WINDOWS SYSTEMS

The term *window system* refers to the software that supports graphical user interfaces (GUI). Informally, the term *window* refers to a screen area for which the system supports the following: (a) connecting user actions to the area and (b) drawing the proper visual information on the area. The term *window object* refers collectively to all the software structures that contain information related to a window, and the term *desktop environment* denotes the collection of windows that appear on the screen.

In many systems (such as Apple's Macintosh or Microsoft's Windows 95) the graphical interface is the dominant access mode of the system and the window system supports many operations that are not graphical in any way. This article does not attempt to cover features that go beyond the direct support of GUIs.

USER REQUIREMENTS

It will be helpful to briefly review what users expect from a GUI, since these requirements guide the design of a window system (see GRAPHICAL USER INTERFACES for more details).

The main mode of user interaction in a GUI is "point and click"; therefore the essential program code should be in the procedures that are invoked upon user action. Such procedures are generally called callbacks. Users expect certain visual clues to be associated with the functionality of a window. Therefore windows where a click causes the invocation of a single process have the appearance of a button that looks pressed when the mouse button is pressed and pops out when the mouse button is released. If the user action should result in the selection of a specific discrete parameter value, the button should stay depressed. Such buttons are called *toggles* or *radio buttons*. Windows that select from a continuum of values usually have the appearance of a slider or a scrollbar.

The selection windows (buttons) are arranged in menus that may be permentally displayed or appear only upon user action (popup or pulldown menus). Buttons may be labeled with text or with images. The term *icon* refers to small images that are used to label not only buttons but any window of small size. A *dialogue box* is a temporary window that displays a message to the user, who must provide a simple response.

When an application is running and provides complete functionality, its windows tend to occupy a significant area in

the screen. Because the user may stop using the application for awhile and resume later, the window system must provide means for reducing the size of window and using a symbolic representation to signify the presence of the application. That is typically done through an icon and a label. This process is known by the terms *minimization, closing a window,* or *iconifying a window*. (You should be aware that in some systems the term "closing a window" implies terminating an application. Unfortunately, there is a great anarchy of terms in window systems.)

In addition to minimizing windows, the system must provide means for resizing windows as well as moving them in front of others. This imposes the requirement of being able to reconstruct on demand the appearance of a window and adapting to larger or smaller drawing areas. Also, the system must be able to allocate resources for new windows and keep track of their *stacking order,* since windows may overlap.

Users often want to transfer data from one window into another; for example, they want to cut a piece of text from one editing window and paste it onto another. Such a transfer may take place through a "drag and drop" interface. Therefore a window system must support communication between applications.

SYSTEM OVERVIEW

There are many ways to implement a window system. Typically, a window corresponds to a structure whose members include at least the position and size of a rectangle and a pointer to a memory area in the display where what is drawn on the window will be displayed. (If nothing is drawn, the window will appear blank.) Other information about the window may be stored in the same structure or to other structures that are linked together.

Windows are usually rectangles, but there is no fundamental reason for not having other shapes. We could have windows that are oval, or star-shaped, or even doughnut-shaped. Regardless of the shape we must provide a procedure that determines whether a given screen point is inside the window area. We need this information both for confining a drawing within the window area and for finding the window where the pointer (cursor) is located. Because checking the point containment is much simpler for rectangles with sides parallel to the coordinate axes than for any other shape, most systems support only rectangular windows with vertical and horizontal sides.

When the user interacts with the display by manipulating the mouse, the keyboard, or any other input device, the operating system receives interrupts. Such interrupts are mapped into *events,* structures that contain not only information about the interrupt itself (for example that the mouse moved) but also other related information (for example, coordinates of the cursor, time of the day, etc). In modern window systems events are generated not only by interrupts but by numerous other conditions, including program actions.

When an event is generated, it is associated with a particular window (usually the one where the cursor is in). Normally, for each window there is a function, the *callback,* to be called in response for each particular type of event.

Application programs running in a window system start by creating one or more windows, initializing various structures, and then suspending execution. When an event occurs that is associated with one of the windows, an appropriate function is executed. In other words, the program does nothing most of the time, except when an event occurs. Such programs are called event-driven programs.

Today all major window systems encourage event driven programming. These systems include the X Window System (X) (1–11), Microsoft (MS) Windows (including Windows 95, Windows NT, etc) (12–16), the BeBox Interface Kit (Be Kit) (17), and the Abstract Windowing Toolkit (AWT) of Java (18,19). The latter is not surprising since Java's AWT always runs on top of another window system. X has an extension (SHAPE) that supports nonrectangular windows (see Chapter 18 of Ref. 10).

While all these systems have many similarities, they use incompatible terminology. MS Windows and the Be Kit use the term *message* instead of "event," so they refer to application programs running in them as message-driven programs. On the other hand, similar terms are used for different concepts (for example, the word "resources" means different things in X and in MS Windows). Most confusing is the use of the term "window" with similar but not identical meaning in these systems. In this article we use the term informally to refer to the screen object that the user sees.

