CHAPTER SEVEN

INTERACTION

INTRODUCTION

Computer graphics gives us added dimensions for communication between the user and the machine. Data about size, shape, position, and orientation can be presented to the user in a natural manner. Complex organizations and relationships can be conveyed clearly to the user. But communication should be a two-way process. It is desirable to allow the user to respond to this information. The most common form of computer output is a string of characters printed on the page or on the surface of a CRT terminal. The corresponding form of input is also a stream of characters coming from a keyboard. Computer graphics extends the form of output to include two-dimensional images, lines, polygons, arcs, colors, and intensities. The graphic display can show the positions of objects and the relationships between objects. We might ask what is the appropriate form of input for a user's response to these images. First consider what sorts of responses a user might wish to make to such a display. The user may wish to select a particular object on the display. He may wish to specify a position on the display. He may wish to alter an object's orientation, size, or location. He may wish to enter a new object onto the display. We find the usual keyboard character input to be quite unnatural for these functions. The user does not wish to determine the coordinates of some point relative to some reference system and then enter them into the machine as decimal digits; he would much rather just point to the position he is interested in. It is awkward to create a drawing by typing coordinates of the endpoints of line segments or by typing instructions for moving the imaginary pen. Most users would prefer to be able to take a real pen and move it across the screen. Some effective methods have been developed for the input of graphic information. A user might move a penlike stylus and see lines appear on the display as if real pen and ink were being used. He may "attach" a portion of an image to the stylus and reposition it by moving the stylus. Scale and orientation might be altered by pushing levers or twisting dials. A portion of the display may be selected by just pointing at it. In this chapter we shall consider the devices which allow the user to respond to graphical information in a natural manner. We shall also consider some of the techniques which may be employed to take advantage of this interaction between man and machine.

HARDWARE

Various hardware devices have been developed to enable the user to interact in this more natural manner. These devices can be separated into two classes, *locators* and *selectors*. Locators are devices which give position information. The computer typically receives from a locator the coordinates for a point. Using a locator we can indicate a position on the screen. Selector devices are used to select a particular graphical object. A selector may pick a particular item but provide no information about where that item is located on the screen.

Let us first consider some locator devices. One example of a locator is a pair of *thumbwheels* such as are found on the Tektronix 4010 graphics terminal. These are two potentiometers mounted on the keyboard, which the user can adjust. One potentiometer is used for the x direction, the other for the y direction. Analog-to-digital converters change the potentiometer setting into a digital value which the computer can read. The potentiometer settings may be read whenever desired. The two potentiometer readings together form the coordinates of a point. To be useful, this scheme must also present the user with information as to which point the thumbwheels are specifying. Some *feedback* mechanism is needed. This may be in the form of a special *screen cursor*, that is, a special marker placed on the screen at the point which is being indicated. It might also be done by a pair of *cross hairs* which cross at the indicated point. As a thumbwheel is turned, the marker or cross hair moves across the screen to show the user which position is being read. (See Figure 7-1.)

Another locator device is a *joystick*. A joystick has two potentiometers, just as a pair of thumbwheels. However, they have been attached to a single lever. Moving the

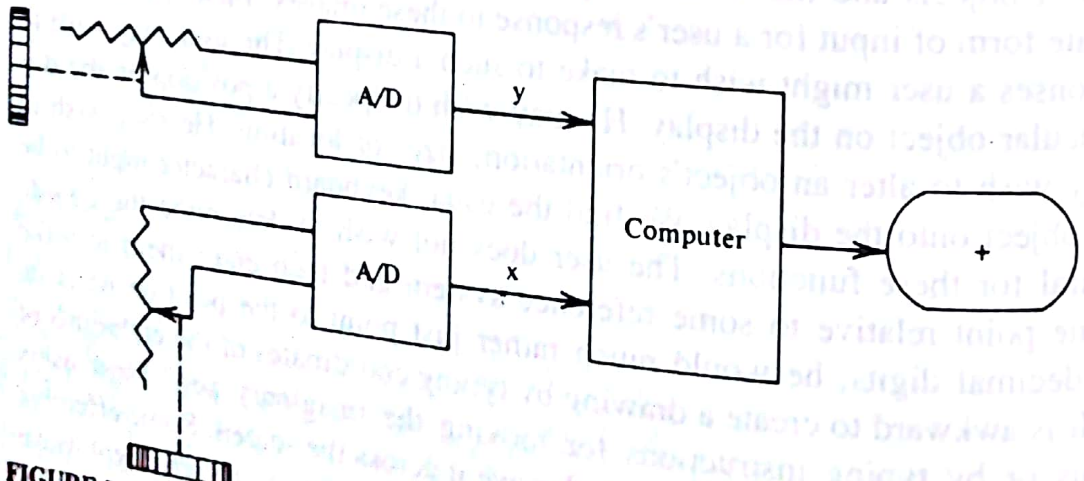

FIGURE 7-1
Thumbwheel entry of position.

lever forward or back changes the setting on one potentiometer. Moving it left or right changes the setting on the other potentiometer. Thus with a joystick both x- and y-coordinate positions can be simultaneously altered by the motion of a single lever. The potentiometer settings are processed in the same manner as they are for thumbwheels. Some joysticks may return to their zero position when released, whereas thumbwheels remain at their last position until changed. Joysticks are inexpensive and are quite common on displays where only rough positioning is needed. (See Figure 7-2.)

Some locator devices use switches attached to wheels instead of potentiometers. As the wheels are turned, the switches produce pulses which may be counted. The count indicates how much a wheel has rotated. This mechanism is often found in *mice* and *track balls*. A mouse is a palm-sized box with a ball on the bottom connected to such wheels for the x and y directions. As the mouse is pushed across a surface, the wheels are turned, providing distance and direction information. This can then be used to alter the position of a cursor on the screen. A mouse may also come with one or more buttons which may be sensed. A track ball is essentially a mouse turned upside down. The ball which turns the wheels is large and is moved directly by the operator. (See Figure 7-3.) There are also mice which use photocells rather than wheels and switches to sense position. An optical mouse is moved across a surface which contains a special grid pattern. Photocells in the bottom of the mouse sense the movement across the grid and produce pulses to report the motion.

If we had a paper drawing or a blueprint which we wished to enter into the machine, we would find that the joystick was not very useful. Although the joystick could indicate a position on the screen, it could not match the screen position to the corresponding blueprint position. For applications such as tracing we need a device called a *digitizer*, or a *tablet*. A tablet is composed of a flat surface and a penlike stylus or windowlike tablet cursor. (See Figure 7-4.) The tablet is able to sense the position of the *stylus* or *tablet cursor* on the surface. A number of different physical principles have been employed for the sensing of the stylus. Most do not require actual contact between the stylus and the tablet surface, so that a drawing or blueprint might be placed upon the surface and the stylus used to trace it. A feedback mechanism on the screen is

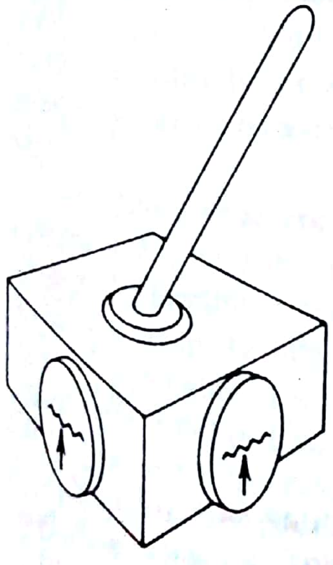

FIGURE 7-2
Joystick.

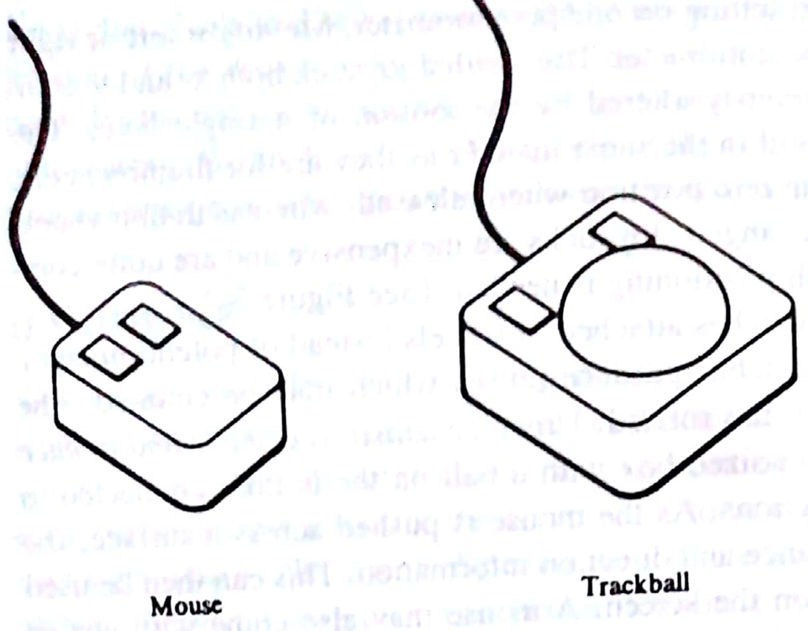

FIGURE 7-3
Mouse and track ball.

not as necessary for a graphics tablet as it is for a joystick because the user can look at the tablet to see what position he is indicating. Nevertheless, if tablet entries are to be coordinated with items already on the screen, then some form of feedback, such as a screen cursor, is useful.

The user may not wish to have every stylus position entered into the machine. Some of the time he may be moving the stylus about in order to position it for the next entry. It is therefore desirable to have some means of turning the tablet off and on, so that the computer can only read coordinate values when the user is ready. A convenient way of doing this is to build a switch into the tip of the stylus which turns on only when the stylus is pressed down. The user can then lift the stylus, position it, and press down to enter a point.

An example of a selector device is a *light pen*. A light pen is composed of a photocell mounted in a penlike case. (See Figure 7-5.) This pen may be pointed at the screen on a refresh display. The pen will send a pulse whenever the phosphor below it is illuminated. While the image on a refresh display may appear to be stable, it is in fact blinking on and off faster than the eye can detect. This blinking is not too fast, however, for the light pen. The light pen can easily determine the time at which the

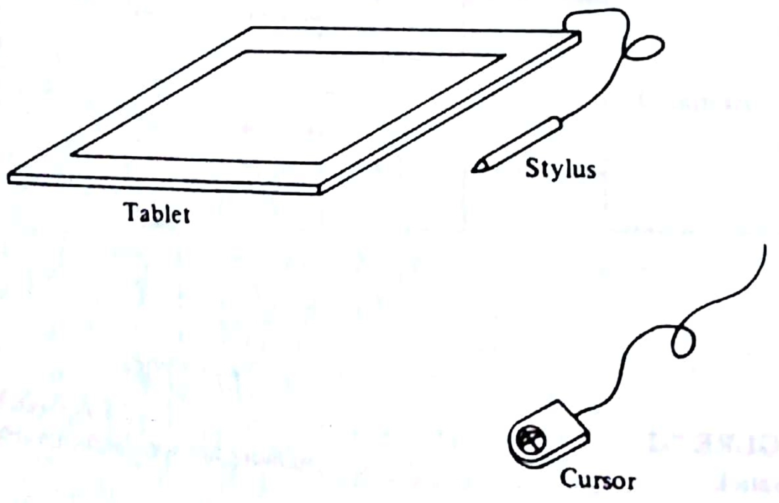

FIGURE 7-4
Tablet.

