51. Graphical User Interface Programming

Brad A. Myers*

Human Computer Interaction Institute

Carnegie Mellon University

5000 Forbes Avenue

Pittsburgh, PA 15213

bam@cs.cmu.edu

http://www.cs.cmu.edu/~bam

(412) 268-5150

FAX: (412) 268-1266

*This paper is revised from an earlier version that appeared as: Brad A. Myers. "User Interface Software Tools," *ACM Transactions on Computer-Human Interaction*. vol. 2, no. 1, March, 1995. pp. 64-103.

Draft of: January 27, 2003

To appear in: *CRC HANDBOOK OF COMPUTER SCIENCE AND ENGINEERING – 2nd Edition*, 2003. Allen B. Tucker, Editor-in-chief

51.1. 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 proven 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 for 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 recognition, vision from cameras, 3-D, intelligent agents and integrated multi-media, 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 re-design (and therefore re-implement) 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, and 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 3 million programmers use Microsoft's Visual Basic tool because it allows them to create GUIs for Windows significantly more quickly.

This chapter surveys user interface software tools, and explains the different types and classifications. However, it is now impossible to discuss all user interface tools, since 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 PhD theses on user interface tools every year. Therefore, 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]).

51.2. 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. This is because of the following:

- 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 that is a crucial component of achieving high-quality user interfaces [Nielsen 1993b].

- The reliability of the user interface will be higher, since 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 user interface 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 since the tool will be used with many different applications.

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

Second, the user interface 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 user interface 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 complexities of the underlying system.

- It will be easier to port an application to different hardware and software environments since the device dependencies are isolated in the user interface tool.

51.3. Overview of User Interface Software Tools

Since 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 multi-media.

Components of User Interface Software

As shown in Figure 1, user interface 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 the 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, and 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, since 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” when referring 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 use the toolkit widgets. The following sections discuss each of these components in more detail.

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 below, 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 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 inter-process communication mechanism to connect to application programs.

Structure of Windowing Systems

A windowing system can be logically divided into two layers, each of which has two parts (see Figure 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 is comprised of the pictures that the window manager displays, and the *commands*, which are how the user manipulates the windows and their contents.

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 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.

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 out of 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 non-rectangular primitives (such as rounded rectangles), the use of BitBlt has significantly decreased. It is primarily used now 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 (since 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 Direct3D. 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 since it is available for various X and Windows platforms. Microsoft supplies its own 3-D graphics model called Direct3D as part of Windows.

As shown in Figure 3, the earlier windowing systems assumed that a graphics package would be implemented using the windowing system (Figure 3-a). 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 (Figure 3-b and c). Now, with Java2D and Java3D, as well as Web-based graphics systems such as VRML for 3-D programming on the Web [Web3D Consortium 1997], we are seeing a return to a model similar to the Figure 3-a, with the graphics on top of the windowing system (see Figure 3-d).

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, where 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 handled in essentially this same way today as in the original windowing systems, even though there are some well-known unsolved problems with this model:

- There is no provision for special stop-output (control-S) or abort (control-C, command-dot) events, so these will be queued with the other input events.

- The same event mechanism is used to pass special messages from the windowing system to the application. When a window gets larger or becomes uncovered, the application must usually be notified so it can adjust or redraw the picture in the window. Most window systems communicate this by enqueuing special events into the event stream, which the program must then handle.

- The application must always be willing to accept events in order to process aborts and redrawing requests. If not, then long operations cannot be aborted, and the screen may have blank areas while they are being processed.

- The model is device dependent, since the event record has fixed fields for the expected incoming events. If a 3-D pointing device or one with more than the standard number of buttons is used instead of a mouse, then the standard event mechanism cannot handle it.

- Because the events are handled asynchronously, there are many race conditions that can cause programs to get out of synchronization with the window system. For example, in the X windowing system, if you press inside a window and release outside, under certain conditions the program will think that the mouse button is still depressed. Another example is that refresh requests from the windowing system specify a rectangle of the window that needs to be redrawn, but if the program is changing the contents of the window, the wrong area may be redrawn by the time the event is processed. This problem can occur when the window is scrolled.

Although these problems have been known for a long time, there has been little research on new input models (an exception is the Garnet Interactors model [Myers 1990b]).

Communication

In the X windowing system and NeWS, all communication between applications and the window system uses inter-process communication through a network protocol. This means that the application program can be on a different computer from its windows. In all other windowing systems, operations are implemented by directly calling the window manager procedures or through special traps into the operating system. The primary advantage of the X mechanism is that it makes it easier for a person to utilize multiple machines with all their windows appearing on a single machine. Another advantage is that it is easier to provide interfaces for different programming languages: for example the C interface (called xlib) and the Lisp interface (called CLX) send the appropriate messages through the network protocol. The primary disadvantage is efficiency, since each window request will typically be encoded, passed to the transport layer, and then decoded, even when the computation and windows are on the same machine.

User Interface Layer

The user interface of the windowing system allows the user to control the windows. In X, the user can easily switch user interfaces, by killing one window manager and starting another. Some of the original window managers under X included uwm (with no title lines and borders), twm, mwm (the Motif window manager), and olwm (the OpenLook window manager). Newer choices include complete desktop environments that combine a window manager with a file browser and other GUI utilities (to better match the capabilities found in Windows and the Macintosh). Two popular desktop environments are KDE ("K" Desktop Environment" — http://www.kde.org/) with its window manager KWin, and Gnome (http://www.gnome.org/), which provides a variety of window manager choices. X provides a standard protocol through which programs and the base layer communicate to the window manager, so that all programs continue to run without change when the window manager is switched. It is possible, for example, to run applications that use Motif widgets inside the windows controlled by the KWin window manager.

A discussion of the options for the user interfaces of window managers was previously published [Myers 1988b]. Also, the video *All the Widgets* [Myers 1990a] has a 30-minute segment showing many different forms of window manager user interfaces.

Some parts of the user interface of a windowing system, which is sometimes called its *look and feel*, can apparently be copyrighted and patented. Which parts is a highly complex issue, and the status changes with decisions in various court cases [Samuelson 1993].

Presentation

The presentation of the windows defines how the screen looks. One very important aspect of the presentation of windows is whether they can overlap or not. *Overlapping windows*, sometimes called *covered windows*, allow one window to be partially or totally on top of another window, as shown in

Figure 4. This is also sometimes called the *desktop metaphor*, since windows can cover each other like pieces of paper can cover each other on a desk. There are usually other aspects to the desktop metaphor, however, such as presenting file operations in a way that mimics office operations, as originated in the Star office workstation [Smith 1982]. The other alternative is called *tiled windows*, which means that windows are not allowed to cover each other. Obviously, a window manager that supports covered windows can also allow them to be side-by-side, but not vice-versa. Therefore, a window manager is classified as "covered" if it allows windows to overlap. The tiled style was popular for a while, and was used by Cedar [Swinehart 1986], and early versions of the Star [Smith 1982], Andrew [Palay 1988], and even Microsoft Windows. A study even suggested that using tiled windows was more efficient for users [Bly 1986]. However, today tiled windows are rarely seen on conventional window systems, because users generally prefer overlapping.

Modern "browsers" for the World-Wide Web, such as Netscape and Microsoft's Internet Explorer, provide a windowing environment inside the computer's main windowing system. Newer versions of browsers support frames containing multiple scrollable panes, which are a form of tiled window. In addition, if an application written in Java is downloaded (see the section on Virtual Toolkits below), it can create multiple, overlapping windows like conventional GUI applications.

Another important aspect of the presentation of windows is the use of *icons*. These are small pictures that represent windows (or sometimes files). They are used because there would otherwise be too many windows to conveniently fit on the screen and manage. Sapphire was the first window manager to group the icons into a window [Myers 1984], which was picked up by the Motif window manager. Now, the "taskbar" provides the icons and names of running and available processes in Windows and other modern window managers. Other aspects of the presentation include whether the window has a title line or not, what the background (where there are no windows) looks like, and whether the title and borders have control areas for performing window operations.

Commands

Since computers typically have multiple windows and only one mouse and keyboard, there must be a way for the user to control which window is getting keyboard input. This window is called the *input* (or *keyboard*) *focus*. Another term is the *listener* since it is listening to the user's typing. Some systems called the focus the *active window* or *current window*, but these are poor terms since in a multi-processing system, many windows can be actively outputting information at the same time. Window managers provide various ways to specify and show which window is the listener. The most important options are:

- *Click-to-type*, which means that the user must click the mouse button in a window before typing to it. This is used by the Macintosh and Microsoft Windows.

- *Move-to-type*, which means that the mouse only has to move over a window to allow typing to it. This is usually faster for the user, but may cause input to go to the wrong window if the user accidentally knocks the mouse.

Some X window managers (including the Motif window manager, mwm) allow the user to choose which method is desired. However, the choice can have significant impact on the user interface of applications. For example, because the Macintosh requires click-to-type, it can provide a single menubar at the top, and the commands can always operate on the focused window. With move-to-type, the user might have to pass through various windows (thus giving them the focus) on the way to the top of the screen. Therefore, Motif applications must have a menubar in each window so the commands will know which window to operate on.

All covered window systems allow the user to bring a window to the top (not covered by other windows), and some allow sending a window to the bottom (covered by all other windows). Other commands allow windows to be changed size, moved, shrunk to an icon, made full size, and destroyed.

Toolkits

A toolkit is a library of *widgets* that can be called by application programs. A *widget* (also called a *control*) is a way of using a physical input device to input a certain type of value. Typically, widgets in toolkits include menus, buttons, scroll bars, text type-in fields, etc. Figure 5 shows some examples of widgets. Creating an interface using a toolkit can only be done by programmers, because toolkits only have a procedural interface.

Using a toolkit has the advantage that the final UI will look and act similarly to other UIs created using the same toolkit, and each application does not have to rewrite the standard functions, such as menus. A problem with toolkits is that the styles of interaction are limited to those provided. For example, it is difficult to create a single slider that contains two indicators, which might be useful to input the upper and lower bounds of a range. In addition, the toolkits themselves are often expensive to create: “The primitives never seem complex in principle, but the programs that implement them are surprisingly intricate” [Cardelli 1985, p.199]. Another problem with toolkits is that they are often difficult to use since they may contain hundreds of procedures, and it is often not clear how to use the procedures to create a desired interface.

As with the graphics package, the toolkit can be implemented either using or being used by the windowing system (see Figure 3). Early systems provided only minimal widgets (e.g., just a menu), and expected applications to provide others (Figure 3-a). In the Macintosh and Microsoft Windows, the toolkit is at a low level, and the window manager user interface is built using it. The advantage of this is that the window manager can then use the same sophisticated toolkit routines for its user interface (Figure 3-b). When the X system was being developed, the developers could not agree on a single toolkit, so they left the toolkit to be on top of the windowing system. In X, programmers can use a variety of toolkits (for example, the Motif, InterViews [Linton 1989], Amulet [Myers 1997], tcl/tk [Ousterhout 1991] and Gnome GTK+ [GNOME 2002] toolkits can be used on top of X), but the window manager must usually

implement its user interface without using the toolkit (Figure 3-c). The Java “Swing” toolkit is implemented on top of the Java 2D graphics package, which in turn is on top of the windowing system (Figure 3-d).

Because the designers of X could not agree on a single look-and-feel, they created an intrinsics layer on which to build different widget sets, which they called Xt [McCormack 1988]. This layer provides the common services, such as techniques for object-oriented programming and layout control. The widget set layer is the collection of widgets that is implemented using the intrinsics. Multiple widget sets with different looks and feels can be implemented on top of the same intrinsics layer (Figure 6-a), or else the same look-and-feel can be implemented on top of different intrinsics (Figure 6-b).