We should also mention that it is possible to access window systems through *scripting* languages. Particular examples include the *Tk* toolkit (20) as well as Javascript and HTML forms.

WINDOW MANAGERS, SERVERS, AND CLIENTS

From the viewpoint of the application user, a window system needs a program that lets the user manipulate windows as well as start applications by pointing and clicking on a particular part of the screen, or by selecting from a menu. This program is the window manager (WM). While it is possible to run a window system without a WM, it is quite awkward to do so.

From the viewpoint of the application programmer a window system must provide means for accessing the graphics hardware for input and output. Instead of requiring programming at the machine code level, modern window systems provide an Applications Programming Interface (API) through a library of graphics functions. The API for the X Window System is *Xlib* and the API for most recent versions of Microsoft Windows (95 and NT) is *Win32*. The Be Kit and Java are strongly object-oriented, and graphics are performed with the methods of various graphics objects.

Because the X window system was designed to run over a network, *Xlib* functions do not interact directly with the hardware. Instead they generate or interpret messages according to the X Protocol. There is a different program that translates such messages to graphics machine code, or converts such code to messages. X uses the counter-intuitive terms *server* for the part that deals with the hardware and *client* for the application part.

A server program is necessary for any window system that allows programs to run on a different machine than the one in front of the user. A key issue is whether the server should be independent of the application. Indeed, it is possible to write the client and server parts for each application. The

server part would be down-loaded at start of the execution of an application. This solution was adopted by the *Blit* (21). X opted for a common server solution: Each display device has a server program running that communicates with any application through a socket mechanism.

If we accept the common server solution, we must decide what functionality to give to the server. Handling the low-level graphics communications protocol and allocation of hardware resources is a minimal functionality. This is the policy that X has adopted, thus simplifying the server design. There are two additional major functionalities that a server may have: that of the window manager and the ability to handle the definition of new functions.

The functions of the window manager are closely connected to the hardware and a system would be more efficient if window manager and server were parts of the same program. Because the design of the window manager places certain restrictions on the "look and feel" of the GUI, X opted to separate the two programs, so users would have more choices. This decision had two important consequences: On one hand it made the server simpler and the other hand it made *Xlib* much larger because it had to contain functions that were needed by window manager programs. The Be Kit uses an Application Server that combines the functions of the window manager and the X Server (drawing and user input handling).

A desirable server property is to allow applications to define functions. This was possible in an early window system, NeWS, where the server interpreted the Postscript language and therefore allowed the definition of functions. The X server does not support function definitions. This is particularly unfortunate because *Xlib* is a very low level library. For example, it does not have a spline drawing function. To draw a spline one must send the individual curve points to the server. If function definition were allowed, one would need send only the control points of the spline. *Win32* provides functions equivalent to most of the low-level functions of *Xlib,* as well as higher-level functions, including spline drawing. *Win32* and the *Be Kit* also provide certain facilities that perform the role of user-defined functions (see section entitled "Drawing Line Segments").

BACKING UP WINDOWS

When the user manipulates the desktop environment, some windows may be hidden behind others and later brought in front again. A key question is what happens to what was drawn on them before and how do we restore it when the window is brought back up again. There are various solutions to this problem. One extreme solution is to have a copy of the window in off-screen memory. We may even follow the policy that the application program never draws on the window itself, but only in the off-screen memory. When the window (or part of it) becomes visible, the window manager or the server copy the appropriate part of that memory to the window.

Another extreme solution is to make no effort to save any off-screen information. Instead each application program is required to redraw the contents of its windows upon demand. This implies that the application must keep its own internal copy of the information displayed. However, that information may be in far more compact form than a copy of the window itself. For example, if we have a window that displays text we need keep only strings of characters rather than their images. (A character is normally represented by 8 bits, while its image requires 100 bits or more.) Both X and Microsoft Windows have opted for the latter solution. Both impose upon the application the need to respond to a "redrawing event."

An intermediate solution is to provide window back up. When a window (or part of it) is obscured, an off-screen copy is made. The creation of such a copy may be transparent to the application, so the application program may continue to draw on the window. This solution was adopted for the early window system of the *Blit* (21). Window backup was also used by another early window system, Sunview. X offers optional backup, but the mechanism is not guaranteed to be available, and it does not always function as the programmer might expect, so that an application is never relieved of the ultimate responsibility to be able to restore the information displayed on a window.

Window backup or redrawing is also needed when a window is resized. If backup is available, the old display is either cropped (if the window becomes smaller) or surrounded by empty spaces (if window is enlarged). Requiring the application to redraw offers the opportunity to adjust the scale of the display to the new size of the window.

EVENTS OR MESSAGES

Window systems rely on a loop where one statement checks to see if there is an event (message) to be processed. If not, the program suspends execution (usually); otherwise it looks at the event structure. The structure contains information about the window where the event occurred, and on that basis a function is involved. Listing 1 shows the basic loop in X (Ref. 3, pp. 754–756) and Listing 2 shows the basic loop in MS Windows (Ref. 16, p. 95).