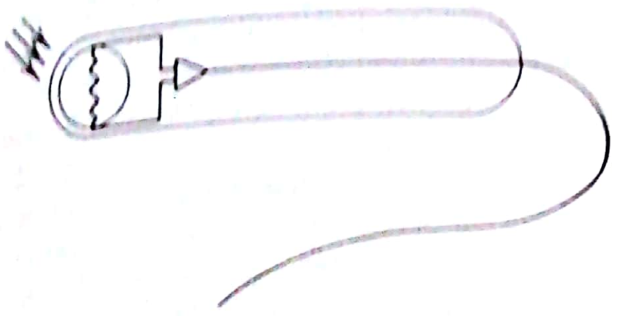

FIGURE 7-5
Light pen.

phosphor is illuminated. Since there is only one electron beam on the refresh display, only one line segment can be drawn at a time and no two segments are drawn simultaneously. When the light pen senses the phosphor beneath it being illuminated, it can interrupt the display processor's interpreting of the display file. The processor's instruction register tells which display-file instruction is currently being drawn. Once this information is extracted, the processor is allowed to continue its display. Thus the light pen tells us which display-file instruction was being executed in order to illuminate the phosphor at which it was pointing. By determining which part of the picture contained the instruction that triggered the light pen, the machine can discover which object the user is indicating. It is often possible to turn the interrupt mechanism on or off during the display process and thereby select or deselect objects on the display for sensing by the light pen.

A light pen is an example of an *event-driven* device. Unlike a locator, which can be sampled at any time, the processor must wait on the light pen until it encounters an illuminated phosphor on the screen. The computer must wait for this to happen before it can obtain the desired information. A keyboard is a more familiar example of an event-driven device. The machine must wait for a key to be pressed before it can determine what character the user wishes to send. Buttons and switches may also be available as input devices. Again, these are event-driven devices. Handling of event-driven devices can be conducted in two different ways. The first is to enter a *polling loop*, that is, a loop which checks the status of the device until an event is indicated. When an event occurs, the loop is exited with the results of the event. (See Figure 7-6.) Reading of terminal input is often handled this way in high-level languages. A disadvantage of this scheme is that the processor is essentially idle while it is waiting for an event. This would be disastrous on a device where the processor is also needed to maintain the display.

An alternative approach is to enable an *interrupt* which is sent by the device when an event occurs. An interrupt is an electrical signal which causes the processor to break the normal execution sequence and transfer control to a special interrupt-handling routine. By using the interrupt mechanism, the processor is free to carry out some other operation while it is waiting for an event to take place. When the event occurs, processing is interrupted and the device which caused it is serviced. After servicing the device, processing can continue. (See Figure 7 -7.)

The interrupt approach to event handling allows handling of events which may occur at times other than when the processor is expecting them. Suppose that one is

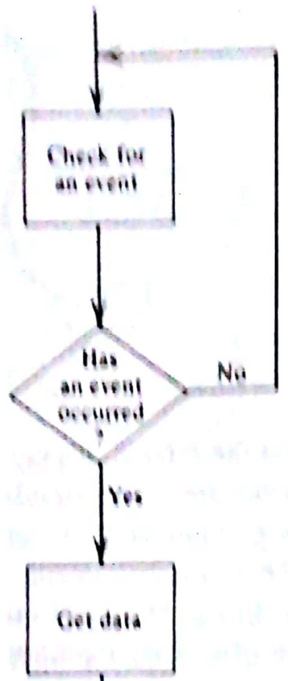

FIGURE 7-6
A polling loop.

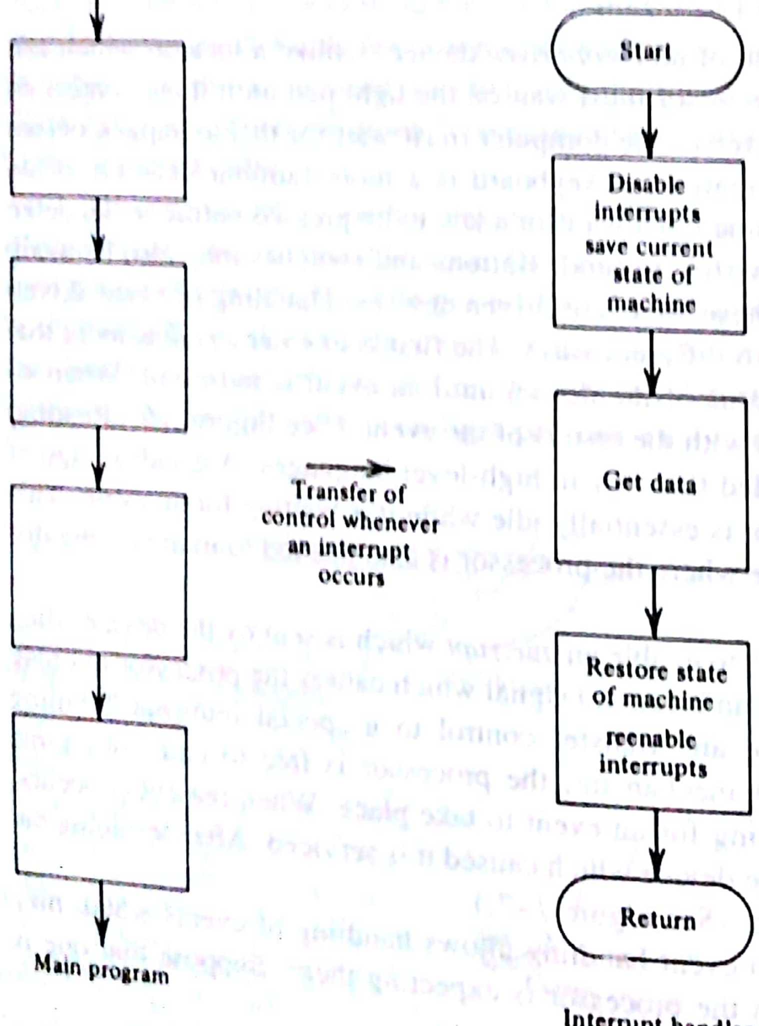
Start
Disable interrupts save current state of machine
Get data
Restore state of machine reenable interrupts
Return
Transfer of control whenever an interrupt occurs
Main program
Interrupt handler

FIGURE 7-7
An interrupt scheme for retrieving input data.

using a program which presents a display and then requests a keyboard input from the user. The familiar user might anticipate the correct keyboard input and enter it before the system has finished processing and presenting the display. If polling techniques are used, this input could be lost because the computer is not ready for it. An interrupt scheme, on the other hand, would temporarily stop processing and store the input so that it can be reexamined at the appropriate time. The processing of events is broken into two parts. When events occur, the associated information is entered onto an *event queue*. Several events could occur before the program is ready for them, but they would all be stored and would therefore be accessible. When information from an event-driven device is finally needed, the event queue is searched to see what events have transpired. If an appropriate event has occurred, the information is removed from the queue. If no event has occurred and the queue is empty, a polling loop can still be employed to repeatedly check the status of the event queue, instead of checking the status of individual devices.

We may have more than one event-driven device causing interrupts. Input data from different devices may have different sizes or forms. We may wish, therefore, not to save this information directly in a queue, but in some other location. We can use the queue entry to save a "pointer" which tells us where to find the information for the event. Another alternative is to have a separate specialized queue for each class of device.

Seeking the information about an event is a two-step operation. The first step is to examine the queue to determine whether the event occurred. The second step is to recover any information which may have been stored when the event transpired. Thus the first step is to await the event, whereas the second is to get the results.

INPUT DEVICE-HANDLING ALGORITHMS

Once again we are dealing with an area which depends heavily on the particular hardware devices available. The graphics standards achieve device independence by specifying what routines should be available and what they should do (but not how they do it); a user can then depend on these features being available. The system acts as an interface between a device-independent user program and the particular devices available. The "insides" of the interface routines will depend on the devices, but the "outsides" which the user sees will always look the same. This is great for the user but difficult for us, because we cannot just give an algorithm. Each particular input device will have its own version of the algorithm. This will make the routines of this chapter a bit more nebulous than previous routines. Nevertheless, we can state in general terms what we want our algorithms to do. All graphic input may be simulated using only a keyboard device if necessary.

For the keyboard simulation, the internal form of the routines is substantially altered from that of the general case. This is because of two factors. First, input from a keyboard via a high-level language appears to be a sampled, rather than event-driven, mechanism. A READ may be done at any time and will always return some value. In effect, processing is suspended until the input is obtained. So, like a sampled device, whenever a READ occurs, a value is returned. To act as an event-driven device, the READ statement would have to return whether or not some new information has been placed in an input buffer. But this is just not the case for most high-level languages. The second area of difference is the fact that we can assume that there is only one actual device and, therefore, need not worry about problems such as what to do when two events occur at the same time.

Let us consider the general forms of routines to handle input devices. We shall consider five classes of devices. The first is the *button*. The button is an event-driven device that sends an interrupt when it is pressed. It sends no information other than the fact that it has been pressed. The second class of device is the *pick*. The pick is typified by a light pen. It is an event-driven device which selects some portion of the display. The third class is the *keyboard*, and the fourth is the *locator*. A locator is a sampled device that returns the coordinates of some position. Finally, we shall include a *valuator*, which, like a locator, is a sampled device, but which returns only one value. A valuator might be a dial or knob which can be set by the user. There is nothing to prevent a system from having several buttons, light pens, keyboards, locators, or valuators. Therefore, to specify a particular piece of hardware we have to indicate not only its class but also which member of that class the device happens to be.

We assume that some mechanism is available for turning these devices logically on or off, so that routines should be provided which allow the user to *enable* or *disable* each device. When a device is disabled, its inputs, if any, are ignored. The user must, therefore, enable the device before it may be used.

Let us begin our discussion of the interactive graphics routines with the enabling and disabling of devices. Instead of enabling and disabling individual devices, we shall simplify this process to the enabling and disabling of device classes. We shall create a flag for each class. The setting of the flags will determine whether or not a particular class is enabled. We can give numerical names to the various classes to aid in their specification. (See Table 7-1.) The disable routine, then, takes as its argument a class number, performs whatever actions are needed to turn this class of device logically off, and sets the corresponding device flag to false. The enable routine similarly takes a class number, performs any necessary actions to turn the devices logically on, and sets the corresponding device flag to true.

```
7.1 Algorithm ENABLE-GROUP(CLASS) Routine to enable an input device class
Argument    CLASS the code for the class to be enabled
Global      BUTTON, PICK, KEYBOARD, LOCATOR, VALUATOR device flags
BEGIN
    IF CLASS = 1 THEN
```

TABLE 7-1
Input class numbering

Input device type	Class number
Button	1
Pick	2
Keyboard	3
Locator	4
Valuator	5

```
        BEGIN
            PERFORM ALL OPERATIONS NEEDED TO PERMIT INPUT FROM THE
                BUTTON DEVICES;
            BUTTON ← TRUE;
        END;
    IF CLASS = 2 THEN
        BEGIN
            PERFORM ALL OPERATIONS NEEDED TO PERMIT INPUT FROM THE
                PICK DEVICES;
            PICK ← TRUE;
        END;
    IF CLASS = 3 THEN
        BEGIN
            PERFORM ALL OPERATIONS NEEDED TO PERMIT INPUT FROM THE
                KEYBOARDS;
            KEYBOARD ← TRUE;
        END;
    IF CLASS = 4 THEN
        BEGIN
            PERFORM ALL OPERATIONS NEEDED TO PERMIT INPUT FROM THE
                LOCATOR DEVICES;
            LOCATOR ← TRUE;
        END;
    IF CLASS = 5 THEN
        BEGIN
            PERFORM ALL OPERATIONS NEEDED TO PERMIT INPUT FROM THE
                VALUATOR DEVICES;
            VALUATOR ← TRUE;
        END;
    RETURN;
END;
```

