From:
Dan R. Olsen
Developing User Interfaces
Morgan Kaufman publishers
1998

4

Basics of Event Handling

At the heart of user interface software is the event model for communication between interactive objects and the input/output system. This event model was developed to support the creation of direct-manipulation interfaces. *Direct manipulation* is when the user is presented with concrete objects on the display and then manipulates those objects using the interactive devices.

Before the advent of direct-manipulation interfaces, the interaction architecture was much simpler. Interfaces such as the DOS or UNIX command lines or the LISP interpreter had a very simple architecture. The software would read a line of text from the input, parse the text, evaluate the result, and repeat. Slightly more complicated question-answer dialogs had a similar architecture. Whenever the program, which was always in control, wanted some information from the user, a prompt would be written out to indicate what was wanted. The input line would then be read, parsed, and executed as before. The control architecture was simple because the program was in control. Users did what the program wanted them to do, and when the program wanted them to do it. The architecture for direct-manipulation interfaces is much more complicated than that because the user, not the program, is in control.

Before we look at exactly how user interface software is constructed, we must first understand the environment in which such programs run. It is assumed that our graphical user interfaces will run under the control of a *windowing system*. In this chapter, we will discuss the services performed by such systems and how interactive applications work within them. A modern graphical user interface communicates with other applications, with the windowing system, among parts of itself, and with the code that implements the interface's functionality. The communication is performed by means of

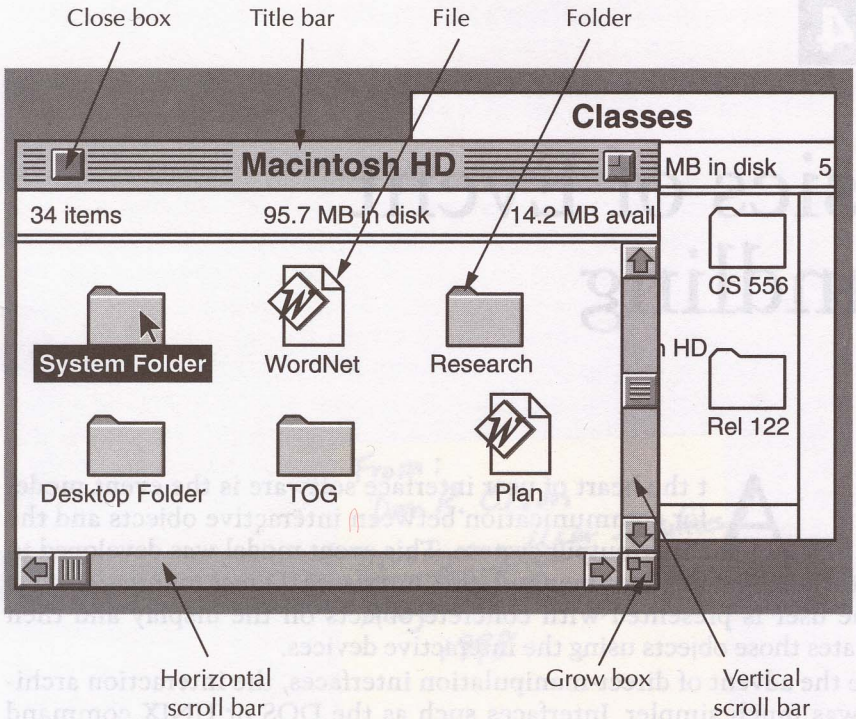

Figure 4-1 Macintosh finder

events. This communication between parts is crucial to the understanding of interactive software and is discussed in detail in this chapter.

To illustrate the kinds of problems that must be addressed, we will consider an example. Figure 4-1 shows a screen dump of the Macintosh finder that is the user interface to the file system. In this example, there are a variety of interactive options available to the user. The user can do these things, among others:

- close the window by clicking in the close box
- move the window by dragging the title bar
- bring the Classes window forward by clicking on it
- drag the WordNet file into the Classes folder

It is this complexity of what can be done that leads us to the event model architecture.