Toolkit Intrinsics

Toolkits come in two basic varieties. The most conventional is simply a collection of procedures that can be called by application programs. Examples of this style include the SunTools toolkit for the SunView windowing system, and the original Macintosh Toolbox [Apple Computer Inc. 1985]. The second variety uses an object-oriented programming style that makes it easier for the designer to customize the interaction techniques. Examples include Smalltalk [Tesler 1981], Andrew [Palay 1988], InterViews [Linton 1989], Xt [McCormack 1988], Amulet [Myers 1997], the Java Swing toolkit [Sun Microsystems 2003], and Gnome’s GTK+ [GNOME 2002].

The advantages of using object-oriented intrinsics are that it is a natural way to think about widgets (the menus and buttons on the screen *seem* like objects), the widget objects can handle some of the chores that otherwise would be left to the programmer (such as refresh), and it is easier to create custom widgets (by subclassing an existing widget). The advantage of the older, procedural style is that it is easier to implement, no special object-oriented system is needed, and it is easier to interface to multiple programming languages.

To implement the objects, the toolkit might invent its own object system, as was done with Xt, Andrew, Amulet and GTK+, or it might use an existing object system, as was done in InterViews [Linton 1989] which uses C++, NeXTStep from NeXT which uses Objective-C, and Swing which uses Java.

The usual way that object-oriented toolkits interface with application programs is through the use of *call-back procedures*. These are procedures defined by the application programmer that are called when a widget is operated by the end user. For example, the programmer might supply a procedure to be called when the user selects a menu item. Experience has shown that real interfaces often contain hundreds of call-backs, which makes the code harder to modify and maintain [Myers 1992c]. In addition, different toolkits, even when implemented on the same intrinsics like Motif and OpenLook, have different call-back protocols. This means that code for one toolkit is difficult to port to a different toolkit. Therefore, research has been directed at reducing the number of call-backs in user interface software [Myers 1991b].

Some research toolkits have added novel features to the toolkit intrinsics. For example, Garnet [Myers 1990d], Rendezvous [Hill 1994], Amulet [Myers 1997], and SubArctic [Hudson 1996] allow the objects to be connected using constraints, which are relationships that are declared once and then maintained automatically by the system. For example, the designer can specify that the color of a rectangle is constrained to be the value of a slider, and then the system will automatically update the color if the user moves the slider.

Many toolkits include a related capability for handling graphical layouts in a declarative manner. This is often called *geometry management.* Widgets can be specified to stay at the sides or center of a container, or to expand to fill up a specified space. This is particularly important when the size of objects might change, for example in systems that can run on multiple architectures. An early example of this was in InterViews [Linton 1989] and layout managers are important parts of Motif and Java Swing.

Widget Set

Typically, the intrinsics layer is look-and-feel independent, which means that the widgets built on top of it can have any desired appearance and behavior. However, a particular widget set must pick a look-and-feel. The video *All the Widgets* shows many examples of widgets that have been designed over the years [Myers 1990a]. For example, it shows 35 different kinds of menus. Like window manager user interfaces, the widgets' look-and-feel can be copyrighted and patented [Samuelson 1993].

As was mentioned above, different widget sets (with different looks and feels) can be implemented on top of the same intrinsics. In addition, the same look-and-feel can be implemented on top of different intrinsics. For example, there are Motif look-and-feel widgets on top of the Xt, InterViews and Amulet intrinsics (Figure 5-b). Although they all look and operate the same (so would be indistinguishable to the end user), they are implemented quite differently, and have completely different procedural interfaces for the programmer.

Specialized Toolkits

A number of toolkits have been developed to support specific kinds of applications or specific classes of programmers. For example, the SUIT system [Pausch 1992] (which contains a toolkit and an interface builder) is specifically designed to be easy to learn and is aimed at classroom instruction. Its successor, Alice [Pausch 1995], provides an easy way to program 3-D graphics and animation. Amulet [Myers 1997] provides high-level support for graphical, direct manipulation interfaces, and handles input as hierarchical command objects, making Undo easier to implement [Myers 1996a]. Rendezvous [Hill 1994], Visual Obliq [Bharat 1994], and GroupKit [Roseman 1996] are designed to make it easier to create applications that support multiple users on multiple machines operating synchronously. Whereas most toolkits provide only 2-D interaction techniques, the Brown 3-D toolkit [Stevens 1994] and the Silicon Graphics' Inventor toolkit [Wernecke 1994] provide preprogrammed 3-D widgets and a framework for creating others.

Special support for animations has been added to Artkit [Hudson 1993], Amulet [Myers 1996b] and Alice [Pausch 1995]. Tk [Ousterhout 1991] is a popular toolkit for the X window system (and also Windows) because it can be attached to a variety of interpretive languages, including the original called tcl, as well as Perl and Python, which makes it possible to dynamically change the user interface. Tcl supports the Unix style of programming where many small programs are glued together.

Virtual Toolkits

Although there are many small differences among the various toolkits, much remains the same. For example, all have some type of menu, button, scroll bar, text input field, etc. Although there are fewer windowing systems and toolkits than there were ten years ago, people are still finding it to be a lot of work to port software so it works on the Macintosh, Microsoft Windows and X on Linux.

Therefore, a number of systems, called *Virtual Toolkits*, have been developed that try to hide the differences among the various toolkits, by providing virtual widgets that can be mapped into the widgets of each toolkit. Another name for these tools is *cross-platform development systems*. The programmer writes the code once using the virtual toolkit and the code will run without change on different platforms and still look like it was designed with that platform's widgets. For example, the virtual toolkit might provide a single menu routine, which always has the same programmer interface, but connects to a Motif menu, Macintosh menu, or a Windows menu depending on which machine the application is run on.

There are two styles of virtual toolkits. In the first, the virtual toolkit links to the different actual toolkits on the host machine. For example, XVT [XVT Software Inc 1997] provides a C or C++ interface that links to the actual Motif, Macintosh, MS-Windows, and OS/2-PM toolkits (and also character terminals) and hides their differences. The Java AWT toolkit also works this way and uses the underlying widget sets. The second style of virtual toolkit reimplements the widgets in each style. For example, Galaxy [Visix Software Inc. 1997], Amulet [Myers 1997], and Java Swing provide libraries of widgets that look like those on the various platforms. The advantage of the first style is that the user interface is more likely

to be look-and-feel conformant (since it uses the real widgets). The disadvantages are that the virtual toolkit must still provide an interface to the graphical drawing primitives on the platforms. Furthermore, they tend to only provide functions that appear in all toolkits. Many of the virtual toolkits that take the second approach, for example Java Swing, provide a sophisticated graphics package and complete sets of widgets on all platforms. However, with the second approach, there must always be a large run-time library, since in addition to the built-in widgets that are native to the machine, there is the reimplementation of these same widgets in the virtual toolkit's library.

All of the toolkits that work on multiple platforms can be considered virtual toolkits of the second type. For example, SUIT [Pausch 1992] and Garnet [Myers 1990d] work on X, Macintosh and Windows. However, these use the same look-and-feel on all platforms (and therefore do not look the same as the other applications on that platform), so they are not classified as virtual toolkits.

Higher Level Tools

Since programming at the toolkit level is quite difficult, there is a tremendous interest in higher level tools that will make the user interface software production process easier. These are discussed next.

Phases

Many higher-level tools have components that operate at different times. The *design time component* helps the user interface designer design the user interface. For example, this might be a graphical editor that can lay out the interface, or a compiler to process a user interface specification language. The next phase is when the end-user is using the program. Here, the *run-time component* of the tool is used. This usually includes a toolkit, but may also include additional software specifically for the tool. Since the run-time component is "managing" the user interface, the term *User Interface Management System* (UIMS) seems appropriate for tools with a significant run-time component.

There may also be an *after-run-time component* that helps with the evaluation and debugging of the user interface. Unfortunately, very few user interface tools have an after-run-time component. This is partially because tools that have tried, such as MIKE [Olsen Jr. 1988], discovered that there are very few metrics that can be applied by computers. Some tools try to evaluate how people will interact with interfaces by automatically creating cognitive models from high-level descriptions of the user interface. For example, the GLEAN system generates quantitative predictions of performance of a system from a GOMS model [Kieras 1995].

Specification Styles

High-level user interface tools come in a large variety of forms. One important way that they can be classified is by how the designer specifies what the interface should be. Some tools require the programmer to program in a special-purpose language, some provide an application framework to guide the programming, some automatically generate the interface from a high-level model or specification, and others allow the interface to be designed interactively. Each of these types is discussed below. Of course, some tools use different techniques for specifying different parts of the user interface. These are classified by their predominant or most interesting feature.

Language Based

With most of the older user interface tools, the designer specifies the user interface in a special-purpose language. This language can take many forms, including context-free grammars, state transition diagrams, declarative languages, event languages, etc. The language is usually used to specify the syntax of the user interface; i.e., the legal sequences of input and output actions. This is sometimes called the *dialogue*. Green [Green 1986] provides an extensive comparison of grammars, state transition diagrams, and event languages, and Olsen [Olsen Jr. 1992] surveys various older UIMS techniques.

State Transition Networks

Since many parts of user interfaces involve handling a sequence of input events, it is natural to think of using a state transition network to code the interface. A transition network consists of a set of states, with arcs out of each state labeled with the input tokens that will cause a transition to the state at the other end of the arc. In addition to input tokens, calls to application procedures and the output to display can also be put on the arcs in some systems. Newman implemented a simple tool using finite state machines in 1968 [Newman 1968] that handled textual input. This was apparently the first user interface tool. Many of the assumptions and techniques used in modern systems were present in Newman's tool: different languages for defining the user interface and the semantics (the semantic routines were coded in a normal programming language), a table-driven syntax analyzer, and device independence.

State diagram tools are most useful for creating user interfaces where the user interface has a large number of modes (each state is really a mode). For example, state diagrams are useful for describing the operation of low-level widgets (e.g., how a menu or scroll bar works), or the overall global flow of an application (e.g., this command will pop-up a dialogue box, from which you can get to these two dialogue boxes, and then to this other window, etc.). However, most highly-interactive systems attempt to be mostly "mode-free," which means that at each point, the user has a wide variety of choices of what to do. This requires a large number of arcs out of each state, so state diagram tools have not been successful for these interfaces. In addition, state diagrams cannot handle interfaces where the user can operate on multiple objects at the same time. Another problem is that they can be very confusing for large interfaces, since they get to be a "maze of wires" and off-page (or off-screen) arcs can be hard to follow.

Recognizing these problems, but still trying to retain the perspicuousness of state transition diagrams, Jacob [Jacob 1986] invented a new formalism, which is a combination of state diagrams with a form of event languages. There can be multiple diagrams active at the same time, and flow of control transfers from one to another in a co-routine fashion. The system can create various forms of direct manipulation

interfaces. VAPS is a commercial system that uses the state transition model, and it eliminates the maze-of-wires problem by providing a spreadsheet-like table in which the states, events, and actions are specified [eNGENUITY 2002]. Transition networks have been thoroughly researched, but have not proven particularly successful or useful either as a research or commercial approach.

Context-Free Grammars

Many grammar-based systems are based on parser generators used in compiler development. For example, the designer might specify the user interface syntax using some form of BNF. Examples of grammar-based systems are Syngraph [Olsen Jr. 1983] and parsers built with YACC and LEX in Unix.

Grammar-based tools, like state diagram tools, are not appropriate for specifying highly-interactive interfaces, since they are oriented towards batch processing of strings with a complex syntactic structure. These systems are best for textual command languages, and have been mostly abandoned for user interfaces by researchers and commercial developers.