7.2 Algorithm DISABLE-GROUP(CLASS) Routine to disable an input device class

```
Argument    CLASS the code for the class to be disabled
Global      BUTTON, PICK, KEYBOARD, LOCATOR, VALUATOR device flags
BEGIN
    IF CLASS = 1 THEN
        BEGIN
            PERFORM ALL OPERATIONS NEEDED TO PROHIBIT INPUT FROM THE
                BUTTON DEVICES;
            BUTTON ← FALSE;
        END;
    IF CLASS = 2 THEN
        BEGIN
            PERFORM ALL OPERATIONS NEEDED TO PROHIBIT INPUT FROM THE
                PICK DEVICES;
            PICK ← FALSE;
        END;
    IF CLASS = 3 THEN
    BEGIN
      PERFORM ALL OPERATIONS NEEDED TO PROHIBIT INPUT FROM THE
        KEYBOARDS;
      KEYBOARD ← FALSE;
    END;
  IF CLASS = 4 THEN
    BEGIN
      PERFORM ALL OPERATIONS NEEDED TO PROHIBIT INPUT FROM THE
        LOCATOR DEVICES;
      LOCATOR ← FALSE;
    END;
  IF CLASS = 5 THEN
    BEGIN
      PERFORM ALL OPERATIONS NEEDED TO PROHIBIT INPUT FROM THE
        VALUATOR DEVICES;
      VALUATOR ← FALSE;
    END;
  RETURN;
END;
```

We can also provide the user with a single routine for disabling all input devices.

```
7.3 Algorithm DISABLE-ALL  Routine to disable all input devices
Local        CLASS for stepping through the possible classes of devices
BEGIN
  FOR CLASS = 1 TO 5 DO DISABLE-GROUP(CLASS);
  RETURN;
END;
```

EVENT HANDLING

In theory we picture an event-driven device as one which generates an interrupt. When the processor detects the interrupt, it stops its current activity and services the device. When the device has been serviced, normal processing resumes. What is involved in servicing an interrupt? What will an algorithm to do this servicing look like? An outline for such an algorithm is given below. It assumes that there is a single event queue. In general, however, there could be many queues instead of only one. Servicing the interrupt means identifying which device caused the interrupt and obtaining the input data from that device. This information is then stored on the event queue. Strings are handled a little differently because they may require more storage. The string may be stored in a special string storage area. On the event queue, instead of trying to store the entire string in a data field, we store a pointer which tells us where the string may be found.

```
7.4 Algorithm EVENT  This algorithm is a model for the processing of an input-device
interrupt
BEGIN
  DISABLE THE PHYSICAL INTERRUPT;
  to prevent interruption of the interrupt processing
```

```
    SAVE PROCESSOR STATUS;
    DETERMINE WHICH DEVICE (CLASS AND NUMBER) CAUSED THE
       INTERRUPT;
    IF DEVICE IS LOGICALLY ENABLED THEN
       BEGIN
          IF EVENT FROM A STRING INPUT DEVICE THEN
             BEGIN
                GET THE INPUT STRING;
                ADD-STRING-Q(STRING, DATA);
             END
          ELSE GET THE DATA FROM THE DEVICE;
          ADD-EVENT-Q(CLASS, NUMBER, DATA);
       END;
    RESTORE PROCESSOR STATUS;
    REENABLE PHYSICAL INTERRUPT;
    RETURN;
END;
```

A *queue* is a first-in–first-out data structure like a ticket line. The first one in line is the first to get a ticket. As new patrons arrive, they take their place at the rear of the line. We want our algorithm to add new events to the rear of the queue. We set up arrays to hold the queue data and a pointer QREAR which tells us at which point the last entry was made. We increment this position to store the next event (if we step past the end of the array, we wrap around to the first array position). We also have a pointer to the next element to be removed from the queue, QFRONT. A special value of QFRONT = 0 means that the queue is empty, so this should be checked and set to 1 when we enter the first queue element. (See Figure 7-8.)

7.5 Algorithm ADD-EVENT-Q(CLASS, NUMBER, DATA) Adds an event to the event queue

Arguments	CLASS class of the input device
	NUMBER number of the input device
	DATA data from the input device
Global	EVENTQC, EVENTQN, EVENTQD
	arrays of size QSIZE which form the event queue
	QFRONT, QREAR pointers to front and rear of event queue
Constant	QSIZE the maximum size of the event queue

```
BEGIN
   IF QREAR = QSIZE THEN QREAR ← 1 ELSE QREAR ← QREAR + 1;
   IF QFRONT = QREAR THEN RETURN ERROR 'EVENT Q OVERFLOW';
```

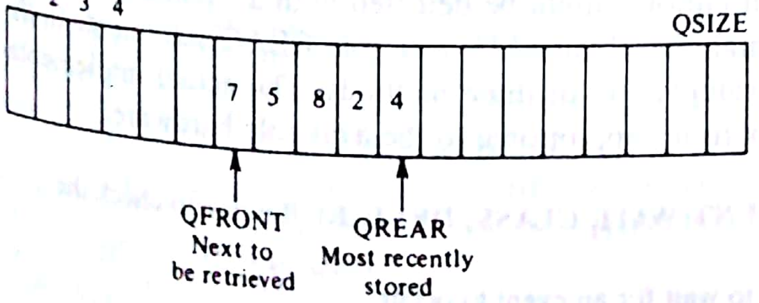

FIGURE 7-8
An array used as a queue containing 7, 5, 8, 2, and 4.

```
    EVENTQC[QREAR] ← CLASS;
    EVENTQN[QREAR] ← NUMBER;
    EVENTQD[QREAR] ← DATA;
    IF QFRONT = 0 THEN QFRONT ← 1;
    RETURN;
END;
```

The string queue is handled in a simpler fashion. Each new string is added to an array of strings. When the end of the array is reached, we wrap around to the beginning.

7.6 Algorithm ADD-STRING-Q(STRING, DATA) Saves STRING and returns a pointer to it in DATA

Arguments	STRING a string to be saved in the string array
	DATA for return of the index of the stored string
Global	STRINGQ an array of strings
	SQREAR next free string storage area
Constant	SQSIZE the size of the string queue array

```
BEGIN
    IF SQREAR = SQSIZE THEN SQREAR ← 1 ELSE SQREAR ← SQREAR + 1;
    STRINGQ[SQREAR] ← STRING;
    DATA ← SQREAR;
    RETURN;
END;
```

The user must be able to determine if an event has taken place. We must therefore have an ''await-event'' routine. This routine checks the event queue and returns information about which device has caused an event. If no event has taken place and the queue is empty, the routine may poll the queue for a specified period of time. If at the end of this time the queue is still empty, a failure indicator may be returned.

Now let us consider the routine for checking the event queue. In the interrupt case, this routine polls the queue until the event has occurred or time runs out. It then obtains from the queue the class and number of the device which caused the event. However, if the input device does not generate interrupts, then the event process should be simulated through polling the device directly. There are two ways that the polling may be done. It may be built into the programming language and/or operating system we are using. Input will then act as if from a sampled device; an example is a high-level language READ statement. Polling may also be done in a loop which we have written ourselves. We call this a polled device. In general, our AWAIT-EVENT routine may involve a combination of these techniques. For example, picks may be done through true interrupts, but buttons might be detected with a ''homemade'' polling loop, and keyboard input might be obtained by a simple READ statement. In the upcoming algorithm we give examples of all three methods. The actual implementation should include only those portions appropriate to the available hardware.

7.7 Algorithm AWAIT-EVENT(WAIT, CLASS, DEVICE) Routine to check the event queue

Arguments	WAIT the time to wait for an event to occur

```
            CLASS, DEVICE to return the type of event which occurred
            BUTTON, PICK, KEYBOARD device enabled flags
Global      INPUT-STRING, PICKED-SEGMENT storage for keyboard and pick input
            DETECTABLE segment detectability attribute array
            BUTTON-POLL, PICK-POLL, KEYBOARD-POLL flag indicating
            the status of polled devices
            TIME-LIMIT the time at which to stop waiting
Local       DATA for receiving data from the event queue

BEGIN
  if buttons are simulated by a sampled device, then include the following conditional
  statement
  IF BUTTON THEN
    BEGIN
      READ DEVICE;
      CLASS ← 1;
      RETURN;
    END;
  if picks are simulated by a sampled device, then include the following conditional
  statement
  IF PICK THEN
    BEGIN
      PICKED-SEGMENT ← 0;
      WHILE PICKED-SEGMENT < 1 OR PICKED-SEGMENT > NUMBER-OF-
          SEGMENTS OR NOT DETECTABLE(PICKED-SEGMENT) DO
          READ PICKED-SEGMENT;
      CLASS ← 2;
      DEVICE ← 1;
      RETURN;
    END;
  if the keyboard is simulated by a sampled device, then include the following condi-
  tional statement
  IF KEYBOARD THEN
    BEGIN
      READ INPUT-STRING;
      CLASS ← 3;
      DEVICE ← 1;
      RETURN;
    END;
  if interrupt-generating or polled devices are available, then include the following loop
  TIME-LIMIT ← TIME( ) + WAIT;
  WHILE TIME( ) ≤ TIME-LIMIT DO
    BEGIN
      if interrupt-generating devices are adding to the event queue, then check the
      queue with the following statements
        BEGIN
          GETQ(CLASS, DEVICE, DATA);
          IF CLASS ≠ 0 THEN
            IF CLASS = 2 THEN PICKED-SEGMENT ← DATA
            ELSE IF CLASS = 3 THEN INPUT-STRING ← STRINGQ[DATA];
          RETURN;
        END;
    if buttons are simulated on a polled device, then include the following condi-
    tional statement
    IF BUTTON AND BUTTON-POLL THEN
        BEGIN
            CLASS ← 1;
            READ-BUTTON(DEVICE);
            RETURN;
        END;
    if a pick is simulated on a polled device, then include the following conditional
    statement
    IF PICK AND PICK-POLL THEN
        BEGIN
            READ-PICK(DEVICE,PICKED-SEGMENT);
            IF DETECTABLE[PICKED-SEGMENT] THEN
                BEGIN
                    CLASS ← 2;
                    RETURN;
                END;
        END;
    if the keyboard is treated as a polled device, then include the following condi-
    tional statement
    IF KEYBOARD AND KEYBOARD-POLL THEN
        BEGIN
            CLASS ← 3;
            READ-KEYBOARD(DEVICE, INPUT-STRING);
            RETURN;
        END;
  END;
  CLASS ← 0;
  DEVICE ← 0;
  RETURN;
END;
```

If interrupts and a true event queue are used, then we must have a routine to get the foremost item in queue. The algorithm uses the pointer QFRONT to indicate the position of the leading element. If QFRONT is zero, the queue is empty; if not, the value of the leading item is returned and the QFRONT pointer is advanced to the next entry.

```
7.8 Algorithm GETQ(CLASS, DEVICE, DATA) Returns the event at the front of the
event queue
If the queue is empty, zero is returned
Arguments   CLASS class of the event
            DEVICE device of the event
            DATA input data from the event
Global      EVENTQC, EVENTQN, EVENTQD the event queue arrays
            QFRONT, QREAR pointers to front and rear of event queue
```

```
Constant    QSIZE the size of the event queue
BEGIN
   CLASS ← 0;
   IF QFRONT = 0 THEN RETURN;
   CLASS ← EVENTQC[QFRONT];
   DEVICE ← EVENTQN[QFRONT];
   DATA ← EVENTQD[QFRONT];
   IF QFRONT = QREAR THEN
      BEGIN
         QFRONT ← 0;
         QREAR ← 0;
      END
   ELSE IF QFRONT = QSIZE THEN QFRONT ← 1
      ELSE QFRONT ← QFRONT + 1;
   RETURN;
END;
```

The event queue allows several events to occur before the information is used. The queue provides storage for the event information until it is needed. But we may sometimes want to start fresh, ignoring old events which have not been processed. We can design a routine which will clear the event queue, discarding the unwanted events.