Listing 1

```
XEvent event;
while (1) [
 XtAppNextEvent(. . ., &event);
 XtDispatchEvent(&event);
]
```

Listing 2

```
MSG msg;
while ( GetMessage(&msg, . . .) ) [
 TranslateMessage(&msg);
 DispatchMessage(&msg);
]
```

The key function in both systems is a "dispatcher." Each system maintains a window-object tree and the tree is traversed until a match is found between the window of the event (message) and the object, thus "dispatching" the event to the window object. Each object has certain methods that are called when an event is received. While the generic term *callback* is used for such functions, there are many varieties of them, and in some systems the term "callback" is reserved for a particular class. The most general type of an event processing function is called an event handler in X, or a window procedure in MS Windows. The event/message contains a "type" member and the function is built around a switch with the type as argument. Listings 3 and 4 show switch examples for X and MS Windows, respectively.

Listing 3

```
Xevent *ep;
switch (ep->type) [
 case Expose:
   /* Draw on the Window */
   return;
 case MotionNotify:
   /* Respond to Mouse Motion */
   return;
```

Listing 4

```
UINT iMsg;
switch (iMsg) [
 case WM_PAINT:
   /* Draw on the Window */
   return 0;
 case WM_MOUSEMOVE:
   /* Respond to Mouse Motion */
   return 0;
```

The prefix "WM" stands for "Window Message." In X, messages are generated by the server. In the case of mouse motion the Window Manager is not involved at all. However, the Window Manager is the one that has initiated the event that requires drawing on the window since the Window Manager keeps track of which windows are visible.

While details are different and there are other ways of responding to events what is shown in the four tables is the prevailing way and it illustrates the structure of event driven programs.

In Java's AWT the event loop is hidden. Instead the application must provide a `handleEvent()` method whose code is similar to that shown in Listings 3 and 4. Java also has names for methods for specific events: `paint()` and `mouseMove()` are examples with the obvious meaning. The application must provide the code for these functions. Similarly the Be Kit has `Draw()` and `MouseMoved()` methods that must be implemented by the application.

COMPONENTS OF WINDOW SYSTEMS

Windows, Widgets, and Classes

A characteristic of graphical user interfaces is that there is a relatively small number of operations that the window system must support, while the operations themselves can be quite complex. This has led to the building block approach in window systems. A relatively small number of window objects (classes) are defined and applications are built through combinations of them.

The X Window System distinguishes between a low-level window object that describes a screen area and a high-level window object that encompasses event handling and drawing on the window. It uses the term *window* for the low-level object and *widget* for the high-level. Because the functions of *Xlib* are too low level, another library of functions (the X Intrinsics) and objects (the X Widgets) has been built on top of it. Collectively the Intrinsics and the Widgets are known as the X Toolkit (Xt). Additional toolkits have been built on top of Xt, the most widely used being Motif. Therefore direct window creation is rarely found in X applications. Instead it is preferable to create a widget.

Microsoft Windows uses the term "window" for an object that is closer to an X widget than to an X window. The Microsoft Foundation Classes provide formal objects that enhance the basic window classes. The approach used by MS Windows is to provide a basic window class that has considerable functionality. It is the only object that simple applications need. It also provides window classes that may serve as control buttons, text labels, and so on. These classes are simple to use, although the application writer has minimal control over their appearance. In contrast, X allows for unlimited control at the expense of considerable complexity.

The Be Kit has a `BWindow` object that is used only for the top window of an application. It establishes a connection to the Application Server and is associated with a computational thread. `BView`s are the Be objects that most closely correspond to widgets in X and to windows in MS Windows.

Customization is an important question in the design of a window system. Users should be able to adjust the appearance of their desktop without difficulty. X provides a general resource that makes customization quite easy, some might say too easy. Microsoft Windows uses the same term, resource, with a different meaning that overlaps only partly with the meaning of the term in X. Profile files and the Registry support customization that parallels that provided by X resources.

Because of the significant differences in the handling of resources and complex window objects we discuss each system separately. Java creates only partial window objects of its own. For full functionality, it relies on peer objects that are taken from whatever toolkit happens to be available in the system the program runs. Therefore it relies always on another window system.

Resources in X

The X Toolkit (Xt) uses a mechanism that links character strings with arbitrary values. A detailed discussion of Xt Resources is beyond the scope of this article, so we provide only a simplified description. Consider the labels of items in a menu. We would like to use different labels depending on the user's language; thus instead of having their text in the program, we leave the labels undefined and specify them at execution time on the basis of resource file. Let us assume that the application name is `draw` and the buttons have internal names `button_1`, `button_2`, and so on. The resource file takes the following form

```
draw*button_1.label: Save
draw*button_2.label: Save As
draw*button_3.label: Discard
```

The resource values may be changed either by editing the resource file or by using a special dialogue box that is usually part of the application. This is a powerful mechanism, but also a dangerous one. There is no security safeguard to limit resource modification. An incorrect label translation can lead to disaster. A safer solution in this case would have been to have an application resource, "language" and have the resource file specify

```
draw*language: English
```