Event Languages

With event languages, the input tokens are considered to be *events* that are sent to individual event handlers. Each handler will have a condition clause that determines what types of events it will handle, and when it is active. The body of the handler can cause output events, change the internal state of the system (which might enable other event handlers), or call application routines.

Sassafras [Hill 1986] is an event language where the user interface is programmed as a set of small event handlers. The Elements-Events and Transitions (EET) language provides elaborate control over when the various event handlers are fired [Frank 1995]. In these earlier systems, the event handlers were global. With more modern systems, the event handlers are specific to particular objects. For example, the HyperTalk language that is part of HyperCard for the Apple Macintosh can be considered an event

language. Microsoft's Visual Basic also contains event-language features, since code is written to handle the response to events on objects.

The advantages of event languages are that they can handle multiple input devices active at the same time, and it is straightforward to support non-modal interfaces, where the user can operate on any widget or object. The main disadvantage is that it can be very difficult to create correct code, especially as the system gets larger, since the flow of control is not localized and small changes in one part can affect many different pieces of the program. It is also typically difficult for the designer to understand the code once it reaches a nontrivial size. However, the success of HyperTalk, Visual Basic and similar tools shows that this approach is appropriate for small- to medium-size programs. The style of programming used by Java Swing and related systems, where the programmer overrides methods that are called when events happen, is similar to the event style.

Declarative Languages

Another approach is to try to define a language that is declarative (stating what should happen) rather than procedural (how to make it happen). Cousin [Hayes 1985] and HP/Apollo's Open-Dialogue [Schulert 1985] both allow the designer to specify user interfaces in this manner. The user interfaces supported are basically forms, where fields can be text that is typed by the user, or options selected using menus or buttons. There are also graphic output areas that the application can use in whatever manner desired. The application program is connected to the user interface through *variables*, which can be set and accessed by both. As researchers have extended this idea to support more sophisticated interactions, the specification has grown into full application "models," and newer systems are described in the section on "Model-Based Automatic Generation."

The layout description languages that come with many toolkits are also a type of declarative language. For example, Motif's User Interface Language (UIL) allows the layout of widgets to be defined. Since the UIL is interpreted when an application starts, users can (in theory) edit the UIL code to customize the

interface. UIL is not a complete language, however, in the sense that the designer must still write C code for many parts of the interface, including any areas containing dynamic graphics and any widgets that change.

The advantage of using declarative languages is that the user interface designer does not have to worry about the time sequence of events, and can concentrate on the information that needs to be passed back and forth. The disadvantage is that only certain types of interfaces can be provided this way, and the rest must be programmed by hand in the "graphic areas" provided to application programs. The kinds of interactions available are preprogrammed and fixed. In particular, these systems provide no support for such things as dragging graphical objects, rubber-band lines, drawing new graphical objects, or even dynamically changing the items in a menu based on the application mode or context. However, these languages have been used as intermediate languages describing the layout of widgets (such as UIL) that are generated by interactive tools.

Constraint Languages

A number of user interface tools allow the programmer to use *constraints* to define the user interface [Borning 1986b]. Early constraint systems include Sketchpad [Sutherland 1963], which pioneered the use of graphical constraints in a drawing editor, and Thinglab [Borning 1981], which used constraints for graphical simulation. Subsequently, Thinglab was extended to aid in the generation of user interfaces [Borning 1986b].

The discussion of toolkits above mentioned the use of constraints as part of the intrinsics of a toolkit. A number of research toolkits now supply constraints as an integral part of the object system (e.g., Garnet [Myers 1990d], Amulet [Myers 1997], and SubArctic [Hudson 1996]). In addition, some systems have provided higher-level interfaces to constraints. Graphical Thinglab [Borning 1986a] allows the designer to create constraints by wiring icons together, and NoPump [Wilde 1990] and C32 [Myers 1991a] allow constraints to be defined using a spreadsheet-like interface.

The advantage of constraints is that they are a natural way to express many kinds of relationships that arise frequently in user interfaces. For example, that lines should stay attached to boxes, that labels should stay centered within boxes, etc. A disadvantage with constraints is that they require a sophisticated run-time system to solve them efficiently, and it can be difficult for programmers to specify and debug constraint systems correctly. As yet, there are no commercial user interface tools using general-purpose constraint solvers.

Database Interfaces

A very important class of commercial tools supports form-based or GUI-based access to databases. Major database vendors such as Oracle [Oracle Tools 1995] provide tools that allow designers to define the user interface for accessing and setting data. Often these tools include interactive form editors (which are essentially interface builders) and special database languages. Fourth generation languages (4GLs), that support defining the interactive forms for accessing and entering data, also fall into this category.

Visual Programming

Visual programs use graphics and two (or more) dimensional layout as part of the program specification [Myers 1990c]. Many different approaches to using visual programming to specify user interfaces have been investigated. Most systems that support state transition networks use a visual representation. Another popular technique is to use dataflow languages. In these, icons represent processing steps, and the data flow along the connecting wires. The user interface is usually constructed directly by laying out pre-built widgets, in the style of interface builders. Examples of visual programming systems for creating user interfaces include Labview [National Instruments 2003], which is specialized for controlling laboratory instruments, and Prograph [Pictorius 2002]. Using a visual language seems to make it easier for novice programmers, but large programs still suffer from the familiar “maze of wires” problem. Other papers (e.g., [Myers 1990c]) have analyzed the strengths and weaknesses of visual programming in detail.

Another popular language is Visual Basic from Microsoft. However, this is more of a structure editor for Basic combined with an Interface Builder (see below), and therefore does not really count as a visual language.

Summary of Language Approaches

In summary, many different types of languages have been designed for specifying user interfaces. One problem with all of these is that they can only be used by professional programmers. Some programmers have objected to the requirement for learning a new language for programming just the user interface portion [Olsen Jr. 1987]. This has been confirmed by market research [X Business Group 1994, p.29]. Furthermore, it seems more natural to define the graphical part of a user interface using a graphical editor. However, it is clear that for the foreseeable future, much of the user interface will still need to be created by writing programs, so it is appropriate to continue investigations into the best language to use for this. Indeed, an entire book is devoted to investigating the languages for programming user interfaces [Myers 1992b].

Application Frameworks

After the Macintosh Toolbox had been available for a little while, Apple discovered that programmers had a difficult time figuring out how to call the various toolkit functions, and how to ensure that the resulting interface met the Apple guidelines. They therefore created a software system that provides an overall *application framework* to guide programmers. This was called MacApp [Wilson 1990] and used the object-oriented language Object Pascal. Classes are provided for the important parts of an application, such as the main windows, the commands, etc., and the programmer specializes these classes to provide the application-specific details, such as what is actually drawn in the windows and which commands are provided. MacApp was very successful at simplifying the writing of Macintosh applications. Today, there are multiple frameworks to help build applications for most major platforms, including the Microsoft Foundation Classes (MFC) for Windows, and the portable Java Swing framework. A definition of

framework would be a software architecture, often object-oriented, that guides the programmer so that implementing user interface software is easier.

The Amulet framework [Myers 1997] is aimed at graphical applications, but due to its graphical data model, many of the built-in routines can be used without change (the programmer does not usually need to write methods for subclasses). Newer frameworks aim to help implement applications that take advantage of *ubiquitous computing* [Weiser 1993] (also called *pervasive computing*), multiple users (also called *Computer Supported Cooperative Work—CSCW*), various sensors that tell the computer where it is and who is around (called *context-aware computing* [Moran 2001]), and user interfaces that span multiple computers (called *multi-computer user interfaces* [Myers 2001]). For example, the BEACH framework [Tandler 2001] provides facilities to handle all of these kinds of user interfaces.

The *component* approach aims to replace today's large, monolithic applications with smaller pieces that attach together. For example, you might buy a separate text editor, ruler, paragraph formatter, spell checker, and drawing program, and have them all work together seamlessly. This approach was invented by the Andrew environment [Palay 1988], which provides an object-oriented document model that supports the embedding of different kinds of data inside other documents. These "insets" are unlike data that is cut and pasted in systems like the Macintosh because they bring along the programs that edit them, and therefore can always be edited in place. Furthermore, the container document does not need to know how to display or print the inset data since the original program that created it is always available. The designer creating a new inset writes subclasses that adhere to a standard protocol so the system knows how to pass input events to the appropriate editor. Microsoft OLE [Petzold 1991], Apple's OpenDoc [Curbow 1995], and JavaBeans [JavaSoft 1996] use this approach. The Microsoft ".Net" initiative provides a component architecture for web services.

All of these frameworks require the designer to write code, typically by creating application-specific subclasses of the standard classes provided as part of the framework.

Model-Based Automatic Generation

A problem with all of the language-based tools is that the designer must specify a great deal about the placement, format, and design of the user interfaces. To solve this problem, some tools use automatic generation so that the tool makes many of these choices from a much higher-level specification. Many of these tools, such as Mickey [Olsen Jr. 1989], Jade [Vander Zanden 1990], and DON [Kim 1993] have concentrated on creating menus and dialogue boxes. Jade allows the designer to use a graphical editor to edit the generated interface if it is not good enough. DON has the most sophisticated layout mechanisms and takes into account the desired window size, balance, symmetry, grouping, etc. Creating dialogue boxes automatically has been very thoroughly researched, but there still are no commercial tools that do this.

UIDE (the User-Interface Design Environment) [Sukaviriya 1993] requires that the semantics of the application be defined in a special-purpose language, and therefore might be included with the language-based tools. It is placed here instead because the language is used to describe the functions that the application supports and not the desired interface. UIDE is classified as a model-based approach because the specification serves as a high-level, sophisticated model of the application semantics. In UIDE, the description includes pre- and post-conditions of the operations, and the system uses these to reason about the operations, to automatically generate an interface, and to automatically generate help [Sukaviriya 1990].

The ITS [Wiecha 1990] system also uses rules to generate an interface. ITS was used to create the visitor information system for the EXPO 1992 worlds fair in Seville, Spain. Unlike the other rule-based systems, the designer using ITS is expected to write many of the rules, rather than just writing a specification that the rules work on. In particular, the design philosophy of ITS is that all design decisions should be codified as rules so that they can be used by subsequent designers, which will hopefully mean that interface designs will become easier and better as more rules are entered. As a result, the designer should

never use graphical editing to improve the design, since then the system cannot capture the reason that the generated design was not sufficient.

Recently, there has been a resurgence of interest in model-based interfaces to try to provide interfaces that work on the many kinds of handheld devices. For example, the Wireless Access Protocol (WAP) contains high-level descriptions of the information to be displayed, which the handhelds must convert to use specific layouts and interaction techniques. Research continues on ways to convert high-level specifications of appliances and other devices into appropriate remote control interfaces for handhelds, for use in "smart rooms" [Ponnekanti 2001], by the handicapped [Zimmerman 2002], and for home appliances [Banavar 2000] [Nichols 2002].

Direct Graphical Specification

The tools described next all allow the user interface to be defined, at least partially, by placing objects on the screen using a pointing device. This is motivated by the observation that the visual presentation of the user interface is of primary importance in graphical user interfaces, and a graphical tool seems to be the most appropriate way to specify the graphical appearance. Another advantage of this technique is that it is usually much easier for the designer to use. Many of these systems can be used by non-programmers. Therefore, psychologists, graphic designers and user interface specialists can more easily be involved in the user interface design process when these tools are used.

These tools can be distinguished from those that use *visual programming* since with direct graphical specification, the actual user interface (or a part of it) is drawn, rather than being generated indirectly from a visual program. Thus, direct graphical specification tools have been called *direct manipulation programming* since the user is directly manipulating the user interface widgets and other elements.