With the variety of options possible, there are several issues that must be addressed in handling events. The major issues discussed in this chapter are

- partitioning between the application and the windowing system
- the kinds of events a window might receive
- distributing inputs to the interactive objects
- models for communication between objects

4.1 Windowing System

On most graphical screens, a variety of interactive applications can be running simultaneously. Each of these applications is decomposed into smaller parts, where each performs some part of the interface functionality. In order to prevent chaos on the screen and in the software, graphical user interfaces are generally built on top of a *windowing system.* There are two aspects to such a windowing system. The first is a set of software services that allow programs to create and organize windows as well as to implement the interactive fragments within those windows. Together these fragments form the user interface. In addition to this software view of the windowing system, a user interface allows the user to control the size and placement of windows. This is called the *window manager.*

4.1.1 Software View of the Windowing System

It is almost universal in graphical user interfaces to decompose interactive objects into a tree, based on the screen geometry of the interactive objects. We can decompose our finder example from Figure 4-1 into the tree in Figure 4-2.

Notice that except for the files and folders in the folder contents area, the geometry of each of the subpieces in Figure 4-1 is a rectangle that is fully contained within the rectangle of its parent. This hierarchy of nested rectangles is the basis for the vast majority of windowing systems used in graphical user interfaces. The primary reason is that nested rectangles form a simple, consistent model for decomposing screen space and for dispatching events. It should be noted that some systems, such as X, provide for nonrectangular windows such as the two ovals required for the program X-eyes 👀 .

There is a wide variation in how much of this hierarchy is managed by the system software. On the Macintosh, only the top-level areas ("Macintosh HD" and "Classes") in Figure 4-1 are considered windows. All windowing systems would call these top-level items windows and would manage them in the system software. In most systems, these are referred to as *root windows.* In systems other than the Macintosh, however, there is only one root window, which is the display screen. For our purposes, we refer to the top-level windows rather than the whole display as the root windows.

Root windows are special because in most systems they provide the user interface abstraction for multiple applications. Each root window belongs to a

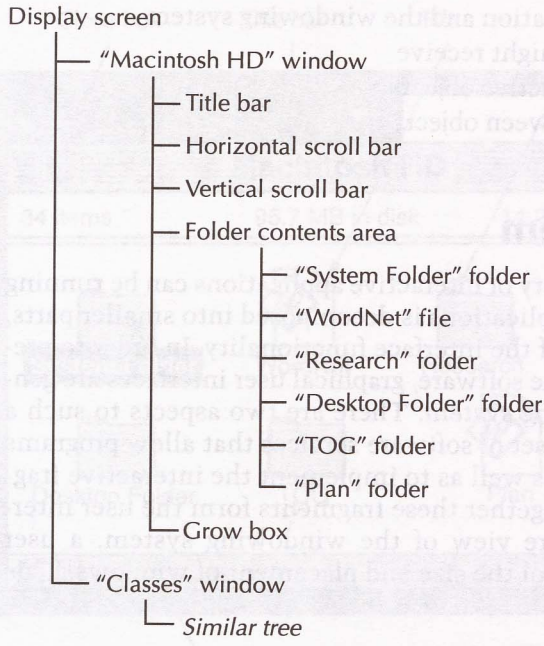

Figure 4-2 Interactor tree for finder example

particular application and is managed by that application. In all modern systems, there can be a variety of applications, with each having a set of root windows. A primary purpose of a windowing system, then, is to arbitrate interactive resources and in particular screen space among these independent applications.

The concept that each root window corresponds to a given application and that all descendant windows belong to that application is generally true in all windowing systems. This distinction, however, has recently been broken down by systems such as Object Linking and Embedding (OLE) and OpenDoc. These systems allow applications to partition a piece of their window space to be managed by some other application. An example of this would be a word processor that allows a spreadsheet fragment to be pasted into a document and then allows the spreadsheet application to manage that screen space. Such architectures, which allow the embedding of one application in another, are very powerful and will be discussed much later in this text in a section by themselves.

In some systems, such as X, NeXTSTEP, or Microsoft Windows, all of the items in this hierarchy would be considered windows and would behave in the same uniform way. It would depend on the application as to whether

items such as the contents of the folder contents area would be windows or would be handled in some other way by the application. In older systems, such as the Macintosh, only the root windows are considered windows. Items such as scroll bars, grow boxes, and buttons are handled under the separate concept of *controls*.

