X Window System

Editor's Note: Cross-references in the text refer to chapters in the companion book, *UNIX: The Complete Reference, Second Edition,* by Rosen, Host, Klee, Farber, and Rosinski.

The *user interface* is the part of the UNIX System that defines how you interact with it—how you enter commands and other information, and how the system displays prompts and information to you. Chapter 7 discusses the KDE (K Desktop Environment), which is the primary Linux user interface. Chapter 7 also discusses the CDE (Common Desktop Environment), which is the user interface for Solaris, HP-UX, and a few other UNIX systems (you can also get CDE as an add-on package for Linux). Both of these environments were developed to provide a common user interface built on a graphical user interface (GUI) concept called the X Window System, which is the topic discussed in this chapter. This chapter is dedicated to UNIX users who either still use the X environment or wish to understand how CDE, KDE, and other visual user interfaces, such as GNOME and UDE, evolved, as well as the many toolkits that have been developed for these windows management environments. This chapter does not attempt to help you configure X, since the procedures are specific to the variant of UNIX that you run and can be very detailed. Much better sources of information are available with the X implementation that you choose to run on your system. Rather, this chapter is devoted to understanding the underlying principles for the many GUIs that have been built on top of the X Window System.

The user interface is sometimes referred to as the system's "look and feel." For a number of users, the primary interface to the UNIX System is the command-line interface provided by the shell. But GUIs such as X provide a more visual way to interact with the system that is easier, more effective, and more enjoyable.

GUIs replace the command-line style of interacting with the UNIX System with one based on menus, icons, and the selection and manipulation of objects. Instead of having to remember commands and command options, you work directly with graphical representations of objects (files, programs, pictures, lists) and select actions from menus rather than typing their names.

Graphical interfaces are now in common use on PCs running environments such as Windows 2000/XP, and UNIX System GUIs share many features with them. However, graphical user interfaces for the UNIX System have some special characteristics that meet the particular needs of UNIX System applications. Specifically, to be generally usable with the UNIX System, graphics environments must support networked applications, must permit applications to be independent from specific display and terminal hardware, and must allow graphics applications to be easily portable across the variety of hardware that the UNIX System runs on. The original standard UNIX System graphics environment that meets these needs is the X Window System.

Like most readers of this book, you have probably already used some windowing system—perhaps the Apple Macintosh or some version of Microsoft Windows—and you are probably familiar with the basic ideas of graphical user interface technology, such as windows, a mouse, and mouse-operated menu selection. Although windows and graphical user interfaces have become familiar through PC products such as Microsoft Windows as well as the Macintosh, it was the modular, innovation-friendly architecture of the UNIX System that enabled and pioneered the development of early windowing systems, such as the Bell Laboratories BLIT and Sun Microsystems' SUNVIEW, from which the developers of the Apple Macintosh and of Microsoft Windows took their cue.

The X Window System incorporates all the user-interface capabilities of its contemporaries, and it adds some very useful ones of its own. At the same time, it follows the UNIX philosophy of being modular—and therefore innovation-friendly—because experimental replacements for small modular tools are easier to build than replacements for a complex conglomeration of operating system, windowing system, and user interface manager such as the Apple Macintosh or Microsoft Windows 2000/XP. And following the UNIX tradition, it empowers the user to customize the user interface to match his or her individual aesthetic preferences, cognitive style, and work skills. This is the primary focus of this chapter: showing you how you can take advantage of the flexibility and customizability of the X Window System on UNIX to customize and individualize your work environment. Whatever system you have as a starting point, this chapter will show you how to make it look, feel, and work the way you want it to.

This chapter deals with the user's view of the X graphical user interface, and how you can use it to make your own UNIX System GUI environment. The discussion begins with an overview of the X Window System and some popular UNIX GUIs. The rest of the chapter focuses on how to customize the X environment to suit your own needs and preferences.

What Is the X Window System?

The X Window System is a comprehensive graphical interface and windowing environment for developing and running applications having networked, graphical user interfaces. It was developed by Project Athena at the Massachusetts Institute of Technology and is now owned and distributed by the X Consortium. It became a standard component of most UNIX systems, available as an add-on package from various software vendors or in a public domain version.

The main concepts on which the X Window System is based include a *client/server model* for how applications interact with terminal devices, a *network protocol,* various *software tools* that can be used to create X Window–based applications, and a collection of *utility applications* that provide basic application features.

The X Window System is the standard basic windowing system platform for current versions of the UNIX System operating system, and for many others as well. The X Window server, the software that actually controls the user interface hardware—your keyboard, pointer, and one or more screens—often runs as a process under the UNIX system of a personal computer or workstation; but it may run on other platforms, including computers running Microsoft Windows or Apple Macintosh operating systems, and even stand-alone X Window terminals, running the X server from firmware, without any operating system at all.

The X Window System is a *network* windowing system. That means that an X server can provide a user interface not only to client processes running on the same UNIX computer or workstation as the X server itself, but also to programs running on other computers connected to the same network—even a very large network such as the global Internet. You can use the X server running on your own desktop from a computer located at a remote location, even on the other side of the world. Some X Window System users in New Jersey have used their desktop X servers to run client programs on machines as far away as New Zealand and Singapore.

Under UNIX, the client connects at startup with the X server designated by the environment variable value *$DISPLAY,* or by the argument that follows the **–display** option on its command line. This value starts with an endpoint identifier, such as a DNS name (such as "mymachine. myorg.net") or an IP numerical address (such as 123.45.67.89). Three reserved identifiers—*unix, localhost,* and (blank)—refer to X server processes running under the same UNIX system as the client. The endpoint identifier is followed by a colon (:), and a "display" number. This is a small number that identifies a specific X server at the given endpoint address, usually a 0 (zero) on a platform that supports only one X server at a time. Note that each X server is meant to control a complete set of user interface hardware: screen(s), keyboard, and a pointer device such as a trackball or mouse. Some UNIX systems, such as Sun workstations, can support several complete sets of user interface hardware through backplane plug-in boards and associated X server processes. The final, optional part of the display value is a period followed by the number of the screen on which the client program is to display its windows. Platforms that support more than one video frame buffer, such as Sun workstations, have X server software that can display client windows on any one of them. The default screen of a display is always numbered "0." The complete value of *$DISPLAY,* or the argument after the **–display** command-line option, usually looks something like mymachine:0.1 or 123.45.67.89:1.

A *server* is a process that lets several other processes—its *clients*—share some physical or logical resource. Just as a file server lets several processes—usually the kernels of several workstations—share the files on a central file system, an X server lets several client processes share access to hardware that provides them with an interface to their human user. This hardware—screen area, keyboard keys, pointer position and buttons—needs to be shared among client processes in some way. With rare exceptions, an X server shares its resources among client processes under the direction of a master client process called the *window manager.*

The sharing of resources, such as screen area and keyboard keys, is done under user control. This user control requires an interactive user interface: Most window managers create and control a frame around each client window to let the user identify, move, resize, and restack—to the top or bottom—application windows on the screen. Most window managers also provide one or more menus, usually invoked from the window frame for window-specific functions, and from the background, or *root window,* for other functions, for example to refresh all windows (in case of a graphics malfunction) or to execute a UNIX command (such as the **xlock** command to lock the screen and the keyboard until the user unlocks it by typing the password).