The tools that support graphical specification can be classified into four categories: *prototyping tools*, those that support a sequence of cards, *interface builders*, and editors for application-specific graphics.

Prototyping Tools

The goal of prototyping tools is to allow the designer to quickly mock up some examples of what the screens in the program will look like. Sometimes, these tools cannot be used to create the real user interface of the program; they just show how some aspects will look. This is the chief factor that distinguishes them from other high-level tools. Many parts of the interface may not be operable, and some of the things that look like widgets may just be static pictures. In many prototypers, no real toolkit widgets are used, which means that the designer has to draw simulations that look like the widgets that will appear in the interface. The normal use is that the designer would spend a few days or weeks trying out different designs with the tool, and then completely reimplement the final design in a separate system. Most prototyping tools can be used without programming, so they can, for example, be used by graphic designers.

Note that this use of the term *prototyping* is different from the general phrase *rapid prototyping*, which has become a marketing buzzword. Advertisements for just about all user interface tools claim that they support "rapid prototyping," by which they mean that the tool helps create the user interface software more quickly. The term "prototyping" is being used in this paper in a much more specific manner.

Probably the first prototyping tool was Dan Bricklin's Demo program. This is a program for an IBM PC that allows the designer to create sample screens composed of characters and *character graphics* (where the fixed-size character cells can contain a graphic such as a horizontal, vertical or diagonal line). The designer can easily create the various screens for the application. It is also relatively easy to specify the actions (mouse or keyboard) that cause transitions from one screen to another. However, it is difficult to define other behaviors. In general, there may be some support for type-in fields and menus in prototyping tools, but there is little ability to process or test the results.

For graphical user interfaces, designers often use tools like Macromedia's Director [Macromedia 2003a], which is actually an animation tool. The designer can draw example screens, and then specify that when

the mouse is pressed in a particular place, an animation should start or a different screen should be displayed. Components of the picture can be reused in different screens, but again the ability to show behavior is limited. HyperCard and Visual Basic are also often used as prototyping tools. Research tools such as SILK [Landay 1995] and DENIM [Lin 2002] provide a quick sketching interface and then convert the sketches into actual interfaces.

The primary disadvantage of these prototyping tools is that sometimes the application must be re-coded in a "real" language before the application is delivered. There is also the risk that the programmers who implement the real user interface will ignore the prototype.

Cards

Many graphical programs are limited to user interfaces that can be presented as a sequence of mostly static pages, sometimes called *frames*, *cards*, or *forms*. Each page contains a set of widgets, some of which cause transfer to other pages. There is usually a fixed set of widgets to choose from, which have been coded by hand.

An early example of this is Menulay [Buxton 1983a], which allows the designer to place text, graphical potentiometers, iconic pictures, and light buttons on the screen and see exactly what the end user will see when the application is run. The designer does not need to be a programmer to use Menulay.

Probably, the most famous example of a card-based system is HyperCard from Apple. There are many similar programs, such as GUIDE [Owl International Inc. 1991], and ToolBook [Click2learn 1995]. In all of these, the designer can easily create cards containing text fields, buttons, etc., along with various graphic decorations. The buttons cause transfers to other cards. These programs provide a scripting language to provide more flexibility for buttons. HyperCard's scripting language is called HyperTalk, and as mentioned above, is really an event language, since the programmer writes short pieces of code that are executed when input events occur. In its original instantiation, the World-Wide-Web was represented as a

sequence of pages, which were like “cards,” with embedded links that transfer to other pages that replace the previous page.

Interface Builders

An *interface builder* allows the designer to create dialogue boxes, menus and windows that are to be part of a larger user interface. These are also called *Interface Development Tools* (IDTs) or *GUI Builders*. Interface builders allow the designer to select from a pre-defined library of widgets, and place them on the screen using a mouse. Other properties of the widgets can be set using property sheets. Usually, there is also some support for sequencing, such as bringing up sub-dialogues when a particular button is hit. The Steamer project at BBN demonstrated many of the ideas later incorporated into interface builders and was probably the first object-oriented graphics system [Stevens 1983]. Other examples of research interface builders are DialogEditor [Cardelli 1988], and Gilt [Myers 1991b]. There are hundreds of commercial interface builders, including the *resource editors* that come with professional development environments such as Metrowerks Codewarrior. Microsoft’s Visual Basic is essentially an interface builder coupled with an editor for an interpreted language. Many of the tools discussed above, such as the virtual toolkits, visual languages, and application frameworks, also contain interface builders.

Interface builders use the actual widgets from a toolkit, so they can be used to build parts of real applications. Most will generate C or C++ code templates that can be compiled along with the application code. Others generate a description of the interface in a language that can be read at run-time. It is sometimes important that the programmers not edit the output of the tools (such as the generated C code) or else the tool can no longer be used for later modifications.

Although interface builders make laying out the dialogue boxes and menus easier, this is only part of the user interface design problem. These tools provide little guidance toward creating good user interfaces, since they give designers significant freedom. Another problem is that for any kind of program that has a graphics area (such as drawing programs, CAD, visual language editors, etc.), interface builders do not

help with the contents of the graphics pane. Also, they cannot handle widgets that change dynamically. For example, if the contents of a menu or the layout of a dialogue box changes based on program state, this must be programmed by writing code.

Data Visualization Tools

An important commercial category of tools is that of dynamic *data visualization systems*. These tools, which tend to be quite expensive, emphasize the display of dynamically changing data on a computer, and are used as front ends for simulations, process control, system monitoring, network management, and data analysis. The interface to the designer is usually quite similar to an interface builder, with a palette of gauges, graphers, knobs and switches that can be placed interactively. However, these controls usually are not from a toolkit and are supplied by the tool. Example tools in this category include DataViews [DataViews 2001] and SL-GMS [SL Corp 2002].

Editors for Application-Specific Graphics

When an application has custom graphics, it would be useful if the designer could draw pictures of what the graphics should look like rather than having to write code for this. The problem is that the graphic objects usually need to change at run time, based on the actual data and the end user's actions. Therefore, the designer can only draw an *example* of the desired display, which will be modified at run-time, and so these tools are called *demonstrational programming* [Myers 1992a]. This distinguishes these programs from the graphical tools of the previous three sections, where the full picture can be specified at design time. As a result of the generalization task of converting the example objects into parameterized prototypes that can change at run-time, these systems are still in the research phase.

Peridot [Myers 1988a] allows new, custom widgets to be created. The primitives that the designer manipulates with the mouse are rectangles, circles, text, and lines. The system generalizes from the designer's actions to create parameterized, object-oriented procedures such as those that might be found

in toolkits. Experiments showed that Peridot could be used by non-programmers. Lapidary [Vander Zanden 1995] extends the ideas of Peridot to allow general application-specific objects to be drawn. For example, the designer can draw the nodes and arcs for a graph program. The DEMO system [Fisher 1992] allows some dynamic, run-time properties of the objects to be demonstrated, such as how objects are created. The Marquise tool [Myers 1993] allows the designer to demonstrate when various behaviors should happen, and supports palettes that control the behaviors. With Pavlov [Wolber 1997], the user can demonstrate how widgets should control a car's movement in a driving game. Gamut [McDaniel 1999] has the user give hints to help the system infer sophisticated behaviors for games-style applications. Research continues on making these ideas practical.

Specialized Tools

For some application domains, there are customized tools that provide significant high-level support. These tend to be quite expensive, however (i.e., US$20,000 to US$50,000). For example, in the aeronautics and real-time control areas, there are a number of high-level tools, such as InterMAPhics [Gallium 1991].

51.4. Tools for the World Wide Web

Implementing user interfaces for the World Wide Web generally uses quite different tools than building GUIs, and is covered in depth in other chapters of this volume. Furthermore, the technology and tools are changing quite rapidly. Therefore, this section just provides a brief overview.

Simple web pages may be composed of static text and graphics with embedded links, and these can be authored by directly writing the underlying *html* (*Hyper-Text Markup Language*). Alternatively, the author can use interactive tools, such as Microsoft FrontPage, which therefore serves as a kind of Interface Builder. FrontPage can also author pages that contain forms for filling in information (text fields, buttons, etc.). More dynamic pages can use a scripting language embedded in the html, such as

Javascript or VBscript (Visual Basic Script). Alternatively, a specialized animation language can be used, such as Macromedia's Flash language, which might be authored using an interactive tool such as Dreamweaver [Macromedia 2003b].

In all cases, the back-end that provides the pages, processes any input provided in form fields, and delivers new pages as a result, must be implemented using some kind of server-side scripting or database tool, which is typically quite different from the tools used to author the client-side pages that the user sees.

51.5. Models of User Interface Software

Since creating user interface software is so difficult, there have been a number of efforts to describe the software organization at a very abstract level, by creating *models* of the software. The earliest attempts used the same levels that had been defined for compilers, and talked about the *semantic*, *syntactic* and *lexical* parts of the user interface, but this proved to be mostly only useful for parser-based implementations [Buxton 1983b]. Another early model is the *Seeheim model* [Pfaff 1985], which separates the *presentation* aspects (output), from the *dialogue management* (what happens in what order, based on what input), from the *application interface model* (what the resulting changes in data are). This model does not work well with GUIs since they de-emphasize dialogue in favor of mode-free interaction. The *Model-View-Controller* concept [Krasner 1988] was first used by Smalltalk, and separates the output handling (View) from the input handling (Controller). Both of these are separated from the underlying data (the Model). Later systems, such as InterViews [Linton 1989], found it difficult to separate the View from the Controller, and therefore used a simpler Model-View organization, where the View includes the Controller. A new model for software organization that tries to handle the various aspects required for multiple users, distributed processing, and ubiquitous computing, is being developed as part of BEACH [Tandler 2002].

51.6. Technology Transfer

User interface tools are an area where research has had a tremendous impact on the current practice of software development [Myers 1998]. Of course, window managers and the resulting “GUI style” come from the seminal research at the Stanford Research Institute, Xerox Palo Alto Research Center (PARC), and MIT in the 1970s. Interface builders and “card” programs like HyperCard were invented in research laboratories at BBN, the University of Toronto, Xerox PARC, and others. Now, interface builders are widely used for commercial software development. Event languages, as widely used in HyperTalk and Visual Basic, were first investigated in research laboratories. The current generation of environments, such as OLE and Java Beans, are based on the component architecture that was developed in the Andrew environment from Carnegie Mellon University. Thus, whereas some early UIMS approaches such as transition networks and grammars may not have been successful, overall, the user interface tool research has changed the way that software is developed.

51.7. Research Issues

Although there are many user interface tools, there are plenty of areas in which further research is needed. Previous reports discuss future research ideas for user interface tools at length [Myers 2000] [Olsen Jr. 1993]. Here, a few of the important ones are summarized.

New Programming Languages

The built-in input/output primitives in today’s programming languages, such as printf/scanf or cout/cin, support a textual question-and-answer style of user interface that is modal and well-known to be poor. Most of today’s tools use libraries and interactive programs which are separate from programming languages. However, many of the techniques, such as object-oriented programming, multiple-processing, and constraints, are best provided as part of the programming language. Even new languages, such as

Java, make much of the user interface harder to program by leaving it in separate libraries. Furthermore, an integrated environment, where the graphical parts of an application can be specified graphically and the rest textually, would make the generation of applications much easier. How programming languages can be improved to better support user interface software is the topic of a book [Myers 1992b].