7.9 Algorithm FLUSH-ALL-EVENTS Removes all events from event queue

```
Global      QFRONT, QREAR event queue front and rear pointers
            SQREAR string file pointer
BEGIN
   QFRONT ← 0;
   QREAR ← 0;
   SQREAR ← 1;
   RETURN;
END;
```

The AWAIT-EVENT routine simulates the occurrence of some event. For keyboard or pick events, we still must retrieve the information saved when the event occurred. We must therefore provide the user with a GET-KEYBOARD-DATA routine. This routine returns the string which was input, along with its length. A GET-PICK-DATA routine returns the name of the segment which the user selected.

7.10 Algorithm GET-KEYBOARD-DATA(STRING, LEN) Routine to return the stored keyboard input

```
Arguments   STRING for the return of the string
            LEN the string's length
Global      INPUT-STRING keyboard input storage
BEGIN
   LEN ← LENGTH(INPUT-STRING);
   STRING ← INPUT-STRING;
   RETURN;
END;
```

```
7.11 Algorithm GET-PICK-DATA(SEGMENT-NAME) Routine to return the selected
pick value
Argument   SEGMENT-NAME the name of the selected segment
Global     PICKED-SEGMENT pick input storage
BEGIN
    SEGMENT-NAME ← PICKED-SEGMENT;
    RETURN;
END;
```

The AWAIT-EVENT routine allows several input devices to be in use at the same time. Many applications, however, require input from only one device at a time, and furthermore, some time-sharing systems prohibit the simultaneous use of several devices. For these situations, the generality of the event queue becomes a needless overhead. To avoid this, we can provide routines which await input from only a single class of device, either button, pick, or keyboard. Once again, the routines will be device- and system-dependent, but outlines for possible implementations for button and pick routines are given below.

```
7.12 Algorithm AWAIT-BUTTON(WAIT, BUTTON-NUM) User routine to await the
pressing of a button
Arguments  WAIT the time to wait for a button event
           BUTTON-NUM for return of the number of the button device
Global     BUTTON device enabled flag
           BUTTON-POLL status flag if button is a polled device
Local      TIME-LIMIT the time at which to stop waiting
           DUMMY a dummy argument
BEGIN
    IF NOT BUTTON THEN RETURN ERROR 'BUTTON NOT ENABLED';
    if buttons are simulated by a sampled device, then include the following statement
    READ BUTTON-NUM;
    if interrupt-generating or polled devices are used, then include the following loop
    TIME-LIMIT ← TIME( ) + WAIT;
    WHILE TIME( ) ≤ TIME-LIMIT DO
        BEGIN
            if interrupt-generating buttons are used, they may be found by
            BEGIN
                GETQ(CLASS, BUTTON-NUM, DUMMY);
                IF CLASS = 1 THEN RETURN;
            END;
            if buttons are simulated on a polled device, then include the following conditional statement
            IF BUTTON-POLL THEN
            BEGIN
                READ-BUTTON(BUTTON-NUM);
                RETURN;
            END;
        END;
    BUTTON-NUM ← 0;
    RETURN;
END;
```

7.13 Algorithm AWAIT-PICK(WAIT, PICK-NUM) User routine to await a pick

```
Arguments   WAIT the time to wait for a pick event
            PICK-NUM for return of the number of the picked segment
Global      PICK device enabled flag
            PICK-POLL status flag if pick is a polled device
Local       TIME-LIMIT the time at which to stop waiting
BEGIN
    IF NOT PICK THEN RETURN ERROR 'PICK NOT ENABLED';
    if picks are simulated by a sampled device, then include the following statement
    READ PICK-NUM;
    if interrupt-generating or polled devices are used, then include the following loop
    TIME-LIMIT ← TIME( ) + WAIT;
    WHILE TIME( ) ≤ TIME-LIMIT DO
        BEGIN
            if interrupt-generating picks are used, they may be found by
                BEGIN
                    GETQ(CLASS, DUMMY, PICK-NUM);
                    IF CLASS = 2 THEN RETURN;
                END;
            if picks are simulated on a polled device, then include the following conditional
            statement
            IF PICK-POLL THEN
                BEGIN
                    READ-PICK(PICK-NUM);
                    RETURN;
                END;
        END;
    PICK-NUM ← 0;
    RETURN;
END;
```

SAMPLED DEVICES

The locator is a sampled device and, therefore, may be read at any time; there is no need to wait for an event to be placed on the event queue. We need only a routine which reads the locator and returns its x, y values.

Since locators are sampled devices, our simulation of a locator need not involve the AWAIT-EVENT routine. For an actual locator device, the current coordinate values are read from the device and returned. In a keyboard simulation, coordinate values will be read from the keyboard and returned.

7.14 Algorithm READ-LOCATOR(X, Y)

```
Global      LOCATOR the locator enabled flag
BEGIN
    IF NOT LOCATOR THEN RETURN ERROR 'LOCATOR NOT ENABLED';
    OBTAIN (X, Y) FROM LOCATOR DEVICE OR ITS SIMULATION;
    CONVERT TO NORMALIZED DEVICE COORDINATES IF NECESSARY;
    RETURN;
END;
```

Valuators are treated in essentially the same way as locators—one variable is returned instead of a coordinate pair.

7.15 Algorithm READ-VALUATOR(V)

```
Global      VALUATOR the valuator enabled flag
BEGIN
    IF NOT VALUATOR THEN RETURN ERROR 'VALUATOR NOT ENABLED';
    OBTAIN V FROM VALUATOR DEVICE OR ITS SIMULATION;
    RETURN;
END;
```

THE DETECTABILITY ATTRIBUTE

A useful feature for pick devices is the ability to set the *detectability* of portions of the display. We may wish to be able to pick an item from a subset of the items actually appearing on display. We want all other items of the display to be ignored by the pick device. This can be done by disabling the interrupt mechanism when portions of the display which we wish to ignore are being drawn. The interrupt mechanism is reenabled when detectable items are being drawn. We would therefore like to present the user with some routine for setting the detectability of portions of his display. We can give segments a detectability attribute so that a segment may or may not be detectable by a pick device. We do this by giving each segment a detectability flag. We provide the user with a routine to set this flag to on or off (the default value would be off). The detectability flag may be checked by the display-generating routines in a vector refresh device to disable or enable light-pen interrupts. For other simulations the flag can still be used to determine whether a segment is a candidate for a pick. An example is the above AWAIT-EVENT routine for sampled pick simulations. In the case where a pick for a particular segment is simulated by a keyboard, the detectability flag for that segment is checked by the AWAIT-EVENT routine. If the segment turns out not to be detectable, then another input is requested. If the segment is detectable, then its name is saved and AWAIT-EVENT returns, indicating that a valid pick has occurred.

If actual interrupts are used for picks, then the MAKE-PICTURE-CURRENT or INTERPRET routine should be extended to include enabling and disabling of interrupts according to the detectability flag for the segment being interpreted.

The following algorithm allows the user to set the detectability attribute.

7.16 Algorithm SET-DETECTABILITY(SEGMENT-NAME, ON-OFF) User routine to set the detectability attribute

```
Arguments   SEGMENT-NAME the display-file segment being set
            ON-OFF the detectability setting
Global      DETECTABLE the detectability attribute array
Constant    NUMBER-OF-SEGMENTS the size of the DETECTABLE array
```

```
BEGIN
  IF SEGMENT-NAME < 1 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
  THEN
     RETURN ERROR 'INVALID SEGMENT';
  DETECTABLE[SEGMENT-NAME] ← ON-OFF;
  RETURN;
END;
```

We have added a new segment attribute. This means that our routines which manage the segment table must be extended to include this new property. There are two routines which must be changed: the CREATE-SEGMENT routine, which initializes all attributes, and the RENAME-SEGMENT routine, which copies all attributes to a new position in the table. The modified versions of these routines are given below.

7.17 Algorithm CREATE-SEGMENT(SEGMENT-NAME) (Modification of algorithm 6.4) User routine to create a named segment

```
Argument   SEGMENT-NAME the segment name
Global     NOW-OPEN the segment currently open
           FREE the index of the next free display-file cell
           SEGMENT-START, SEGMENT-SIZE, VISIBILITY
           ANGLE, SCALE-X, SCALE-Y, TRANSLATE-X, TRANSLATE-Y
           DETECTABLE the segment-table arrays
Constant   NUMBER-OF-SEGMENTS size of the segment table
BEGIN
  IF NOW-OPEN > 0 THEN RETURN ERROR 'SEGMENT STILL OPEN';
  IF SEGMENT-NAME < 1 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
  THEN
     RETURN ERROR 'INVALID SEGMENT NAME';
  IF SEGMENT-SIZE[SEGMENT-NAME] > 0 THEN
     RETURN ERROR 'SEGMENT ALREADY EXISTS';
  NEW-VIEW-2
  SEGMENT-START[SEGMENT-NAME] ← FREE;
  SEGMENT-SIZE[SEGMENT-NAME] ← 0;
  VISIBILITY[SEGMENT-NAME] ← VISIBILITY[0];
  ANGLE[SEGMENT-NAME] ← ANGLE[0];
  SCALE-X[SEGMENT-NAME] ← SCALE-X[0];
  SCALE-Y[SEGMENT-NAME] ← SCALE-Y[0];
  TRANSLATE-X[SEGMENT-NAME] ← TRANSLATE-X[0];
  TRANSLATE-Y[SEGMENT-NAME] ← TRANSLATE-Y[0];
  DETECTABLE[SEGMENT-NAME] ← DETECTABLE[0];
  NOW-OPEN ← SEGMENT-NAME;
  RETURN;
END;
```