Most windowing systems provide a built-in set of interactive objects, such as text type-in boxes, buttons, scroll bars, or color selectors. These are referred to as either *widgets* or *controls*. Implementing each widget as a separate window is conceptually useful but may incur significant overhead in space and time. In systems like X or NeXTSTEP, a root window must have sufficient information to arbitrate between multiple asynchronous processes. That facility is usually overkill for a simple scroll bar. That is the reason why the Macintosh created the separate notion of controls that manage a rectangular area of screen space within a window. Sun/View has a similar notion of panes within windows. Panes have far less system overhead but provide similar interactive functionality. The Motif tool kit provides gadgets that behave similarly to widgets except that they do not have their own window.

For purposes of this text, we will refer to all of the rectangular regions known to the windowing system as *windows*. This means that windows can recursively contain other windows for the purpose of handling input events and screen space.

Since windows are an important part of the user interface system, many parts of the software must refer to them in some way. This is done by means of a window ID. In some systems, this is as simple as a pointer to the data structures that have information about the window. In others, it is a specially allocated integer that can be used to access internal tables. In almost all systems, the exact nature of this identifier is hidden from the programmer. The window ID is used in conjunction with various system calls to get information about windows and to perform operations on them.

4.1.2 Window Management

In modern windowing systems, the entire screen is viewed as a work area on which various applications can interact with a user. The management of this work area must be under the control of the user, just as a person generally has complete control over how his desk is organized. In order to manage this work space, the user must be provided with a standard user interface. On most windowing systems, root windows usually implement their interfaces using standard code provided by the windowing system. This standard code provides the user's interface for window placement and size and is known as the *window management* code. Providing a standard user interface for the manipulation of windows is very helpful to the users. On most systems such as MS Windows or the Macintosh, the window manager is fixed and standard.

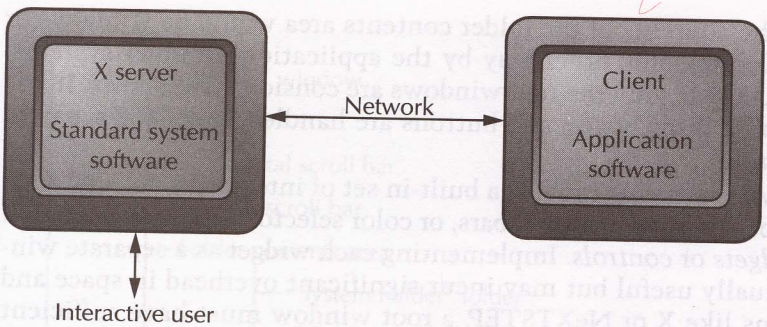

Figure 4-3 X client/server architecture

4.1.3 Variations on the Windowing System Model

There are several variations on the basic windowing system approach. The most notable of these are when the workstation serving the user is not the same as the one running the program. This network-based interaction forces several changes in the basic architecture that are addressed in different ways by X Windows, NeWS, and Java.

X Windows

There are several features of the X Windows system that impose special requirements on its implementation. The first is that the user and the application can be on separate machines, provided that there is a network connection between them, and the second is that the window manager is a separate process.

Most windowing systems are designed to allow the user of a workstation to interact with software running on that workstation. X, however, was designed so that the application software and the user do not need to be on the same machine. Figure 4-3 shows the overall architecture of X.

In the X system, there is a separate process called the *window manager* that takes care of the sizing and positioning of root windows. Notice in Figure 4-4 that there is a frame around the window with various regions that can be used to resize or reposition the window. There is also a title bar with standard buttons for opening, closing, and iconifying windows. This frame area is managed by the window manager, which in Figure 4-4 is the Motif window manager. Having the window manager as a separate process allows a variety of window managers to be used and even allows the implementation of special personal window managers.