The window manager client is special in some respects; for example, only one window manager can connect to a single X server at a given time. But in many ways it is just another client. For example, there is no requirement that the window manager run on the same UNIX machine as the X server it controls—and for X servers running on dedicated hardware "X terminals," the window manager, like all clients, must be run from UNIX systems on the network. In addition to customization with resource variables, to be discussed shortly, most window managers have one or more special initialization files to define special capabilities, such as the content of menus.

The Client/Server Model

A fundamental X Window concept is the separation of applications from the software that handles terminal input and output. All interactions with terminal devices—displaying information on a screen, collecting keystrokes or mouse button presses—are handled by a dedicated program (the *server*) that is totally responsible for controlling the terminal. Applications *(clients)* send the server the information to be displayed, and the server sends applications information about user input.

Separating applications (clients) from the software that manages the display (the server) means that only the server needs to know about the details of the terminal hardware or how to control it. The server "hides" the hardware-specific features of terminals from applications. This makes it easier to develop applications, and it makes it relatively easy to port existing X Window System applications to new terminals.

For example, suppose the instructions for drawing a line differ on two different terminals. If an application communicates directly with the terminal, then different terminals require different versions of the same application. However, if the specific hardware instructions are handled by servers, one application can send the same instruction to the server associated with each terminal, and the terminal server can map it into the corresponding control signals for the terminal. As a result, the same application can be used with many different terminal devices.

With the client/server model, each new terminal device requires a new server. But once a server is provided, existing applications can work with that terminal without modification.

X applications (clients) can run on multiple hosts and on a workstation. These applications are accessible from workstations or X terminals (servers) either on the same machine or distributed in a network. Note that on the display there is no distinction between an X application running on the local machine and one running on a remote machine.

The existence of a special server for each type of terminal is one part of the client/server model. The other is the use of a standard way for client applications to communicate with servers. This is provided by the X Window System protocol.

The X Protocol

The X protocol is a standard language used by client applications to send instructions to X servers and used by servers to send information (for example, mouse movements) to clients. In the X Window System, clients and servers communicate *only* through the X protocol.

The X protocol is designed to work over a network or within a single processor. The messages that go between a client and a server are the same whether the client and server are on the same workstation or on separate machines.

This use of a network protocol as the single, standard interface between client and server means that X Window System applications, initially developed to run on a workstation that has its own attached display, can automatically run over a network.

The X Library

The X protocol is designed to work efficiently over a network. However, it is not a good language for developers to use for developing applications. The X Window System provides a standard set of C language routines that developers can use to program basic graphics functions, and that automatically produce the corresponding X protocol. These routines are referred to as the X library routines, or *xlib.* Xlib provides a standard programmer's interface to the X Window System.

Toolkits

Xlib itself provides relatively low-level functions. It deals with basic graphics elements like drawing a line, filling a region, and so forth. To further simplify application development, higher-level routines have been developed to produce more complex elements, for example windows, menus, or scrollbars. Higher-level elements like these are called *widgets.* A toolkit is sometimes called a *widget set.* Typical widgets include scrollbars, buttons, forms, and similar components.

The X Window System distribution includes a library called the *Toolkit Intrinsics* (libXt, with functions whose names begin with the "Xt" prefix). The Toolkit Intrinsics library is a foundation on which different vendors can build toolkits that support their graphical user interfaces. To provide vendors with an example of how libXt can be used to build a toolkit, the X distribution includes a simple toolkit—the Athena toolkit from the MIT Athena Project—and several sample applications, such as the popular terminal emulator XTerm, built on top of the Athena toolkit.

Two groups of vendors developed widely used intrinsics-based toolkits. Olit, a toolkit produced by Sun, AT&T, and Novell, supports the Open Look GUI standard, which is still used by many UNIX users running System V Release 4. The Open Group (formerly the Open Software Foundation), sponsored by many vendors including Hewlett-Packard, IBM, DEC, and NCR, developed an intrinsics-based toolkit called Motif, which supports the Common Desktop Environment (CDE) GUI, discussed in Chapter 7. Although these two efforts started out as competing GUIs, the continued development efforts on both of the native interfaces and their toolkits resulted in a richer, more useful set of capabilities that have been merged into the CDE and its toolkit. This effort was largely accomplished by the Common Open Software Environment (COSE) initiative. A brief history of this evolution is also discussed in Chapter 7.

The Common Desktop Environment (CDE)

The Open Group (then OSF) produced the Motif toolkit and the Motif Window Manager (**mwm**) to support the CDE GUI standard, favored by vendors who support both UNIX-based workstations and Intel-based (Microsoft Windows, IBM OS/2) personal computing environments. The CDE specification was originally developed by IBM to provide GUIs compatible with the user interface of Microsoft Windows on other platforms such as OS/2 and UNIX/X. This specification was then standardized by the Open Group and has been adopted for use by many different UNIX system vendors.

CDE addresses the need for user interface compatibility for people who must use multiple computing platforms. For example, you might do most of your work under UNIX, using its extensive customization capabilities and easy assembly of automated "work engines" for specific work from application piece-parts. However, you may also need to run some packaged applications—for example, a print optimizer to enhance the production of sophisticated color graphics on a specific color printer—that are simply not available for UNIX and can only be used under Microsoft Windows or OS/2. A person who works in any given user interface environment develops automatic work-optimizing habits that will be carried, sometimes without conscious intention or awareness, to all their other work environments. If the several environments used by the same human are incompatible—if they require different actions and habits for equivalent steps in the human's work—this "transfer" of automatic habits from one environment to another would be *negative,* meaning that it would interfere with doing the job, with results that might range from annoying to disastrous. By specifying user interface components that look and work much like their counterparts from Microsoft Windows, CDE not only prevents disasters that could result from *negative transfer* of skills but encourages *positive transfer,* so that user habits formed in each environment enhance the user's performance in the other.

The CDE standard includes, in addition to the Motif toolkit and the Motif window manager, a suite of applications designed to emulate, under the X Window System, most of the frequently used Microsoft Windows tools: a file manager, session and application managers, a calendar, a mailer, and a windowing shell. Some users find these tools to be less attractive than the many sophisticated applications that are available for free in the UNIX environment, which can be invoked with commands from a UNIX shell window. But for users who must use equivalent applications under both MS Windows and UNIX, these CDE applications are very useful. Some vendor distributions of the X Window System include the Motif toolkit and **mwm**, but not the rest of the CDE applications and tools. The latter are typically distributed in the directory */usr/dt* ("dt" stands for "desktop"); if you have this directory, then CDE applications and tools are available on your machine. They may be customized with resource variables, in much the same way as other X Window System applications built with any intrinsics-based toolkit.

The Window Manager

The window manager in the X Window System is a client application that provides the basic window management and manipulation functions that you use in interacting with the system. This includes the basic layout of windows, borders, menu appearance, the creation and elimination of windows, moving windows, managing keyboard and color mappings, and iconifying windows. Together, these functions make the window manager the main determinant of the overall look and feel of your system.