7.18 Algorithm RENAME-SEGMENT(SEGMENT-NAME-OLD, SEGMENT-NAME-NEW) (Modification of algorithm 5.5) User routine to rename SEGMENT-NAME-OLD to be SEGMENT-NAME-NEW

```
Arguments  SEGMENT-NAME-OLD old name of segment
           SEGMENT-NAME-NEW new name of segment
```

```
Global      SEGMENT-START, SEGMENT-SIZE, VISIBILITY
            ANGLE, SCALE-X, SCALE-Y, TRANSLATE-X, TRANSLATE-Y
            DETECTABLE the segment-table arrays
            NOW-OPEN the segment currently open
Constant    NUMBER-OF-SEGMENTS the size of the segment table
BEGIN
    IF SEGMENT-NAME-OLD < 1 OR SEGMENT-NAME-NEW < 1
        OR SEGMENT-NAME-OLD > NUMBER-OF-SEGMENTS
        OR SEGMENT-NAME-NEW > NUMBER-OF-SEGMENTS THEN
        RETURN ERROR 'INVALID SEGMENT NAME';
    IF SEGMENT-NAME-OLD = NOW-OPEN OR
        SEGMENT-NAME-NEW = NOW-OPEN THEN
        RETURN ERROR 'SEGMENT STILL OPEN';
    IF SEGMENT-SIZE[SEGMENT-NAME-NEW] ≠ 0 THEN
        RETURN ERROR 'SEGMENT ALREADY EXISTS';
    copy the old segment-table entry into the new position
    SEGMENT-START[SEGMENT-NAME-NEW]
        ← SEGMENT-START[SEGMENT-NAME-OLD];
    SEGMENT-SIZE[SEGMENT-NAME-NEW]
        ← SEGMENT-SIZE[SEGMENT-NAME-OLD];
    VISIBILITY[SEGMENT-NAME-NEW] ← VISIBILITY[SEGMENT-NAME-OLD];
    ANGLE[SEGMENT-NAME-NEW] ← ANGLE[SEGMENT-NAME-OLD];
    SCALE-X[SEGMENT-NAME-NEW] ← SCALE-X[SEGMENT-NAME-OLD];
    SCALE-Y[SEGMENT-NAME-NEW] ← SCALE-Y[SEGMENT-NAME-OLD];
    TRANSLATE-X[SEGMENT-NAME-NEW]
        ← TRANSLATE-X[SEGMENT-NAME-OLD];
    TRANSLATE-Y[SEGMENT-NAME-NEW]
        ← TRANSLATE-Y[SEGMENT-NAME-OLD];
    DETECTABLE[SEGMENT-NAME-NEW]
        ← DETECTABLE[SEGMENT-NAME-OLD];
    delete the old segment
    SEGMENT-SIZE[SEGMENT-NAME-OLD] ← 0;
    RETURN;
END;
```

As in previous chapters, we shall provide the user with an initialization routine. This routine sets the default values so that no device is enabled, no events are in the event queue, and no segments are detectable.

7.19 Algorithm INITIALIZE-7 Initialization

```
Global      DETECTABLE the detectability attribute array
BEGIN
    INITIALIZE-6;
    DISABLE-ALL;
    FLUSH-EVENT-Q;
    DETECTABLE[0] ← FALSE;
    RETURN;
END;
```

SIMULATING A LOCATOR WITH A PICK

While the light pen can be used to select an object on the screen, it does not give that object's position. Nor can it be used to indicate a position where there is no object, as can be done with a joystick or tablet. In order to use a light pen for position information, a *tracking cross* is employed. A tracking cross is a small cross made of four or more separate line segments. This is placed at some known position on the screen and selected so that it is detectable by the light pen. (See Figure 7-9.)

The pen is positioned at the center of the cross. Then as the pen is moved, it will encounter one of the cross arms. If the pen is moved slightly to the right, it will encounter the right arm of the cross. This information is used to move the cross slightly to the right. After each refresh the cross will be moved until the light pen is once again centered. (See Figure 7-10.)

A similar action is taken for all of the other arms of the cross so that the tracking cross will follow the movements of the light pen. Since the position of the center of the tracking cross is known, positional information can be entered by "grabbing" the cross with the light pen and moving it to the desired location.

SIMULATING A PICK WITH A LOCATOR

Suppose we have a locator, but no pick device. How can we use the locator to get the effect of a pick? We know that the user is interested in the part of the picture located at position (x, y), but do not know which segment is being drawn there. To find out, we shall step through the display-file instructions segment by segment as if we were drawing the image. But instead of generating line and character images, we will ask how close the lines and characters are to the locator position. If a line or a character is sufficiently close, we will assume that it is the item at which the user is pointing. We will note what segment we are considering and return it as the value of the pick selection.

How can we determine the distance between a point (x, y) and a character? We simplify the problem by treating the character as if it were a point (x_c, y_c). Then we can use the distance formula to give

$$D = [(x - xc)^2 + (y - yc)^2]^{1/2} \qquad (7.1)$$

If this distance is close enough

$$D < \text{APERTURE} \qquad (7.2)$$

then we have found the segment. (See Figure 7-11.)

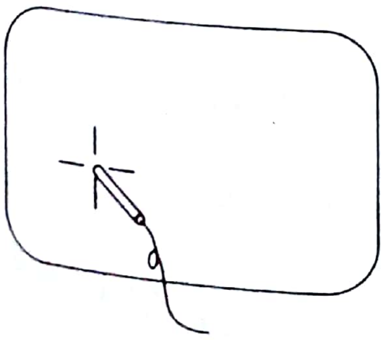

FIGURE 7-9
Light pen and tracking cross.

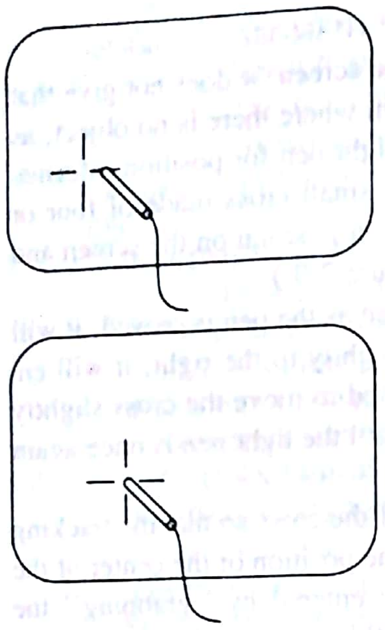

FIGURE 7-10
If the light pen encounters one of the arms of the cross, the position of the cross is shifted.

This formula will select a character if its point lies within a circle of radius APERTURE centered on the locator point. But this formula involves multiplications which are computationally costly. A more efficient test is to select a point lying within a square with side 2 * APERTURE centered on the locator point. In this case, the test condition becomes

$$|(x - x_c)| + |(y - y_c)| < \text{APERTURE} \qquad (7.3)$$

Absolute values are much easier to calculate than squares. (See Figure 7-12.)

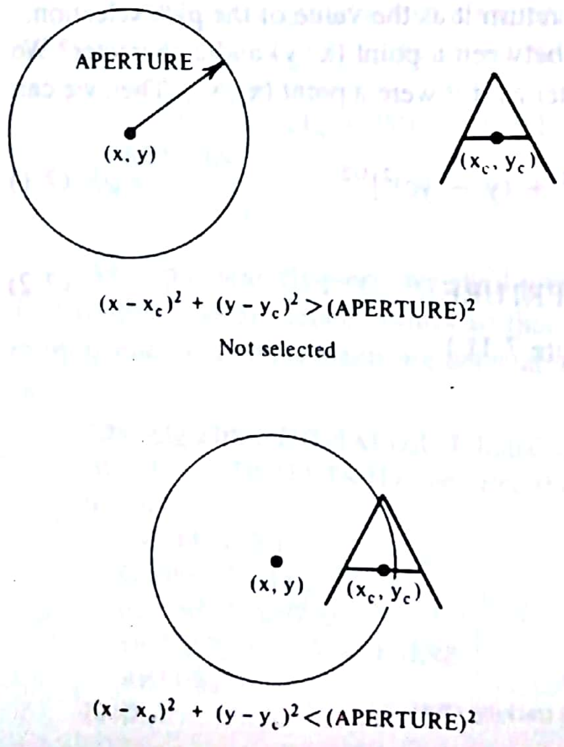
APERTURE
(x, y)
(x_c, y_c)
(x - x_c)² + (y - y_c)² > (APERTURE)²
Not selected
(x, y)
(x_c, y_c)
(x - x_c)² + (y - y_c)² < (APERTURE)²

FIGURE 7-11
Selection of a point.

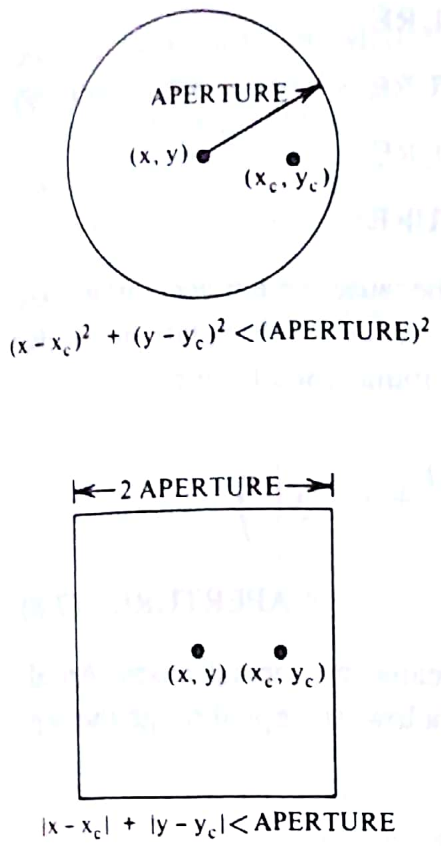
APERTURE
(x, y)
(x_c, y_c)
(x - x_c)² + (y - y_c)² < (APERTURE)²
2 APERTURE
(x, y) (x_c, y_c)
|x - x_c| + |y - y_c| < APERTURE

FIGURE 7-12
A square test area results in a more efficient test.

We can also find the distance between a point and a line segment. We already found an expression for this in Equation 1.30 which is

$$D = |rx + sy + t| \tag{7.4}$$

where (x, y) is the locator point, and r, s, and t are parameters which describe the line (see Equation 1.6) and also satisfy the normalization condition (Equation 1.7). The values for r, s, and t satisfying these constraints for a line with endpoints (x_1, y_1), (x_2, y_2) are

$$\begin{aligned} r &= (y_2 - y_1)/d \\ s &= -(x_2 - x_1)/d \\ t &= (x_2 y_1 - x_1 y_2/d \end{aligned} \tag{7.5}$$

where

$$d = [(x_2 - x_1)^2 + (y_2 - y_1)^2]^{1/2}$$

With a little algebra, Equations 7.4 and 7.5 can be distilled into the formula

$$D = \frac{|(x - x_1)(y_2 - y_1) - (y - y_1)(x_2 - x_1)|}{[(x_2 - x_1)^2 + (y_2 - y_1)^2]^{1/2}} \tag{7.6}$$

This measure applies to the entire line containing the line segment. A preliminary test should be made to determine if the point lies too far beyond the segment endpoints. That is, we can reject the segment if

$$\begin{aligned} & x < \min(x_1, x_2) - \text{APERTURE} \\ \text{or } & x > \max(x_1, x_2) + \text{APERTURE} \\ \text{or } & y < \min(y_1, y_2) - \text{APERTURE} \\ \text{or } & y > \max(y_1, y_2) + \text{APERTURE} \end{aligned} \qquad (7.7)$$

Here again we have a number of costly multiplications because we are accepting lines which fall within a circle about the locator point. By considering a square about the locator point we can use the following more efficient formula. (See Figure 7-13.)

$$\min\left(\left|\frac{(y_2-y_1)(x-x_1)}{x_2-x_1}+y_1-y\right|, \left|\frac{(x_2-x_1)(y-y_1)}{y_2-y_1}+x_1-x\right|\right) - < \text{APERTURE} \qquad (7.8)$$

Vertical and horizontal line segments must be treated as special cases. An algorithm for simulating a pick with a light pen is given below. It steps through the seg-

