48

Graphical User Interface Programming

Brad A. Myers
Carnegie Mellon University

48.1 Introduction*

Almost as long as there have been user interfaces, there have been special software systems and tools to help design and implement the user interface software. Many of these tools have demonstrated significant productivity gains for programmers and have become important commercial products. Others have proved less successful at supporting the kinds of user interfaces people want to build. Virtually all applications today are built using some form of **user interface tool** [Myers 2000].

User interface (UI) software is often large, complex, and difficult to implement, debug, and modify. As interfaces become easier to use, they become harder to create [Myers 1994]. Today, direct-manipulation interfaces (also called GUIs for **graphical user interfaces**) are almost universal. These interfaces require that the programmer deal with elaborate graphics, multiple ways of giving the same command, multiple asynchronous input devices (usually a keyboard and a pointing device such as a mouse), a mode-free interface where the user can give any command at virtually any time, and rapid "semantic feedback" where determining the appropriate response to user actions requires specialized information about the objects in the program. Interfaces on handheld devices, such as a Palm organizer or a Microsoft PocketPC device, use similar metaphors and implementation strategies. Tomorrow's user interfaces will provide speech

*This chapter is revised from an earlier version: Brad A. Myers. 1995. "User Interface Software Tools," *ACM Transactions on Computer–Human Interaction*. 2(1): 64–103.

1-58488-360-X/$0.00+$1.50

recognition, vision from cameras, 3-D, intelligent agents, and integrated multimedia, and will probably be even more difficult to create. Furthermore, because user interface design is so difficult, the only reliable way to get good interfaces is to iteratively redesign (and therefore reimplement) the interfaces after user testing, which makes the implementation task even harder.

Fortunately, there has been significant progress in software tools to help with creating user interfaces. Today, virtually all user interface software is created using tools that make the implementation easier. For example, the MacApp system from Apple, one of the first GUI frameworks, was reported to reduce development time by a factor of four or five [Wilson 1990]. A study commissioned by NeXT claimed that the average application programmed using the NeXTStep environment wrote 83% fewer lines of code and took one-half the time, compared to applications written using less advanced tools, and some applications were completed in one-tenth the time. Over three million programmers use Microsoft's Visual Basic tool because it allows them to create GUIs for Windows significantly more quickly.

This chapter surveys UI software tools and explains the different types and classifications. However, it is now impossible to discuss all UI tools, because there are so many, and new research tools are reported every year at conferences such as the annual ACM User Interface Software and Technology Symposium (UIST) (see http://www.acm.org/uist/) and the ACM SIGCHI conference (see http://www.acm.org/sigchi/). There are also about three Ph.D. theses on UI tools every year. This article provides an overview of the most popular approaches, rather than an exhaustive survey. It has been updated from previous versions (e.g., [Myers 1995]).

48.2 Importance of User Interface Tools

There are many advantages to using user interface software tools. These can be classified into two main groups. First, the *quality* of the resulting user interfaces might be higher, for the following reasons:

Designs can be rapidly prototyped and implemented, possibly even before the application code is written. This, in turn, enables more rapid prototyping and therefore more iterations of iterative design, which is a crucial component of achieving high-quality user interfaces[Nielsen 1993b].

The reliability of the user interface will be higher, because the code for the user interface is created automatically from a higher-level specification.

Different applications are more likely to have consistent user interfaces if they are created using the same UI tool.

It will be easier for a variety of specialists to be involved in designing the user interface, rather than having the user interface created entirely by programmers. Graphic artists, cognitive psychologists, and usability specialists may all be involved. In particular, professional user interface designers, who may not be programmers, can be in charge of the overall design.

More effort can be expended on the tool than may be practical on any single user interface, because the tool will be used with many different applications.

Undo, Help, and other features are more likely to be available because the tools might support them.

Second, the UI code might be *easier and more economical* to create and maintain. This is because of the following:

Interface specifications can be represented, validated, and evaluated more easily.

There will be less code to write, because much is supplied by the tools.

There will be better modularization, due to the separation of the UI component from the application. This should allow the user interface to change without affecting the application, and a large class of changes to the application (such as changing the internal algorithms) should be possible without affecting the user interface.

The level of programming expertise of the interface designers and implementers can be lower, because the tools hide much of the complexity of the underlying system.

It will be easier to port an application to different hardware and software environments because the device dependencies are isolated in the UI tool.

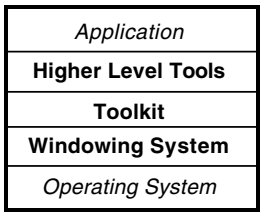

FIGURE 48.1 The components of user interface software.