This assumes that the program has internal lists of all button labels in different languages. Since this is not practical except

for a small number, the safer solution is not general. Resources also provide a powerful conversion mechanism. For example, a color may be specified by name rather than by RGB (red, green, blue) values:

```
draw*button_1.background: lightblue
draw*button_2.background: greenyellow
```

Translation tables are a special type of resources that allow the linking of events to callbacks. For example:

```
draw*: translations: ⟨key⟩q: quit()
```

or

```
draw*: translations: ⟨Btn3Down⟩: quit()
```

The first statement links the event of pressing the "q" key to the callback `quit()`. The second statement links pressing the right mouse button (No. 3) to the same callback. Note that the callback is fixed, and what may be customized is the event activating the callback.

Windows and Widgets in X

In X the term "window" is used to refer to a server window object. A window in X has geometry parameters, attributes, and properties. Parameters include the width and height of the window area, the coordinates of its upper left corner, border width, and so on. Attributes include the background color, events that should be tracked for this window, information about redrawing policies after resizing, and so on. Finally properties are a set of character strings that are used for communication between applications.

The prototype *Xlib* function for creating a simple window is shown in Listing 5. This function call specifies the geometry parameters and two attributes. The type `Window` is not a pointer because the window may be on a different machine than the application that created it. Instead `Window` is an integer that serves as a handle to the window structure. X uses the term X ID (or XID) instead of handle.

Listing 5

```
Window XCreateSimpleWindow(
 Display *, /* pointer to the server */
 Window, /* parent */
 int, int, /* x, y of upper left corner */
 int, int, /* width and height */
 int, /* border width */
 unsigned long, /* border color */
 unsigned long, /* background color */
 )
```

Widget is a client window object that includes a reference to a server window object. There is a clear distinction between members of the object that vary with each implementation (instance record) and those that do not (class record). The former include dimensions, location, and so on. The latter are mostly methods and parameters that remain the same for all members of the class.

The prototype *Xt* function to create a widget is shown in Listing 6.

Listing 6

```
Widget XtCreateWidget(
 String, /* widget name */
 WidgetClass, /* widget class */
 Widget, /* parent */
 ArgList, /* array of structures with
 parameters */
 Cardinal, /* length of array in previous
 argument */
 )
```

Note that *Xt* defines many data types for its own use, even if there is a corresponding C type. Thus `String` is `char *` and `Cardinal` is `int`. `Widget` is a pointer to an instance record. `WidgetClass` is a pointer to a class record. `ArgList` is an array of `Arg` structures. Each one of the latter has two members: a resource name and a resource value. Widget class pointers are known by symbolic names, for example `labelWidgetClass`. These are defined in the public definition file of the class, for example, in `Label.h`. Constructing a new class under Xt is a major effort, thus there is a limited and well-defined set of widget classes that may be used. This is true, even if the class is going to be very simple—a menu button or a drawing widget for example. There is always a very large overhead involved.

The X Toolkit has a hierarchy of widget classes. The basic class is `Core` that has `Composite` as a subclass. The latter has two subclasses: `Constraint` and `Shell`. A shell widget provides means for interacting with the window manager, and a widget of that class serves always as the top widget of an application. A constraint widget provides means for specifying the arrangement of its children through layout rules that remain in effect even after the window is resized.

These classes are not sufficient for most applications, and toolkits such as Motif are used to provide major support. Motif has a `Primitive` class that is a subclass of `Core` and a `Manager` class that is a subclass of `Constraint`. The subclasses of `Manager` include: `RowColumn` for simple layouts (usually vertical or horizontal arrays) and `Form` that allows layouts of considerable complexity. `Label` is a subclass of `Primitive` and has various button types as subclasses: `PushButton` and `ToggleButton`. A `Text` class is a subclass of `Primitive` and can be used to edit text. Motif also supports compound widgets, widgets that consist of a collection of others. Typical examples are popup menus and viewports. A high-level programming language, UIL, is available for prototyping Motif applications (8).

The widget class hierarchy discussed here should be distinguished from the widget tree that is based on the window containment relations in an application. Figure 1 shows on the left the appearance of an application and on the right the widget tree. The top-level shell widget wraps itself tightly

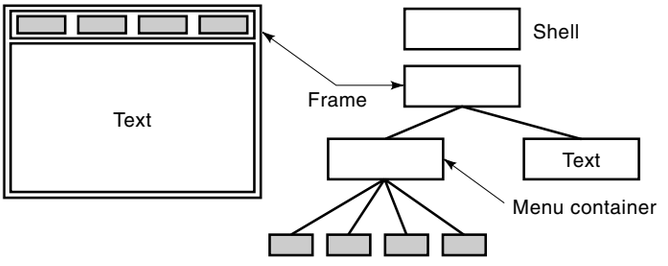

Figure 1. (Left) Appearance of an application. **(Right)** The widget tree.

around the frame. The frame contains a menu container and a text widget. Buttons are marked in gray.