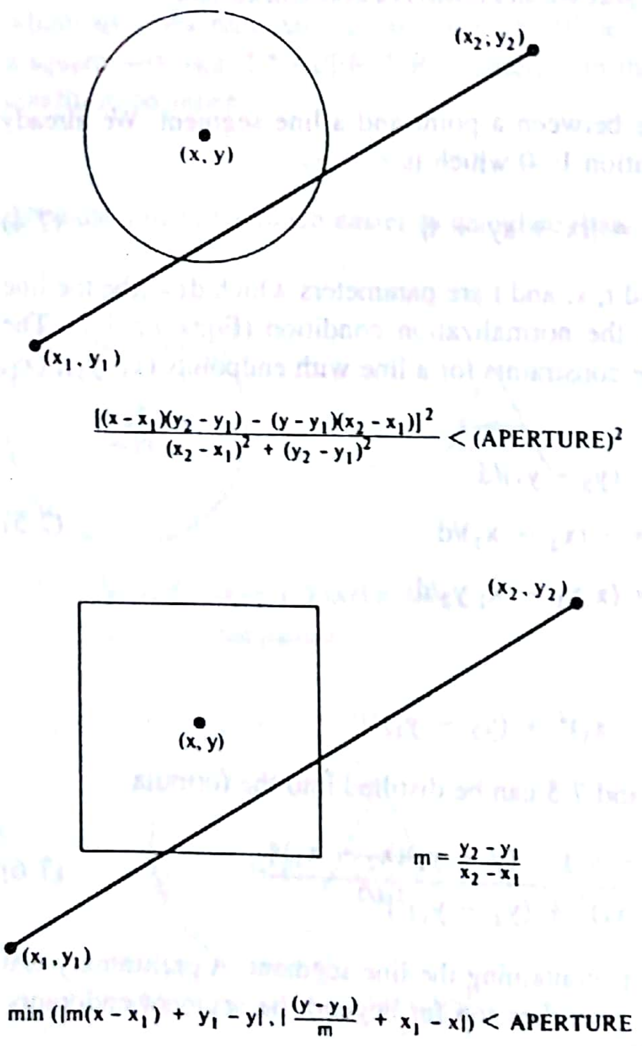
(x_2, y_2)
(x, y)
(x_1, y_1)
$\frac{[(x-x_1)(y_2-y_1)-(y-y_1)(x_2-x_1)]^2}{(x_2-x_1)^2+(y_2-y_1)^2} < (\text{APERTURE})^2$
(x_2, y_2)
(x, y)
$m = \frac{y_2-y_1}{x_2-x_1}$
(x_1, y_1)
$\min(|m(x-x_1)+y_1-y|, |\frac{(y-y_1)}{m}+x_1-x|) < \text{APERTURE}$

FIGURE 7-13
Tests for selecting a line segment.

ment table as does our MAKE-PICTURE-CURRENT routine, but considers only visible, detectable, named segments. For each segment, it steps through the display file as does our INTERPRET routine, but instead of performing the instruction, it applies the above tests. The first segment which satisfies a test is returned as the result. If no such segment is found, then zero is returned.

7.20 Algorithm PICK-SEARCH (PICK, X, Y) Routine used to simulate a pick with a locator
It indicates which segment image is at location X, Y

Arguments	PICK used to return the discovered segment
	X, Y position of the simulated pick
Global	APERTURE sensitivity of the pick
	SEGMENT-START, SEGMENT-SIZE the segment-table arrays
	DETECTABLE the segment detectability attribute array
	VISIBILITY the segment visibility attribute array
	IMAGE-XFORM a 3 × 2 array containing the image transformation
Local	SEGMENT for stepping through the possible segments
	INSTRUCTIONS for stepping through the display-file segments
Constant	ROUNDOFF some small number greater than any round-off error
	NUMBER-OF-SEGMENTS size of the segment table

```
BEGIN
  X1 ← 0;
  Y1 ← 0;
  FOR SEGMENT = 1 TO NUMBER-OF-SEGMENTS DO
    IF SEGMENT-SIZE[SEGMENT] ≠ 0
      AND VISIBILITY[SEGMENT] AND DETECTABLE[SEGMENT] THEN
      BEGIN
        BUILD-TRANSFORMATION(SEGMENT);
        DO-TRANSFORMATION(X1, Y1, IMAGE-XFORM);
        PICK ← SEGMENT;
        FOR INSTRUCTION = SEGMENT-START[SEGMENT] TO
          SEGMENT-START[SEGMENT] + SEGMENT-SIZE[SEGMENT] -
          1 DO
          BEGIN
            GET-POINT(INSTRUCTION, OP, X2, Y2);
            IF OP > 0 OR OP < -31 THEN
              BEGIN
                DO-TRANSFORMATION(X2, Y2, IMAGE-XFORM);
                IF OP < -31
                  AND |X - X2| + |Y - Y2| < APERTURE THEN
                  RETURN;
                IF OP = 2 THEN
                  IF X > MIN(X1, X2) - APERTURE AND
                     X < MAX(X1, X2) + APERTURE AND
                     Y > MIN(Y1, Y2) - APERTURE AND
                     Y < MAX(Y1, Y2) + APERTURE THEN
                        IF |Y2 - Y1| < ROUNDOFF OR
                           |X2 - X1| < ROUNDOFF THEN RETURN
                        ELSE IF MIN(|(Y2 - Y1) * (X - X1) / (X2 - X1)
                                + Y1 - Y|,|(X2 - X1) * (Y - Y1) / (Y2 - Y1)
                                + X1 - X|)
                                < APERTURE THEN RETURN;
                          X1 ← X2;
                          Y1 ← Y2;
                        END;
                    END;
              END;
          PICK ← 0;
          RETURN;
      END;
```

ECHOING

An important part of an interactive system is *echoing*. Echoing provides the user with information about his actions. This allows the user to compare what he has done against what he wanted to do. For keyboard input, echoing usually takes the form of displaying the typed characters. Locators may be echoed by a screen cursor displayed at the current locator position. This allows the user to see the current locator setting and to relate its position to the objects on the display. (See Figure 7-14.)

A pick may be echoed by identifying the selected objects on the display. The selected objects may be flashed or made brighter or, perhaps, altered in color. This will allow the user to determine whether or not he has selected the intended objects. (See Figure 7-15.)

Buttons, when used to select menu items, can be echoed by flashing or highlighting the selected items. (See Figure 7-16.)

For a valuator, display of the current setting in numerical form or as a position on a scale is possible. (See Figure 7-17.)

It is important that some form of echoing be present, no matter what form it takes. Without it, the user feels that he is working in the dark and may never feel comfortable with the program.

INTERACTIVE TECHNIQUES

We should include in our discussion of graphics input some of the techniques for interactively creating and modifying pictures. Let us first consider how to add new pic-

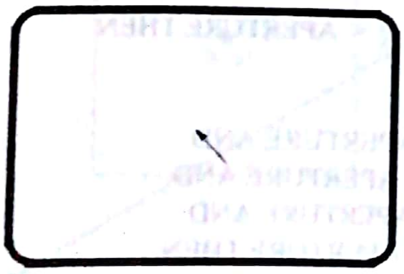
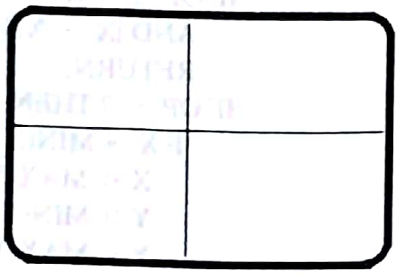

FIGURE 7-14
Echoing a location.

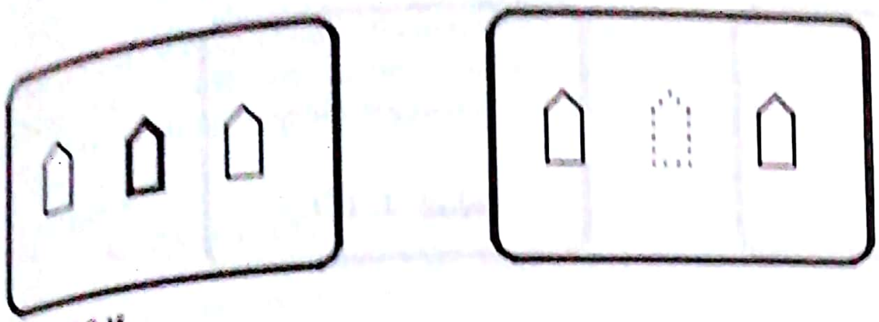

FIGURE 7-15
Echoing a pick.

ture elements to the image. *Point plotting* gives the user the capability of selecting a particular point on the screen. This is usually done by a combination of a locator and a button. The locator is used to tell which point the user selects. The button indicates when the locator is correctly positioned. The algorithm to perform point plotting must await the button event; as soon as it occurs, the locator may be read. The selection of points can be used in many different ways. For example, it can give endpoints of line segments, positions for text, or translations of picture segments. It occurs so often that a system routine for it is worthwhile.

7.21 Algorithm AWAIT-BUTTON-GET-LOCATOR(WAIT, BUTTON-NUM, X, Y)
User routine to interactively select a point

Arguments	WAIT the time to wait for a button event
	BUTTON-NUM for return of the user's button selection
	X, Y the point the user selects

```
BEGIN
   AWAIT-BUTTON(WAIT, BUTTON-NUM);
   READ-LOCATOR(X, Y);
   RETURN;
END;
```

One use for the point-plotting routine is to enter line segments. The idea here is to connect the successive points selected by the user with line segments, like a child's "connect-the-dots" drawing. The user can indicate whether to continue or terminate the process by which button is pushed. (See Figure 7-18.)

A program fragment for point plotting is as follows:

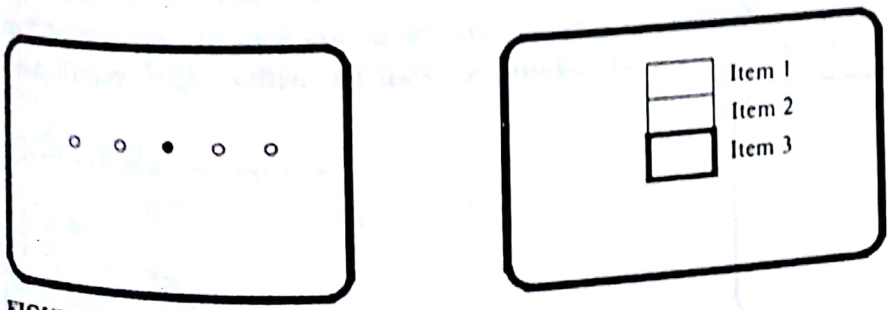

FIGURE 7-16
Echoing buttons and menu selection.