48.2.1 Overview of User Interface Software Tools

Because user interface software is so difficult to create, it is not surprising that people have been working for a long time to create tools to help with it. Today, many of these tools and ideas have progressed from research into commercial systems, and their effectiveness has been amply demonstrated. Research systems also continue to evolve quickly, and the models that were popular five years ago have been made obsolete by more effective tools, changes in the computer market, and the emergence of new styles of user interfaces, such as handheld computing and multimedia.

48.2.1.1 Components of User Interface Software

As shown in Figure 48.1, UI software may be divided into various layers: the **windowing system**, the **toolkit**, and higher-level tools. Of course, many practical systems span multiple layers.

The windowing system supports the separation of the screen into different (usually rectangular) regions, called **windows**. The X system [Scheifler 1986] divides window functionality into two layers: the window system, which is the functional or programming interface, and the **window manager**, which is the user interface. Thus, the window system provides procedures that allow the application to draw pictures on the screen and get input from the user; the window manager allows the end user to move windows around and is responsible for displaying the title lines, borders, and **icons** around the windows. However, many people and systems use the name "window manager" to refer to both layers, because systems such as the Macintosh and Microsoft Windows do not separate them. This article will use the X terminology, and use the term *windowing system* to refer to both layers.

Note that Microsoft confusingly calls its entire system *Windows* (for example, *Windows 98* or *Windows XP*). This includes many different functions that here are differentiated into the operating system part (which supports memory management, file access, networking, etc.), the windowing system, and higher-level tools.

On top of the windowing system is the toolkit, which contains many commonly used **widgets** (also called *controls*) such as menus, buttons, scroll bars, and text input fields. On top of the toolkit might be higher-level tools, which help the designer to use the toolkit widgets. The following sections discuss each of these components in more detail.

48.2.1.2 Windowing Systems

A windowing system is a software package that helps the user monitor and control different contexts by separating them physically onto different parts of one or more display screens [Myers 1988b]. Although most of today's systems provide toolkits on top of the windowing systems, as will be explained later, toolkits generally only address the drawing of widgets such as buttons, menus, and scroll bars. Thus, when the programmer wants to draw application-specific parts of the interface and allow the user to manipulate these, the window system interface must be used directly. Therefore, the windowing system's programming interface has significant impact on most user interface programmers.

The first windowing systems were implemented as part of a single program or system. For example, the EMACs text editor [Stallman 1979], and the Smalltalk [Tesler 1981], and DLISP [Teitelman 1979] programming environments had their own windowing systems. Later systems implemented the windowing

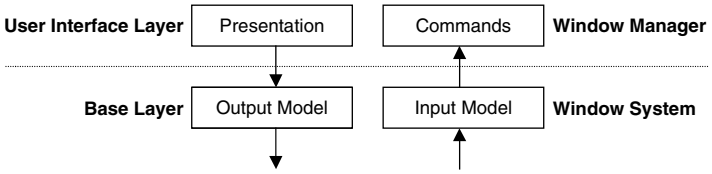

FIGURE 48.2 The windowing system can be divided into two layers, called the base (or window system) layer and the user interface (or window manager) layer. Each of these can be divided into parts that handle output and input.

system as an integral part of the operating system, such as Sapphire for PERQs [Myers 1984], SunView for Suns, and the Macintosh and Microsoft Windows systems. In order to allow different windowing systems to operate on the same operating system, some windowing systems, such as X and Sun's NeWS [Gosling 1986], operate as a separate process and use the operating system's interprocess communication mechanism to connect to application programs.

48.2.1.2.1 Structure of Windowing Systems

A windowing system can be logically divided into two layers, each of which has two parts (see Figure 48.2). The window system, or base layer, implements the basic functionality of the windowing system. The two parts of this layer handle the display of graphics in windows (the output model) and the access to the various input devices (the input model), which usually includes a keyboard and a pointing device such as a mouse. The primary interface of the base layer is procedural and is called the windowing system's application programmer interface (API).

The other layer of windowing system is the window manager or user interface. This includes all aspects that are visible to the user. The two parts of the user interface layer are the presentation, which comprises the pictures that the window manager displays, and the commands, which are how the user manipulates the windows and their contents.

48.2.1.2.2 Base Layer

The base layer is the procedural interface to the windowing system. In the 1970s and early 1980s, there were a large number of different windowing systems, each with a different procedural interface (at least one for each hardware platform). People writing software found this to be unacceptable because they wanted to be able to run their software on different platforms, but they would have to rewrite significant amounts of code to convert from one window system to another. The X windowing system [Scheifler 1986] was created to solve this problem by providing a hardware-independent interface to windowing. X has been quite successful at this, and it drove all other windowing systems out of the workstation hardware market. X continues to be popular as the windowing system for Linux and all other UNIX implementations. In the rest of the computer market, most machines use some version of Microsoft Windows, with the Apple Macintosh computers having their own windowing system.