An Example of a Motif Widget

We discuss here in some detail an example of a Motif widget. Listing 7 shows the creation of a push button widget with a fixed label and the assignment of a callback to it. Widget names are used mainly in resource files. The specification of the label is quite cumbersome because Motif provides for labels of alphabets other than Latin. If we want to mark the button with an icon, rather than a text label, we can do so with the code of Listing 8.

Listing 7

```
#include ⟨Xm/PushB.h⟩
 /* . . . */
void save_file(Widget, XtPointer, XtPointer);
 /* . . . */
static char *file_name;
 /* . . . */
static Widget frame, choice[8];
 /* . . . */
choice[0] = XtVaCreateManagedWidget(
 ''button_1'', /* widget name */
 xmPushButtonWidgetClass, /* class */
 frame, /* parent */
 XmNlabelString, /* resource name */
 XmStringCreateLocalized(''Save''), /* label
 */
 NULL);

XtAddCallback(choice[0], /*widget */
 XmNactivateCallback, /* resource name */
 save_file, /* callback */
 (XtPointer)file_name /* to be used as second
 argument in callback */
 );
 /* . . . */
```

Listing 8

```
#include ⟨label_1.bitmap⟩
/* . . . */
Pixmap px_1 = XCreatePixmapFromBitmapData(
 XtDisplay(frame),
 DefaultRootWindow(XtDisplay(frame)),
 label_1_bits, label_1_width, label_1_height,
 foreground_color, background_color,
 DefaultDepth(XtDisplay(frame),
   DefaultScreen ( XtDisplay(frame)))
 );

choice[0] = XtVaCreateManagedWidget(
 ''button_1'', /* widget name */
 xmPushButtonWidgetClass, /* class */
 frame, /* parent */
 XmNlabelType, XmPIXMAP, /* type of label */
 XmNlabelPixmap, /* resource name */
 px_1, /* pixmap */
 NULL);
 /* . . . */
```

The above code assumes that a bitmap of one bit per pixel has been drawn and placed the file `label_1.bitmap`. (See article on RASTER GRAPHICS ARCHITECTURES for the definition of bitmaps.) The file is written in the form of C code and it contains variables `label_1_bits`, `label_1_width`, and `label_1_height` referring, respectively, to a character array with the figure data, the width, and height of the bitmap.

It is tempting to try to use the resource mechanism to construct the pixmap from the bitmap file. One could have the line

```
*button_1.labelPixmap: label_1.bitmap
```

in a resource file. This will not work because the bitmap produced will be only one bit per pixel deep while the Motif button expects a bitmap with full depth. The call to the pixmap creation function includes two arguments (foreground and background colors that were not part of the original bitmap file).

The code for a button labeled with a bitmap is sufficiently complex to discourage anyone from complaining that the basic Microsoft Window classes do not support such a feature. The numerous cryptic default arguments point to another problem with X. The intention is to provide full generality and in particular support applications that use multiple servers, and servers that have multiple screens. However, this provision increases the difficulty of writing the vast majority of applications that use only one server with one screen.

Resources in Microsoft Windows

The term "resources" is used with a different meaning in Microsoft Windows. Resources are part of the executable file, but are loaded in main memory only as they are needed. They are not specified in a C source file, but in a separate resource script. Listing 9 shows a resource script for some text labels.

Listing 9

```
#include ''mylabels.h''

STRINGTABLE
 {
 SAVE, ''Save''
 SAVE_AS, ''Save As''
 TRASH, ''Discard''
 }
```

The file `mylabels.h` contains the definitions of the symbolic constants `SAVE`, etc. The resource script should be part of a file called `draw.rc`. That file is compiled with an `rc` command with the result placed in a file `draw.res` that is then linked with the `draw.obj` and the various library files into the executable `draw.exe`. The `draw.c` file should contain the code shown in Listing 10.

Listing 10

```
#include ''mylabels.h''
/* . . . */
char label_buf[64];
/* . . . */
LoadString( hInstance, SAVE, label_buf, 64);
/* . . . */
```

Then the string `label_buf` could be used as the button label. The argument `hInstance` refers to the running process and is needed to identify the resources from the disc.

Windows in Microsoft Windows

In Microsoft Windows the term "window" has a meaning closer to that of an X widget. The Microsoft Foundation Classes (MFC) provide an object-oriented interface to Windows plus some enhancements in their functionality. The prototype of the function to create a window is shown in Listing 11.

Listing 11

```
HWND CreateWindow(
 char *, /* class name */
 char *, /* window title */
 int, /* window style */
 int int, /* x, y of upper left corner */
 int int, /* width and height */
 HWND, /* parent window handle */
 HWND, /* handle for menu subwindow */
 HINSTANCE, /* program instance */
 pointer, /* place for optional data, if none
 pass NULL */
 )
```

The type `HWND` stands for `window handle`, exactly the same role that the type `Window` (XID) plays in X. There are several similarities with X, but also significant differences. The function arguments appear to be a mixture of the `XCreateSimpleWindow()` (position and dimension specification) and `XtVaCreateWidget()` functions (class name).