Just as the UNIX System encouraged innovation in character-oriented user interfaces, or "shells," by moving user interface functions out of the operating system into a separate module that could exist in many versions such as the C shell, Bourne shell, Korn shell, and so on, so the X Window System puts its own interface with the user into a separate software module called the *window manager.* And just as the modular nature of the shell led to alternative shells, many alternative window managers have been developed. The X Window System does not dictate a specific "look and feel," the way a PC GUI such as Microsoft Windows does, for example. Two different X Window System–based applications can have very different appearances and styles of operation. They may differ in the ways in which menus and actions are represented, in the way your application turns a window into an icon, and in other fundamental features. Although this flexibility has value, the resulting inconsistencies can defeat the potential benefits of having graphical interfaces.

To avoid this problem, products were developed to provide a consistent user interface both for the UNIX System as a whole and for applications from different vendors. One of the two original UNIX window managers is the Motif Window Manager, **mwm**, from the Open Group and its descendants, upon which CDE was built. The other is the Open Look window manager, **olwm**, from Sun Microsystems, together with relatives like **olvwm**, which manages windows on a "virtual screen" much larger than the real screen actually in front of the user.

Most X Window System applications are built from general-purpose reusable software objects called *widgets* and *gadgets.* Libraries of those objects are known as *toolkits.* The most popular toolkits follow either the Motif/CDE or the Open Look user interface conventions; **mwm** and **olwm** were written to work in ways consistent with the widgets of Motif and Open Look toolkits.

Motif has a GUI look and feel that was developed by the Open Group (formerly OSF), based on work by DEC and Hewlett-Packard. It was designed to be similar to Microsoft Windows and IBM's Presentation Manager.

Open Look (OL) was developed by Sun Microsystems and AT&T, based on previous work by Xerox, and on previous Sun GUIs. It was originally the most common X Window System GUI on Sun platforms, and it is still used heavily even though CDE exists.

Although there are clear differences in graphic design and appearance between Motif and Open Look, and although there are differences in specific features (for example, Open Look's pinned menus), both Motif and Open Look will seem familiar to users of current PC GUIs.

You should keep in mind that these CDE- or OL-compliant default environments are just starting points. As you develop individual work habits and preferences that optimize your personal productivity, you will be able to adjust your own X Window System environment to whatever works best for you.

Functions of the Window Manager

One of the main functions of the window manager is to arbitrate the sharing of screen space among the simultaneously active windows of different applications. For this reason, the window manager controls the user's interface for moving, resizing, and reshaping application windows. Many window managers use the metaphor of overlapping sheets of paper on a desktop to set up the stacking order of overlapping windows, giving the user the ability to move windows "back," to lie under others directly on the desk surface, or to the "front," metaphorically on top of all the other windows or papers, unobscured to the user. The windows of temporarily unused applications can be *iconified* (referred to as *minimized* in the CDE environment or *closed* in OpenLook) into a small, usually pictorial window called an *icon,* again under the direction of the window manager. To control all of these window-specific functions, most window managers display a *frame* around each application window, with special mouse-draggable controls such as corners for resizing a window. The full set of other window-specific functions is normally available from a menu, which appears when the appropriate mouse button (the ACTION button in Motif, or the MENU button in OL) is pressed in the title bar of the application window.

Besides the frames it puts up around each application window, the window manager also controls the "root" window, the backdrop—analogous to the desktop surface in the papers-on-desk metaphor—on top of which application windows are displayed. When you operate the menu-evoking mouse button over the root window, you will get a menu of window manager functions that pertain to the whole X Window System server, and not just to the window of some specific application. Some of these functions, such as locking up your display until you type a password, or refreshing the content of all visible windows (very useful if some malfunction messes up what you see—in more primitive windowing systems, you would have to reboot your computer to do that!), or exiting from the X Window System, may be built-in. You can add more functions—including menu items for starting up additional applications—by editing a window manager startup file, such as *.mwmrc* for **mwm**, or *.openwin-menu* for **olwm**. The format of the files that specify the content of window manager menus is described in the window manager's manual page.

Learning about Your Window Manager

One of the first things you will want to do when starting to use the X Window System is to read the manual page—actually a technical document that may contain a dozen pages or more—that describes your system's window manager. You can do this by typing the **man** command into any UNIX shell window. For example, use the following to get information about the **mwm** window manager:

```
$ man mwm
```

This will tell you how the specific functions operate on your window manager. Remember that on some systems, the **man** command will automatically invoke a pager such as **more** so that you can read the man page one page at a time; on others, the content will scroll through to the end, and you will need to use the scrollbar to page through it, or explicitly invoke a pipe to your pager to read it, with a command such as this:

```
$man mwm | more
```

Client Applications

A large number of useful client applications have been written for the X Window System. They are the GUI analogy to the standard UNIX System tools. A few of these clients are **xterm**, **xclock**, **xbiff**, **xlock**, and **xcalc**.

The xterm Client

xterm is the standard X Window System terminal emulation client. It is probably the most frequently used X application, because it provides a window in which you can run a shell, run another UNIX command, or start up another X client. When you log in, you will probably be placed in an xterm window. To start a new xterm window from an existing one, simply type this:

```
$ xterm &
```

By default, your **xterm** window will include your shell (as specified in your *SHELL* variable).

If you are using **xterm**, you should set your *TERM* variable this way:

```
TERM=xterm
```

You can also create an **xterm** window for other commands. For example, you can run the **vi** text editor in an **xterm** window with this command:

```
xterm -e vi doc
```

Quitting the shell (with the **exit** command or CTRL-D) also kills the **xterm** window in which the shell was running. You can do the same thing by selecting **exit** in the **xterm** menu. In general, when you kill an **xterm** window in this way, all applications running in it are killed unless you used **nohup** to run them.

Some Other Useful X Clients

A large number of useful X clients are available. You may have many of these on your system already, and you can obtain many others from software archives on the Internet. Here are just a few X clients you may find useful:

- **xemacs** is an X Window implementation of the **emacs** editor, which is discussed in detail in Chapter 5.
- **xclock** is a simple graphical clock application. You can display it by including a line like this in your *.xinitrc* file:

  ```
  xclock -analog geometry 113x113-5+-4 &
  ```

- **xbiff** notifies you of new e-mail messages. It displays an icon of a mailbox. When a message arrives, it beeps and a flag on the mailbox is raised.
- **xlock** is a locking screen saver that keeps other people from viewing or using your terminal until you enter your password.
- **xcalc** displays a calculator (either a TI-30 or an HP-10C). To run it use this command:

  ```
  $ xcalc &
  ```

Starting and Ending an X Window Session

To start an X Window Session, you log in and run a startup script. Exactly how you do this varies slightly, depending on whether your system provides an X Window System server running permanently on your display, or whether you have to explicitly start it. In either case, the set of applications that will appear on your screen when you first start up the X Window System is controlled by a file containing an executable shell script, either *.xinitrc* or *.xsession.* If an X Window System server is not set up to be always running on your system, you will probably start your X Window session by running a script that invokes **xinit**. This is a program that starts up your X server and then executes the shell script in your *$HOME/.xinitrc.* When the last command in your *.xinitrc* exits, **xinit** kills the X server process and returns control to the shell from which it was invoked. On systems that have a permanent X Window server you use a login window provided by a program called **xdm** (X Display Manager) to log in. **xdm** then executes the shell script it finds in your *$HOME/.xsession.* When the last command in your *.xsession* exits, **xdm** terminates your session and replaces it with a new login window on the display.