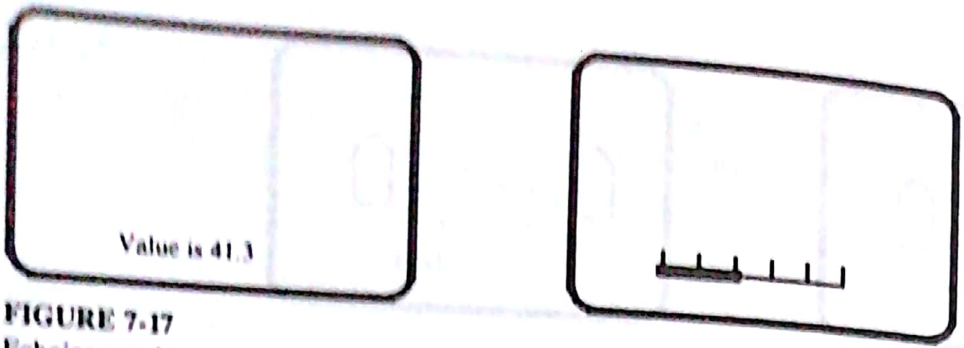

FIGURE 7-17
Echoing a value.

```
BEGIN
    BUTTON-NUM ← CONTINUE;
    WHILE BUTTON-NUM = CONTINUE DO
        BEGIN
            AWAIT-BUTTON-GET-LOCATOR(WAIT, BUTTON-NUM, X, Y);
            LINE-ABS-2(X, Y);
            MAKE-PICTURE-CURRENT;
        END;
END;
```

A variation of this technique is called *inking*. Inking makes the locator automatically leave a trail of line segments the way a pen leaves a trail of ink. It does not re-

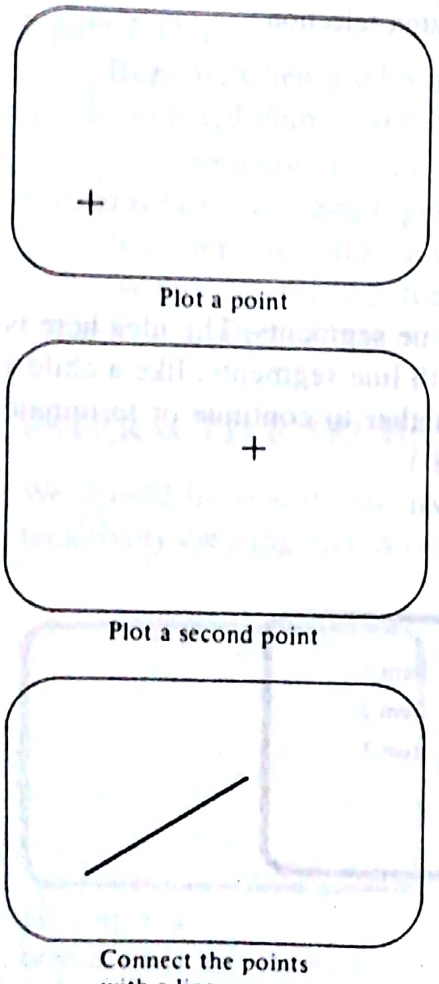

FIGURE 7-18
Point plotting.

Scanned by CamScanner

quire the user to push a button for every line segment; instead, a new segment is drawn whenever the locator moves a sufficient distance. (See Figure 7-19.)

The following program fragment outlines the inking techniques.

```
BEGIN
  PEN-ON ← TRUE;
  READ-LOCATOR(XOLD, YOLD);
  WHILE PEN-ON DO
    BEGIN
      READ-LOCATOR(X, Y);
      IF |(X - XOLD)| + |Y - YOLD| > NO-MOVEMENT THEN
      BEGIN
        LINE-ABS-2(X, Y);
        XOLD ← X;
        YOLD ← Y;
        MAKE-PICTURE-CURRENT;
      END;
    UPDATE(PEN-ON);
  END;
END;
```

The UPDATE(PEN-ON) determines whether inking should cease. This may be governed by an event, such as lifting the stylus, or by the passing of a time limit or a limit on the allowed number of line segments. Note that we could collect the sequence of locator positions without connecting them with line segments. Such a sequence of points is called a *stroke*. Inking displays strokes.

Raster displays can extend inking to *painting* with a *brush*. A brush is a pattern of pixel values. When painting, we repeatedly sample the locator position. Each time the position changes we copy the brush pattern into the frame buffer. The pattern is centered on the current locator position. The effect is to draw this pattern repeatedly on the display along the path of the locator. The instances of the pattern usually overlap, resulting in a shape that looks as though the brush has moved across the display leaving a trail of paint. The shape of the trail depends on both the path of the locator and the shape of the brush. (See Figure 7-20.)

It is often the case that the user wants to connect endpoints of several lines, and frequently horizontal and vertical lines predominate. But it can be difficult to position to a single point, and it may be hard to get a line perfectly horizontal or vertical. A technique to aid in such constructions is a *grid*. A grid is a pattern of special points. (See Figure 7-21.) Often the user can make these points either visible or invisible.

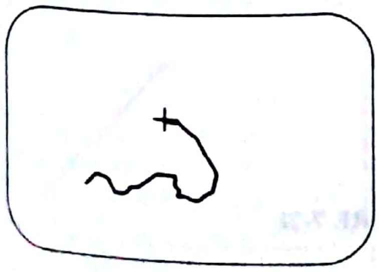

FIGURE 7-19
Inking.

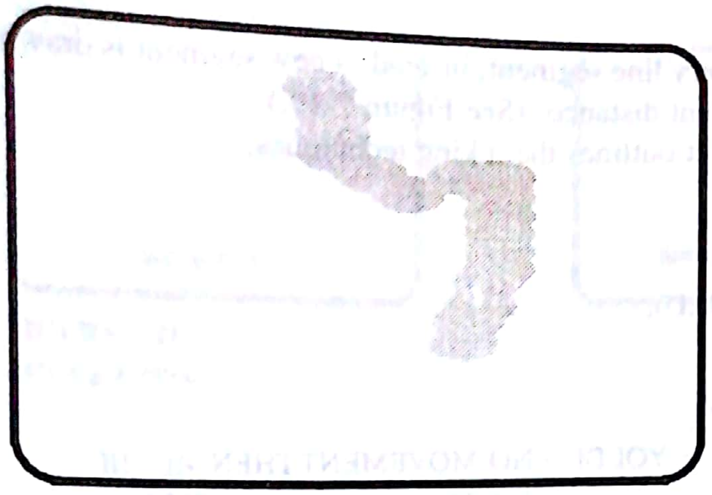

FIGURE 7-20
Painting.

What is special is that values obtained from the locator are rounded to the nearest grid point. All lines begin and end exactly on a grid point. This makes it easy to ensure that endpoints meet and that lines are vertical or horizontal. The following program fragment draws a grid; it should be placed in its own display-file segment so that it can be made visible or invisible.

```
BEGIN
    FOR X = 0 TO NUMBER-OF-GRIDPOINTS DO
        BEGIN
            FOR Y = 0 TO NUMBER-OF-GRIDPOINTS DO
                BEGIN
                    MOVE-ABS-2(X/NUMBER-OF-GRIDPOINTS – DX/2,
                        Y/NUMBER-OF-GRIDPOINTS);
                    LINE-REL-2(DX, 0);
                    MOVE-REL-2( – DX / 2, – DX / 2);
```

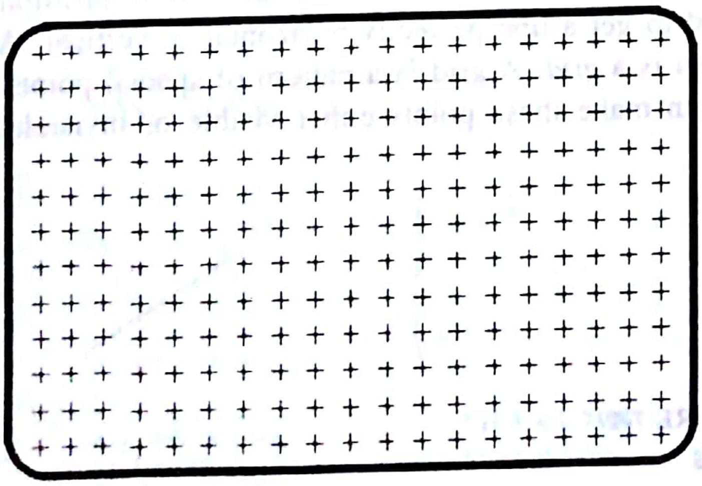

FIGURE 7-21
A grid.

```
                LINE-REL-2(0, DX);
            END;
        END;
    END;
```

In the above fragment, NUMBER-OF-GRIDPOINTS is a count of how many grid points to display across the screen and DX is the width of a small cross which will be imaged at each grid point.

The rounding to the nearest grid point would look like the following:

```
READ-LOCATOR(X, Y);
X ← INT(X * NUMBER-OF-GRIDPOINTS + 0.5) / NUMBER-OF-GRIDPOINTS;
Y ← INT(Y * NUMBER-OF-GRIDPOINTS + 0.5) / NUMBER-OF-GRIDPOINTS;
```

Vector refresh and some raster displays allow a technique called *rubber band lines*. This technique allows the user to see what a line will look like before fixing it in place. The basic process is the same as line plotting; the user adjusts the locator to place the endpoint of the next line segment. The user presses a button when the locator is correctly positioned. The difference is that a line is constantly drawn between the last endpoint and the locator. As the locator moves, the line is redrawn in a new position, giving the appearance of a rubber band stretched between the fixed endpoint and the locator point. (See Figure 7-22.)

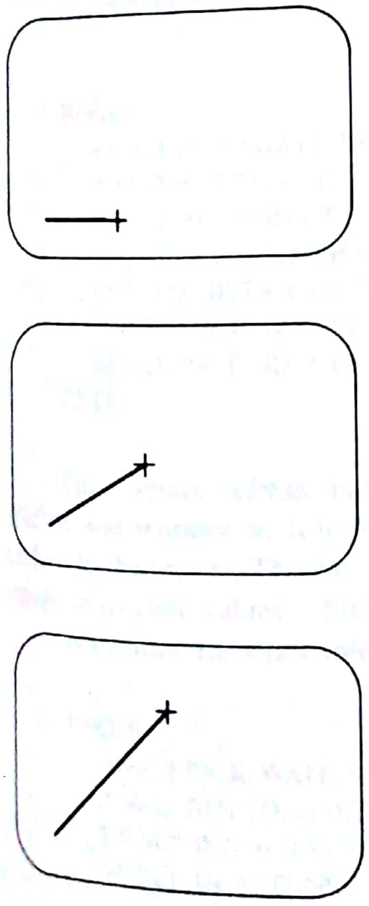

FIGURE 7-22
Rubber band lines.

This technique is difficult to implement in a straightforward manner in our system without adding either the ability to alter portions of the display file without opening and closing segments, or the ability to append new instructions onto a previously closed segment. It can, however, be done by using the image transformation on a single fixed line in a closed segment to create the rubber band line, and another open segment to hold the user's final drawing. The details of this are left as an exercise.

The user might wish not only to create new images but to alter and adjust existing images as well. The first thing necessary is often identifying which part of the image is to be changed. This is naturally a pick operation. Suppose the user wants to remove portions of the image. An AWAIT-PICK operation will determine which segment should be removed and the SET-VISIBILITY routine can make it invisible. Notice that this technique cannot be used to bring the item back, since one cannot pick what one cannot see. (See Figure 7-23.)

Another desirable modification might be a change in the image transformation. For example, the user may wish to change the position of a segment's image. It is usually most convenient for the user to indicate the new position by pointing at it with a locator. The user can use a pick to select the segment to be moved and then select some point on the image with the locator. Finally, the locator is moved to the place to where the point on the image should be moved. (See Figure 7-24.)

A program fragment to do this is given below. INQUIRE-IMAGE-TRANSLATION(TX, TY) is a routine which returns the current image translation parameters.