The basic Microsoft Window provides the functionality of a collection of several X widgets: a shell widget for interaction with the window manager, a container widget, and, optionally, a menu. The menu subwindow can be omitted by setting the respective argument to NULL.

Some parameters correspond to X window properties: the window title and the program instance. The latter identifies the program that created the window. There is a major difference on how this information is handled in X and in Microsoft Windows. In X the program may attach the command line arguments (that include the program name) as a property to a window. Since this is an ASCII character string there is no restriction on what can be placed there. In contrast, the handle used in Microsoft Windows has a value provided at execution time that cannot be modified in an obvious way to yield another legitimate value. In addition, the value corresponds to an instance, so if we have two copies of an application running, we may distinguish between them. Because X may run over a network, there is no way to ensure such a close link between window and program.

In contrast to X, classes can be defined easily by initializing a class structure of 12 members and then registering it. Part of the code is shown in Listing 12.

Listing 12

```
LRESULT CALLBACK PlayAround( /* . . . */ )
{
  /* message handler code */
}

WNDCLASS WndClass;

Wndclass.lpszClassName = ''Play
 Program'' ; /* name */
Wndclass.lpfnWndProc = PlayAround; /* event
 handler */
/* . . . etc . . . */

(void)RegisterClass(&WndClass);
```

Once a class has been registered, the same structure, `WndClass`, can be used to create a new class. There exist predefined classes whose names might be used in the `CreateWindow()` without any other preparation.

Examples of Microsoft Window Classes

We provide here examples of Microsoft Window buttons. In contrast to X where each specific button type is a separate class, Windows has a general button class with name "button" and 10 different button types are defined as *styles:* `BS_PUSHBUTTON`, `BS_RADIOBUTTON`, `BS_CHECKBOX`, and so on. One of the styles is `BS_OWNERDRAW` that allows the application to create the label, possibly with a bitmap. Listing 13 is the counterpart of Listing 7.

Listing 13

```
static HWND hwndFrame;
static HWND hwndChoice[8];
 /* . . . */
hwndChoice[0] = CreateWindow(
 ''button'', /* class name */
 ''Save'', /* label */
 WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON, /*
   style */
 /* . . . position and size parameters . . .
   */
 hwndFrame, /* parent window */
 (HMENU)1, /* window ID number */
 hInst, /* Instance handle */
 );
```

The callback code is included in window procedure in response to the message of type `WM_COMMAND`.

The organization of the different types of classes is much simpler than in Motif (or other X Toolkits) which have almost as many button classes as MS Windows has styles. While the application designer has much more control over the appearance and functionality of Motif buttons compared to most MS buttons styles, that flexibility comes at the cost of complexity. For most applications the MS Windows button styles are quite adequate. For the few situations where these styles are not adequate, there is the `BS_OWNERDRAW` style. The applications programmer has to be concerned with button appearance only when it is absolutely necessary.

BeBox Interface Kit

The Be Kit software is written in C++ making full use of the object oriented features of the language. Applications start by creating a `BWindow` with a constructor function that has the prototype

```
BWindow(BRect frame, const char *title,
 window_type type, ulong flags, ulong
 workspaces = B_CURRENT_WORKSPACE)
```

The `frame` specifies the dimensions and location of the window, and `title` has the obvious meaning. `type` specifies

the window class. `B_DOCUMENT_WINDOW` has a title and scroll bars, so it is well suited for the display of text files. The parameter `flags` specifies whether the window can be moved, resized, minimized, and so on. In short, those two arguments specify characteristics that are associated with the X Shell widget. A window object is associated with a computational thread.

Subwindows are created as `BView` objects. Be Kit classes derived from `BView` include `BButton`, `BMenu`, and so on. A `BView` object is responsible for drawing and for handling messages delivered to window thread, basically the same functionality as an X Widget. The `BView` constructor is

```
BView(BRect frame, const char *name, ulong
 resizingMode, ulong flags)
```

The first two arguments have the obvious meaning, and `resizingMode` specifies how the redrawing should be done after resizing. `flags` specifies the types of notifications the object receives.

DOING GRAPHICS ON WINDOWS

BitBlt

All window systems rely on raster graphics that rely in turn on television technology. The display is drawn on piece of memory (refresh memory or frame buffer) that is continuously read, and its contents are used to specify the color of a particular screen location. (See GRAPHICS HARDWARE for more details.) A pixel is a location in the refresh memory that controls the color and intensity of a single spot on the screen.

The term BitBlt stands for bit block transfer and refers to the basic drawing operation in window systems: copying a block of pixels from one memory location to another. This appears to be a simple operation, except for one thing. Computer memory organization does not necessarily correspond to pixel organization. For example, suppose we have a screen with one bit per pixel and 16 bit words in memory. Since most machines require the same amount of time to copy a whole word than a part of it, if we copy pixel by pixel it may take 16 times as long than if we copy word by word.