The order of programs in your *.xinitrc* or *.xsession* is fixed, as follows:

1. First, you run the programs that customize your X Window environment: **xrdb**, **xset**, **xsetroot**, **xhost**, **xmodmap**, and so on. Because application programs inherit the resource variable values and keyboard maps that were in effect at their startup, the customization programs have to be run first, synchronously, in the foreground, so that the application programs won't be started until the X Window environment has been customized for them.
2. Next, all the automatically started application programs must be fired up, asynchronously (with an ampersand [&] to indicate asynchronous execution, that is, *not* waiting for each application to terminate before starting up the next, to the UNIX shell).
3. If you want to have an application always available but don't need to use it immediately, you can start it off pre-iconified, usually with the **–iconic** option on the application startup command line.
4. The last item of business in the *.xinitrc* or *.xsession* file is to start your window manager *synchronously* (without the "&"). This is essential so that when the window manager exits, the session script will also exit. Otherwise, you may wind up with an X Window session without having a window manager to control it or terminate it. If that happened, the only way to end the session and make your station available for subsequent work might be to reboot the hardware.

Selection Buffers

The X Window System was designed for powerful engineering workstations with lots of screen area, so that many different applications can appear on the screens and work simultaneously. It is even possible to use two or more different display terminals, with different application windows on separate screens. Unlike other windowing systems, which often limit you to a full-size display of only one application at a time, the X Window System enables the user to interact simultaneously with several active applications. A major function of the X Window System is to facilitate communication among its concurrently active clients. Most of this communication takes place automatically, without intervention by the user. However, one very important form of communication between applications is normally operated *by* the user. This is the *selection buffer,* which is used to transfer text between different windows, simultaneously on the screen.

Using the selection buffer is similar to the cut-and-paste operations in Microsoft Windows and other GUIs. You place text in the selection buffer by selecting it onscreen with the "select" button on the mouse, usually the one operated by your index finger. To select text, you move the mouse until the visible mouse cursor is located over the first character of the text you wish to select, depress the select button, and with the button depressed, move the mouse to the last character in your selection. The selection may span several lines. It includes the newline character at the end of a line, if the region highlighted to show the selection includes the area between the last character on the line and the edge of the text window. You terminate the selection by lifting the select button. On some systems, you may add text to an existing selection by sweeping it out with the opposite ("extend") mouse button depressed. Some X Window System applications provide other means of populating the selection buffer. For example, the font selection utility **xfontsel** has a screen button that, when "pressed" (selected from the mouse), deposits the name of the currently displayed font into the selection buffer.

Once selected, the content of the selection buffer may be entered into any text-based widget in any application, or into a text-based application such as a terminal emulator (for instance, **xterm**), just as though it had been typed from the keyboard. To do this, just shift the input focus to the object you want to drop text into—this is often done by moving the mouse until its cursor overlaps the text-accepting object—and press the "draw" or "deposit" mouse button, usually the middle button of a three-button mouse. Because different windows may belong to applications running on different machines, this is often a convenient way to transfer small pieces of information from one UNIX system to another. (Of course, some applications may limit the amount of text they will accept in this manner; others may differ in their interpretation of newline characters included in the selection buffer.) One of the most useful applications of selection buffers is to save the output of a program for future reference *after* the program has run. This is like rerunning the command with output to **tee** or a file in standard UNIX without X. It also gives you an easy way to edit text or output on the fly.

Customization: Becoming a Power User of the X Window System

If you look at the display on the screen of a wizard or expert user, you will probably be struck by the differences between it and your own screen. The screen background may show a different picture every day. The scrollbars of the wizard's XTerm windows may be unobtrusively narrow with a solid yellow scrolling indicator on a deep red background, instead of the wider, fuzzy gray that you seemed to be stuck with when you used XTerm. The fonts and the background and foreground colors in the windows may be different. The cursor may have a different shape from yours, maybe the shape of a little sailboat. The icons for XTerm windows to different systems may have different shapes instead of the uniform, easily confused appearance they have on your screen. And the "wizard" may seem rarely to have to type anything, complicated commands appearing on the screen at the touch of a single key, lines and paragraphs appearing highlighted under the mouse and then typing themselves into other windows. And you may notice that when the wizard is typing and needs to refer to something several screens back, the text scrolls without the wizard's ever having to move any fingers away from the keyboard to manipulate the mouse into the scrollbar.

This section is all about how to do all these things, and a lot more. It is about how you can make X Window System applications do things in ways that match the way you work best, and look the way you like things to look when you work with them for hours at a time. Unlike applications for other operating systems, which only can do the work they were written to do in exactly the way they were written to do it, X Window System applications were designed to be flexible for doing work that their authors could not anticipate, in ways limited only by the knowledge and intelligence of the user. Challenges that in other environments cannot be met without writing new programs, in X under UNIX often require no more than putting together existing pieces with some new resource variable values. By the end of this chapter, you will know what you need to be in full control of your X Window System environment and applications. You can be as impressive and productive as your "wizard" was.

Before you go on, though, you should know one more thing: Wizards learn what they know less from reading books than from experimentation, from trying things out. Trying things out is often discouraged in school courses, and even on the job in some fields. A programmer, for example, has the job of writing programs that will work on any processor for a standard language, not that just happen to work on the one specific platform they were experimentally tested on. But setting up your own work environment is different: You are not trying to customize everybody's work environment; just your own. So don't hesitate to try what you learn here, modify it, improvise. Experiment. Your knowledge of how things were meant to work is just a starting point.

Using X Window System Resources

With rare exceptions, the behavior and appearance of X Window System applications are controlled by a hierarchy of structured variables called *resources.* The values of resource variables are stored in a database in the X server process; this permits any client application that connects to your X Window System server to obtain their values, regardless of where on the network it happens to be running. The shell script for starting up your X session, usually *.xsession* or *.xinitrc,* includes, at the beginning, a command such as this:

```
xrdb -load .Xdefaults
```

This reads the content of a resource file, such as *.Xdefaults* in this example, into the resource database on your X server. When an application program connects itself to your X server, it reads these values and customizes itself accordingly. Most applications only read the resource database once and then maintain a private copy of its values. Thus, it is possible to start up one copy of an application, then change the content of the resource database, and start up another copy of the application with an identical command line, and have it behave differently because the values of some resource variables have changed.

The file from which the values in the resource database are read in has a very specific format. It consists of lines separated by newline characters. Each line assigns a value to an individual resource, or to a class of resources, pertaining to a specific object within some application or to a class of such objects. If a single assignment of a value to a resource or class of resources is too long to fit on one line in the file, intervening newlines may be escaped with a backslash (\e) at the end of a physical line, to merge two or more physical lines into a single "logical line." And if a literal newline character needs to be incorporated into the value assigned to a resource variable, it is written as "\en". The resource file may even include comments: Lines that start with an exclamation mark (!) in the first column are ignored by **xrdb** but remain in the file for humans to read.

A Simple Specification

The simplest possible resource value assignment line in a resource file such as *.Xdefaults* would assign a value to one specific resource of a specific X Window System object in a specific application. It might look like this:

```
xclock.clock.hands: red
```

This specification has three parts: *xclock.clock, hands,* and *red.* “Red” is the *value* being assigned to the *resource variable* “hands” of the *object* with the *object path* “xclock.clock”. Let’s look at each of these in turn.