Each system that supports X provides an X server program that can interpret X commands from the network and can send interactive events over the network. X client systems also provide Xlib, which provides the software

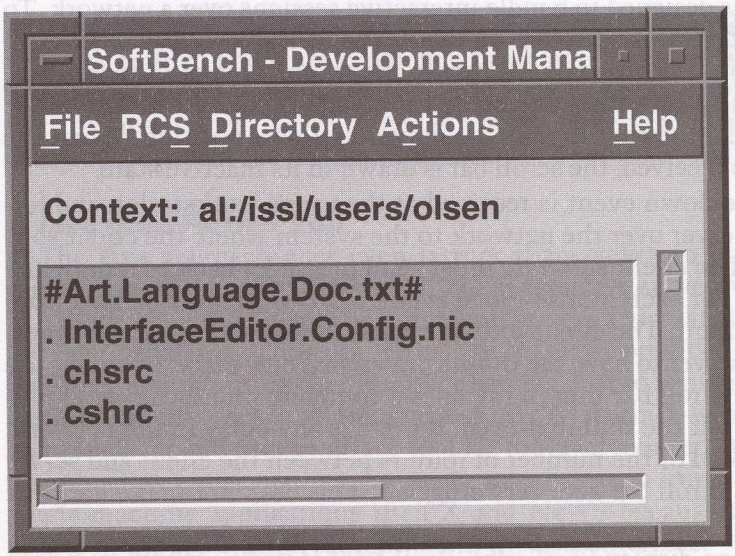

Figure 4-4 Example Motif window

interface to X. Xlib assembles the output from these routines into packets for transmission to the appropriate X server. When the user runs a program, he designates which machine on the network will be the user workstation. Xlib automatically forwards all output to the appropriate server on the appropriate workstation.

When a user generates interactive inputs, X must uniquely determine which window should receive those inputs and then forward the input events over the network to the system that is running the client software for that window. This means that, unlike the other windowing systems that must only arbitrate interactive resources among multiple applications on the same machine, X must arbitrate between applications running on different machines and possibly controlled by different users.

This client/server architecture provides great flexibility in the way that users interact over the network. It does, however, complicate the way in which X must handle input events. For the most part, however, X hides these problems from the application programmer. Therefore, we can discuss X as if the network issues did not exist.

NeWS Goes One Step Farther

There are two problems with the X system that are addressed in NeWS. The first of these is that all user interactions must make a round trip over the network. The second is the graphics model used. NeWS uses display PostScript, which is based on splines rather than the more line-oriented model of X.[1]

Like X, NeWS is designed to handle interactive sessions over a network. To understand the differences between the two, let us consider the case of a scroll bar. When the mouse-down event occurs in the thumb area of the scroll bar, the thumb area must be redrawn to show that it is selected. As mouse-move events are received, the thumb must be drawn in a new position. When the mouse-up event is received, the scroll bar is drawn in its inactive state.

In X, the mouse-down event is received at the user's workstation and forwarded by the X server over the network to the system where the client software is running. The client software must send messages back to redraw the scroll bar thumb. For each mouse-move event, a message must be sent from the server to the client. The client software must then send one or more messages back to the server to have the thumb drawn in a new place. Every mouse movement on the scroll bar thumb involves a complete round-trip set of network messages to get the thumb moved. If there is any delay in the network, due to excessive load or the number of routers between the client and server workstations, the scroll bar acts very sluggish. This does not make for great usability.

NeWS is based on display PostScript. PostScript is itself a graphics system built around the language FORTH.[2] FORTH is a relatively fast interpretive language. PostScript defines all of its graphics routines as commands in FORTH and was originally designed to drive laser printers. Instead of sending the pixels for what is to be printed to the printer, application software sends a FORTH program with PostScript commands. The printer then executes this FORTH program to generate the printed image. This approach greatly reduces the network traffic required for high-resolution printing. FORTH also provides the ability to create new FORTH commands at run time, which can then be used by other parts of the FORTH program.

Display PostScript uses the same FORTH base and the same PostScript drawing commands. In addition, it defines commands for handling input events. A program can either download FORTH code into the NeWS server that will handle input events locally or it can have the inputs forwarded back to the client software. Remember that, as with X, the client software supports the application code and the server supports the user.

In our scroll bar example, we could do the following. When the mouse-down event is received, we can send the complete code for drawing the scroll bar as a FORTH procedure to the server. In addition, we can send code that converts the mouse-move events into invocations of the draw procedure for the scroll bar. Using this technique, there is no network traffic involved while dragging the scroll bar's thumb. When the mouse-up event is received, the client software is notified and whatever is being controlled by the scroll bar can be updated.