Increased Depth

Many researchers are trying to create tools that will cover more of the user interface, such as application-specific graphics and behaviors. The challenge here is to allow flexibility to application developers while still providing a high level of support. Tools should also be able to support Help, Undo, and Aborting of operations.

Today's user interface tools mostly help with the *generation* of the code of the interface, and assume that the fundamental user interface design is complete. What are also needed are tools to help with the generation, specification, and analysis of the *design* of the interface. For example, an important first step in user interface design is task analysis, where the designer identifies the particular tasks that the end user will need to perform. Research should be directed at creating tools to support these methods and techniques. These might eventually be integrated with the code generation tools, so that the information generated during early design can be fed into automatic generation tools, possibly to produce an interface directly from the early analyses. The information might also be used to automatically generate documentation and run-time help.

Another approach is to allow the designer to specify the design in an appropriate notation, and then provide tools to convert that notation into interfaces. For example, the UAN [Hartson 1990] is a notation for expressing the end user's actions and the system's responses.

Finally, much work is needed in ways for tools to help evaluate interface designs. Initial attempts, such as in MIKE [Olsen Jr. 1988], have highlighted the need for better models and metrics against which to

evaluate the user interfaces. Research in this area by cognitive psychologists and other user interface researchers (e.g., [Kieras 1995]) is continuing.

Increased Breadth

We can expect the user interfaces of tomorrow to be different from the conventional window-and-mouse interfaces of today, and tools will have to change to support the new styles. For example, already we are seeing tiny digital pagers and phones with embedded computers and displays, palm-size computers such as the PalmOS devices, notebook-size panel computers such as Microsoft's TabletPCs, as well as wall-size displays. Furthermore, computing is appearing in more and more devices around the home and office. An important next wave will appear when the devices can all easily communicate with each other, probably using wireless radio technologies like 802.11 ("Wi-Fi") or BlueTooth [Haartsen 1998]. Sound, video, and animations will increasingly be incorporated into user interfaces. New input devices and techniques will probably replace the conventional mouse and menu styles. For example, there will be substantially more use of techniques such as gestures, handwriting, and speech input and output. These are called *recognition-based* because they require software to interpret the input stream from the user to identify the content. In these "non-WIMP" [Nielsen 1993a] applications (WIMP stands for Windows, Icons, Menus and Pointing devices), designers will also need better control over the timing of the interface, to support animations and various new media such as video. Although a few tools are directed at multiple-user applications, there are no direct graphical specification tools, and the current tools are limited in the styles of applications they support. A further problem is supporting *multiple* interfaces for the same application, so it can run on small and large devices in a consistent manner, and sometimes a person might be using multiple devices *at the same time*, such as a PalmOS device *and* a big display screen [Myers 2001].

Another concern is supporting interfaces that can be moved from one natural language to another (like English to French). Internationalizing an interface is much more difficult than simply translating the text

strings, and includes different number, date, and time formats, new input methods, redesigned layouts, different color schemes, and new icons [Russo 1993]. How can future tools help with this process?

End User Programming and Customization

One of the most successful computer programs of all time is the spreadsheet. The primary reason for its success is that end users can program (by writing formulas and macros). However, *end user programming* is rare in other applications, and where it exists, usually requires learning conventional programming. For example, AutoCAD provides Lisp for customization, and many Microsoft applications use Visual Basic. More effective mechanisms for users to customize existing applications and create new ones are needed [Myers 1992b]. However, these should not be built into individual applications as is done today, since this means that the user must learn a different programming technique for each application. Instead, the facilities should be provided at the system level, and therefore should be part of the underlying toolkit. Naturally, since this is aimed at end users, it will not be like programming in C, but rather at some higher level.

Application and User Interface Separation

One of the fundamental goals of user interface tools is to allow better modularization and separation of user interface code from application code. However, a survey reported that conventional toolkits actually make this separation more difficult, due to the large number of call-back procedures required [Myers 1992c]. Therefore, further research is needed into ways to better modularize the code, and how tools can support this.

Tools for the Tools

It is very difficult to create the kinds of tools described in this chapter. Each one takes an enormous effort. Therefore, work is needed in ways to make the tools themselves easier to create. For example, the Garnet

toolkit explored mechanisms specifically designed to make high-level graphical tools easier to create [Myers 1992d]. The Unidraw framework has also proven useful for creating interface builders [Vlissides 1991]. However, more work is needed.

51.8. Conclusions

Generally, research and innovation in *tools* trail innovation in user interface *design*, since it only makes sense to develop tools when you know for what kinds of interfaces you are building tools. Given the consolidation of the user interface interaction style in the last 15 years, it is not surprising that tools have matured to the point where commercial tools have fairly successfully covered the important aspects of user interface construction. It is clear that the research on user interface software tools has had enormous impact on the process of software development. Now, user interface design is poised for a radical change, primarily brought on by the rise of the World-Wide-Web, ubiquitous computing, recognition-based user interfaces, handheld devices, wireless communication, and other technologies. Therefore, we expect to see a resurgence of interest and research on user interface software tools in order to support the new user interface styles.

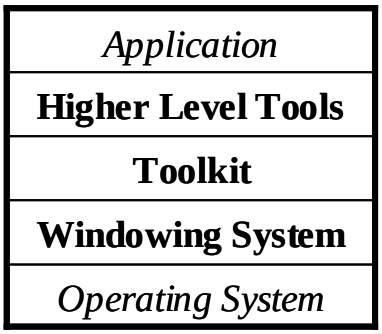


Figure 1: The components of user interface software.

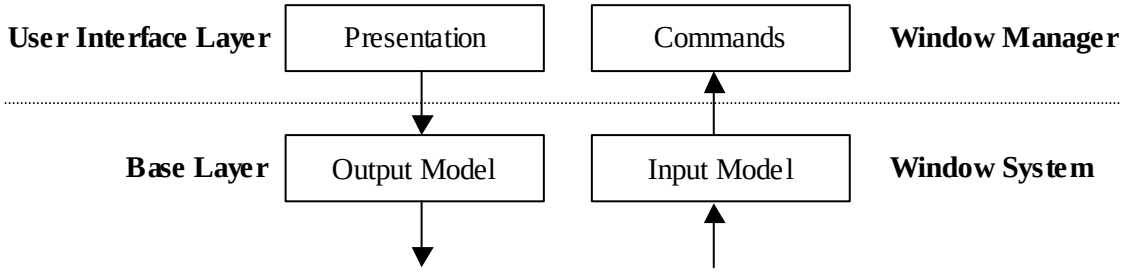


Figure 2: The windowing system can be divided into two layers, called the base or window system layer, and the user interface or window manager layer. Each of these can be divided into parts that handle output and input.

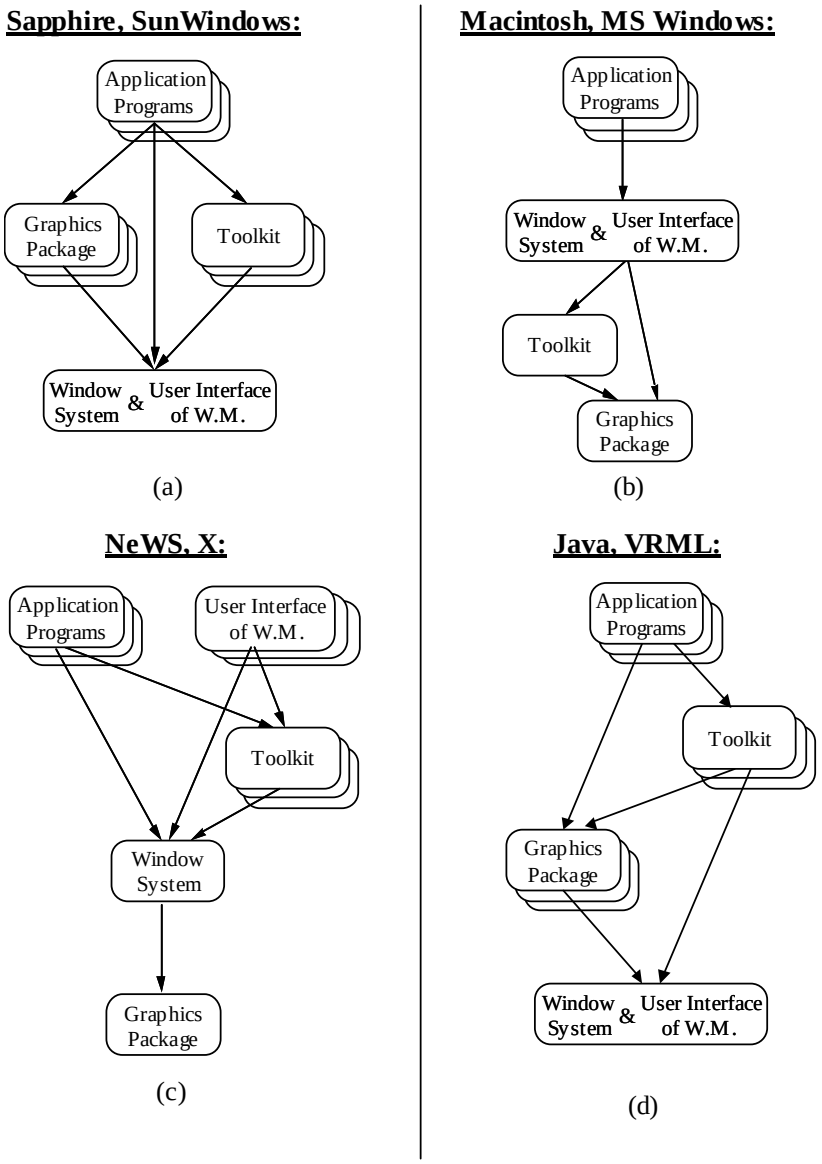


Figure 3: Various organizations that have been used by windowing systems. Boxes with extra borders represent systems that can be replaced by users. Early systems (a) tightly coupled the window manager and the window system, and assumed that sophisticated graphics and toolkits would be built on top. The next step in designs (b), was to incorporate into the windowing system the graphics and toolkits, so that the window manager itself could have a more sophisticated look and feel, and so applications would be more consistent. Other systems (c) allow different window managers and different toolkits, while still embedding sophisticated graphics packages. Newer systems (d) hark back to the original design (a) and implement the graphics and toolkit on top of the window system.

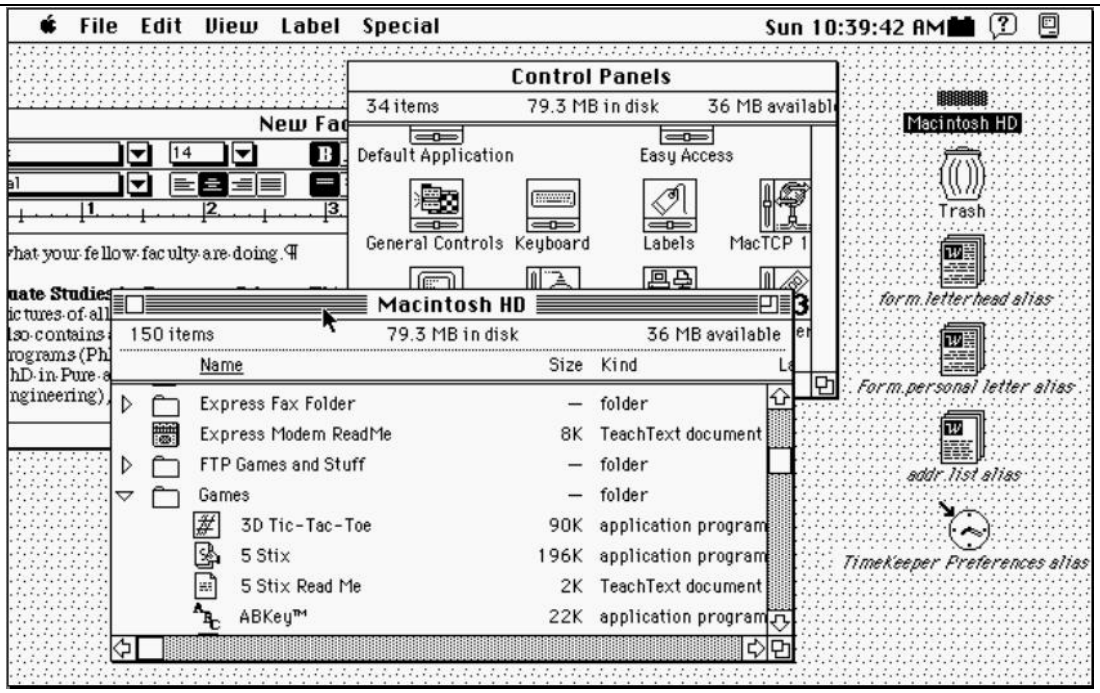


Figure 4: A screen from the original Macintosh showing 3 windows covering each other, and some icons along the right margin.