The value “red” is a color, defined by a specific combination of intensities of red, green, and blue light (RGB values) from the corresponding pixels on the screen. To find out what pixel RGB values correspond to a named color, we could use this shell command:

```
$ showrgb | grep 'red$'
199  21 133    medium violet red
219 112 147    pale violet red
255  69   0    orange red
255   0   0    red
199  21 133    mediumvioletred
205  92  92    indianred
205  92  92    indian red
208  32 144    violetred
208  32 144    violet red
255  69   0    orangered
219 112 147    palevioletred
```

This tells you that “red” has the highest possible intensity of red light (255), and zero intensity of green and blue. Note that the “d” in “red” must be the very last character on the specification line. Although any blanks, tabs, and escaped newline characters between the colon (:) and the first visible character of the value are discarded when the resource file is being read in, any subsequent occurrences of these characters are included in the value being assigned. Trailing spaces are easy to miss, but if you were to leave one in the file, you’d be likely to get a diagnostic to the effect that the value “red ” (note the trailing space) can’t be converted to a color pixel value. If the complaining software leaves out the delimiting quotes, the trailing space is not visible, and the diagnostic may leave you questioning the sanity of your software. Note also that a color need not be specified by *name,* because a set of hexadecimal intensities is also acceptable. The latter is written as a sharp sign (#) followed by three sets of two hexadecimal digits specifying red, green, and blue intensity values. “Red,” for example, may also be written “#FF0000” (the hexadecimal for 255 0 0).

The *object path,* “xclock.clock,” is a sequence of period-separated object names beginning with the name of the running application object, and ending with the name of the specific object that the resource variable being specified pertains to. **xclock** is a trivially simple application with only one subordinate object (an instance of the “Clock” widget from the Athena toolkit), but most applications are much more complicated, with objects within objects within objects. Relevant objects (usually “widgets” or “gadgets”) within applications are usually documented in the manual page. For example, you can get information about **xclock** with this shell command:

```
$ man xclock
```

This manual page states that the instance name of the “Clock” widget is “clock.” This is an example of the convention of giving widget *instances* names that are lowercase transliterations of the widget *class* name. The default application instance name, used whenever a different name has not been specified on the command line that started the application and also documented in the manual page, is usually the same as the application's startup command—in this case, **xclock**. A different instance name may be specified for most X Window System applications with the **–name** command-line option. For example, to assign the name "GMT" to an **xclock** showing Greenwich Mean Time, the shell command might be this:

```
$ TZ=GMT xclock -name GMT &
```

The color of the hands for the GMT **xclock** would then be specified by a line in the resource file such as this:

```
GMT.clock.hands:blue
```

The initial resource setting would still apply to instances of "Xclock" with the default name "xclock." Using the application's *class name,* "Xclock," would apply the resource to all instances of the class, unless overridden by the higher precedence of a specification by instance name. For example, the following lines will give a cyan background to all Xclocks except for the one for Rome, which will be painted magenta:

```
Xclock.clock.background:cyan
Rome.clock.background:magenta
```

Another example is shown here:

```
xclock.clock.hands:  red
```

Here, "hands" is the instance name of the resource variable to be assigned the value "red." According to the **xclock** man page, "hands" is a resource that controls the color of the inside portion of the hands of the clock. It is one of three instance resource variables in the class *Foreground.* The other two are "highlight," the color of the edges of the hands, and "foreground," the color of the ticks. It is possible to specify the colors of all three separately, by using the names of the individual resources. It is also possible to specify a color for all three together, or with individually specified exceptions, by using the resource class *Foreground.* The following, for example, specifies blue ticks and hand edges, with red hand bodies:

```
xclock.clock.Foreground:blue
xclock.clock.hands:  red
```

You also have the option of creating a multilevel hierarchy of resource (or object) classes and subclasses. And the same class mechanism that applies to instances and classes of application objects and of resource variables also applies to other objects such as widgets and gadgets within applications. For example, all the pushbuttons in an application typically belong to the class "PushButton," horizontal and vertical scrollbars belong to the class "scrollbar," and so on.

The Asterisk Notation

The asterisk (*) notation lets you assign resource variable values to all members of any named class in an application, regardless of the details of the object hierarchy intervening between any of them and the corresponding application object. You can even use the asterisk notation to specify resource values for all objects (except for more specifically detailed exceptions) of a given class across applications, or a value for all resources that share a specific name, or that belong to the same named class regardless of the object to which they pertain. In a resource specification line, an asterisk may be substituted for any number, from zero on up, of dot-separated objects in the object hierarchy. For example, consider these specifications:

```
*fontList: lucidasans-typewriterbold
Mosaic*XmTextField*fontList: lucidasans-bold
```

This means that the value "lucidasans-typewriterbold" will be assigned to *all* resource variables named "fontList" in *all* objects in *all* applications, with the exception of those assigned more specifically. The second line is such an assignment: the value "lucidasans-bold" (a similar font, but with proportional rather than fixed character spacing) is assigned to the fontList resources that pertain to XmTextField widgets in applications of class "Mosaic."

Sometimes more than one line in a resource file will appear to apply to some resource variable. The X Window System toolkit library "Xt" (also called "Toolkit Intrinsics") tries to apply the potentially conflicting specifications from left to right, and it chooses the more specific one—the one for the instance name rather than the class name, or the one bound to its parent in the hierarchy by a period (.) instead of an asterisk (*). Most often, though, the best way to determine the effect of a resource specification is to experiment. Indeed, experimentation is generally the best way anyone has to settle any question about how things actually work in the X Window System, and what actually needs to be done.

In experimenting, or to override some specific resource value in specific cases, you can use the **–xrm** command-line option with most X Window System applications. Want to see how **xclock** would look with yellow hands? From your shell, try this:

```
$ xclock -xrm "*hands:yellow" &
```

If you want to be even more sophisticated, there is a whole hierarchy of files and other sources of resource values. They are, from the lowest and most easily overridden, to the highest and stickiest:

- A resource file supplied by the author of the application, bearing the name of the application class, in directory */usr/lib/X11/app-defaults* or another directory pointed to by the environment (shell exported) variable value *$XFILESEARCHPATH.*
- An application-specific resource file, bearing the application class name, in directory *$XUSERFILESEARCHPATH* or *$XAPPLRESDIR.*
- Resources that were loaded into the server with the **xrdb** command, or if none were loaded, those in file *$HOME/.Xdefaults* on the machine on which the application is being executed.
- Resources specified in the value of *$XENVIRONMENT.*
- Resources specified with **–xrm** command-line options.
- Resource values hard-coded into the application by user-hostile programmers.

If your workstation supports several screens, you may wish to set different resources for them, for example, color resources for a color screen and monochrome resources for a monochrome screen. X Window System releases 5 and higher support SCREEN_ RESOURCES properties for the different screens in addition to the RESOURCE_ MANAGER property that holds the resource database for all the screens together. If you get that far, the manual page of **xrdb** will tell you how to use this capability.