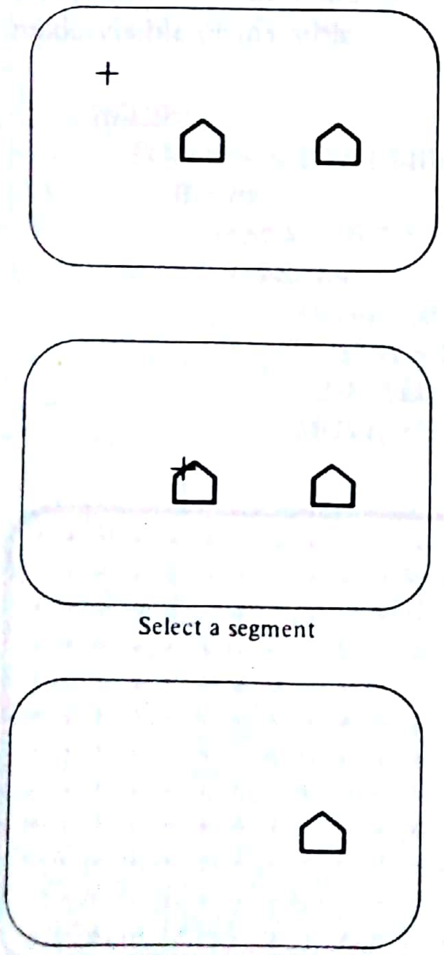

FIGURE 7-23
Selecting a segment with a pick.

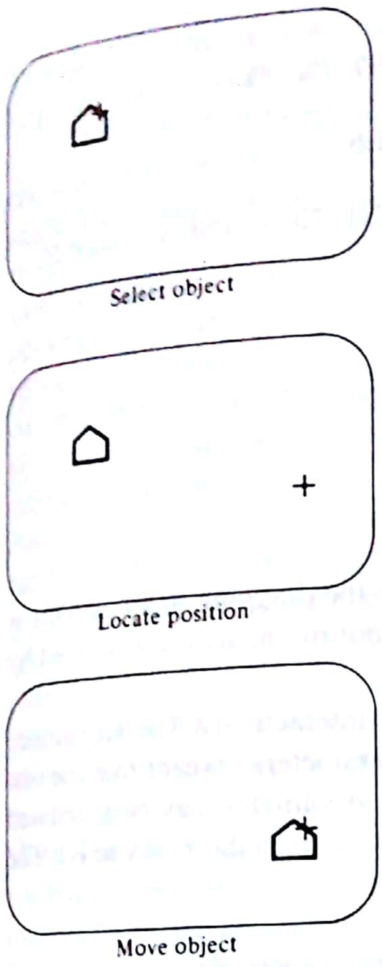

FIGURE 7-24
Positioning.

```
BEGIN
   AWAIT-PICK(WAIT, SEGMENT);
   AWAIT-BUTTON-GET-LOCATOR(WAIT, BUTTON-NUM, XOLD, YOLD);
   AWAIT-BUTTON-GET-LOCATOR(WAIT, BUTTON-NUM, XNEW, YNEW);
   INQUIRE-IMAGE-TRANSLATION(SEGMENT, TX, TY);
   SET-IMAGE-TRANSLATION(SEGMENT, TX + XNEW - XOLD,
      TY + YNEW - YOLD);
   MAKE-PICTURE-CURRENT;
END;
```

On a vector refresh and some raster displays, the image can be continuously shifted and redrawn to follow the locator movements. The image appears to be attached to the locator. The user can see how the picture will look before fixing the new image translation values. This technique is called *dragging*. (See Figure 7-25.)

An outline for a possible dragging procedure is given below.

```
BEGIN
   AWAIT-PICK(WAIT, SEGMENT);
   AWAIT-BUTTON-GET-LOCATOR(WAIT, BUTTON-NUM, XOLD, YOLD);
   DRAGGING ← TRUE;
   WHILE DRAGGING DO
```

```
        BEGIN
            INQUIRE-IMAGE-TRANSLATION(SEGMENT, TX, TY);
            READ-LOCATOR(XNEW, YNEW);
            IF XNEW ≠ XOLD OR YNEW ≠ YOLD THEN
                BEGIN
                    SET-IMAGE-TRANSLATION(SEGMENT, TX + XNEW − XOLD,
                        TY + YNEW − YOLD);
                    XOLD ← XNEW;
                    YOLD ← YNEW;
                    MAKE-PICTURE-CURRENT;
                END;
            UPDATE(DRAGGING);
        END;
END;
```

The UPDATE(DRAGGING) routine is used to stop the dragging process. This is usually done by detecting the occurrence of some event, but might also be triggered by the passing of a time limit.

Changes in orientation and scale may also be done interactively. The techniques are basically the same as for changes of the translation parameters, except that the natural input device might be a valuator instead of a locator. A valuator may be simulated using a locator and a scale drawn on the screen (usually parallel to the x or y axis). The

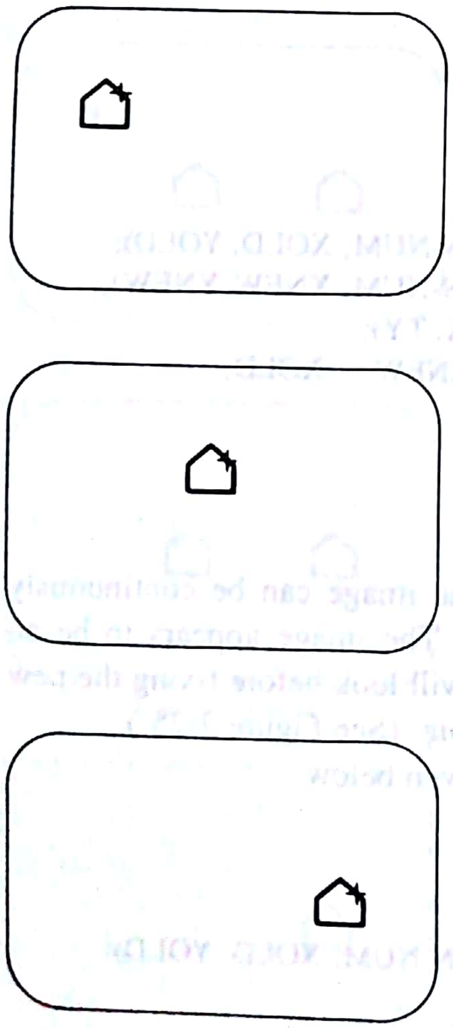

FIGURE 7-25
Dragging.

locator cursor can be positioned on the scale by the user and this position can then be converted into a valuator value. (See Figure 7-26 and Plate 3.)

An interactive program may provide the user with several options. For example, the user might be able to enter a single new line, ink, make a segment visible, move a segment, or quit. The user must select which option to follow. Such a selection can be made with the buttons by assigning a button for each option. The user makes a selection by pressing the appropriate button. We may wish to inform the user just what his options and what the corresponding buttons are. Such a list of options is called a menu. (See Figure 5-7 and Plate 1.) When there are a large number of options, a menu can become excessively long and hard to read. To avoid this the menu can be broken up into several smaller submenus. A top-level menu can be used to select the submenu containing the desired item. Thus menus can be structured into a hierarchy. In a complex selection process one might have to step through several levels of menus to reach the final selection, but at each level the menu is small and the selection easy.

Menus should be placed in a segment or segments different from the main picture so that they can be made invisible when no longer needed. A user may find it inconvenient to transfer his attention back and forth between buttons (or keyboard) and locator (or pick). It's for this reason that a light pen or tablet stylus will often have a switch built into its tip or handle. To avoid switching back and forth between different devices, we may wish to use the light pen or stylus to indicate the desired menu item. Menu selections can be made by a pick instead of a button if we place each selection in a separate segment so that it can be identified. When a light pen is the pick device, such selection items are called *light-buttons*. They behave as buttons, but do not require the user to shift his attention from the screen.

FURTHER READING

Description and classification of graphics input devices are found in [OHL78]. Early descriptions of tablets are given in [DAV64] and [TEI68]. One input device we did not

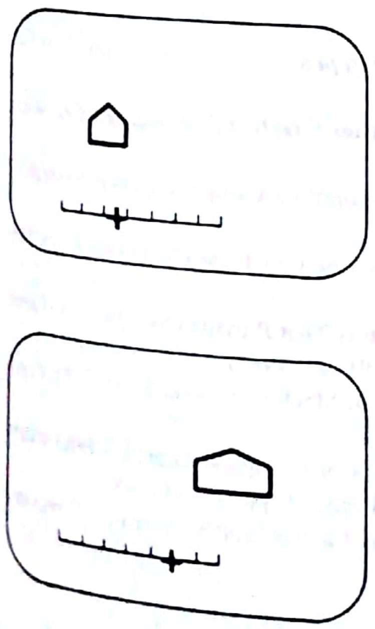

FIGURE 7-26
Using scales.

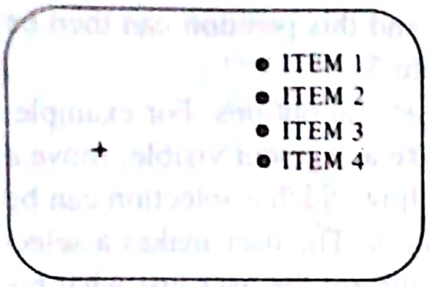

FIGURE 7-27
A menu.

mention is a touch panel, which acts as a locator attached to the display so that positions can be indicated by touching the screen with a finger or with a special pen. An example of this class of device is described in [NAG85]. A version of a mouse which detects changes in position optically is described in [LYO82]. An event queue for graphical input is suggested in [SPR75]. Graphic interaction techniques are described in [BER78], [FOL74], and [NEW68]. A detailed model of device-independent virtual input devices is given in [ROS82]. A description of virtual input is given in [WAL76]. In some cases, the virtual devices we have described may not be a good match to the actual devices; an alternative approach is to describe devices by the state they manage, the events they notice, and the actions they cause [ANS79]. A discussion of some interactive techniques for raster displays and encodings of raster data to support them may be found in [BAU80]. The distinction between the handling of an interaction and the applications program which uses the information is made clear in [KAM83]. The use of picks in interactive graphics is described in [WEL80]. An algorithm for simulating a pick with a locator is given in [GAR80]. Simulating valuators with a locator is described in [EVA81] and [THO79]. In [TUR84] a distinction is made between program variables (set by assignment statements) and graphics variables (set by hardware). An extension of graphics systems to allow attributes and coordinates to be graphics variables allows the display to be dynamically altered for interaction. Issues in interactive graphics and a bibliography are given in [THO83].

[ANS79] Anson, E., "The Semantics of Graphical Input," *Computer Graphics*, vol. 13, no. 2, pp. 113–120 (1979).

[BAU80] Baudelaire, P., Stone, M., "Techniques for Interactive Raster Graphics," *Computer Graphics*, vol. 14, no. 3, pp. 314–320 (1980).

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

[DAV64] Davis, M. R., Ellis, T. O., "The Rand Tablet: A Man-Machine Graphical Communication Device," *AFIPS FJCC*, vol. 26, pp. 325–331 (1964).

[EVA81] Evans, K. B., Tanner, P. P., Wein, M., "Tablet-Based Valuators That Provide One, Two, or Three Degrees of Freedom," *Computer Graphics*, vol. 15, no. 3, pp. 91–97 (1981).

[FOL74] Foley, J. D., Wallace, V. L., "The Art of Natural Graphic Man-Machine Conversation," *Proceedings of the IEEE*, vol. 62, no. 4, pp. 462–470 (1974).

[FOL84] Foley, J. D., Wallace, V. L., Chan, P., "The Human Factors of Computer Graphics Interaction Techniques," *IEEE Computer Graphics and Applications*, vol. 4, no. 11, pp. 13–48 (1984).

[GAR80] Garret, M. T., "Logical Pick Device Simulation Algorithms for the CORE System," *Computer Graphics*, vol. 13, no. 4, pp. 301–313 (1980).