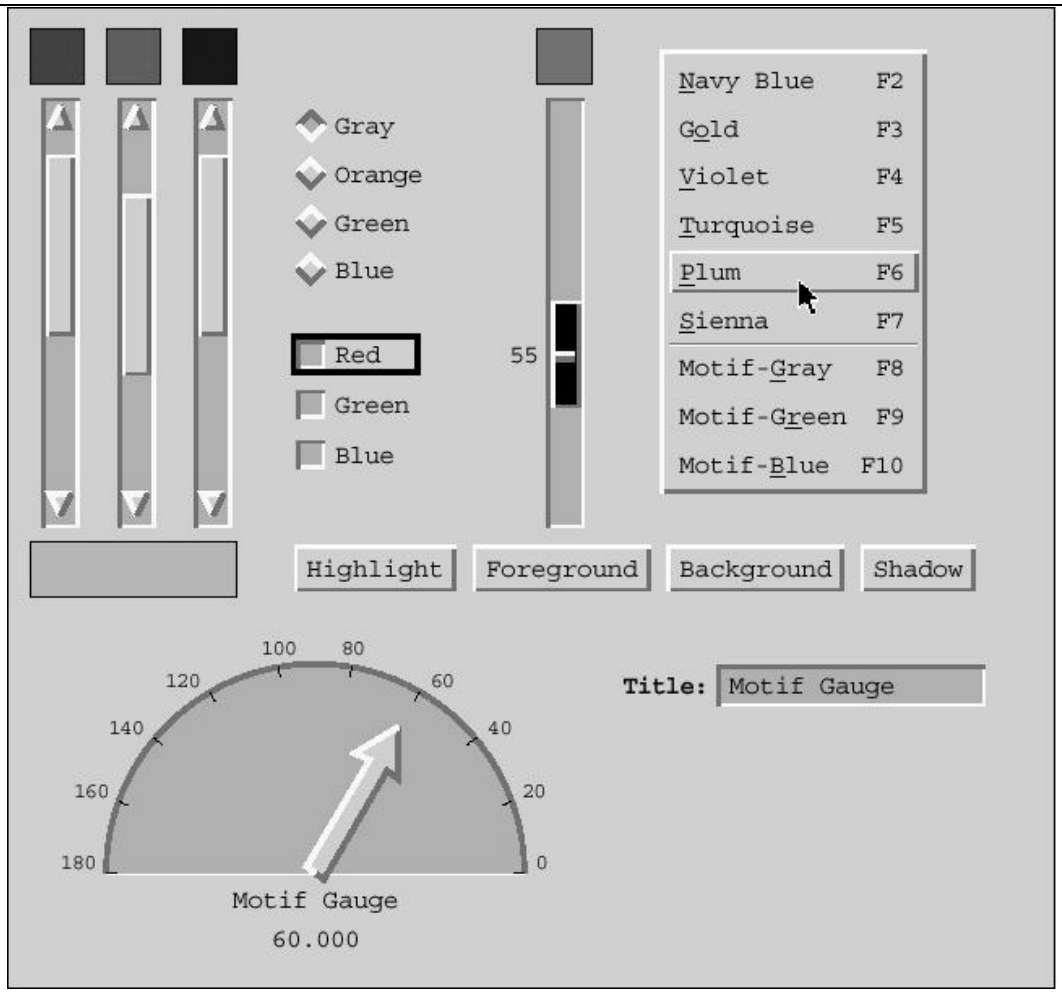
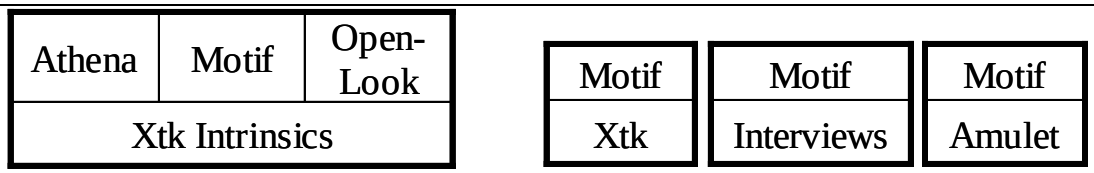


Figure 5: Some of the widgets with a Motif look-and-feel provided by the Garnet toolkit.



(a) (b)

Figure 6: (a) At least three different widget sets that have different looks and feels were implemented on top of the Xt intrinsics. (b) The Motif look-and-feel has been implemented on many different intrinsics.

Defining Terms

Application or application semantics: The part of the software that is *not* the user interface.

Application Framework: A software architecture, often object-oriented, that guides the programmer so that implementing user interface software is easier.

Call-back procedures: Procedures defined by the application programmer that are called when a widget is operated by the end user.

Constraints: Relationships that are declared once and then maintained automatically by the system.

Geometry Management: Part of the toolkit intrinsics that handles the placement and size of widgets.

GUI: Graphical User Interface: A form of user interface that makes significant use of the direct manipulation style using pointing with a mouse.

Icons: Small pictures that represent windows (or sometimes files) in window managers.

Interface Builder: Interactive tool that lays out widgets to create dialogue boxes, menus and windows that are to be part of a larger user interface. These are also called *Interface Development Tools* and *GUI Builders*.

Intrinsics: The layer of a toolkit on which different widgets are implemented.

Model-View-Controller: Model of how user interface software might be organized, separating the application data (model), presentation (view), and input handling (controller) aspects.

Prototyping tools: These allow the designer to quickly mock up some examples of what the screens in the program will look like. Often, these tools cannot be used to create the real user interface of the program; they just show how some aspects will look.

Seeheim model: Model of how user interface software might be organized, separating the presentation, dialogue and application aspects.

Toolkit: A library of widgets that can be called by application programs.

User Interface (UI): The part of the software that handles the output to the display and the input from the person using the program.

User Interface Development Environments (UIDE): General term for comprehensive user interface tools.

User Interface Management System (UIMS): An older term, not much used now. Sometimes used to cover all user interface tools, but usually limited to tools that handle the sequencing of operations (what happens after each event from the user).

User Interface Tool: Any software that helps create user interfaces.

Virtual toolkits: Also called *cross-platform development systems,* these are programming interfaces to multiple toolkits that allow code to be easily ported to Macintosh, Microsoft Windows and Unix environments.

Visual Programming: Using graphics and two (or more) dimensional layout as part of the program specification.

Widget: A way of using a physical input device to input a certain type of value. Typically, widgets in toolkits include menus, buttons, scroll bars, text type-in fields, etc.

Window: Region of the screen (usually rectangular) that can be independently manipulated by a program and/or user.

Window manager: The user interface of the windowing system. Also used to mean the entire windowing system.

Windowing system: Software that separates different processes into different rectangular regions (*windows*) on the screen.

References

[Adobe Systems Inc 1985] Adobe Systems Inc. *Postscript Language Reference Manual*. Addison-Wesley. 1985.

[Apple Computer Inc. 1985] Apple Computer Inc. *Inside Macintosh*. Addison-Wesley. 1985.

[Banavar 2000] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman and Deborra Zukowski. "Challenges: An Application Model for Pervasive Computing," *Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000),* 2000.http://www.research.ibm.com/PIMA/

[Bharat 1994] Krishna Bharat and Marc H. Brown. "Building Distributed, Multi-User Applications by Direct Manipulation," *ACM SIGGRAPH Symposium on User Interface Software and Technology,* Proceedings UIST'94. Marina del Rey, CA, Nov, 1994, 1994. pp. 71-81.

[Bly 1986] Sara A. Bly and Jarrett K. Rosenberg. "A Comparison of Tiled and Overlapping Windows," *Human Factors in Computing Systems,* Proceedings SIGCHI'86. Boston, Mass, Apr, 1986, 1986. pp. 101-106.

[Borning 1981] Alan Borning. "The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory," *ACM Transactions on Programming Languages and Systems. ACM Transactions on Programming Languages and Systems.* 1981. **3**(4). pp. 353-387.

[Borning 1986a] Alan Borning. "Defining Constraints Graphically," *Human Factors in Computing Systems,* Proceedings SIGCHI'86. Boston, MA, Apr, 1986, 1986a. pp. 137-143.

[Borning 1986b] Alan Borning and Robert Duisberg. "Constraint-Based Tools for Building User Interfaces," *ACM Transactions on Graphics.* 1986b. **5**(4). pp. 345-374.

[Buxton 1983a] W. Buxton, M.R. Lamb, D. Sherman and K.C. Smith. "Towards a Comprehensive User Interface Management System," *Computer Graphics*, Proceedings SIGGRAPH'83. Detroit, Mich, Jul, 1983, 1983a. pp. 35-42.

[Buxton 1983b] William Buxton. "Lexical and Pragmatic Considerations of Input Structures," *Computer Graphics. Computer Graphics.* 1983b. **17**(1). pp. 31-37.

[Cardelli 1988] Luca Cardelli. "Building User Interfaces by Direct Manipulation," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'88. Banff, Alberta, Canada, Oct, 1988, 1988. pp. 152-166.

[Cardelli 1985] Luca Cardelli and Rob Pike. "Squeak: A Language for Communicating with Mice," *Computer Graphics*, Proceedings SIGGRAPH'85. San Francisco, CA, July 22-26, 1985, 1985. pp. 199-204.

[Click2learn 1995] Click2learn. *ToolBook. Click2learn, Inc. (formerly Asymetrix Corporation), 110 110th Ave NE, Bellevue, WA 98004*. http://www.asymetrix.com/en/toolbook/index.asp

[Curbow 1995] Dave Curbow, Elizabeth Dykstra-Erickson, Kerry Orteg and Geoff Schuller. *Human Interface Specification for the Macintosh Implementation*. Apple Computer, Inc. OpenDoc Version 1.0, Specification Version 1.0.3. November 6, 1995.

[DataViews 2001]DataViews. *GE Fanuc Automation NA. 1 Columbia Circle Drive, Albany, NY 12203-5189, USA*. www.dvcorp.com/

[eNGENUITY 2002] eNGENUITY. *VAPS. eNGENUITY Technologies (fomerly: Virtual Prototypes, Inc.), 4700 de la Savane, Suite 300, Montreal, Quebec CANADA, H4P 1T7*. http://www.engenuitytech.com/

[Fisher 1992] Gene L. Fisher, Dale E. Busse and David A. Wolber. "Adding Rule-Based Reasoning to a Demonstrational Interface Builder," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'92. Monterey, CA, Nov, 1992, 1992. pp. 89-97.