Color

As in the earlier **xclock** example, colors are one of the easiest features to customize through resources. Almost every object has at least two color resources, *background* and *foreground.* Widgets that appear inside other widgets often also have an optional border around them, specified with resources *borderWidth* (in pixels, 0 means no visible border) and *borderColor.*

If any additional color resources can be specified, they are listed in the object's (application's or widget's) manual page. Most application manual pages bear the lowercase name of the application. Widget manual pages bear the name of the widget, usually starting with an uppercase letter. The manual page will also list the classes—usually Background, Foreground, or BorderColor—to which each color resource belongs. Colors may be specified by name, using names from the color database as reported by **showrgb**, or with three two-digit hexadecimal numbers in #RRGGBB format. The set of colors available on a given screen is known, in X Window System terminology, as the screen's *visual.*

Common visuals include monochrome (black and white), grayscale (usually with 8 bits of resolution), true color (24-bit, with 8 bits for each of the three color intensities), and mapped color, also called pseudo-color. Pseudo-color is the most frequently encountered X Window System visual. Pseudo-color screens can display up to 256 different colors at one time—the number that can be represented with one byte value per color—out of 16 million (2 to the 24th power) specifiable true colors. The one-byte values in the computer's video memory are mapped to the actually displayed colors by a colormap with up to 256 color cells. Unless you are a graphic artist or plan to view color photographs on your computer screen, you probably won't have to deal with the details of color mapping. Some window managers (such as **olwm**) and platform customization tools (**xset**) have facilities for manipulation of colormaps. If they do, they are described in their manual pages.

Using color resources is usually fairly straightforward unless you use both color and monochrome screens. If you do, it may be worth your while to write, and load, different resource files for color and monochrome screens. There is no simple way to determine whether any specific color will be displayed on a monochrome screen as black or as white; the X Window System libraries actually use a computational model of the human visual system to decide whether a given combination of red, green, and blue light intensities should be represented by black or white for the human eye. The need to customize colors is frequently associated with applications that make default color assignments without considering the usability of the application on a monochrome screen.

Bitmaps and Pixmaps

Bitmaps and pixmaps are native image representation formats in the X Window System. Pixmaps may have any *depth* that corresponds to the number of bits per pixel in some X Window System visual. Bitmaps are monochrome (depth 1) pixmaps. Wherever a pixmap is expected, a bitmap may be used, but not vice versa. Under the UNIX System operating system, X Window System bitmaps and pixmaps are stored as ordinary files.

Besides their obvious application in the storage of images, bitmaps are used in the X Window System in a number of ways, including defining the background textures of object windows, for the mask and image shapes of the pointer cursor, for icons, and for glyphs (pictorial elements of composed graphics). A font is just a collection of numbered bitmaps and doesn't have to be used only for the characters of a written language. There are fonts that contain icons, cursors, glyphs, even images of game pieces for chess and other games. Fonts used to represent characters are discussed later on in this chapter.

Most widgets that present text or graphics have a *backgroundPixmap* resource and a *borderPixmap* resource. These resources were provided so that object windows on monochrome screens could have backgrounds that would contrast with both white and black characters and graphics, and borders that would be visible against both white and black backgrounds of other windows. The bitmap usually used for these purposes is */usr/include/X11/bitmaps/gray.* In theory, however, any other bitmap could be substituted. This means that many X Window System widgets may be shown with backgrounds of fancy patterns or pictures. This is occasionally useful but should not be casually applied to widgets whose use, such as readable display of text, would conflict with a distracting background.

Pixmap resources accept the path of any pixmap file as a legal value, not just the files in the standard X11 bitmaps directory. You can edit your own bitmaps with the **bitmap** client (for information, read the manual page: **man bitmap**). Different software vendors' editions of the UNIX System often include additional tools, such as Sun Microsystems' **iconedit**, which can create multicolor pixmaps.

You can use your own bitmaps to customize the icons of most applications (with resources *iconPixmap* and *iconMask*), or the window manager's (root window's) cursor, with this command:

```
$ xsetroot -cursor \e
/usr/include/X11/bitmaps/star \e
  /usr/include/X11/bitmaps/starMask
```

You can substitute any other pair of foreground and mask bitmaps.

Bitmaps are also customizable in applications that display pictorial elements, such as **xbiff**, or that use variable graphic elements. One use of this capability is in customizing the scrollbar of **xterm**:

```
XTerm*scrollbar.width:  7
XTerm*scrollbar.background:  red
XTerm*scrollbar.foreground:  yellow
XTerm*scrollbar.thumb: \e
  /usr/openwin/share/include/X11/bitmaps/black
```

By default, the scrollbar "thumb" is given a gray bitmap that contrasts with both available colors on a monochrome screen. These settings give you red-with-yellow contrast instead; and because solid colors look much better than patterns on a color screen, this example includes a black (all bits on) bitmap.

Fonts

Fonts are used in the X Window System for many collections of bitmaps. For example, most applications choose their cursors from the standard cursor font. If you like the sailboat cursor in XTerm, use this:

```
XTerm*pointerShape:sailboat
XTerm*pointerColor:blue
XTerm*pointerColorBackground:yellow
```

You can examine all the glyphs in any given font with **xfd**, the X font display utility. For example, you can see all the available fonts with this command:

```
$ xfd -fn cursor&
```

You can read their names, prefixed with "XC_", in */usr/include/X11/cursorfont.h.*

As you would expect, the main use of fonts is to display text. On a typical workstation, the X Window System may include hundreds of fonts to display ordinary characters. Many more are available over the Internet in various FTP archives, including the *contrib* archive of contributed X Window System software at *ftp.x.org*. You can bring any font over to your workstation and make it available to your X Window System server by including its directory in *$FONTPATH.*

A complete list of the fonts available in the directories in your *$FONTPATH* can be obtained from the **xlsfonts** command.

Some fonts have brief aliases that also appear in the output of **xlsfonts**, but the typical full name of a font is a list of attributes that usually looks something like this:

```
-b&h-lucida sans typewriter-medium-r-normal-sans-18-180-72-72-m-110-iso8859-1
```

Every dash-separated element identifies some specific attribute of the font. When setting the various font and fontList resources, you can use either an alias, or the full name, or one with asterisks standing in for those attributes you don't care about.