The key to NeWS is that, unlike X, the server can be programmed at run time using the FORTH language. This allows significant functionality to be

programmed into the server by each application and thus can greatly reduce network traffic. There are problems, however, with the fact that the server may have limited resources and may be serving for multiple applications. In spite of these issues, NeWS provides a much more flexible and efficient interactive system than the simpler X model.

Java Goes Farther Still

The Java approach goes even farther than NeWS in downloading entire interactive applets into the server. The interactive server (usually embedded in a World Wide Web browser) is implemented as a virtual machine for executing Java programs. Thus, as with FORTH, Java provides a mechanism for remotely programming user interface servers. These applets function as complete interactive applications and use AWT (abstract window tool kit, the graphics layer and user interface tool kit for Java) to interact with the user. By downloading entire applets, Java user interfaces are implemented more like single-machine applications than X and NeWS because the networking is deeper in the application rather than in the user interface.

4.1.4 Windowing Summary

A windowing system breaks the screen space into semi-independent fragments. The programmer writes software for such fragments or reuses previously implemented fragments (widgets) to provide user interface functionality for an application. These fragments are then composed, generally in a tree structure, to form complete applications. The windowing system is responsible for arbitrating interactive resources such as screen space and input events among the various interactive fragments. Generally each fragment is a separate window. The window manager is software that provides a user interface for controlling placement of the root windows.

4.2 Window Events

As a user interacts with the input devices, these actions are translated into software events that are then distributed to the appropriate window. In some systems, such events are called *messages,* but the concept is the same. It is important to note that these messages or events do not always involve interprocess communication. Most message passing has the approximate cost of invoking a procedure rather than the cost of an operating system message.

Before discussing how events are distributed to windows, it is helpful to characterize the kinds of events that a window might receive. All events are identified by an *event type* that allows the receiving software to distinguish the kind of information that the event is intended to communicate. In

addition to the event type, there is other information, such as the mouse position, the window to which the event is directed, or the character key that was pressed.

4.2.1 Input Events

Among the most important events are the input events generated by the user. Most such events carry with them the current mouse position, since that information is necessary to dispatch the event appropriately. In general, event records carry with them the identifier of the window that should receive the event. On older systems, such as the Macintosh, the programmer must make a separate call to convert the mouse position into a window ID.

Mouse Button Events

In mouse-based systems, the mouse button events are among the most important because they provide the point-and-click functionality found in most modern interfaces. A large amount of interactive functionality is associated with handling the mouse-up and mouse-down events. The mouse button events always carry the current mouse location with them.

To understand how these events are used, take the example of the finder application shown in Figure 4-1. If the user positions the mouse over the close box (the square box on the far left of the title) and presses the mouse button, an event signaling a button down is sent to the title bar object. The title bar must compare the mouse location with the position of the close box. If the mouse is inside the close box, the title bar interactor highlights the close box and remembers that fact. When the user releases the button, a button-up event is sent to the title bar. The close button is unhighlighted and the window is closed.

The interactive button syntax is actually more complicated than this because the mouse may have moved. The button-down event may have occurred outside of the close box, with the button-up event occurring inside the close box. Designing the complexities of button syntax is left for a later discussion.

The way in which mouse events are defined by a windowing system has a lot to do with how that system was designed relative to a particular hardware configuration. For example, the Macintosh tool box assumes that all mice have only one button and therefore there is only one event for button down (when the button is pressed) and one for button up (when the button is released). Microsoft Windows, in contrast, defines separate events for the left, middle, and right buttons of a three-button mouse.

X, however, was defined to run on a variety of platforms across the network and therefore defines a single button-down event and a corresponding button-up event. Each of these events carries with them the number of the button

that was actually pressed. This allows application software to support any number of buttons. The flexibility of X is a problem, however, in that software that is written to rely on a three-button mouse is very difficult to use on hardware that only has a one-button mouse. For example, running UNIX programs—which use three buttons—from an X server program on a one-button Macintosh can be difficult.

Modifiers

Focusing so much interactive behavior through a limited number of mouse buttons has its problems. It is difficult to do 10 different interactive actions on a given object with only one or even three buttons. Many applications solve this by using special-function keys and double-clicking. All keyboards supply a Shift key and a Control key; many of them provide Alt, Meta, or Option. Many applications do different things in response to mouse button events, depending on whether or not one of these modifier keys is pressed.