Therefore, an efficient implementation of bit block transfer is essential. All window systems have a set of functions that copy a rectangular area from one piece of memory (source) to another (destination). The main function in MS Windows is actually called `BitBlt()`, in X it is called `XCopyArea()`. In general, these functions perform a Boolean operation between the contents of source and destination rather than simple copying. The type of the operation is specified by a constant that either is passed as an argument in the bit block transfer function (in the case of Windows) or is a parameter of the graphics environment (in the case of X). Table 1 shows some common bitwise logical operations and the name of the symbolic constants used in Windows and in X.

Table 1. Some Common Bitwise Logical Operations and the Names of the Symbolic Constants Used in Windows and in X

Operation	Symbol in X	Symbol in Windows
0	GXclear	BLACKNESS
src AND dst	GXand	SRCAND
src	GXcopy	SRCCOPY
src OR dest	GXor	SRCPAINT
src XOR dest	GXxor	SRCINVERT
1	GXset	WHITENESS

The exclusive OR (XOR) deserves some comments. It has been a popular operation because it allows for the use of the same call for drawing and for erasing a figure. XOR sets to 1 all bits where source and destination differ. All other bits are set to 0. Therefore the XOR of a pattern with itself is 0. In particular:

```
src XOR (src XOR dest) = (src XOR src) XOR
 dest = dest
```

This works fine in 1 bit displays, but there are problems in multibit displays if the drawing area has a background color that corresponds to a nonzero bit pattern. As an example, suppose that we have 3 bits per pixel with 100 corresponding to red, 010 to green, and 001 to blue. To achieve a yellow background, all pixels should contain the pattern 110. If we want to draw a red line using XOR and assign the source the 100 bit pattern, the result will not be red but green! (100 XOR 110 = 010). To have a red output we must use 010 as the source bit pattern. In general, to make sure we obtain the right color we must adjust the source color by applying one extra XOR operation with the destination color. Keeping track of all the XOR operation adds to the complexity of the code, especially if we want to draw in more than one color. This might not be a problem for experienced programmers, but it is troublesome for beginners. While both X and Microsoft Windows support the XOR operation, the Be Kit does not.

Drawing Line Segments

Listing 14 shows the code that draws a line segment from a point with coordinates `x1`, `y1` to one with coordinates `x2`, `y2` for X and MS Windows. We observe that there are three "mysterious" arguments in X and one in MS Windows.

Listing 14
Xlib:

```
XDrawLine(Dpy, win, gc, x1, y1, x2, y2);
```

Win32:

```
MoveToEx(hdc, x1, y1, NULL);
LineTo(hdc, x2, y2);
```

`Dpy` is pointer to the server, `win` is the window handle (XID), and `gc` the *graphics context;* `Dpy` points to a structure that contains information about the color, thickness, and style (dashed or solid) of the line segment to be drawn. Those values are set by earlier calls—for example, the call

```
XSetLineAttributes(Dpy, gc, 2, LineOnOffDash,
 CapButt, JoinMiter);
```

sets line thickness to 2 pixels, sets style to dashed, and specifies the shape of endpoints and corners according to predefined rules expressed by symbolic constants, a favorite practice in X.

`hdc` is a handle for device context that encompasses both window and graphics context information. It is usually obtained by a call such as

```
hdc = BeginPaint(hwnd, &ps);
```

where `hwnd` is a window handle and `ps` is a paint structure (type `PAINTSTRUCT`). The latter structure contains a clipping

rectangle that limits what you may draw on the window. If the window was partially obscured and must be redrawn because it is no longer obscured, the clipping rectangle encompasses only the area that needs to be redrawn. In X the information about the area to be redrawn is contained in the event structure while a clipping polygon (and not just a rectangle) is part of the graphics context.

The device context structure contains information both about the area to be painted on the window and the parameters that usually go with the X graphics context. Some of them are members of the device context structure itself (for example, the background color), others are grouped in substructures. One of them is the pen structure (type `HPEN`) that contains information about style, line thickness, and foreground color. To specify a dashed green line with thickness 2 we need the call

```
hPen = CreatePen( PS_DASH, 2, RGB(0, 255, 0)
 );
```

There is another function that creates a pen with additional attributes, including line ends and joints.

```
hPen = ExtCreatePen( PS_DASH | PS_ENDCAP_FLAT
 | PS_JOIN_MITER, 2, &lBrush, 0, NULL );
```

The Be Kit takes advantage of function name overloading that is supported in C++ and provides different versions of the same function for drawing straight lines.

Listing 15

```
BPoint p1, p2;

v.MovePenTo(p1);
v.StrokeLine(p2, pattern);
```

or

```
v.StrokeLine(p1, p2, pattern);
```

`pattern` corresponds to style; solid, dashed, and so on. The pen structures contains information only about thickness.