The following are some font resources that you may find useful:

```
*BoldFont:   -b&h-*-bold-r-*-*-14-*-*-*-m-*-*-*
*ButtonFont: -b&h-*-bold-r-*-*-14-*-*-*-p-*-*-*
*Font.Name:  -b&h-*-bold-r-*-*-12-*-*-*-m-*-*-*
*Font:    -b&h-*-bold-r-*-*-14-*-*-*-m-*-*-*
*IconFont:   avantgarde-demi
*ItalicFont: -*-*-*-o-*-*-14-*-*-*-m-*-*-*
*TextFont:   -b&h-*-bold-r-*-*-14-*-*-*-m-*-*-*
*TitleFont:  -b&h-*-bold-r-*-*-12-*-*-*-p-*-*-*
Mosaic*AddressFont: -adobe-times-medium-i-normal-*-14-*-*-*-*-*-iso8859-1
Mosaic*BoldFont: -adobe-times-bold-i-normal-*-14-*-*-*-*-*-iso8859-1
Mosaic*FixedFont:    -b&h-*-bold-r-*-*-14-*-*-*-m-*-*-*
Mosaic*Font: -adobe-times-medium-r-normal-*-14-*-*-*-*-*-iso8859-1
Mosaic*Header3Font: -adobe-times-bold-i-normal-*-14-*-*-*-*-*-iso8859-1
Mosaic*ItalicFont: -adobe-times-medium-i-normal-*-14-*-*-*-*-*-iso8859-1
Mosaic*XmText*fontList: lucidasans-typewriterbold
Mosaic*XmTextField*fontList: lucidasans-bold
Mosaic*fixedboldFont:-b&h-*-bold-r-*-*-14-*-*-*-m-*-*-*
```

```
Mosaic*fixeditalicFont:-adobe-courier-bold-o-*-*-14-*-*-*-m-*-*-*
Mosaic*font:\-adobe-new century schoolbook-bold-r-normal—14-140-75-75-p-87-
iso8859-1
Mosaic*fontList:\ -b&h-lucida sans-bold-r-normal-sans-10-100-72-72-p-67-
iso8859-1
Mosaic*listingFont:-b&h-*-bold-r-*-*-14-*-*-*-m-*-*-*
Mosaic*plainFont:-b&h-*-bold-r-*-*-14-*-*-*-m-*-*-*
Mosaic*plainboldFont:-b&h-*-bold-r-*-*-14-*-*-*-m-*-*-*
Mosaic*plainitalicFont:-adobe-courier-bold-o-*-*-14-*-*-*-m-*-*-*
```

Note that the type-of-spacing attribute is always "p" for proportionally spaced fonts but may be either "c" or "m" for constant-width fonts.

There is also a tool, called **xfontsel**, that you can use to interactively select the fonts that you prefer for a particular application. For each attribute in the standard font name format you get a menu from which you can select either some specific value of the relevant attribute or an asterisk (*) for "any." **xfontsel** can display the string of your choice in the selected font (read the manual page for details), and it has a button for placing the current selection string in the test selection buffer, so you can "drop" it directly into a resource or other file you might be editing.

The very wide variety of available fonts can be used not only to enliven the appearance of your screen, but also to display different kinds of text differently, so that they can be quickly distinguished from each other.

The Keyboard and Mouse

The user communicates with X Window System applications and with other clients, such as the window manager, through a pointer such as a mouse or joystick and a keyboard. Mouse movement has two customizable parameters: acceleration and threshold. Mouse acceleration and threshold are X server parameters. Like all server parameters, they can be customized with the **xset** utility. The mouse, or whatever pointer your machine is connected to, will go *acceleration* times as fast when it travels more than *thresholdpixels* in a short time. This allows you to set the mouse so that it can be used for precise alignment when it is moved slowly, and still travel across the screen in a flick of the wrist when you move it quickly. **xset** is typically used at the beginning of the *.xinitrc* shell script. The following setting accelerates the cursor movement four times whenever the mouse is moved rapidly more than two pixels:

```
$ xset m 4 2
```

All user input, other than mouse movement, consists of depressing and releasing pointer buttons and keyboard keys. The server sends an event message to the application each time the user presses or releases a mouse button or a keyboard key. The client-side X Window System library, through which the client application communicates with the server, keeps a mapping table, or *keyboard map,* to translate the hardware-generated *key codes* identifying the keys and buttons to meaningful *key symbols* that the application can use. The keyboard map is loaded when the application starts, so separate instances of the same application can be started at different times with different keyboard maps.

The symbol received by the application may depend not only on which keyboard key was pressed or released, but also on the up-or-down state of up to eight other keys, called modifiers: *shift, control, caps lock, number lock, alt,* and *meta.* You can get the modifier list for your X server with this command:

```
$ xmodmap -pm
```

On a Sun workstation, this will typically produce something like the following:

```
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):
shift    Shift_L (0x6a),  Shift_R (0x75)
lock  Caps_Lock (0x7e)
control  Control_L (0x53)
mod1  Meta_L (0x7f),  Meta_R (0x81)
mod2  Mode_switch (0x14)
mod3  Num_Lock (0x69)
mod4  Alt_L (0x1a)
mod5  F13 (0x20),  F18 (0x50),  F20 (0x68)
```

On a GraphOn terminal, the same command will produce something like this:

```
xmodmap:  up to 3 keys per modifier, (keycodes in parentheses):

shift    Shift_R (0xad),  Shift_L (0xae)
lock  Caps_Lock (0xb0)
control  Control_L (0xaf)
mod1  Alt_L (0x5c),  Alt_R (0xac)
mod2  Num_Lock (0xa5)
mod3  F13 (0x73),  F18 (0x81)
mod4
mod5
```

Note that keys other than SHIFT, CAPSLOCK, and CTRL may have different *modifier designations* on different servers. For example, *numeric lock* is *mod3* on the Sun but *mod2* on the GraphOn. The remaining, noncontrol keys may each carry up to four key symbols in the keyboard map. The symbol returned to the application will be determined by the X library from the map and the state of the modifier keys. You can get the current keyboard map with this command:

```
$ xmodmap -pk
```

The most important use of **xmodmap** is to customize the keyboard by changing the keyboard map. Typically this is done to move keys to where you are accustomed to finding them, or to obtain key symbols that are not in the default keyboard map. Suppose, for example, you have an old and obsolete AT&T 730X X terminal. For efficient editing with the UNIX **vi** editor the ESC key should be close to the letter area of the keyboard. The default keyboard of that terminal has it uncomfortably far up, while the key with the grave and ASCII tilde symbols is where you would want the ESC key. You can swap these key assignments with these commands:

```
$ xmodmap -e 'keycode 106 = grave asciitilde'
$ xmodmap -e 'keycode 93 = Escape'
```

Of course, if you want this key swap every time you use the terminal, you should put those commands in your *.xinitrc.*

In the preceding example, and in most applications of **xmodmap** listed in its manual page, you need to know the keycodes of the keys whose mappings you are going to modify before you use **xmodmap**. An easy way to do this is to use **xev**.

When you start up **xev**, it will pop up a small window, with a second smaller window inside it. Move this window out of the way, so it does not cover the shell window from which you started **xev**. (The output of **xev** will go to the shell window, and it is important that you be able to read it.) Next, move the mouse cursor into the **xev** window. Once the flurry of activity from moving the window and the mouse subsides, you can press any key you want and read the resulting event message. If you click the key, you will see two event descriptions: *key press* and *key release.* Each description will include the key code you can then use with **xmodmap**. To terminate **xev**, use the menu provided in its window-manager frame, or type your interrupt character (usually CTRL-C or DELETE) into the shell window from which you started **xev**.

You can add keys for input options omitted by the manufacturer of your equipment. For example, you can scroll through many Motif applications page by page if you have PRIOR and NEXT keys. The Sun type 4 keyboard has keys labeled PGUP and PGDN, but the default keyboard map does not assign the corresponding symbols to these keys. You can do it yourself in your *.xinitrc*:

```
xmodmap -e 'keycode 77 = Prior F29 KP_9 Prior' \
    -e 'keycode 121 = Next F35 KP_3 Next
```

Or maybe you miss a right-hand control key on that keyboard, but you don't need its "Compose" key. The following will fix the situation:

```
xmodmap -e "keycode 20 = Control_R" -e "add control = Control_R"
```

Note that assigning the key symbol "Control_R" was not enough; the key symbol also had to be added to the "control" modifier list. If you wanted to change the symbols assigned to a key already on a modifier list, you would have to remove it first. The **xmodmap** manual page provides the example of swapping the left CTRL and CAPSLOCK keys. The two keys must first be removed from their respective modifier lists, then swapped, and then put back on.