There is a problem in how such modifiers should be handled relative to mouse button events. Defining separate events for each modifier key (down and up) leads to a lot of events and requires a lot of event logic for each handler. It also requires that the application software pay attention to when the modifiers are pressed relative to when buttons are pressed. Most systems define a "modifiers" or "flags" word, which is passed with the button events. This word assigns one bit to each modifier key. If a modifier key is down at the time the button event occurs, its corresponding bit in the modifiers word will be on. This simple mechanism provides the application software with great flexibility in ignoring or assigning meaning to such actions as Control-Alt-Shift-MiddleButton_Up. Many systems also add a modifier flag for the state of each of the mouse buttons. This allows applications to assign meaning to having multiple buttons pressed at the same time.

Double-Clicking

When the original Macintosh appeared with only one mouse button and only the Shift and Command keys, the set of possible mouse actions was limited. The Macintosh associated meaning with the notion of a double-click. A double-click is a second button-down event that occurs within a short amount of time from the previous button-down event. The length of time used to determine if a button-down event should be recast as a double-click event is settable by the user in all systems that define this concept. In Microsoft Windows, there is a separate double-click event for each of the three buttons.

On the Macintosh, the double-click is universally associated with opening an object or accessing more information about it. Interpreting double-clicks can be problematic because when an object receives the first button-down event it has no idea whether a double-click event will follow. This is resolved

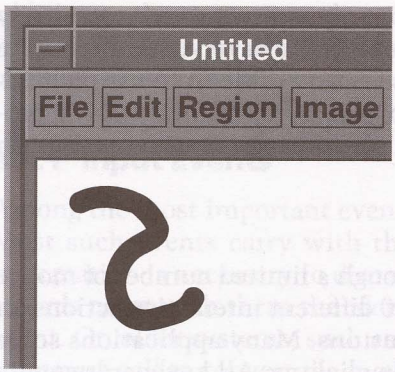

Figure 4-5 XPaint using mouse motion

in most situations where a double-click has meaning by defining the first button-down event as a selection of the object and the double-click as the opening of the object. If the double-click never arrives, the original button-down event still has meaning. This, however, is only a convention, not a built-in feature of the system. Because of the influence of the Macintosh on graphical user interfaces, this double-click convention is widely used.

Because X was defined to run over a network on machines that—at the time—were relatively slow, the double-click was a problem. Double-clicking depends on accurate timing of the input events. With multiple processes and network delays, such timings were not practical. X, therefore, has no double-click event. Because of the Macintosh influence, however, many X applications simulate the double-click in the application code by comparing the time stamp of the events.

Function Buttons

Most hardware configurations provide many more buttons than those on the mouse. In earlier graphics systems, all buttons were treated uniformly. On most current workstations, these function buttons are physically mounted on the keyboard and so most windowing systems treat nonmouse buttons as special keys on the keyboard. In X, the resolution of this issue depends on the implementation of a particular X server. The X model is flexible enough to accommodate either choice. The client software, however, must adapt to whatever choices the actual server makes.

Mouse Movement Events

Figure 4-5 is a snapshot of XPaint in the midst of sketching a curve. In order to sketch this curve, the paint window object needs to know the position of the mouse at much more frequent intervals than on the button-down and -up events.

In very early graphics systems, mice and other locator devices were treated as sampled devices; that is, whenever the application software needed to know the location of the mouse, it would call a routine (or read a hardware I/O register) to obtain the current mouse location. With the advent of event-driven interfaces, this sampled model was no longer appropriate. In particular, user interfaces running on multitasked systems such as UNIX have problems with sampled devices.

In order to implement a tool like XPaint using a sampled approach, the application would enter a loop of reading the mouse and then updating the screen. Such a tight inner loop steals machine cycles from all other processes. This is a problem because process-time slices are on the order of 1/60 of a second and interactive input requires only about 1/5 of a second. Such a loop is extracting information and performing processing at 12 times the rate actually needed by the user interface. Thus a large amount of computing resource is wasted and cannot be used by other applications.

In the early Macintosh, only one application could be running at once. Based on this, the system provided a NULL event whenever there were no other events pending. Mouse motion dialogs such as painting would be driven by this NULL event. The advantage was that software was now entirely event driven. This simplified the software architecture. Such an approach in a multitasking system, however, would have the same problems as the sampled approach. In the Macintosh, with only one process, the system had nothing better to do. This made running through code to process NULL events as good a use of the CPU as doing nothing.