48.2.1.2.3 Output Model

The output model is the set of procedures that an application can use to draw pictures on the screen. It is important that all output be directed through the window system so that the graphics primitives can be clipped to the window's borders. For example, if a program draws a line that would extend beyond a window's borders, it must be clipped so that the contents of other, independent, windows are not overwritten. Most computers provide graphics hardware that is optimized to work efficiently with the window system.

In early windowing systems, such as Smalltalk [Tesler 1981] and Sapphire [Myers 1986], the primary output operation was BitBlt (also called RasterOp, and now sometimes CopyArea or CopyRectangle). These early systems primarily supported monochrome screens (each pixel is either black or white). BitBlt takes

a rectangle of pixels from one part of the screen and copies it to another part. Various Boolean operations can be specified for combining the pixel values of the source and destination rectangles. For example, the source rectangle can simply replace the destination, or it might be XORed with the destination. BitBlt can be used to draw solid rectangles in either black or white, display text, scroll windows, and perform many other effects [Tesler 1981]. The only additional drawing operation typically supported by these early systems was drawing straight lines.

Later windowing systems, such as the Macintosh and X, added a full set of drawing operations, such as filled and unfilled polygons, text, lines, arcs, etc. These cannot be implemented using the BitBlt operator. With the growing popularity of color screens and nonrectangular primitives (such as rounded rectangles), the use of BitBlt has significantly decreased. Now, it is primarily used for scrolling and copying off-screen pictures onto the screen (e.g., to implement double-buffering).

A few windowing systems allowed the full PostScript imaging model [Adobe Systems Inc. 1985] to be used to create images on the screen. PostScript provides device-independent coordinate systems and arbitrary rotations and scaling for all objects, including text. Another advantage of using PostScript for the screen is that the same language can be used to print the windows on paper (because many printers accept PostScript). Sun created a version used in the NeWS windowing system, and then Adobe (the creator of PostScript) came out with an official version called Display PostScript, which was used in the NeXT windowing system. A similar imaging model is provided by Java 2D [Sun Microsystems 2002], which works on top of (and hides) the underlying windowing system's output model.

All of the standard output models only contain drawing operations for two-dimensional objects. Extensions to support 3-D objects include PEX, OpenGL, and Direct3-D. PEX [Gaskins 1992] is an extension to the X windowing system that incorporates much of the PHIGS graphics standard. OpenGL [Silicon Graphics Inc. 1993] is based on the GL programming interface that has been used for many years on Silicon Graphics machines. OpenGL provides some machine independence for 3-D because it is available for various X and Windows platforms. Microsoft supplies its own 3-D graphics model, called Direct3-D, as part of Windows.

As shown in Figure 48.3, the earlier windowing systems assumed that a graphics package would be implemented using the windowing system. See Figure 48.3a. For example, the CORE graphics package was implemented on top of the SunView windowing system. Next, systems such as the Macintosh, X, NeWS, NeXT, and Microsoft Windows implemented a sophisticated graphics system as part of the windowing system. See Figure 48.3b and Figure 48.3c. Now, with Java2D and Java3-D, as well as Web-based graphics systems such as VRML for 3-D programming on the Web [Web3-D Consortium 1997], we are seeing a return to a model similar to the one shown in Figure 48.3a, with the graphics on top of the windowing system. See Figure 48.3-D.

48.2.1.2.4 Input Model

The early graphics standards, such as CORE and PHIGS, provided an input model that does not support the modern, direct-manipulation style of interfaces. In those standards, the programmer calls a routine to request the value of a virtual device, such as a locator (pointing device position), string (edited text string), choice (selection from a menu), or pick (selection of a graphical object). The program would then pause, waiting for the user to take action. This is clearly at odds with the direct-manipulation mode-free style, in which the user can decide whether to make a menu choice, select an object, or type something.

With the advent of modern windowing systems, a new model was provided: a stream of event records is sent to the window that is currently accepting input. The user can select which window is getting events using various commands, described subsequently. Each event record typically contains the type and value of the event (e.g., which key was pressed), the window to which the event was directed, a timestamp, and the x and y coordinates of the mouse. The windowing system queues keyboard events, mouse button events, and mouse movement events together (along with other special events), and programs must dequeue the events and process them. It is somewhat surprising that, although there has been substantial progress in the output model for windowing systems (from BitBlt to complex 2-D primitives to 3-D), input is still

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.