Finally, since the goal of the X Window System is to create an easy-to-use graphical interface, it provides an X tool called **xkeycaps** that enables you to change your keyboard mapping by using a GUI representation of your keyboard to do it.

Translations

Many X Window System applications have a *Translation table,* or *"Translations"* resource. The translation table enables the user to direct the performance of nearly any action that the application can perform with nearly any output. Although most applications have well-thought-out default translation tables that really don't need any user customization, and others don't have much in the way of interesting actions to assign events to, nearly everyone will wish to customize some aspect of their shell/terminal emulation windows. The following examples of *Translations* resource customization illustrate this for **xterm**.

(The same methods are, of course, reusable with any application whose manual page documents a translation table.)

The most common use of translation tables for **xterm** is to define strings sent by certain keys. For example, suppose you leave your office open but lock your workstation. The following example shows how you can translate a readily located key on your keyboard (L9 in the example) to send the command to lock your X Window server with **xlock**, allowing you to leave your office quickly:

```
XTerm*VT100.Translations:    #override\
<Key>L9:string("xlock -remote -mode random")string(0x0d)
```

With this translation, hitting the key sends the specified command to the shell running under the **xterm**. Sending the terminating RETURN is specified as a separate action, because the hexadecimal code needed to specify a control character such as CR and quoted strings of ordinary characters cannot be mixed in the same argument list to the action *string*( ).

This *Translations* resource applies to the VT100 emulation object in XTerm. The qualification "*#override*" means that, with the exception of the events specified in the file, all other event-to-action translations in the preexisting (default) translation table will remain in force. Without this qualification, specifying a *Translations* resource for this one key would wipe out nearly all the normal functionality of **xterm**.

Binding whole commands to single keys is the most common application of **xterm**'s *Translations* resource, but it is just the starting point. For one thing, the keyboard is by no means the only way to generate events that can trigger actions specified in a translation table. For example, changing an **xterm**'s width or height results in an event called "ConfigureNotify" or "Configure" for short. The command to reset the values of *$LINES* and *$COLUMNS* so that size-dependent programs like **vi** will work correctly is shown here:

```
$ eval 'resize'
```

This can be automatically triggered by the "Configure" event:

```
XTerm*VT100.Translations:    #override\e
<Configure>:string("eval 'resize'")string(0x0d)\en\e
<Key>L9:string("xlock -remote -mode random")string(0x0d)
```

Thus, as long as you only resize an XTerm window when a shell is running, you can be sure that size-dependent programs will run correctly after every resize.

Why are all translations but the last terminated with "\n"? The reason is that the whole translation table is the value of a single resource variable, and so the specification of this value must be continued, at the end of every line but the last, with a "\" at the end of the line. But the value of that resource variable is itself a *table,* consisting of separate lines, each of which must be separated by its own newline character from the next. This newline character is written as "\n" just before the "\" that continues the specification of the translations to the next line.

Any action can be triggered by any event, so there is no enforced distinction between what can be done by the pointer versus the keyboard. With the default translations, for example, scrolling can only be done with the mouse. If you would like to be able to refer to earlier text while typing, without having to take your hand off the keyboard to operate the mouse, you can add translations to operate the scrollbar with the PGUP and PGDN keys. The previous section on keyboard mapping with **xmodmap** mentioned assigning the symbols Prior and Next to those keys. The translations are shown here:

```
<Key>Prior:scroll-back(1,halfpage)\n\
<Key>Next:scroll-forw(1,halfpage)\n\
```

Because some continuity of context is helpful, this only scrolls half a page at a time. By using windows with an odd number of lines, you can keep a line of continuous context when you scroll through two half-pages by hitting the PGUP or PGDN key twice.

Just as the keyboard may be used for things that are usually done with the mouse, so can mouse buttons be used for keyboard functions such as sending characters or strings to applications and the shell. To send a carriage return (the default keyboard input) without taking your hand off the mouse, you can assign CR to mouse button 3:

```
~Shift ~Ctrl ~Meta <Btn3Down>:string(0x0d)\n\
~Shift ~Ctrl ~Meta <Btn3Motion>:ignore()\n\
```

These translations are qualified by making them applicable only when neither SHIFT, nor CTRL, nor META is pressed down. This prevents you from losing the ability to use XTerm's very useful "button 3 menu." With these qualifications, you can still get the default translation of *button 3 down* (presenting the menu) by pressing button 3 while holding down one of the three tilde modifiers. Because the default translation for moving the mouse with button 3 down is to extend the current text selection (something you don't want to happen by accident if you happen to move the mouse while clicking button 3 to go to the next mail or news item), that translation is set to "ignore( )".

How to Find Out More

A number of excellent books are available on the X Window System, ranging from the elementary to the highly sophisticated. Here are a few recommendations:

Asente, Paul, Donna Converse, and Ralph Swick. *X Window System Toolkit: The Complete Programmer's Guide and Specification,* X Version 11, Release 6 and 6.1. Woburn, MA: Digital Press, 1998.

Mansfield, Niall. *The Joy of X.* Reading, MA: Addison-Wesley, 1994.

Quercia, Valerie, and Tim O'Reilly. *X Window System User's Guide.* Sebastopol, CA: O'Reilly, 1993.

Smith, Jerry D. *X—A Guide for Users.* Englewood Cliffs, NJ: Prentice Hall, 1994.

The Asente book provides detailed information about two of the newer releases of X. The Mansfield book is a good overview of the system and includes a very handy format for describing key features at a middle level of detail. Smith's *Guide for Users* provides a more detailed treatment and contains much description of specific screens and applications. For information on more advanced X Window System topics such as protocols and programming, consult the books in O'Reilly's *X Window System* series.

To understand the evolution from Motif to CDE in developing X Window applications, try the following book:

Mione, Antonino. *CDE and Motif: A Practical Primer.* Englewood Cliffs, NJ: Prentice Hall, 1997.

You will also want to read some of the periodicals devoted to the X Window System, including *The X Resource: A Practical Journal of the X Window System* and *X Journal.*

Several newsgroups are devoted to the X Window System, including *comp.windows.x,* which provides a general discussion of the X Window System; *comp.x.announce* for announcements for the X Consortium; *comp.windows.x.apps* for a discussion on obtaining and using applications that run on X; *comp.windows.x.i386unix* for a discussion of X Window Systems for Intel-based UNIX PCs; and *comp.windows.x.motif* for a discussion on the Motif graphical user interface. You will also want to read the FAQs that are posted periodically to *comp.windows.x* and to *news.answers.*

Several useful web sites are devoted to the X Window System. In particular, to learn more about how to configure and use the system, you should consult the official X site, the X Consortium Web Server at *http://www.x.org/.* To find other sources of information about X, look at *http://www.x.org/ consortium/x_info.html.*

Also take a look at *http://www.rahul.net/kenton/xsites.html,* which has links to well over 500 sites that pertain to the X Window System.