In order to resolve the mouse motion issue for multitasking, most systems now provide a mouse motion event. The windowing system generates a special event each time the mouse actually changes position. This means that if a user starts a compile while a paint program is running and walks away to talk with someone, the paint program does not compete with the compiler for cycles because there are no events being generated for the paint program if the mouse is not moving.

Our XPaint example points out an additional optimization that can be made. Most cases where a window needs continual notification of the mouse movement occur when one of the mouse buttons is currently pressed. This applies to most painting, drawing, and dragging operations. X, MS Windows, and NeXTSTEP all provide mechanisms for masking off all mouse motion events when there is no button being pressed.

In X, where the input devices may be on a different machine from where the client software will handle the inputs, further refinements become necessary. The problem is the latency in the network connection between the server and the client. X provides a PointerMotionHint option that only reports motion events when 1) the pointer (mouse) has actually moved and 2) the client software actually asks for the next event. In situations like echoing

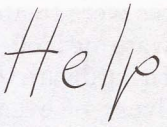

Figure 4-6 Handwriting recognition

menu selections, the exact trace of the mouse location over time is not important. What is important is the current location at the time the menu selection is to be updated.

The network and process latency can also cause a problem when very fine information about the mouse location is required. Such a case occurs when trying to recognize handwritten gestures, as in Figure 4-6. Many current handwriting-recognition algorithms sample the input devices at a much higher resolution than the screen display. Some such algorithms also use the timing of the motion events as part of their recognition. Recent versions of the X server support the concept of a motion history buffer that allows the server to accumulate motion events and to provide them in a block when they are requested by the client software.

Mouse-Enter and Mouse-Exit Events

A more common set of motion events are the mouse-enter and mouse-exit events. These events are generated whenever the mouse cursor enters or leaves a window. An example use of this is the action of a button on the screen. When a button on the screen is pressed, it must be highlighted. If, while the mouse button is still down, the user moves the mouse outside of the window associated with the screen button, then the screen button is unhighlighted. We could program this with mouse motion events but that is much more precision than is required. Sending mouse-enter and mouse-exit events to a window is relatively simple for the windowing system and has very little overhead.

Keyboard Events

Keyboard events are special for a variety of reasons. In its simplest form, a keyboard can be considered an array of buttons. Each button (key) can generate an event for key down and another for key up. There are several complications with the handling of keyboard events.

First, keyboards are not standard. In the US, the set of letters and special symbols is fairly standard; however, the placement of some special characters (physical button assignment) is not standard. When we leave the US for the UK, some symbols—such as currency—change. When we go onto the European continent, we encounter variations in the letters with various inflections

and accents. When we enter Asia, we encounter nonphonetic systems such as Kanji characters in China and Japan. This array of buttons called a keyboard can have an enormous range of meanings.

A second problem is that keys tend to work together. The "A" key has a different meaning when the Shift key is pressed. This is further complicated by the Control, Alt, Meta, and Command modifiers on various keyboards. All of these combinations must be accounted for. At the extreme are computer games, in which holding various keys down simultaneously has meaning, much like playing chords on a piano.

The last problem in deciding what to do with a keyboard event is the notion of accelerator keys, as provided by most windowing systems. On most Macintosh applications, any key that is pressed in conjunction with the Command or Apple key ( ) is not passed directly to the application windows but is captured and forwarded to the menu system as a surrogate for selecting a menu item. If no menu item has this key associated with it, then it can be forwarded to the application. In Motif, the Alt or Meta key serves the same function. In fact, in many applications a single modifier key is not sufficient for all such accelerator keys because there are more than 24 choices that could profit from a key binding.

The complexities of keyboard handling will not be discussed at this time. For our current discussion, it is sufficient to understand that each key press and release will generate an event. With the event is an accompanying *scan code*, which essentially defines the actual key that was pressed or released. All windowing systems provide a routine or set of routines that can translate these raw key events into a more acceptable form. Scan codes can be replaced by ASCII or UNICODE characters. Accelerator keys can be filtered out and appropriately handled. Modifier keys will have their actual events intercepted and instead will participate in the conversion to ASCII codes and the provision of modifiers/flags as discussed earlier.