[Frank 1995] Martin R. Frank. *Model-Based User Interface by Demonstration and by Interview*. Computer Science Department, Georgia Institute of Technology. 1995. PhD Thesis.

[Gallium 1991] Gallium. *InterMAPhics. Gallium Software (formerly: Prior Data Systems), Suite 4000. 303 Moodie Drive, Ottawa, Ontario, K2H 9R4 Canada*. http://www.gallium.com/

[Gaskins 1992] Tom Gaskins. *PEXlib Programming Manual*. 103 Morris Street, Suite A, Sebastopol CA, O'Reilly and Associates, Inc. 1992.

[GNOME 2002] GNOME. *GNU Network Object Model Environment (GNOME)*. 2002. http://www.gnome.org/

[Gosling 1986] James Gosling. *NeWS: A Definitive Approach to Window Systems*. Mountain View, Calif, Sun Microsystems Corp. 1986.

[Green 1986] Mark Green. "A Survey of Three Dialog Models," *ACM Transactions on Graphics. ACM Transactions on Graphics.* 1986. **5**(3). pp. 244-275.

[Haartsen 1998] Jaap Haartsen, Mahmoud Naghshineh, Jon Inouye, Olaf J. Joeressen and Warren Allen. "Bluetooth: Vision, Goals, and Architecture," *ACM Mobile Computing and Communications Review.* 1998. **2**(4). pp. 38-45. Oct. www.bluetooth.com.

[Hartson 1990] H. Rex Hartson, Antonio C. Siochi and Deborah Hix. "The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs," *ACM Transactions on Information Systems. ACM Transactions on Information Systems.* 1990. **8**(3). pp. 181-203.

[Hayes 1985] Philip J. Hayes, Pedro A. Szekely and Richard A. Lerner. “Design Alternatives for User Interface Management Systems Based on Experience with COUSIN,” *Human Factors in Computing Systems*, Proceedings SIGCHI'85. San Francisco, CA, Apr, 1985, 1985. pp. 169-175.

[Hill 1986] Ralph D. Hill. “Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction -- The Sassafras UIMS,” *ACM Transactions on Graphics.* 1986. **5**(3). pp. 179-210.

[Hill 1994] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson and Wayne Wilner. “The Rendezvous Architecture and Language for Constructing Multiuser Applications,” *ACM Transactions on Computer-Human Interaction.* 1994. **1**(2). pp. 81-125.

[Hudson 1996] Scott E. Hudson and Ian Smith. “Ultra-Lightweight Constraints,” *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'96. Seattle, WA, Nov, 1996. pp. 147-155. http://www.cc.gatech.edu/gvu/ui/sub_arctic/.

[Hudson 1993] Scott E. Hudson and John T. Stasko. “Animation Support in a User Interface Toolkit: Flexible, Robust, and Reusable Abstractions,” *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'93. Atlanta, GA, Nov, 1993. pp. 57-67.

[Jacob 1986] Robert J.K. Jacob. “A Specification Language for Direct Manipulation Interfaces,” *ACM Transactions on Graphics.* 1986. **5**(4). pp. 283-317.

[JavaSoft 1996] JavaSoft. *JavaBeans*. Sun Microsystems. JavaBeans V1.0. December 4, 1996. http://java.sun.com/beans.

[Kieras 1995] David E. Kieras, Scott D. Wood, Kasen Abotel and Anthony Hornof. “GLEAN: A Computer-Based Tool for Rapid GOMS Model Usability Evaluation of User Interface Designs,”

Eighth Annual Symposium on User Interface Software and Technology, Proceedings UIST'95. Pittsburgh, PA, Nov, 1995. pp. 91-100.

[Kim 1993] Won Chul Kim and James D. Foley. "Providing High-level Control and Expert Assistance in the User Interface Presentation Design," *Human Factors in Computing Systems*, Proceedings INTERCHI'93. Amsterdam, The Netherlands, Apr, 1993. pp. 430-437.

[Krasner 1988] Glenn E. Krasner and Stephen T. Pope. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system," *Journal of Object Oriented Programming. Journal of Object Oriented Programming.* 1988. **1**(3). pp. 26-49.

[Landay 1995] James Landay and Brad A. Myers. "Interactive Sketching for the Early Stages of User Interface Design," *Human Factors in Computing Systems*, Proceedings SIGCHI'95. Denver, CO, May, 1995. pp. 43-50.

[Lin 2002] James Lin, Michael Thomsen and James A. Landay. "A Visual Language for Sketching Large and Complex Interactive Designs," *ACM CHI'2002 Conference Proceedings: Human Factors in Computing Systems*, Minn, MN, April 20-25, 2002. pp. 307-314.

[Linton 1989] Mark A. Linton, John M. Vlissides and Paul R. Calder. "Composing user interfaces with InterViews," *IEEE Computer. IEEE Computer.* 1989. **22**(2). pp. 8-22.

[Macromedia 2003a] Macromedia. *Director MX. Macromedia, Inc. 600 Townsend Street, San Francisco, CA 94103*. http://www.macromedia.com/software/director/

[Macromedia 2003b] Macromedia. *Dreamweaver MX. Macromedia, Inc. 600 Townsend Street, San Francisco, CA 94103*. http://www.macromedia.com/software/dreamweaver/

[McCormack 1988] Joel McCormack and Paul Asente. "An Overview of the X Toolkit," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'88. Banff, Alberta, Canada, Oct, 1988, 1988. pp. 46-55.

[McDaniel 1999] Richard G. McDaniel and Brad A. Myers. "Getting More Out Of Programming-By-Demonstration," *Human Factors in Computing Systems*, Proceedings CHI'99. Pittsburgh, PA, May 15-20, 1999. pp. 442-449.

[Moran 2001] Thomas P. Moran and Paul Dourish. "Special Issue on Context-Aware Computing," *HCI Journal*. 2001. **16**(2-4). pp. 87-419.

[Myers 1996a] Brad A Myers and David Kosbie. "Reusable Hierarchical Command Objects," *Proceedings CHI'96: Human Factors in Computing Systems*, Vancouver, BC, Canada, April 14-18, 1996a. pp. 260-267.

[Myers 1984] Brad A. Myers. "The User Interface for Sapphire," *IEEE Computer Graphics and Applications. IEEE Computer Graphics and Applications.* 1984. **4**(12). pp. 13-23.

[Myers 1986] Brad A. Myers. "A Complete and Efficient Implementation of Covered Windows," *IEEE Computer. IEEE Computer*. 1986. **19**(9). pp. 57-67.

[Myers 1988a] Brad A. Myers. *Creating User Interfaces by Demonstration*. Boston, Academic Press. 1988a.

[Myers 1988b] Brad A. Myers. "A Taxonomy of User Interfaces for Window Managers," *IEEE Computer Graphics and Applications. IEEE Computer Graphics and Applications.* 1988b. **8**(5). pp. 65-84.

[Myers 1990a] Brad A. Myers. "All the Widgets," *SIGGRAPH Video Review. SIGGRAPH Video Review.* 1990a. **57**

[Myers 1990b] Brad A. Myers. "A New Model for Handling Input," *ACM Transactions on Information Systems.* 1990b. **8**(3). pp. 289-320.

[Myers 1990c] Brad A. Myers. "Taxonomies of Visual Programming and Program Visualization," *Journal of Visual Languages and Computing.* 1990c. **1**(1). pp. 97-123.

[Myers 1991a] Brad A. Myers. "Graphical Techniques in a Spreadsheet for Specifying User Interfaces," *Human Factors in Computing Systems,* Proceedings SIGCHI'91. New Orleans, LA, Apr, 1991a. pp. 243-249.

[Myers 1991b] Brad A. Myers. "Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs," *ACM SIGGRAPH Symposium on User Interface Software and Technology,* Proceedings UIST'91. Hilton Head, SC, Nov, 1991b. pp. 211-220.

[Myers 1992a] Brad A. Myers. "Demonstrational Interfaces: A Step Beyond Direct Manipulation," *IEEE Computer.* 1992a. **25**(8). pp. 61-73.

[Myers 1992b] Brad A. Myers, Ed. *Languages for Developing User Interfaces*. Boston, MA, Jones and Bartlett. 1992b.

[Myers 1994] Brad A. Myers. "Challenges of HCI Design and Implementation," *ACM Interactions. ACM Interactions.* 1994. **1**(1). pp. 73-83.

[Myers 1995] Brad A. Myers. "User Interface Software Tools," *ACM Transactions on Computer Human Interaction.* 1995. **2**(1). pp. 64-103.

[Myers 1998] Brad A. Myers. "A Brief History of Human Computer Interaction Technology," *ACM interactions.* 1998. **5**(2). pp. 44-54. March.

[Myers 2001] Brad A. Myers. "Using Hand-Held Devices and PCs Together," *Communications of the ACM*. 2001. **44**(11). pp. 34-41. http://www.cs.cmu.edu/~pebbles/papers/pebblescacm.pdf

[Myers 1990d] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces," *IEEE Computer*. 1990d. **23**(11). pp. 71-85.

[Myers 1993] Brad A. Myers, Richard G. McDaniel and David S. Kosbie. "Marquise: Creating Complete User Interfaces by Demonstration," *Human Factors in Computing Systems*, Proceedings INTERCHI'93. Amsterdam, The Netherlands, Apr, 1993. pp. 293-300.

[Myers 1997] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski and Patrick Doane. "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Transactions on Software Engineering*. 1997. **23**(6). pp. 347-365. June.

[Myers 1996b] Brad A. Myers, Robert C. Miller, Rich McDaniel and Alan Ferrency. "Easily Adding Animations to Interfaces Using Constraints," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'96. Seattle, WA, Nov, 1996b. pp. 119-128. http://www.cs.cmu.edu/~amulet.

[Myers 1992c] Brad A. Myers and Mary Beth Rosson. "Survey on User Interface Programming," *Human Factors in Computing Systems*, Proceedings SIGCHI'92. Monterey, CA, May, 1992c. pp. 195-202.

[Myers 1992d] Brad A. Myers and Brad Vander Zanden. "Environment for Rapid Creation of Interactive Design Tools," *The Visual Computer; International Journal of Computer Graphics. The Visual Computer; International Journal of Computer Graphics*. 1992d. **8**(2). pp. 94-116.

[Myers 2000] Brad Myers, Scott E. Hudson and Randy Pausch. "Past, Present and Future of User Interface Software Tools," *ACM Transactions on Computer Human Interaction.* 2000. **7**(1). pp. 3-28.

[National Instruments 2003] National Instruments. *LabVIEW. National Instruments Corporation, 11500 N Mopac Expwy, Austin, TX 78759-3504*. http://www.ni.com/

[Newman 1968] William M. Newman. "A System for Interactive Graphical Programming," *AFIPS Spring Joint Computer Conference,* April 30-May 2, 1968. pp. 47-54.

[Nichols 2002] Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joe Hughes, Thomas K. Harris, Roni Rosenfeld and Mathilde Pignol. "Generating Remote Control Interfaces for Complex Appliances," *CHI Letters: ACM Symposium on User Interface Software and Technology, UIST'02,* Paris, France, Oct, 2002. pp. 161-170. http://www.cs.cmu.edu/~pebbles/papers/PebblesPUCuist.pdf

[Nielsen 1993a] Jakob Nielsen. "Noncommand User Interfaces," *CACM. CACM.* 1993a. **36**(4). pp. 83-99.

[Nielsen 1993b] Jakob Nielsen. *Usability Engineering*. Boston, Academic Press. 1993b.

[Olsen Jr. 1987] Dan R. Olsen Jr. "Larger Issues in User Interface Management," *Computer Graphics.* 1987. **21**(2). pp. 134-137.