X uses the term `Pixmap` for a piece of memory where drawing operations are valid and which can be copied on a window. MS Windows uses the term `Bitmap` for the same concept. The term `Bitmap` is also used by X to denote a `Pixmap` with one bit per pixel. Device Independent Bitmaps is a Windows concept and refers to Bitmaps that are accompanied by a color correspondence table. All drawing operations that can be performed on window and can also be performed on a bitmap/pixmap and later copied to the window with a bitblt operation.

MS Windows drawing functions are closer to traditional graphics functions and reflect the historical development of graphics, and in particular vector graphics. There was a time when the pen structure was the only context that could be defined. The design also mixes window and graphics context information.

The designers of X were brave enough to start afresh, and therefore its graphics functions are much cleaner. Window and graphics context are separate. On the other hand the argument structure is needlessly cumbersome. All *Xlib* functions have a pointer to the server as their first argument. However, most applications use only one server, and thus the first argument of all their calls to *Xlib* are the same. It is a good idea to define macros in application programs that take only the essential arguments.

The Be Kit has left the line style outside the pen structure. A pattern contains information both about the style and color. Just to keep things interesting, the term "high color" is used for foreground and "low color" for background.

Communication between Applications

A popular feature of most GUIs is the ability to share data between applications, often through a "drag and drop" mechanism. The underlying process for such a transfer is provided by the window system. When the user selects a block of data in X the server is informed about the selection and registers a function that can recover the data. When the user "drops" the data in another window, that application requests the data from the server which recovers them by calling the function registered by the owner of the selection. Such a transfer requires that the first application be running at the time of the selection.

A *clipboard* selection uses a third application which keeps a copy of the selection, so the original owner need not be running when the selection is requested. MS Windows supports only clipboard selections.

From the application user viewpoint, selections through the clipboard mechanism require two steps: copy to the clipboard and copy from the clipboard. This is evident when text editors move a word from one place to another, which requires a selection action that highlights the word, a copy operation, a selection of the new place, and a paste operation. ("Paste" is the term used for copying from the clipboard.)

BIBLIOGRAPHY

1. R. W. Scheifler and J. Gettys, The X window system, *ACM Trans. Graphics,* **5** (2): 79–109, 1986.
2. R. W. Scheifler and J. Gettys, *X Window System,* 3rd ed., Burlington, MA: Digital Press, 1992.
3. P. J. Asente and R. R. Swick, *X Window System Toolkit,* Newton, MA: Digital Press/Butterworth-Heinemann, 1990.
4. A. Nye, *Xlib Programming Manual,* The Definite Guides to the X Window System, Vol. 1, Sebastopol, CA: O'Reilly & Associates, 1991.
5. A. Nye (ed.), *Xlib Reference Manual,* The Definite Guides to the X Window System, Vol. 2, 3rd ed., Sebastopol, CA: O'Reilly & Associates, 1992.
6. A. Nye and T. O'Reilly, *X Toolkit Intrinsics Programming Manual,* The Definite Guides to the X Window System, Vol. 4, Motif edition, Sebastopol, CA: O'Reilly & Associates, 1993.
7. D. Flanagan, *X Toolkit Intrinsics Reference Manual,* The Definite Guides to the X Window System, Vol. 5, 3rd ed., Sebastopol, CA: O'Reilly & Associates, 1992.
8. P. M. Ferguson, *Motif Reference Manual,* The Definite Guides to the X Window System, Vol. 6B, Sebastopol, CA: O'Reilly & Associates, 1993.
9. P. E. Kimball, *The X Toolkit Cookbook,* Englewood Cliffs, NJ: Prentice-Hall PTR, 1995.
10. E. F. Johnson and K. Reichard, *Advanced X Window Applications Programming,* New York: M&T Books, 1994.
11. T. Pavlidis, *Fundamentals of X Programming,* Boston: PWS-Kent Publishing, 1997.
12. A. King, *Inside Windows 95,* Redmont, WA: Microsoft Press, 1994.

13. N. W. Cluts, *Programming the Windows 95 User Interface,* Redmont, WA: Microsoft Press, 1995.
14. *Programmer's Guide to Microsoft Windows 95,* Redmont, WA: Microsoft Press, 1995.
15. *Programming with MFC,* Vol. 2 of Microsoft Visual C++ six-volume collection, Redmont, WA: Microsoft Press, 1995.
16. C. Petzold, *Programming Windows 95,* Redmont, WA: Microsoft Press, 1996.
17. *The Be Book* Accessible through the web site www.be.com/documentation/be_book/index.html.
18. A. von Hoff, S. Shaio, and O. Starbuck, *Hooked on Java,* Reading, MA: Addison-Wesley, 1996.
19. P. Niemeyer and J. Peck, *Exploring Java,* Sebastopol, CA: O'Reilly & Associates, 1996.
20. J. K. Ousterhout, *Tcl and the Tk Toolkit,* Reading, MA: Addison-Wesley, 1994.
21. R. Pike, Graphics in overlapping bitmap layers, *ACM Trans. Graphics,* **2** (2): 135–160, 1983.

THEO PAVLIDIS
SUNY