In general, most interactive objects handle a key-down event that has been translated into ASCII. Dealing with the additional complexities of keyboards will be left for later in this chapter.

4.2.2 Windowing Events

The next major class of events that a window can receive has to do with managing the window itself. Most windowing systems will send an event when the window is first created. This event allows the window to be initialized. A corresponding event is sent when the window is being destroyed, which allows the application code to clean up any data associated with the window. A similar pair of events can be associated with opening and closing the window. In some window systems—such as X, MS Windows, or NeXTSTEP—a window can be reduced to an icon and set aside; there is a pair of events

associated with iconifying and deiconifying a window. Some windowing systems such as the Macintosh have the notion of a currently selected window. A window receives one event when it is selected and another when it is deselected. These events allow the window to change its appearance depending on whether or not it is selected.

One very important event notifies the application code that the window has been resized. This event allows the software to adjust what is displayed in the window to accommodate the change in size.

4.2.3 Redrawing

An important issue is how a window's visual display gets updated in response to resizing, obscuring, data changes, and all other things that might happen. Let us again consider our example from the Macintosh finder in Figure 4-1. In this example, the "Macintosh HD" window is obscuring most of the "Classes" window. If the user were to select the "Classes" window by clicking on it, the Macintosh style guidelines indicate that the "Classes" window would be brought to the top. This would mean that the portion of the "Classes" window that is obscured by the "Macintosh HD" window would need to be drawn over the top of the "Macintosh HD" window. Such drawing actions are also required if a NeXTSTEP window is converted from an icon to a full window.

The problem is that the windowing system itself has no capability to draw what has been hidden without help from the application. Some older systems such as the AT&T Blit terminal provided off-screen bitmaps for the hidden portions and modified the drawing primitives to actually draw into these hidden areas if necessary.[3] Some form of this is provided in the X Windows concept of *backing store.* In general, however, saving copies of hidden portions of a window can be very costly in space and is not frequently done.

The windowing system needs a mechanism for having the application redraw the newly exposed portions of the window. This is done by sending the window a *redraw event.* This event carries with it a rectangle or region of the screen that should be redrawn. In X, these are called expose events; in NeXTSTEP, it is the drawSelf event; in MS Windows, it is the WM_PAINT event; and on the Macintosh, it is the updateEvt event.

One of the most confusing concepts for new programmers of window-based interfaces is the fact that you can't just draw in the window. Yes, there are routines for drawing in the window but these draw instructions must be performed in response to a redraw event. The application draws when user actions require it to draw, not when the software decides it wants to draw. The implementation of handlers for redraw events will be discussed in great detail in Chapter 5. The issue again is that an application is not in control of all of

```
Initialization
while(not time to quit)
  { Get next event E
    Dispatch event E
  }
```

Figure 4-7 Generic main program

the situations that may modify the screen display. Such applications must therefore be designed to respond to whatever might occur.

4.3 The Main Event Loop

Having discussed how windows are organized in a windowing system and the set of events that a window must be prepared to receive, we now need a general understanding of how events are actually processed. Event-driven programs are very different from traditional software architectures.

When most programmers first start working with windowing systems, they take a sample program, locate "main," and start reading to understand how the program works. This is very frustrating in interactive programs because there is little or nothing in the main program. The main program for all windows-based applications is shown in Figure 4-7.

In the initialization, the application will define windows, load information about the application, and generally set up things. This usually involves creating a main window for the application. In many systems, the main window will not appear until the event processing starts. The reason for this is that no drawing is done in the new window until a redraw event has been received and dispatched to that window, after which the window will appear.

The actual calls to create windows vary from system to system and will not be discussed in this text. Suffice it to say that the initialization must create at least one window that the user can interact with. Subsequent actions may create more windows but one is required to get started. That window may be as simple as a menu bar (as on the Macintosh), but there must be some visible object to manipulate and all such objects exist in windows.

If you read this main program with the intent of understanding the application, you find that it is fruitless. The routine to get the next event and in many cases the routine to dispatch the events are system calls that cannot be read. Reading the main program is like following a creek only to have it immediately go underground. The real functionality of the program is in the code that handles events that are being passed to windows.