
```

Plotting of all but the first data point can usually be accomplished by a loop which gets the measurement, calculates the corresponding display position, and calls LINE-ABS-2 to display it.

FURTHER READING

Introductory discussions of various imaging technologies may be found in [ALD84], [BAL84], [HOB84], [HUB84], [JER77], [LEE84], [LUC78], [MCC84], [MIL84], [PRE78], [SLO76], [SPR83], [WAT84], [ZAP75], and [ZUC84]. Use of magnetic drum memory as a frame buffer was described in [OPH68], and the first example of the core memory frame buffer is [NOL71]. A frame buffer is also described in [KAJ75]. A display processor and display-file interpreter is described in [DOO84]. Use of a display file for a device-independent system and suggested primitives are presented in [NEW74]. A discussion of graphics primitives is given in [LUC76], and those in the CORE system in [BER78] and [GRA79].

[ALD84] Aldersey-Williams, H., "Liquid Crystals in Flat Panel Displays," *Electronic Imaging*, vol. 3, no. 11, pp. 54–57 (1984).

[BAL84] Baltazzi, E. S., "Reprographic Imaging Techniques," *Electronic Imaging*, vol. 3, no. 12, pp. 53–55 (1984).

[BER78] Bergeron, R. D., Bono, P. R., and Foley, J. D., "Graphics Programming Using the CORE System," *ACM Computing Surveys*, vol. 10, no. 4, pp. 389–394 (1978).

[DOO84] Doornink, D. J., Dalrymple, J. C., "The Architectural Evolution of a High-Performance Graphics Terminal," *IEEE Computer Graphics and Applications*, vol. 4, no. 4, pp. 47–54 (1984).

[GRA79] Graphic Standards Planning Committee, "Status Report Part II: General Methodology and the Proposed Core System," *Computer Graphics*, vol. 13, no. 3, pp. II-1–II-179 (1979).

[HOB84] Hobbs, I. C., "Computer Graphics Display Hardware," *Computer Graphics and Applications*, vol. 1, no. 1, pp. 25–32 (1981).

[HUB84] Hubbold, R. J., "Computer Graphics and Displays," *Computer-Aided Design*, vol. 16, no. 3, pp. 127–133 (1984).

[JER77] Jern, M., "Color Jet Plotter," *Computer Graphics*, vol. 11, no. 1, pp. 18–31 (1977).