[Olsen Jr. 1989] Dan R. Olsen Jr. "A Programming Language Basis for User Interface Management," *Human Factors in Computing Systems*, Proceedings SIGCHI'89. Austin, TX, Apr, 1989, 1989. pp. 171-176.

[Olsen Jr. 1992] Dan R. Olsen Jr. *User Interface Management Systems: Models and Algorithms*. San Mateo, CA, Morgan Kaufmann. 1992.

[Olsen Jr. 1983] Dan R. Olsen Jr. and Elizabeth P. Dempsey. “Syngraph: A Graphical User Interface Generator,” *Computer Graphics*, Proceedings SIGGRAPH'83. Detroit, MI, July 25-29, 1983, 1983. pp. 43-50.

[Olsen Jr. 1993] Dan R. Olsen Jr., James D. Foley, Scott E. Hudson, James Miller and Brad Myers. “Research Directions for User Interface Software Tools,” *Behaviour and Information Technology*. 1993. **12**(2). pp. 80-97.

[Olsen Jr. 1988] Dan R. Olsen Jr. and Bradley W. Halversen. “Interface Usage Measurements in a User Interface Management System,” *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'88. Banff, Alberta, Canada, Oct, 1988, 1988. pp. 102-108.

[Oracle Tools 1995] Oracle Tools. *Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA, 94065, (800) 633-0521*.

[Ousterhout 1991]John K. Ousterhout. “An X11 Toolkit Based on the Tcl Language,” *Winter USENIX Technical Conference*, 1991. pp. 105-115.

[Owl International Inc. 1991] Owl International Inc. *Guide 2. 2800 156th Avenue SE, Second Floor, Bellevue, WA 98007. (206) 747-3203.*

[Palay 1988] Andrew J. Palay, Wilfred J. Hansen, Michael Kazar, Mark Sherman, Maria Wadlow, Tom Neuendorffer, Zalman Stern, Miles Bader and Thom Peters. “The Andrew Toolkit - An Overview,” *Proceedings Winter Usenix Technical Conference*, Dallas, Tex, Feb, 1988. pp. 9-21.

[Pausch 1995] Randy Pausch, Tommy Burnette, A.C. Capehart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Suichi Koga and Jeff White. “Alice: A Rapid Prototyping System for 3D Graphics,” *IEEE Computer Graphics and Applications.* 1995. **15**(3). pp. 8-11. May.

[Pausch 1992] Randy Pausch, Matthew Conway and Robert DeLine. “Lesson Learned from SUIT, the Simple User Interface Toolkit,” *ACM Transactions on Information Systems. ACM Transactions on Information Systems.* 1992. **10**(4). pp. 320-344.

[Petzold 1991] C. Petzold. “Windows 3.1 - Hello to TrueType, OLE, and Easier DDE; Farewell to Real Mode,” *Microsoft Systems Journal. Microsoft Systems Journal.* 1991. **6**(5). pp. 17-26.

[Pfaff 1985] Gunther R. Pfaff, Ed. *User Interface Management Systems*. Berlin, Springer-Verlag. 1985.

[Pictorius 2002] Pictorius. *Prograph. Pictorius Incorporated, 2000 Barrington Street, Suite 506, Halifax, Nova Scotia, B3J 3K1, CANADA.* http://www.pictorius.com/

[Ponnekanti 2001] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan and T.Winograd. “ICrafter: A service framework for ubiquitous computing environments,” *UBICOMP 2001,* Atlanta, Georgia, 2001. pp. 56-75.

[Roseman 1996] M. Roseman and S. Greenberg. “Building Real Time Groupware with GroupKit, A Groupware Toolkit,” *ACM Transactions on Computer Human Interaction.* 1996. **3**(1). pp. 66-106.

[Russo 1993] Patricia Russo and Stephen Boor. “How Fluent is Your Interface? Designing for International Users,” *Human Factors in Computing Systems*, Proceedings INTERCHI'93. Amsterdam, The Netherlands, Apr, 1993. pp. 342-347.

[Samuelson 1993]Pamela Samuelson. “Legally Speaking: The Ups and Downs of Look and Feel,” *CACM. CACM.* 1993. **36**(4). pp. 29-35.

[Scheifler 1986] Robert W. Scheifler and Jim Gettys. “The X Window System,” *ACM Transactions on Graphics. ACM Transactions on Graphics.* 1986. **5**(2). pp. 79-109.

[Schulert 1985] Andrew J. Schulert, George T. Rogers and James A. Hamilton. "ADM-A Dialogue Manager," *Human Factors in Computing Systems*, Proceedings SIGCHI'85. San Francisco, CA, Apr, 1985. pp. 177-183.

[Silicon Graphics Inc 1993] Silicon Graphics Inc. *Open-GL. 2011 N. Shoreline Blvd. Mountain View, CA 94039-7311. (415) 960-1980.*

[SL Corp 2002] SL Corp. *SL-GMS*. Suite 110 Hunt Plaza, 240 Tamal Vista Blvd., Corte Madera, CA, 94925. http://www.sl.com/

[Smith 1982] David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank and Erik Harslem. "Designing the Star User Interface," *Byte. Byte.* 1982. **7**(4). pp. 242-282.

[Stallman 1979] Richard M. Stallman. *Emacs: The Extensible, Customizable, Self-Documenting Display Editor*. MIT Artificial Intelligence Lab. 519. Aug, 1979, 1979.

[Stevens 1983] Albert Stevens, Bruce Roberts and Larry Stead. "The Use of a Sophisticated Graphics Interface in Computer-Assisted Instruction," *IEEE Computer Graphics and Applications. IEEE Computer Graphics and Applications.* 1983. **3**(2). pp. 25-31.

[Stevens 1994] Marc P. Stevens, Robert C. Zeleznik and John F. Hughes. "An Architecture for an Extensible 3D Interface Toolkit," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'94. Marina del Rey, CA, Nov, 1994, 1994. pp. 59-67.

[Sukaviriya 1990] Piyawadee Sukaviriya and James D. Foley. "Coupling A UI Framework with Automatic Generation of Context-Sensitive Animated Help," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, Proceedings UIST'90. Snowbird, Utah, Oct, 1990, 1990. pp. 152-166.

[Sukaviriya 1993] Piyawadee Sukaviriya, James D. Foley and Todd Griffith. “A Second Generation User Interface Design Environment: The Model and The Runtime Architecture,” *Human Factors in Computing Systems,* Proceedings INTERCHI'93. Amsterdam, The Netherlands, Apr, 1993. pp. 375-382.

[Sun Microsystems 2002] Sun Microsystems. *Java 2D API*. http://java.sun.com/products/java-media/2D/.

[Sun Microsystems 2003] Sun Microsystems. *Java: Programming for the Internet*. http://java.sun.com/.

[Sutherland 1963] Ivan E. Sutherland. “SketchPad: A Man-Machine Graphical Communication System,” *AFIPS Spring Joint Computer Conference,* 1963. pp. 329-346.

[Swinehart 1986] Daniel Swinehart, Polle Zellweger, Richard Beach and Robert Hagmann. “A Structural View of the Cedar Programming Environment,” *ACM Transactions on Programming Languages and Systems. ACM Transactions on Programming Languages and Systems.* 1986. **8**(4). pp. 419-490.

[Tandler 2002] P. Tandler, N. A. Streitz and Th. Prante. “Roomware -- Moving Toward Ubiquitous Computers,” *IEEE Micro.* 2002. **22**(6). pp. 36-47. http://www2.darmstadt.gmd.de/ipsi/ambiente/abstract.asp?Pub_ID=293

[Tandler 2001] Peter Tandler. “Software Infrastructure for Ubiquitous Computing Environments Supporting Synchronous Collaboration with Multiple Single- and Multi-User Devices,” *UbiComp'2001,* Atlanta, Georgia, Springer. Sept 30 - Oct 2, 2001. pp. 96-115. http://ipsi.fraunhofer.de/ambiente/paper/2001/UbiComp-2001-tandler.pdf

[Teitelman 1979] Warren Teitelman. "A Display Oriented Programmer's Assistant," *International Journal of Man-Machine Studies.* 1979. **11**(2). pp. 157-187. Also Xerox PARC Technical Report CSL-77-3, Palo Alto, CA, March 8, 1977.

[Tesler 1981] Larry Tesler. "The Smalltalk Environment," *Byte Magazine.* 1981. **6**(8). pp. 90-147.

[Vander Zanden 1990] Brad Vander Zanden and Brad A. Myers. "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," *Human Factors in Computing Systems,* Proceedings SIGCHI'90. Seattle, WA, Apr, 1990. pp. 27-34.

[Vander Zanden 1995] Brad Vander Zanden and Brad A. Myers. "Demonstrational and Constraint-Based Techniques for Pictorially Specifying Application Objects and Behaviors," *ACM Transactions on Computer-Human Interaction.* 1995. **2**(4). pp. 308-356.

[Visix Software Inc. 1997] Visix Software Inc. *Galaxy Application Environment.* (Company dissolved in 1998. Galaxy was bought by Ambiencia Information Systems, Inc., Campinas, Brazil, support@ambiencia.com. http://www.ambiencia.com).

[Vlissides 1991] John M. Vlissides and Steven Tang. "A Unidraw-Based User Interface Builder," *ACM SIGGRAPH Symposium on User Interface Software and Technology,* Proceedings UIST'91. Hilton Head, SC, Nov, 1991, 1991. pp. 201-210.

[Web3D Consortium 1997] Web3D Consortium. *The Virtual Reality Modeling Language*. 1997. ISO/IEC 14772-1:1997. http://www.web3d.org/Specifications/VRML97/

[Weiser 1993] Mark Weiser. "Some Computer Science Issues in Ubiquitous Computing," *CACM.* 1993. **36**(7). pp. 74-83. July.

[Wernecke 1994] Josie Wernecke. *The Inventor Mentor*. Reading, MA, Addison-Wesley Publishing Company. 1994.

[Wiecha 1990] Charles Wiecha, William Bennett, Stephen Boies, John Gould and Sharon Greene. "ITS: A Tool for Rapidly Developing Interactive Applications," *ACM Transactions on Information Systems.* 1990. **8**(3). pp. 204-236.

[Wilde 1990] Nicholas Wilde and Clayton Lewis. "Spreadsheet-based Interactive Graphics: from Prototype to Tool," *Human Factors in Computing Systems,* Proceedings SIGCHI'90. Seattle, WA, Apr, 1990, 1990. pp. 153-159.

[Wilson 1990] David Wilson. *Programming with MacApp*. Reading, MA, Addison-Wesley Publishing Company. 1990.

[Wolber 1997] David Wolber. "An Interface Builder for Designing Animated Interfaces," *ACM Transactions on Computer-Human Interaction.* 1997. **4**(4). pp. 347-386. Dec.

[X Business Group 1994] Inc. X Business Group. *Interface Development Technology*. 3155 Kearney Street, Suite 160, Fremont, CA 94538. (510) 226-1075. 1994.

[XVT Software Inc 1997] XVT Software Inc. *XVT. 4900 Pearl East Circle, Boulder, CO, 80301, USA, 1-800-678-7988 or (303) 443-4223*. http://www.xvt.com/

[Zimmerman 2002] Gottfried Zimmerman, Gregg Vanderheiden and Al Gilman. "Prototype Implementations for a Universal Remote Console Specification," *Human Factors in Computing Systems,* Extended Abstracts for CHI'2002. Minneapolis, MN, Apr 1-6, 2002. pp. 510-511. see also: http://www.ncits.org/tc_home/v2.htm